WebSphere MQ

Using Java
Using Java
Contents

Figures ........................................... ix  
Tables ............................................ xi  
About this book ................................. xiii  
Abbreviations used in this book.............. xiii  
Who this book is for ............................. xiii  
What you need to know to understand this book xiii  
How to use this book ............................ xiv  
Summary of changes ............................. xv  
Changes for this edition (SC34-6066-00) ........ xv  

Part 1. Guidance for users ................. 1  

Chapter 1. Getting started ................. 3  
What are WebSphere MQ classes for Java? 3  
What are WebSphere MQ classes for Java Message Service? 3  
Who should use WebSphere MQ Java? 3  
Connection options ............................ 4  
  Client connection ............................. 5  
  Bindings connection .......................... 5  
Prerequisites .................................... 6  

Chapter 2. Installation ...................... 7  
What is installed ............................... 7  
  Installation directories ........................ 7  
  Environment variables ........................ 8  
Web server configuration ...................... 10  
Running WebSphere MQ Java applications under the Java 2 Security Manager 10  

Chapter 3. Using WebSphere MQ classes for Java (WebSphere MQ base Java) .............. 13  
Configuring your queue manager to accept client connections ..................... 13  
  TCP/IP client .................................. 13  
Verifying with the sample application .................................................................. 14  
Running your own WebSphere MQ base Java programs .................................... 15  
Solving WebSphere MQ base Java problems ......................................................... 15  
  Tracing the sample application ................................................................. 15  
  Error messages ................................ 16  

Chapter 4. Using WebSphere MQ classes for Java Message Service (WebSphere MQ JMS) .............. 17  
JMS Postcard .................................... 17  
  Setting up JMS Postcard ....................... 17  
  Starting ......................................... 17  

Chapter 5. Using the WebSphere MQ JMS administration tool ............................. 39  
Invoking the Administration tool .......................... 39  
Configuration .................................... 40  
  Using an unlisted InitialContextFactory 41  
  Security ........................................ 41  
Configuring for WebSphere Application Server V3.5 ......................... 42  
Administration commands .......................... 42  
Manipulating subcontexts ......................... 43  
Administering JMS objects ........................ 43  
  Object types .................................... 43  
  Verbs used with JMS objects .................. 45  
  Creating objects ............................... 45  
  Properties ..................................... 46  
  Property dependencies ......................... 51  
SSL properties .................................. 53  
Sample error conditions ......................... 54  

Part 2. Programming with WebSphere MQ base Java .............. 55  

Chapter 6. Introduction for programmers ................................................ 57  
Why should I use the Java interface? .......................... 57  
The WebSphere MQ classes for Java interface 58  
Java Development Kit ............................ 58  
WebSphere MQ classes for Java class library 59
Chapter 8. Environment-dependent behavior

Chapter 9. The WebSphere MQ base Java classes and interfaces
Part 3. Programming with WebSphere MQ JMS

Chapter 10. Writing WebSphere MQ JMS programs

The JMS model
Building a connection
Retrieving the factory from JNDI
Using the factory to create a connection
Creating factories at runtime
Choosing client or bindings transport
Obtaining a session
Sending a message
Setting properties with the 'set' method
Message types
Receiving a message
Message selectors
Asynchronous delivery
Closing down
Java Virtual Machine hangs at shutdown
Handling errors
Exception listener
User Exits
Using Secure Sockets Layer (SSL)
SSL administrative properties

Chapter 11. Programming publish/subscribe applications

Introduction
Getting started with WebSphere MQ JMS and publish/subscribe
Choosing a broker
Setting up the broker to run the WebSphere MQ JMS
Writing a simple publish/subscribe application connecting through WebSphere MQ
Import required packages
Obtain or create JMS objects
Publish messages
Receive subscriptions
Close down unwanted resources
TopicConnectionFactory administered objects
Using topics
Topic names
Creating topics at runtime
Subscriber options
Creating non-durable subscribers
Creating durable subscribers
Using message selectors
Suppressing local publications
Combining the subscriber options
Configuring the base subscriber queue
Subscription stores
Solving publish/subscribe problems
Incomplete publish/subscribe close down
Subscriber cleanup utility
Manual cleanup
Cleanup from within a program
Handling broker reports

Chapter 12. JMS messages

Message selectors
Mapping JMS messages onto WebSphere MQ messages
The MQRFH2 header
JMS fields and properties with corresponding MQMD fields
Mapping JMS fields onto WebSphere MQ fields (outgoing messages)
Mapping WebSphere MQ fields onto JMS fields (incoming messages)
Mapping JMS to a native WebSphere MQ application
Message body

Chapter 13. WebSphere MQ JMS Application Server Facilities

ASF classes and functions
ConnectionConsumer
Planning an application
Error handling
Application server sample code
MyServerSession.java
MyServerSessionPool.java
MessageListenerFactory.java
Examples of ASF use
Load1.java
CountingMessageListenerFactory.java
ASFClient1.java
Load2.java
LoggingMessageListenerFactory.java
ASFClient2.java
TopicLoad.java
ASFClient3.java
ASFClient4.java
ASFClient5.java

Chapter 14. JMS interfaces and classes

Sun Java Message Service classes and interfaces
WebSphere MQ JMS classes
Contents

Appendix D. Connecting with other products ............. 409
Setting up a publish/subscribe broker ............. 409
Transformation and routing with WebSphere MQ Integrator V2 ............. 410

Appendix E. JMS JTA/XA interface with WebSphere Application Server V4 ............. 413
Using the JMS interface with WebSphere Application Server ............. 413
Administered objects ............. 413
Container-managed versus bean-managed transactions ............. 414
Two-phase commit versus one-phase optimization ............. 414
Defining administered objects ............. 414
Retrieving administration objects ............. 414
Samples ............. 414
Sample1 ............. 415
Sample2 ............. 416
Sample3 ............. 416

Appendix F. Using WebSphere MQ Java in applets with Java 1.2 or later ............. 419
Changing browser security settings ............. 419
Copying package class files ............. 420

Appendix G. Information for SupportPac MA1G ............. 421
Environments supported by SupportPac MA1G ............. 421
Obtaining and installing SupportPac MA1G ............. 421
Verifying installation using the sample program ............. 422
Features not provided by SupportPac MA1G ............. 422
Running WebSphere MQ base Java applications under CICS Transaction Server for OS/390 ............. 423
Restrictions under CICS Transaction Server ............. 423

Appendix H. SSL CipherSuites supported by WebSphere MQ ............. 425

Appendix I. JMS exception messages ............. 427

Appendix J. Notices ............. 441
Trademarks ............. 442

Index ............. 445

Sending your comments to IBM ............. 451
### Figures

1. WebSphere MQ classes for Java example applet ........................................ 63
2. WebSphere MQ classes for Java example application .................................. 66
3. WebSphere MQ classes for Java Message Service topic name hierarchy ........ 211
4. How messages are transformed between JMS and WebSphere MQ using the MQRFH2 header ........................................ 229
5. How JMS messages are transformed to WebSphere MQ messages (no MQRFH2 header) ........................................ 241
6. ServerSessionPool and ServerSession functionality .................................... 250
7. WebSphere MQ Integrator message flow ...................................................... 409
<table>
<thead>
<tr>
<th></th>
<th>Tables</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Platforms and connection modes</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Product installation directories</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Samples directories</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>Sample CLASSPATH statements for the product</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Environment variables for the product</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>Classes that are tested by IVT</td>
<td>32</td>
</tr>
<tr>
<td>7</td>
<td>Administration verbs</td>
<td>42</td>
</tr>
<tr>
<td>8</td>
<td>Syntax and description of commands used to manipulate subcontexts</td>
<td>43</td>
</tr>
<tr>
<td>9</td>
<td>The JMS object types that are handled by the administration tool</td>
<td>44</td>
</tr>
<tr>
<td>10</td>
<td>Syntax and description of commands used to manipulate administered objects</td>
<td>45</td>
</tr>
<tr>
<td>11</td>
<td>Property names and valid values</td>
<td>46</td>
</tr>
<tr>
<td>12</td>
<td>The valid combinations of property and object type</td>
<td>49</td>
</tr>
<tr>
<td>13</td>
<td>Character set identifiers</td>
<td>120</td>
</tr>
<tr>
<td>14</td>
<td>Set methods on MQQueueConnectionFactory</td>
<td>192</td>
</tr>
<tr>
<td>15</td>
<td>Property names for queue URIs</td>
<td>194</td>
</tr>
<tr>
<td>16</td>
<td>Symbolic values for queue properties</td>
<td>195</td>
</tr>
<tr>
<td>17</td>
<td>Possible values for NameValueCCSID field</td>
<td>231</td>
</tr>
<tr>
<td>18</td>
<td>MQRFH2 folders and properties used by JMS</td>
<td>231</td>
</tr>
<tr>
<td>19</td>
<td>Property datatype values and definitions</td>
<td>232</td>
</tr>
<tr>
<td>20</td>
<td>JMS header fields mapping to MQMD fields</td>
<td>233</td>
</tr>
<tr>
<td>21</td>
<td>JMS properties mapping to MQMD fields</td>
<td>234</td>
</tr>
<tr>
<td>22</td>
<td>JMS provider specific properties mapping to MQMD fields</td>
<td>234</td>
</tr>
<tr>
<td>23</td>
<td>Outgoing message field mapping</td>
<td>235</td>
</tr>
<tr>
<td>24</td>
<td>Outgoing message JMS property mapping</td>
<td>235</td>
</tr>
<tr>
<td>25</td>
<td>Outgoing message JMS provider specific property mapping</td>
<td>235</td>
</tr>
<tr>
<td>26</td>
<td>Incoming message JMS header field mapping</td>
<td>239</td>
</tr>
<tr>
<td>27</td>
<td>Incoming message property mapping</td>
<td>239</td>
</tr>
<tr>
<td>28</td>
<td>Incoming message provider specific JMS property mapping</td>
<td>240</td>
</tr>
<tr>
<td>29</td>
<td>Load1 parameters and defaults</td>
<td>254</td>
</tr>
<tr>
<td>30</td>
<td>ASFClient1 parameters and defaults</td>
<td>255</td>
</tr>
<tr>
<td>31</td>
<td>TopicLoad parameters and defaults</td>
<td>257</td>
</tr>
<tr>
<td>32</td>
<td>ASFClient3 parameters and defaults</td>
<td>258</td>
</tr>
<tr>
<td>33</td>
<td>Summary of interfaces in package javax.jms</td>
<td>261</td>
</tr>
<tr>
<td>34</td>
<td>Summary of classes in package javax.jms</td>
<td>263</td>
</tr>
<tr>
<td>35</td>
<td>Summary of classes in package com.ibm.mq.jms</td>
<td>264</td>
</tr>
<tr>
<td>36</td>
<td>Summary of classes in package com.ibm.jms</td>
<td>265</td>
</tr>
<tr>
<td>37</td>
<td>Comparison of representations of property values within the administration tool and within programs</td>
<td>399</td>
</tr>
<tr>
<td>38</td>
<td>Utilities supplied with WebSphere MQ classes for Java Message Service</td>
<td>401</td>
</tr>
<tr>
<td>39</td>
<td>Attribute settings for javaCodebase</td>
<td>404</td>
</tr>
<tr>
<td>40</td>
<td>Attribute settings for javaClassNames</td>
<td>404</td>
</tr>
<tr>
<td>41</td>
<td>Attribute settings for javaClassNames</td>
<td>404</td>
</tr>
<tr>
<td>42</td>
<td>Attribute settings for javaFactory</td>
<td>405</td>
</tr>
<tr>
<td>43</td>
<td>Attribute settings for javaReferenceAddress</td>
<td>405</td>
</tr>
<tr>
<td>44</td>
<td>Attribute settings for javaSerializedData</td>
<td>405</td>
</tr>
<tr>
<td>45</td>
<td>objectClass definition for javaSerializedObject</td>
<td>405</td>
</tr>
<tr>
<td>46</td>
<td>objectClass definition for javaObject</td>
<td>406</td>
</tr>
<tr>
<td>47</td>
<td>objectClass definition for javaContainer</td>
<td>406</td>
</tr>
<tr>
<td>48</td>
<td>objectClass definition for javaNamingReference</td>
<td>406</td>
</tr>
<tr>
<td>49</td>
<td>CipherSpecs and matching CipherSuites</td>
<td>425</td>
</tr>
</tbody>
</table>
About this book

This book describes:

- WebSphere MQ classes for Java, which can be used to access WebSphere MQ systems
- WebSphere MQ classes for Java Message Service, which can be used to access both Java Message Service (JMS) and WebSphere MQ applications

Note: The README file should be consulted for information that expands and corrects information in this book. The README file is installed with the WebSphere MQ Java code and can be found in the doc subdirectory.

Abbreviations used in this book

The following abbreviations are used throughout this book:

**WebSphere MQ Java**
WebSphere MQ classes for Java and WebSphere MQ classes for Java Message Service combined

**WebSphere MQ base Java**
WebSphere MQ classes for Java

**WebSphere MQ JMS**
WebSphere MQ classes for Java Message Service

Who this book is for

This information is written for programmers who are familiar with the procedural WebSphere MQ application programming interface as described in the WebSphere MQ Application Programming Guide and shows how to transfer this knowledge to become productive with the WebSphere MQ Java programming interfaces.

What you need to know to understand this book

You should have:

- Knowledge of the Java programming language
- Understanding of the purpose of the Message Queue Interface (MQI) as described in the MQ Application Programming Guide and the chapter about Call Descriptions in the WebSphere MQ Application Programming Reference
- Experience of WebSphere MQ programs in general, or familiarity with the content of the other WebSphere MQ publications

Users intending to use the WebSphere MQ base Java with CICS® Transaction Server for OS/390® should also be familiar with:

- Customer Information Control System (CICS) concepts
- Using the CICS Java Application Programming Interface (API)
- Running Java programs from within CICS
About this book

Users intending to use VisualAge® for Java to develop OS/390 UNIX® System Services High Performance Java (HPJ) applications should be familiar with the Enterprise Toolkit for OS/390 (supplied with VisualAge for Java Enterprise Edition for OS/390, Version 2).

How to use this book

Part 1 of this book describes the use of WebSphere MQ base Java and WebSphere MQ JMS, Part 2 helps programmers wanting to use WebSphere MQ base Java, and Part 3 helps programmers wanting to use WebSphere MQ JMS.

First, read the chapters in Part 1 that introduce you to WebSphere MQ base Java and WebSphere MQ JMS. Then use the programming guidance in Part 2 or 3 to understand how to use the classes to send and receive WebSphere MQ messages in the environment you wish to use.

Remember to check the README file installed with the WebSphere MQ Java code for later or more specific information for your environment.
Summary of changes

This section describes changes in this edition of *WebSphere MQ Using Java*. Changes since the previous edition of the book are marked by vertical lines to the left of the changes.

### Changes for this edition (SC34-6066-00)

This edition includes the following changes:
- Changes for
  - Integration with WebSphere MQ product
  - JMS Postcard
  - Secure Sockets Layer (SSL) support
- Miscellaneous corrections and clarifications.
Changes
Part 1. Guidance for users

Chapter 1. Getting started
What are WebSphere MQ classes for Java? 3
What are WebSphere MQ classes for Java Message Service? 3
Who should use WebSphere MQ Java? 3
Connection options 4
Client connection 5
Bindings connection 5
Prerequisites 6

Chapter 2. Installation
What is installed 7
Installation directories 8
Environment variables 8
Web server configuration 10
Running WebSphere MQ Java applications under the Java 2 Security Manager 10

Chapter 3. Using WebSphere MQ classes for Java (WebSphere MQ base Java)
Configuring your queue manager to accept client connections 13
TCP/IP client 13
Verifying with the sample application 14
Running your own WebSphere MQ base Java programs 15
Solving WebSphere MQ base Java problems 15
Tracing the sample application 15
Error messages 16

Chapter 4. Using WebSphere MQ classes for Java Message Service (WebSphere MQ JMS)
JMS Postcard 17
Setting up JMS Postcard 17
Starting 17
Signon 18
Signon advanced options 18
Sending a postcard 18
Running JMS Postcard with one queue manager 18
Running JMS Postcard with two queue managers 19
JMS Postcard configuration 20
JMS Postcard default configuration 20
How JMS Postcard works 20
Starting up 20
Receiving messages 20
Sending messages 21
How the postcards get there 21
Tidying up undeliverable messages 21
Exchanging messages between different WebSphere MQ Postcard applications 21
Customizing JMS Postcard 22
Post installation setup 23
Additional setup for publish/subscribe mode 24

Chapter 4. Using WebSphere MQ classes for Java Message Service (WebSphere MQ JMS)
JMS Postcard 17
Setting up JMS Postcard 17
Starting 17
Signon 18
Signon advanced options 18
Sending a postcard 18
Running JMS Postcard with one queue manager 18
Running JMS Postcard with two queue managers 19
JMS Postcard configuration 20
JMS Postcard default configuration 20
How JMS Postcard works 20
Starting up 20
Receiving messages 20
Sending messages 21
How the postcards get there 21
Tidying up undeliverable messages 21
Exchanging messages between different WebSphere MQ Postcard applications 21
Customizing JMS Postcard 22
Post installation setup 23
Additional setup for publish/subscribe mode 24

Chapter 5. Using the WebSphere MQ JMS administration tool
Invoking the Administration tool 39
Configuration 40
Using an unlisted InitialContextFactory 41
Security 41
Configuring for WebSphere Application Server V3.5 42
Administration commands 42
Manipulating subcontexts 43
Administering JMS objects 43
Object types 43
Verbs used with JMS objects 45
Creating objects 45
LDAP naming considerations 45
Properties 46
Property dependencies 51
The ENCODING property 52
SSL properties 53
Sample error conditions 54

For a broker running on a remote queue manager 25
Queues that require authorization for non-privileged users 26
Using the sample JMS applet to verify the TCP/IP client 27
Using the sample applet with OS/400 27
Running the sample applet 27
Tracing the sample as an application 28
Running the point-to-point IVT 28
Point-to-point verification without JNDI 29
Point-to-point verification with JNDI 30
IVT error recovery 31
The publish/subscribe installation verification test 32
Publish/subscribe verification without JNDI 32
Publish/subscribe verification with JNDI 34
PSIVT error recovery 35
Running your own WebSphere MQ JMS programs 35
Solving problems 36
Tracing programs 36
Logging 36

© Copyright IBM Corp. 1997, 2002
Chapter 1. Getting started

This chapter gives an overview of WebSphere MQ classes for Java and WebSphere MQ classes for Java Message Service, and their uses.

What are WebSphere MQ classes for Java?

WebSphere MQ classes for Java (also referred to as WebSphere MQ base Java) allow a program written in the Java programming language to:

- Connect to WebSphere MQ as a WebSphere MQ client
- Connect directly to a WebSphere MQ server

WebSphere MQ base Java enables Java applets, applications, and servlets to issue calls and queries to WebSphere MQ. This gives access to mainframe and legacy applications, typically over the Internet, without necessarily having any other WebSphere MQ code on the client machine. With WebSphere MQ base Java, the user of an Internet terminal can become a true participant in transactions, rather than just a giver and receiver of information.

What are WebSphere MQ classes for Java Message Service?

WebSphere MQ classes for Java Message Service (also referred to as WebSphere MQ JMS) is a set of Java classes that implement Sun's Java Message Service (JMS) interfaces to enable JMS programs to access WebSphere MQ systems. Both the point-to-point and publish/subscribe models of JMS are supported.

The use of WebSphere MQ JMS as the API to write WebSphere MQ applications has a number of benefits. Some advantages derive from JMS being an open standard with multiple implementations. Other advantages result from additional features that are present in WebSphere MQ JMS, but not in WebSphere MQ base Java.

Benefits arising from the use of an open standard include:

- The protection of investment, both in skills and application code
- The availability of people skilled in JMS application programming
- The ability to plug in different JMS implementations to fit different requirements


The extra function provided over WebSphere MQ base Java includes:

- Asynchronous message delivery. Messages can be delivered to an application as they arrive, on a separate thread.
- Message selectors.
- Support for publish/subscribe messaging.
- Structured, more abstract, message classes. Implementation details are left to the JMS provider.

Who should use WebSphere MQ Java?

If your enterprise fits any of the following scenarios, you can gain significant advantage by using WebSphere MQ classes for Java and WebSphere MQ classes for Java Message Service:
Who should use WebSphere MQ Java

- A medium or large enterprise that is introducing intranet-based client/server solutions. Here, Internet technology provides low cost easy access to global communications, while WebSphere MQ connectivity provides high integrity with assured delivery and time independence.

- A medium or large enterprise with a need for reliable business-to-business communications with partner enterprises. Here again, the Internet provides low-cost easy access to global communications, while WebSphere MQ connectivity provides high integrity with assured delivery and time independence.

- A medium or large enterprise that wishes to provide access from the public Internet to some of its enterprise applications. Here, the Internet provides global reach at a low cost, while WebSphere MQ connectivity provides high integrity through the queuing paradigm. In addition to low cost, the business can achieve improved customer satisfaction through 24 hour a day availability, fast response, and improved accuracy.

- An Internet Service provider, or other Value Added Network provider. These companies can exploit the low cost and easy communications provided by the Internet. They can also add value with the high integrity provided by WebSphere MQ connectivity. An Internet Service provider that exploits WebSphere MQ can immediately acknowledge receipt of input data from a Web browser, guarantee delivery, and provide an easy way for the user of the Web browser to monitor the status of the message.

WebSphere MQ and WebSphere MQ classes for Java Message Service provide an excellent infrastructure for access to enterprise applications and for development of complex Web applications. A service request from a Web browser can be queued then processed when possible, thus allowing a timely response to be sent to the end user, regardless of system loading. By placing this queue “close” to the user in network terms, the load on the network does not impact the timeliness of the response. Also, the transactional nature of WebSphere MQ messaging means that a simple request from the browser can be expanded safely into a sequence of individual back end processes in a transactional manner.

WebSphere MQ classes for Java also enables application developers to exploit the power of the Java programming language to create applets and applications that can run on any platform that supports the Java runtime environment. These factors combine to reduce the development time for multi-platform WebSphere MQ applications significantly. Also, if there are enhancements to applets in the future, end users automatically pick these up as the applet code is downloaded.

Connection options

Programmable options allow WebSphere MQ Java to connect to WebSphere MQ in either of the following ways:

- As a WebSphere MQ client using Transmission Control Protocol/Internet Protocol (TCP/IP)
- In bindings mode, connecting directly to WebSphere MQ

Table 1 on page 5 shows which of these connection modes can be used for each platform.

In addition, WebSphere MQ JMS publish/subscribe applications can connect directly across TCP/IP to the IBM WebSphere MQ Event Broker program. For more information about this connection see Chapter 11, “Programming publish/subscribe applications” on page 203.
Table 1. Platforms and connection modes

<table>
<thead>
<tr>
<th>Server platform</th>
<th>Standard Client</th>
<th>Bindings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows NT®</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Windows® 2000</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>AIX®</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sun Solaris (v2.6, v2.8, V7, or SunOS v5.6, v5.7)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>OS/400®</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>HP-UX</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>OS/390 and z/OS™</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Linux on Intel</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Linux on zSeries™</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

Notes:
1. HP-UX Java bindings support is available only for HP-UXv11 systems running the POSIX draft 10 pthreaded version of WebSphere MQ.
2. On Linux on zSeries and Windows 98, only TCP/IP client connectivity is supported.

The following sections describe these options in more detail.

Client connection
To use WebSphere MQ Java as a WebSphere MQ client, you can install it either on the WebSphere MQ server machine, which may also contain a Web server, or on a separate machine. If you install WebSphere MQ Java on the same machine as a Web server, an advantage is that you can download and run WebSphere MQ client applications on machines that do not have WebSphere MQ Java installed locally.

Wherever you choose to install the client, you can run it in three different modes:

From within any Java-enabled Web browser
In this mode, the locations of the WebSphere MQ queue managers that can be accessed may be constrained by the security restrictions of the browser that is used.

Using an appletviewer
To use this method, you must have the Java Development Kit (JDK™) or Java Runtime Environment (JRE) installed on the client machine.

As a standalone Java program or in a Web application server
To use this method, you must have the Java Development Kit (JDK) or Java Runtime Environment (JRE) installed on the client machine.

Bindings connection
When used in bindings mode, WebSphere MQ Java uses the Java Native Interface (JNI) to call directly into the existing queue manager API, rather than communicating through a network. This provides better performance for WebSphere MQ applications than using network connections. Unlike the client mode, applications that are written using the bindings mode cannot be downloaded as applets.
Connections

To use the bindings connection, WebSphere MQ Java must be installed on the WebSphere MQ server.

Prerequisites

To run WebSphere MQ base Java, you require the following software:

- WebSphere MQ for the server platform you wish to use.
- Java Development Kit (JDK) for the server platform.
- Java Development Kit, or Java Runtime Environment (JRE), or Java-enabled Web browser for client platforms. (See “Client connection” on page 5.)
- For z/OS & OS/390, OS/390 Version 2 Release 9 or higher, or z/OS, with UNIX System Services (USS).
- For OS/400, the iSeries™ Developer Kit for Java, 5769-JV1, and the Qshell Interpreter, OS/400 (5769-SS1) Option 30.

The following list shows the supported Java 2 Software Development Kits and Java Runtime Environments:

- IBM Developer Kit for AIX, Java Technology Edition, Version 1.3.1
  Some limitations apply to the Linux environment.
- IBM Developer Kit for OS/390, Java Technology Edition, Version 1.3.1
- IBM Developer Kit for Windows, Java Technology Edition, Version 1.3.1
- IBM iSeries Developer Kit for Java, Version 1.3
- HP-UX SDK, for the Java platform, Version 1.3.1
- Java 2 Standard Edition, for the Solaris Operating Environment, SDK 1.3.1

For full provision of Secure Socket Layer (SSL) authentication to enable WebSphere MQ Java and Java Message Service (JMS) applications to benefit from secure connection to the queue manager (authentication, message integrity, data encryption), a Java Runtime Environment at Version 1.4.0 is required for your platform.

Please check the README file for the latest information about operating system levels this product has been tested against.

To use the WebSphere MQ JMS administration tool (see Chapter 5, “Using the WebSphere MQ JMS administration tool” on page 39), you require the following additional software:

- At least one of the following service provider packages:
  - File system - fscontext.jar, providerutil.jar.
- A Java Naming and Directory Service (JNDI) service provider. This is the resource that stores physical representations of the administered objects. Users of WebSphere MQ JMS will probably use an LDAP server for this purpose, but the tool also supports the use of the file system context service provider. If an LDAP server is used, it must be configured to store JMS objects. For information to assist with this configuration, refer to Appendix C, “LDAP schema definition for storing Java objects” on page 405.

To use the X/Open/XA facilities of WebSphere MQ JMS on OS/400 you need a specific PTF. Please check the README file for further information.
Chapter 2. Installation

This chapter describes how the WebSphere MQ classes for Java and WebSphere MQ classes for Java Message Service code is installed.

What is installed

The latest versions of both WebSphere MQ base Java and WebSphere MQ JMS (together known as WebSphere MQ Java) are installed with WebSphere MQ. You may need to override default installation options to make sure this is done.

Refer to the following books for more information about installing WebSphere MQ:

- WebSphere MQ for AIX, V5.3 Quick Beginnings for AIX
- WebSphere MQ for HP-UX, V5.3 Quick Beginnings for HP-UX
- WebSphere MQ for iSeries V5.3 Quick Beginnings for OS/400
- WebSphere MQ for Linux for Intel and Linux for zSeries, V5.3 Quick Beginnings for Linux
- WebSphere MQ for Solaris, V5.3 Quick Beginnings for Solaris
- WebSphere MQ for Windows, V5.3 Quick Beginnings for Windows systems
- WebSphere MQ for z/OS System Setup Guide for z/OS

WebSphere MQ base Java is contained in the following Java .jar files:

- **com.ibm.mq.jar**
  - This code includes support for all the connection options.

- **com.ibm.mqbind.jar**
  - This code supports only the bindings connection and is not supplied or supported on all platforms. We recommend that you do not use it in any new applications.

WebSphere MQ JMS is contained in the following Java .jar file:

- **com.ibm.mqjms.jar**

The following Java libraries from Sun Microsystems are redistributed with the WebSphere MQ JMS product:

- **connector.jar**
  - Version 1.0

- **fscontext.jar**
  - Early Access 4 Release

- **jms.jar**
  - Version 1.0.2

- **jndi.jar**
  - Version 1.1.2 (except for z/OS & OS/390)

- **ldap.jar**
  - Version 1.0.3 (except for z/OS & OS/390)

- **providerutil.jar**
  - Version 1.0

When installation is complete, files and samples are installed in the locations shown in "Installation directories" on page 8.

After installation, you must update your environment variables, as shown in "Environment variables" on page 8.
What is installed

**Note:** Take care if you install the product, then subsequently install or reinstall a version of SupportPac™ MA88. This may cause your WebSphere MQ Java support to revert to an earlier level.

**Installation directories**

The WebSphere MQ Java V5.3 files are installed in the directories shown in Table 2.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX</td>
<td>/usr/mqm/java/</td>
</tr>
<tr>
<td>z/OS &amp; OS/390</td>
<td>install_dir/mqm/java/</td>
</tr>
<tr>
<td>iSeries &amp; AS/400®</td>
<td>/QIBM/ProdData/mqm/java/</td>
</tr>
<tr>
<td>HP-UX and Sun Solaris</td>
<td>/opt/mqm/java/</td>
</tr>
<tr>
<td>Linux</td>
<td>install_dir/mqm/java/</td>
</tr>
<tr>
<td>Windows systems</td>
<td>Program Files\IBM\WebSphere MQ\java</td>
</tr>
</tbody>
</table>

**Note:** install_dir is the directory in which you installed the product. On Linux, this is likely to be /opt and on z/OS & OS/390 it is likely to be /usr/lpp.

Some sample programs, such as the Installation Verification Programs (IVP), are supplied. Table 3 lists the directory path to these on different platforms. WebSphere MQ base Java samples are within a subdirectory base and WebSphere MQ JMS samples are within a subdirectory jms.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX</td>
<td>/usr/mqm/samp/java/</td>
</tr>
<tr>
<td>z/OS &amp; OS/390</td>
<td>install_dir/mqm/java/samples/</td>
</tr>
<tr>
<td>iSeries &amp; AS/400®</td>
<td>/QIBM/ProdData/mqm/java/samples/</td>
</tr>
<tr>
<td>HP-UX and Sun Solaris</td>
<td>/opt/mqm/samp/java/</td>
</tr>
<tr>
<td>Linux</td>
<td>install_dir/mqm/samp/java/</td>
</tr>
<tr>
<td>Windows systems</td>
<td>Program Files\IBM\WebSphere MQ\tools\Java\</td>
</tr>
</tbody>
</table>

**Note:** install_dir is the directory in which you installed the product. On Linux, this is likely to be /opt and on z/OS & OS/390 it is likely to be /usr/lpp.

**Environment variables**

After installation, you must update your CLASSPATH environment variable to include the WebSphere MQ base Java code and samples directories. Table 4 on page 9 shows typical CLASSPATH settings for the various platforms.

WebSphere MQ Java uses other environment variables. Some are platform dependent and are listed in Table 5 on page 10. Two are common across platforms, MQ_JAVA_INSTALL_PATH and MQ_JAVA_DATA_PATH. On Windows systems, these variables are automatically set by the installation program, but on other platforms you need to set them manually to complete installation.

MQ_JAVA_INSTALL_PATH points to the product installation directory, as shown in Table 2. MQ_JAVA_DATA_PATH points to the root directory for logging and tracing, and is included so that you can use the same directory for WebSphere MQ
Java and the base WebSphere MQ product.

Table 4. Sample CLASSPATH statements for the product

<table>
<thead>
<tr>
<th>Platform</th>
<th>Sample CLASSPATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX</td>
<td>CLASSPATH=/usr/mqm/java/lib/com.ibm.mq.jar:</td>
</tr>
<tr>
<td></td>
<td>/usr/mqm/java/lib/connector.jar:</td>
</tr>
<tr>
<td></td>
<td>/usr/mqm/java/lib:</td>
</tr>
<tr>
<td></td>
<td>/usr/mqm/samp/java/base:</td>
</tr>
<tr>
<td>HP-UX and Sun Solaris</td>
<td>CLASSPATH=/opt/mqm/java/lib/com.ibm.mq.jar:</td>
</tr>
<tr>
<td></td>
<td>/opt/mqm/java/lib/connector.jar:</td>
</tr>
<tr>
<td></td>
<td>/opt/mqm/java/lib:</td>
</tr>
<tr>
<td></td>
<td>/opt/mqm/samp/java/base:</td>
</tr>
<tr>
<td>Windows systems</td>
<td>CLASSPATH=mg_root_dir1\java\lib\com.ibm.mq.jar:</td>
</tr>
<tr>
<td></td>
<td>mg_root_dir\java\lib\connector.jar:</td>
</tr>
<tr>
<td></td>
<td>mg_root_dir\java\lib:</td>
</tr>
<tr>
<td></td>
<td>mg_root_dir\tools\java\base;</td>
</tr>
<tr>
<td>z/OS &amp; OS/390</td>
<td>CLASSPATH=install_dir2\mqm/java/lib/com.ibm.mq.jar:</td>
</tr>
<tr>
<td></td>
<td>install_dir/mqm/java/lib/connector.jar:</td>
</tr>
<tr>
<td></td>
<td>install_dir/mqm/java/lib:</td>
</tr>
<tr>
<td></td>
<td>install_dir/mqm/java/samples/base:</td>
</tr>
<tr>
<td>iSeries &amp; AS/400</td>
<td>CLASSPATH=/QIBM/ProdData/mqm/java/lib/com.ibm.mq.jar:</td>
</tr>
<tr>
<td></td>
<td>/QIBM/ProdData/mqm/java/lib/connector.jar:</td>
</tr>
<tr>
<td></td>
<td>/QIBM/ProdData/mqm/java/lib:</td>
</tr>
<tr>
<td></td>
<td>/QIBM/ProdData/mqm/java/samples/base:</td>
</tr>
<tr>
<td>Linux</td>
<td>CLASSPATH=install_dir2\mqm/java/lib/com.ibm.mq.jar:</td>
</tr>
<tr>
<td></td>
<td>install_dir/mqm/java/lib/connector.jar:</td>
</tr>
<tr>
<td></td>
<td>install_dir/mqm/java/lib:</td>
</tr>
<tr>
<td></td>
<td>install_dir/mqm/samp/java/base:</td>
</tr>
</tbody>
</table>

Notes:

1. mg_root_dir stands here for the directory used to install WebSphere MQ on Windows systems. This is normally C:\Program Files\IBM\WebSphere MQ\.
2. install_dir is the directory in which you installed the product.

To use WebSphere MQ JMS, you must include additional jar files in the classpath. These are listed in “Post installation setup” on page 23.

If there are existing applications with a dependency on the deprecated bindings package com.ibm.mqbind, you must also add the file com.ibm.mqbind.jar to your classpath.

You must update additional environment variables on some platforms, as shown in Table 5 on page 10.
### Installation directories

<table>
<thead>
<tr>
<th>Platform</th>
<th>Environment variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX</td>
<td>LD_LIBRARY_PATH=/usr/mqm/java/lib</td>
</tr>
<tr>
<td>HP-UX</td>
<td>SHLIB_PATH=/opt/mqm/java/lib</td>
</tr>
<tr>
<td>Sun Solaris</td>
<td>LD_LIBRARY_PATH=/opt/mqm/java/lib</td>
</tr>
<tr>
<td>Windows systems</td>
<td>PATH=install_dir\lib</td>
</tr>
<tr>
<td>z/OS &amp; OS/390</td>
<td>LIBPATH=install_dir/mqm/java/lib</td>
</tr>
<tr>
<td>Linux</td>
<td>LD_LIBRARY_PATH=install_dir/mqm/java/lib</td>
</tr>
</tbody>
</table>

**Note:** install_dir is the installation directory for the product

**Notes:**

1. To use WebSphere MQ Bindings for Java on OS/400, ensure that the library QMQMJAVA is in your library list.
2. Ensure that you append the WebSphere MQ variables and do not overwrite any of the existing system environment variables. If you overwrite existing system environment variables, the application might fail during compilation or at runtime.

### Web server configuration

If you install WebSphere MQ Java on a Web server, you can download and run WebSphere MQ Java applications on machines that do not have WebSphere MQ Java installed locally. To make the WebSphere MQ Java files accessible to your Web server, you must set up your Web server configuration to point to the directory where the client is installed. Consult your Web server documentation for details of how to configure this.

**Note:** On z/OS & OS/390, the installed classes do not support client connection and cannot be usefully downloaded to clients. However, jar files from another platform can be transferred to z/OS & OS/390 and served to clients.

### Running WebSphere MQ Java applications under the Java 2 Security Manager

WebSphere MQ Java supports operation with the Java 2 Security Manager enabled. To successfully run applications with the Security Manager enabled, you must configure your JVM with a suitable policy definition file.

The simplest way to do this is to change the policy file supplied with the JRE. On most systems this file is stored in the path lib/security/java.policy, relative to your JRE directory. Policy files can be edited by hand or using the policytool program supplied with your JRE.

The changes that need to be made are to give authority to the com.ibm.mq.jar and com.ibm.mqjms.jar files so they can:
- Create sockets (in client mode)
- Load the native library (in bindings mode)
- Read various properties from the environment
Here is an example of a policy file entry which will allow the WebSphere MQ Java to run successfully under the default security manager. The string “/opt/mqm” in this example should be replaced with the location where WebSphere MQ Java is installed on your system.

```java
grant codeBase "file:/opt/mqm/java/lib/com.ibm.mq.jar" {
    permission java.net.SocketPermission "+\*","connect";
    permission java.lang.RuntimePermission "loadLibrary.*";
};

grant codeBase "file:/opt/mqm/java/lib/com.ibm.mqjms.jar" {
    permission java.util.PropertyPermission "MQJMS_LOG_DIR","read";
    permission java.util.PropertyPermission "MQJMS_TRACE_LEVEL","read";
    permission java.util.PropertyPermission "MQJMS_TRACE_DIR","read";
    permission java.util.PropertyPermission "MQ_JAVA_INSTALL_PATH","read";
    permission java.util.PropertyPermission "file.separator","read";
    permission java.util.PropertyPermission "user.name","read";
    permission java.util.PropertyPermission "com.ibm.mq.jms.cleanup","read";
};
```

This example of a policy file will enable the WebSphere MQ Java classes to work correctly under the security manager, but you may still need to enable your own code to run correctly before your applications will work.

The sample code shipped with WebSphere MQ Java has not been specifically enabled for use with the security manager; however the IVT tests do run with the above policy file and the default security manager in place.
Chapter 3. Using WebSphere MQ classes for Java (WebSphere MQ base Java)

This chapter describes:

- How to configure your system to run the sample applet and application programs to verify your WebSphere MQ base Java installation.
- How to modify the procedures to run your own programs.

Remember to check the README file installed with the WebSphere MQ Java code for later or more specific information for your environment.

The procedures depend on the connection option you want to use. Follow the instructions in the section that is appropriate for your requirements.

Configuring your queue manager to accept client connections

Use the following procedures to configure your queue manager to accept incoming connection requests from the clients.

**TCP/IP client**

1. Define a server connection channel using the following procedures:

   **For the OS/400 platform:**
   
   a. Start your queue manager by using the STRMQM command.
   b. Define a sample channel called JAVA.CHANNEL by issuing the following command:

      ```
      CRTMQMCHL CHLNAME(JAVA.CHANNEL) CHLTYPE(*SVRCN) MQMNAME(QMGRNAME) MCAUSERID(SOMEUSERID) TEXT('Sample channel for WebSphere MQ classes for Java')
      ```

      where QMGRNAME is the name of your queue manager, and SOMEUSERID is an OS/400 user ID with appropriate authority to the WebSphere MQ resources.

   **For z/OS or OS/390 platforms:**

   Note: You must have the Client attachment feature installed on your target queue manager in order to connect via TCP/IP.
   
   a. Start your queue manager by using the START QMGR command.
   b. Define a sample channel called JAVA.CHANNEL by issuing the following command:

      ```
      DEFCchl('JAVA.CHANNEL') CHLTYPE(SVRCONN) TRPTYPE(TCP) DESCR('Sample channel for WebSphere MQ classes for Java')
      ```

   **For other platforms:**

   a. Start your queue manager by using the strmqm command.
   b. Type the following command to start the runmqsc program:

      ```
      runmqsc [QMNAME]
      ```
   c. Define a sample channel called JAVA.CHANNEL by issuing the following command:
Using WebSphere MQ base Java

2. Start a listener program with the following commands:

   For OS/2® and Windows operating systems:
   Issue the command:
   runmqlsr -t tcp [-m QMNAME] -p 1414

   Note: If you use the default queue manager, you can omit the -m option.

   For UNIX operating systems:
   Configure the inetd daemon, so that the inetd starts the WebSphere MQ
   channels. See WebSphere MQ Clients for instructions on how to do this.

   For the OS/400 operating system:
   Issue the command:
   STRMQMLSR MQMNAME(QMGRNAME)

   where QMGRNAME is the name of your queue manager.

   For the z/OS or OS/390 operating system:
   a. Ensure your channel initiator is started. If not, start it by issuing the
      START CHINIT command.
   b. Start the listener by issuing the command START LISTENER
      TRPTYPE(TCP) PORT(1414)

Verifying with the sample application

An installation verification program, MQIVP, is supplied with WebSphere MQ base
Java. You can use this application to test all the connection modes of WebSphere
MQ base Java. The program prompts for a number of choices and other data to
determine which connection mode you want to verify. Use the following procedure
to verify your installation:

1. To test a client connection:
   a. Configure your queue manager, as described in "Configuring your queue
      manager to accept client connections" on page 13
   b. Carry out the rest of this procedure on the client machine.

   To test a bindings connection, carry out the rest of this procedure on the
   WebSphere MQ server machine.

2. Change to your samples directory.
   See Table 3 on page 8 to find where this is.

3. Type:
   java MQIVP

   The program tries to:
   a. Connect to, and disconnect from, the named queue manager.
   b. Open, put, get, and, close the system default local queue.
   c. Return a message if the operations are successful.

4. At prompt (1):
   • To use a TCP/IP connection, enter a WebSphere MQ server hostname.
   • To use native connection (bindings mode), leave the field blank. (Do not
     enter a name.)
Installation verification program

Here is an example of the prompts and responses you may see. The actual prompts and your responses depend on your WebSphere MQ network.

Please enter the IP address of the MQ server: ipaddress
Please enter the port to connect to: 1414
Please enter the server connection channel name: channelname
Please enter the queue manager name: qmname
Success: Connected to queue manager.
Success: Opened SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Put a message to SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Got a message from SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Closed SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Disconnected from queue manager

Tests complete -
SUCCESS: This MQ Transport is functioning correctly.
Press Enter to continue ...

Notes:
1. If you choose server connection, you do not see the prompts marked (2).
2. On z/OS & OS/390, leave the field blank at prompt (1).
3. On OS/400, you can run the command java MQIVP only from the Qshell interactive interface (the Qshell is option 30 of OS/400, 5769-SS1). Alternatively, you can run the application by using the CL command RUNJVA CLASS(MQIVP).
4. To use the WebSphere MQ bindings for Java on OS/400, you must ensure that the library QMQMJAVA is in your library list.

Running your own WebSphere MQ base Java programs

To run your own Java applets or applications, use the procedures described for the verification programs, substituting your application name in place of “MQIVP”.

For more information on writing WebSphere MQ base Java applications and applets, see Part 2, “Programming with WebSphere MQ base Java” on page 55

Solving WebSphere MQ base Java problems

If a program does not complete successfully, run the installation verification applet or installation verification program, and follow the advice given in the diagnostic messages. Both of these programs are described in Chapter 3, “Using WebSphere MQ classes for Java (WebSphere MQ base Java)” on page 13.

If the problems continue and you need to contact the IBM service team, you may be asked to turn on the trace facility. Refer to the following sections for the appropriate procedures for your system.

Tracing the sample application

To trace the MQIVP program, enter the following:

```
java MQIVP -trace n
```

where “n” is a number between 1 and 5, depending on the level of detail required. (The greater the number, the more information is gathered.)

For more information about how to use trace, see “Tracing WebSphere MQ base Java programs” on page 86
Error messages

Error messages
Here are some of the more common error messages that you may see:

**Unable to identify local host IP address**
The server is not connected to the network.
Connect the server to the network and retry.

**MQRC_ADAPTER_CONN_LOAD_ERROR**
If you see this error while running on z/OS, ensure that the WebSphere MQ SCSQANLE, and SCSQAUTH datasets are in your STEPLIB statement.
Chapter 4. Using WebSphere MQ classes for Java Message Service (WebSphere MQ JMS)

This chapter describes the following tasks:

- How to set up and use JMS Postcard
- How to set up your system to use the Test and sample programs
- How to run the point-to-point Installation Verification Test (IVT) program to verify your WebSphere MQ classes for Java Message Service installation
- How to run the sample publish/subscribe Installation Verification Test (PSIVT) program to verify your publish/subscribe installation
- How to run your own programs

JMS Postcard

JMS Postcard is a simple, friendly way to do the following:

- Verify that you have successfully installed WebSphere MQ and WebSphere MQ JMS on one computer and, optionally, others as well
- Introduce you to messaging

Setting up JMS Postcard

You must make sure that the Java Messaging feature of WebSphere MQ for Windows (WebSphere MQ JMS) is installed to be able to use JMS Postcard. In turn, WebSphere MQ JMS requires that you have a working Java Runtime Environment (JRE) at Java 1.3 level.

Before you can successful run the JMS Postcard application, you must have defined the environment variables CLASSPATH, LIBPATH, MQ_JAVA_INSTALL_PATH, and MQ_JAVA_DATA_PATH. On Windows systems these variables are set as part of the install process. On other platforms you must set them yourself. For more information about these variables, see “Environment variables” on page 8.

Many operations that the Postcard application carries out on your behalf require the user to be a member of the WebSphere MQ administrators group (mqm). If you are not a member of mqm, get a member of the mqm group to set up the default configuration on your behalf. See “JMS Postcard default configuration” on page 20.

Starting

To start the JMS Postcard application, run the postcard script. This is supplied in the java/bin directory of the WebSphere MQ installation.

The first time JMS Postcard is run on a computer, you are asked to complete the default configuration, which sets up a suitable queue manager to act as mailbox. See “JMS Postcard default configuration” on page 20.

Whenever you start a Postcard application, you are asked to sign on and enter a nickname. (There are advanced options available on the sign-on dialog, see “Signon advanced options” on page 18 for details).
Signon

The sign-on dialog has a check box labelled Advanced. Check this to see the extended dialog where you can choose which queue manager is used by the Postcard program.

Notes:
1. If you have no queue managers at all, or just the default configuration, then the checkbox will be disabled.
2. Depending upon what queue managers and clusters you have, the checkbox and options will be in one of various combinations of enabled, disabled, and preselected.

Signon advanced options

Use default configuration as mailbox
This is the easiest way to use JMS Postcard, on one or several computers. Make sure that the default configuration is installed on all the computers, that one of them holds the repository, and all the others use the first one as their repository; this puts them all in the same cluster.

Choose queue manager as mailbox
Use the drop-down list to choose any one of your local queue managers. If you want to send postcards between two queue managers (on one or more computers) this way, then it is up to you to make sure that one of the following conditions is true:
- The queue managers are in the same cluster (for more information about clusters, see the WebSphere MQ Queue Manager Clusters book).
- There are explicit connections between the queue managers.

Sending a postcard
To send a postcard successfully, you need to have two instances of the Postcard application with different nicknames. For example, suppose you start the Postcard application and use the nickname “Will”, and then start it again using the nickname “Tim”. Will can send postcards to Tim and Tim can send postcards to Will.

If Will and Tim are connected to the same queue manager, see “Running JMS Postcard with one queue manager”.

If Tim is on a different queue manager manager (on the same or a different computer from Will), see “Running JMS Postcard with two queue managers” on page 19.

When the postcard arrives successfully, you know your WebSphere MQ installation and your WebSphere MQ JMS are working correctly.

For an alternative means of verifying installation of WebSphere MQ JMS, run the IVTRun application from the command line. See “Running the point-to-point IVT” on page 28 for more information about this.

Running JMS Postcard with one queue manager
This assumes that you have already started the Postcard application with a nickname, for example “Will”, and you want to send a postcard to a second nickname on this computer. Follow these steps:
1. Move the first Postcard (Will) to one side of your screen, then start a second Postcard by running the postcard shell script again.
2. Enter your second nickname, for example “Tim”.

3. On Will’s Postcard fill in the To field with your second nickname, Tim. (You can leave the On field empty and Postcard will fill it in for you, or you can type in the queue manager name that you see below the Message box after On).

4. Click in the Message box, type your message in, and click the Send button.

5. Look in Tim’s Postcard to see the message arrive, and double-click on it to see the postcard itself.

6. Try using Tim to send a message back to Will. You can do this easily, by selecting the message that arrived in Tim’s list, and clicking the Reply button.

Note: See “JMS Postcard configuration” on page 20 for advice about configuration.

Running JMS Postcard with two queue managers
This assumes that you have already started JMS Postcard with a nickname, for example “Will”, and you want to send a postcard to a second nickname on a second queue manager on this, or another, computer.

1. Start the second Postcard, choosing one of the following:
   - JMS Postcard
     - on this computer:
       run the postcard shell script again, then in the sign-on dialog check Advanced and select the second queue manager you want to use.
     - on another computer:
       run the postcard shell script; or, on Windows systems, open WebSphere MQ First Steps and click on JMS Postcard.
   - MQI Postcard on Windows systems:
     either start from WebSphere MQ First Steps (to use the default configuration), or open the WebSphere MQ Explorer, right-click on the queue manager you want to use and select All Tasks->Start a Postcard...

2. When the sign-on dialog appears, enter your second nickname (for example, “Tim”).

3. In the Postcard application on Will’s computer, fill in the To field with your second nickname (Tim), and in the On field put the queue manager name of the second postcard where Tim is (if you don’t know this name, on Tim’s computer in the Postcard look below the Message box after On; alternatively if both queue managers are in the default configuration cluster you can just type in the short TCP/IP name of Tim’s computer and Postcard will build that into the queue manager name in the same way that the task which creates the default configuration does).

4. Type your message in, and click the Send button.
   Look in Tim’s Postcard to see the message arrive, and double-click on it to see the postcard itself.

5. Try sending a message from Tim’s computer back to Will.
   You can do this easily, by selecting the message that arrived in Tim’s list, and clicking the Reply button.

Note: See “JMS Postcard configuration” on page 20

See also “How JMS Postcard works” on page 20
JMS Postcard configuration

The Postcard application needs a suitable queue manager to act as mailbox. See "JMS Postcard default configuration" for the easiest way to get one. You will be prompted to install this default configuration the first time you start the Postcard application (see “Starting” on page 17).

Instead of using the default configuration, you can also start the Postcard application using any other local queue manager.

If you want to send postcards to another computer, or to other queue managers, the default configuration must include the option of being joined in the same cluster. The other queue managers must either be in the same cluster or you must create a connection explicitly between them.

See also “How JMS Postcard works”.

JMS Postcard default configuration

Installing the default configuration quickly and easily creates a special queue manager (with queues and channels), and optionally joins it to a cluster, to enable you to use the JMS Postcard application to verify your installation and see messaging working.

On WebSphere MQ for Windows, the Default Configuration Wizard will automatically open when JMS Postcard is started and the wizard has not already been run on this computer.

On platforms other than Windows systems, you can also run the DefaultConfiguration script, provided that there are no existing queue managers on this computer. On Windows systems, run Default Configuration from First Steps.

Note: You need to be a member of the WebSphere MQ administrators group (mqm) to complete default configuration successfully. If you are not a member of mqm, get a member of the mqm group to set up the default configuration on your behalf.

How JMS Postcard works

Starting up

When JMS Postcard starts, it checks to see what queue managers exist on this computer, and initializes the sign-on dialog accordingly. If there are no queue managers at all, it will prompt you to install the default configuration.

JMS Postcard uses the Java Message Service method queueConnectionFactory.createQueueConnection() to connect to the default queue manager.

Receiving messages

All the time JMS Postcard is running, it polls a queue called postcard for incoming messages from other Postcard applications. If there is no queue called postcard, JMS Postcard will attempt to create one.

When JMS Postcard starts running, it creates a Java Message Service QueueReceiver object for the local postcard queue, providing as a parameter a selector string which filters the messages to be received from the queue by the Correlation Identifier (CorrelId field). The selector string defines that the postcard
client should only receive messages where the CorrelId field matches the nickname of the user. The words from the message data are then presented in the JMS Postcard window.

**Sending messages**

If you did not enter a computer name in the On: field, then JMS Postcard assumes the recipient is on the same queue manager.

If you did enter a name, postcard checks for the existence of a queue manager with this name - first using the exact name supplied, and then using a prefix in the same format as that created by the default configuration.

In both cases, it issues a session.createQueue('postcard'), and sets the base queue manager name to the string supplied.

Finally it builds a JMS BytesMessage from your nickname and the words you typed in, and runs queueSender.send(theMessage) to put the message onto the queue.

**How the postcards get there**

When other instances of Postcard on this computer are using the same queue manager and queue, the messages are simply being put and got from the one queue. This does, however, verify that the WebSphere MQ code installed on this computer is configured and working correctly.

JMS Postcard can only send to another queue manager if a connection to that queue manager exists. This connection will exist because either both queue managers are members of the same cluster, or you have explicitly created a connection yourself. JMS Postcard can therefore assume that the queue manager will be connectable, and simply connects to it, opens the queue and puts a message, as already described, leaving all the work of getting the message there to the WebSphere MQ cluster code. In other words, JMS Postcard uses only the one piece of code for putting the message, and does not need to know whether or not the message is going to another computer.

In JMS Postcard, when session.createSender('postcard') is called, the cluster code checks the repository to find the other queue manager, and to check that the queue exists, and throws an exception if this was not possible for any reason.

When queueSender.send(theMessage) is called, the cluster code opens a channel to the other queue manager (creating it if necessary) and sends the message.

The channel may be discarded afterwards, if the cluster optimizing code does not need it. If the queue managers are on different computers, that is all handled by the cluster code.

**Tidying up undeliverable messages**

If you sent a postcard message to “John”, but never ran a Postcard application with the nickname “John”, the message would sit on the queue for ever. To prevent this, JMS Postcard sets the Message Lifetime (Expiry) field in the Message Descriptor (MQMD) to 48 hours. After that time, the message will be discarded, wherever it may be (possibly even still in transmission).

**Exchanging messages between different WebSphere MQ Postcard applications**

You can exchange messages between all the different types of Postcard application as follows:
JMS Postcard

- **MQI Postcard** on WebSphere MQ for Windows.
- **JMS Postcard** on Windows systems.
- **JMS Postcard** on other operating systems such as UNIX.
- **MQSeries Postcard** on previous versions of MQSeries for Windows, with the exception that it cannot *receive* messages from JMS Postcard.
- **MQ Everyplace Postcard** on WebSphere MQ Everyplace on pervasive devices. For this a connection must be explicitly set up between the queue managers. See the WebSphere MQ Everyplace product documentation for further information.

**Customizing JMS Postcard**

Normally JMS Postcard uses standard Java Swing settings for font size and background color. But if a `postcard.ini` file is detected on startup, JMS Postcard will use settings specified in this file instead. You can also change the trace setting.

Edit the sample file `postcard.ini` in the `bin` directory of the WebSphere MQ classes for Java installation and set your preferred settings for font size, and screen foreground and background colors.

**Note:** The precise use of upper and lower case letters in the keywords, as in the following examples, must be strictly observed when you set these properties.

**Setting screen colors**

By setting the Background and Foreground properties, you can change the background and foreground colors of controls used in the Postcard application.

```
Background=000000
Foreground=FFFFFF
```

This example selects white text on a black background. The values represent intensity levels for Red, Green, and Blue colors using a hexadecimal scale from 00 to FF. Other examples of colors are FF0000 (bright red), 00FF00 (bright green) and 0000FF (bright blue).

**Setting font size**

```
MinimumFont=20
```

This example selects a minimum font size of 20 points. Any value smaller than 13 is ignored.

**Using an external browser for online help**

```
WebBrowser=nautilus
```

This setting is only applicable on non-Windows systems. The internal browser used for displaying online help information cannot be customized. This setting allows you to identify an alternative browser.

**Tracing the Postcard application**

```
Trace=1
```

Set this to start trace output. Note that the trace output is sent to the `trc` subdirectory of the directory defined by the MQ_JAVA_DATA_PATH system environment variable. If the application cannot write to this directory, trace output is directed to the system console.
You can also use the MQJMS_TRACE_LEVEL parameter on the java
command line to start tracing. See “Tracing programs” on page 36 for more
about tracing applications.

Post installation setup

Note: Remember to check the README file installed with the WebSphere MQ
Java programs for information that may supersede this book.

To make all the necessary resources available to WebSphere MQ JMS programs,
you need to update the following system variables:

Classpath
Successful operation of JMS programs requires a number of Java packages
to be available to the JVM. You must specify these on the classpath after
you have obtained and installed the necessary packages.

Add the following .jar files to the classpath:
• com.ibm.mq.jar
• com.ibm.mqjms.jar
• connector.jar
• jms.jar
• jndi.jar1
• jta.jar
• ldap.jar1
• providerutil.jar

Environment variables
There are a number of scripts in the bin subdirectory of the WebSphere
MQ JMS installation. These are intended for use as convenient shortcuts
for a number of common actions. Many of these scripts assume that the
environment variables MQ_JAVA_INSTALL_PATH and MQ_JAVA_DATA_PATH are
defined, pointing to the directory in which WebSphere MQ JMS is installed
and a directory for log and trace output, respectively. If you do not set
these variables, you must edit the scripts in the bin directory accordingly.

On Windows NT, you can set the classpath and new environment variable by
using the Environment tab of the System Properties. On Windows 2000,
Environment is a button on the Advanced tab of System Properties. On UNIX,
these would normally be set from each user’s logon scripts. On any platform, you
can choose to use scripts to maintain different classpaths and other environment
variables for different projects.

1. For z/OS & OS/390, use ibmjndi.jar and jndi.jar from /usr/lpp/ldap/1ib instead of jndi.jar and ldap.jar. These files are
supplied with the operating system.
Setup for publish/subscribe

Additional setup for publish/subscribe mode

Before you can use the WebSphere MQ JMS implementation of JMS publish/subscribe, some additional setup is required:

- Ensure you have access to a publish/subscribe broker.
- Ensure that the broker is running.
- Create the WebSphere MQ JMS system queues.

This step is not required for direct connection across a TCP/IP socket to a WebSphere MQ Event Broker broker.

You should also be familiar with publish/subscribe concepts as discussed in Chapter 11, “Programming publish/subscribe applications” on page 203.

Ensure you have access to a publish/subscribe broker

With WebSphere MQ JMS you have the choice of three brokers:

- WebSphere MQ with SupportPac MA0C (also known as MQSeries Publish/Subscribe)
- WebSphere MQ Integrator V2
- WebSphere MQ Event Broker

Differences between these brokers are discussed in Chapter 11, “Programming publish/subscribe applications” on page 203. Please look at the documentation for each broker for installation and configuration instructions.

Ensure that the broker is running

MQSeries Publish/Subscribe

To verify that the broker is installed and is running, use the command:

dspmqbrk -m MY.QUEUE.MANAGER

where MY.QUEUE.MANAGER is the name of the queue manager on which the broker is running. If the broker is running, a message similar to the following is displayed:

WebSphere MQ message broker for queue manager MY.QUEUE.MANAGER running.

If the operating system reports that it cannot run the dspmqbrk command, ensure that the MQSeries Publish/Subscribe broker is installed properly.

If the operating system reports that the broker is not active, start it using the command:

strmqbrk -m MY.QUEUE.MANAGER

WebSphere MQ Integrator V2

To verify that the broker provided in WebSphere MQ Integrator V2 is installed and running please refer to the product documentation.

The command to start the broker in WebSphere MQ Integrator V2 is:

mqsistart MYBROKER

where MYBROKER is the name of the broker.
WebSphere MQ Event Broker

To verify that the broker provided in WebSphere MQ Event Broker is installed and running please refer to the product documentation.

The command to start the broker in WebSphere MQ Event Broker is:

wmqpsstart MYBROKER

where MYBROKER is the name of the broker.

Create the WebSphere MQ JMS system queues

This does not apply if you use a direct connection across TCP/IP to WebSphere MQ Event Broker. For a publish/subscribe implementation to work correctly, a number of system queues must be created. A script is supplied, in the bin subdirectory of the WebSphere MQ JMS installation, to assist with this task. To use the script, enter the following commands:

For iSeries & AS/400:

1. Copy the script from the integrated file system to a native file system library using a command similar to

   CPYFRMSTMF FROMSTMF('/QIBM/ProdData/mqm/java/bin/MQJMS_PSQ.mqsc')
   TOMBR('/QSYS.LIB/QGPL.LIB/QCLSRC.FILE/MQJMS_PSQ.MBR')

2. Call the script file using STRMQMMQSC:

   STRMQMMQSC SRCMBR(MQJMS_PSQ) SRCFILE(QGPL/QCLSRC)

For z/OS & OS/390:

1. Copy the script from the HFS into a PDS using a TSO command similar to

   OGET '/usr/lpp/mqm/java/bin/MQJMS_PSQ.mqsc' 'USERID.MQSC(MQJMSPSQ)'

   The PDS should be of fixed-block format with a record length of 80.

2. Either use the CSQUTIL application to execute this command script, or add the script to the CSQINP2 DD concatenation in your queue manager’s started task JCL. In either case, refer to the WebSphere MQ for z/OS System Setup Guide and the WebSphere MQ for z/OS System Administration Guide for further details.

For other platforms:

   runmqsc MY.QUEUE.MANAGER < MQJMS_PSQ.mqsc

If an error occurs, check that you typed the queue manager name correctly, and check that the queue manager is running.

For a broker running on a remote queue manager

For operation with a broker running on a remote queue manager, further setup is required.

1. Define a transmission queue on the remote queue manager with a queue name matching the local queue manager. These names must match for correct routing of messages by WebSphere MQ.

2. Define a sender channel on the remote queue manager and a receiver channel on the local queue manager. The sender channel should use the transmission queue defined in step 1.

3. Setup the local queue manager for communication with the remote broker:
Setup for publish/subscribe

4. To operate the remote broker, take the following steps:
   a. Start the remote broker queue manager.
   b. Start a listener for the remote broker queue manager (TCP/IP channels).
   c. Start the sender and receiver channels to the local queue manager.
   d. Start the broker on the remote queue manager.
   
   An example command is
   
   `strmqbrk -m MyBrokerMgr`

5. To operate the local queue manager to communicate with the remote broker, take the following steps:
   a. Start the local queue manager.
   b. Start a listener for the local queue manager.
   c. Start the sender and receiver channels to the remote broker queue manager.

Queues that require authorization for non-privileged users

Non-privileged users need authorization granted to access the queues used by JMS. For details about access control in WebSphere MQ, see the chapter about protecting WebSphere MQ objects in [WebSphere MQ System Administration Guide](#).

For JMS point-to-point mode, the access control issues are similar to those for the WebSphere MQ classes for Java:

- Queues that are used by QueueSenders require put authority.
- Queues that are used by QueueReceivers and QueueBrowsers require get, inq and browse authorities.
- The QueueSession.createTemporaryQueue method requires access to the model queue that is defined in the QueueConnectionFactory temporaryModel field (by default this will be SYSTEM.DEFAULT.MODEL.QUEUE).

For JMS publish/subscribe mode, the following system queues are used:

```
SYSTEM.JMS.ADMIN.QUEUE
SYSTEM.JMS.REPORT.QUEUE
SYSTEM.JMS.MODEL.QUEUE
SYSTEM.JMS.PS.STATUS.QUEUE
SYSTEM.JMS.ND.SUBSCRIBER.QUEUE
SYSTEM.JMS.D.SUBSCRIBER.QUEUE
SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE
SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE
SYSTEM.BROKER.CONTROL.QUEUE
```

Also, any application that publishes messages requires access to the STREAM queue that is specified in the topic connection factory being used. The default value for this is:

```
SYSTEM.BROKER.DEFAULT.STREAM
```
If you use the functionality of ConnectionConsumer, additional authorization may be needed. Queues to be read by the ConnectionConsumer must have get, inq and browse authorities. The system dead-letter queue, and any backout-requeue queue or report queue used by the ConnectionConsumer must have put and passall authorities.

Using the sample JMS applet to verify the TCP/IP client

WebSphere MQ JMS includes an installation verification applet, test.html. You can use the applet to verify the TCP/IP connected client mode of WebSphere MQ JMS except on the z/OS & OS/390 platform. On the z/OS & OS/390 platform the TCP/IP connected client mode is not supported.

The standard security settings for applets in Java 1.2 and higher require that all referenced classes are loaded from the same location as the applet you wish to run. For information on how to ensure that applets using WebSphere MQ JMS work, see Appendix F, “Using WebSphere MQ Java in applets with Java 1.2 or later” on page 419.

The applet connects to a given queue manager, exercises all the WebSphere MQ calls, and produces diagnostic messages if there are any failures. If the applet does not complete successfully, follow the advice given in the diagnostic messages and try to run the applet again.

Using the sample applet with OS/400

The OS/400 operating system does not have a native Graphical User Interface (GUI). To run the sample applet, you need to use the Remote Abstract Window Toolkit for Java (AWT), or the Class Broker for Java (CBJ), on graphics capable hardware.

Running the sample applet

First make sure that your queue manager can accept client connections. For details of this, see “Configuring your queue manager to accept client connections” on page 13.

There are different ways of running the JMS sample applet. Each has slightly different properties because of the security restrictions on applets imposed by the Java virtual machine.

Normal Java security settings cause the appletviewer or browser to ignore your system CLASSPATH, so the WebSphere MQ base Java and WebSphere MQ JMS libraries must be present in the same location as the applet class file. For further details of applets and security settings, see Appendix F, “Using WebSphere MQ Java in applets with Java 1.2 or later” on page 419.

Running from a web server (in appletviewer or in a browser):
The applet is invoked using a command line like the following:
appletviewer http://<web.server.host/jmsapplet>/test.html

or by pointing your Java 1.3 enabled browser at this web page. The string “<web.server.host/jmsapplet>” should be changed as appropriate to the URI of the web server you are using.

Running in appletviewer from the local machine:
The applet is invoked using a command line like the following:
appletviewer test.html
Verifying TCP/IP client

Remember that the WebSphere MQ base Java and WebSphere MQ JMS libraries must be present in the same local directory as the applet class file. Also, in this case, the applet may only connect to queue managers on the local machine.

Running the applet as an application:
Use a command line like the following:

```
java JMSTestApplet
```

The JMS sample applet contains a main method that allows the applet to run as a standalone Java application.

This option requires the WebSphere MQ base Java and WebSphere MQ JMS libraries to be present in the system CLASSPATH, as for your own WebSphere MQ JMS applications. It allows you to connect to any host and queue manager to which you have TCP/IP access.

Tracing the sample as an application
To trace the sample as an application, simply alter the command line parameters as shown below, in the same way as you would trace your own JMS applications:

```
java -DMQJMS_TRACE_LEVEL=on JMSTestApplet
```

More details can be found in "Tracing programs" on page 36.

Running the point-to-point IVT

This section describes the point-to-point installation verification test program (IVT) that is supplied with WebSphere MQ JMS.

The IVT attempts to verify the installation by connecting to the default queue manager on the local machine, using the WebSphere MQ JMS in bindings mode. It then sends a message to the SYSTEM.DEFAULT.LOCAL.QUEUE queue and reads it back again.

You can run the program in one of two possible modes.

With JNDI lookup of administered objects
JNDI mode forces the program to obtain its administered objects from a JNDI namespace, which is the expected operation of JMS client applications. (See "Administering JMS objects" on page 43 for a description of administered objects). This invocation method has the same prerequisites as the administration tool (see Chapter 5, "Using the WebSphere MQ JMS administration tool" on page 39).

Without JNDI lookup of administered objects
If you do not wish to use JNDI, the administered objects can be created at runtime by running the IVT in non-JNDI mode. Because a JNDI-based repository is relatively complex to set up, we recommend that the IVT is first run without JNDI.
Point-to-point verification without JNDI

A script, named IVTRun on UNIX, or IVTRun.bat on Windows NT, is provided to run the IVT. This file is installed in the bin subdirectory of the installation.

To run the test without JNDI, issue the following command:
IVTRun [-t] -nojndi [-m <qmgr>]

For client mode, to run the test without JNDI, issue the following command:
IVTRun [-t] -nojndi -client -m <qmgr> -host <hostname> [-port <port>]
[-channel <channel>]

where:

- `-t` means turn tracing on (by default, tracing is off)
- `qmgr` is the name of the queue manager to which you wish to connect
- `hostname` is the host on which the queue manager is running
- `port` is the TCP/IP port on which the queue manager's listener is running (default 1414)
- `channel` is the client connection channel (default SYSTEM.DEF.SVRCONN)

If the test completes successfully, you should see output similar to the following:

| 5648-C60, 5724-B41, 5655-F10 (c) Copyright IBM Corp. 2002. All Rights Reserved. |
| WebSphere MQ classes for Java(tm) Message Service 5.300 |
| Installation Verification Test |
| Creating a QueueConnectionFactory |
| Creating a Connection |
| Creating a Session |
| Creating a Queue |
| Creating a QueueSender |
| Creating a QueueReceiver |
| Creating a TextMessage |
| Sending the message to SYSTEM.DEFAULT.LOCAL.QUEUE |
| Reading the message back again |

Got message:
JMS Message class: jms_text
   JMSType: null
   JMSDeliveryMode: 2
   JMSPriority: 4
   JMSExpiration: 0
   JMSMessageID: ID:414d5120415343452492020202020207ce683c03300020
   JMSTimestamp: 1016124013892
   JMSCorrelationID: null
   JMSDestination: queue:///SYSTEM.DEFAULT.LOCAL.QUEUE
   JMSReplyTo: null
   JMSRedelivered: false
   JMS_IBM_PutDate:20020314
   JMSXAppID:java
   JMS_IBM_Format:MQSTR
   JMS_IBM_MsgType:8
   JMS_IBM_PutApplType:6
   JMS_IBM_PutTime:16401390
   JMSXUserID:parkiw
   JMSXDeliveryCount:1

A simple text message from the MQJMSIVT program
Reply string equals original string
Closing QueueReceiver
Closing QueueSender
Point-to-point verification with JNDI

To run the IVT with JNDI, the LDAP server must be running and must be configured to accept Java objects. If the following message occurs, it indicates that there is a connection to the LDAP server, but the server is not correctly configured:

Unable to bind to object

This message means that either the server is not storing Java objects, or the permissions on the objects or the suffix are not correct. See "Checking your LDAP server configuration" on page 403.

Also, the following administered objects must be retrievable from a JNDI namespace:

- MQQueueConnectionFactory
- MQQueue

A script, named IVTSetup on UNIX, or IVTSetup.bat on Windows NT, is provided to create these objects automatically. Enter the command:

IVTSetup

The script invokes the WebSphere MQ JMS Administration tool (see Chapter 5, "Using the WebSphere MQ JMS administration tool" on page 39) and creates the objects in a JNDI namespace.

The MQQueueConnectionFactory is bound under the name ivtQCF (for LDAP, cn=ivtQCF). All the properties are default values:

- TRANSPORT(BIND)
- PORT(1414)
- HOSTNAME(localhost)
- CHANNEL(SYSTEM.DEF.SVRCONN)
- VERSION(1)
- CCSSID(819)
- TEMPMODEL(SYSTEM.DEFAULT.MODEL.QUEUE)
- QMANAGER()

The MQQueue is bound under the name ivtQ (cn=ivtQ). The value of the QUEUE property becomes QUEUE(SYSTEM.DEFAULT.LOCAL.QUEUE). All other properties have default values:

- PERSISTENCE(APP)
- QUEUE(SYSTEM.DEFAULT.LOCAL.QUEUE)
- EXPIRY(APP)
- TARGCLIENT(JMS)
- ENCODING(NATIVE)
- VERSION(1)
- CCSSID(1208)
- PRIORITY(APP)
- QMANAGER()

Once the administered objects are created in the JNDI namespace, run the IVTRun (IVTRun.bat on Windows NT) script using the following command:

IVTRun [ -t ] -url "<providerURL>" [ -icf <initCtxFact> ]

where:

- \texttt{-t} means turn tracing on (by default, tracing is off)
providerURL is the JNDI location of the administered objects. If the default initial context factory is in use, this is an LDAP URL of the form:

ldap://hostname.company.com/contextName

If a file system service provider is used, (see initCtxFact below), the URL is of the form:

file:///directorySpec

Note: Enclose the providerURL string in quotation marks (").

initCtxFact is the classname of the initial context factory. The default is for an LDAP service provider, and has the value:

com.sun.jndi.ldap.LdapCtxFactory

If a file system service provider is used, set this parameter to:

com.sun.jndi.fscontext.RefFSContextFactory

If the test completes successfully, the output is similar to the non-JNDI output, except that the ‘create’ QueueConnectionFactory and Queue lines indicate retrieval of the object from JNDI. The following shows an example.

5648-C60, 5724-B41, 5655-F10 (c) Copyright IBM Corp. 2002. All Rights Reserved.
Websphere MQ classes for Java(tm) Message Service 5.300
Installation Verification Test

Using administered objects, please ensure that these are available

Retrieving a QueueConnectionFactory from JNDI
Creating a Connection
Creating a Session
Retrieving a Queue from JNDI
Creating a QueueSender
Creating a QueueReceiver
Creating a TextMessage
Sending the message to SYSTEM.DEFAULT.LOCAL.QUEUE
Reading the message back again

Got message:
JMS Message class: jms_text
   JMSType: null
   ...
   ...

Although not strictly necessary, it is good practice to remove objects that are created by the IVTSetup script from the JNDI namespace. A script called IVTTidy (IVTTidy.bat on Windows NT) is provided for this purpose.

IVT error recovery

If the test is not successful, the following notes may be helpful:

- For assistance with any error messages involving the classpath, check that your classpath is set correctly, as described in “Post installation setup” on page 23.
- The IVT might fail with a message “failed to create MQQueueManager”, with an additional message including the number 2059. This indicates that WebSphere MQ failed to connect to the default local queue manager on the machine on which you ran the IVT. Check that the queue manager is running, and that it is marked as the default queue manager.
- A message “failed to open MQ queue” indicates that WebSphere MQ connected to the default queue manager, but could not open the SYSTEM.DEFAULT.LOCAL.QUEUE. This may indicate that either the queue
does not exist on your default queue manager, or the queue is not enabled for
PUT and GET. Add or enable the queue for the duration of the test.

Table 6 lists the classes that are tested by IVT, and the package that they come
from:

Table 6. Classes that are tested by IVT

<table>
<thead>
<tr>
<th>Class</th>
<th>Jar file</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebSphere MQ JMS classes</td>
<td>com.ibm.mq.jms.jar</td>
</tr>
<tr>
<td>com.ibm.mq.MQMessage</td>
<td>com.ibm.mq.jar</td>
</tr>
<tr>
<td>javax.jms.Message</td>
<td>jms.jar</td>
</tr>
<tr>
<td>javax.naming.InitialContext</td>
<td>jndi.jar</td>
</tr>
<tr>
<td>javax.resource.cci.Connection</td>
<td>connector.jar</td>
</tr>
<tr>
<td>javax.transaction.xa.XAException</td>
<td>jta.jar</td>
</tr>
<tr>
<td>com/sun/jndi/toolkit/ComponentDirContext</td>
<td>providerutil.jar</td>
</tr>
<tr>
<td>com.sun.jndi.ldap.LdapCtxFactory</td>
<td>ldap.jar</td>
</tr>
</tbody>
</table>

The publish/subscribe installation verification test

The publish/subscribe installation verification test (PSIVT) program is supplied
only in compiled form. It is in the com.ibm.mq.jms package.

The test requires a broker such as the MQSeries Publish/Subscribe broker
(SupportPac MA0C) or WebSphere MQ Integrator V2 to be installed and running.

The PSIVT attempts to:
1. Create a publisher, p, publishing on the topic MQJMS/PSIVT/Information
2. Create a subscriber, s, subscribing on the topic MQJMS/PSIVT/Information
3. Use p to publish a simple text message
4. Use s to receive a message waiting on its input queue

When you run the PSIVT, the publisher publishes the message, and the subscriber
receives and displays the message. The publisher publishes to the broker’s default
stream. The subscriber is non-durable, does not perform message selection, and
accepts messages from local connections. It performs a synchronous receive,
waiting a maximum of 5 seconds for a message to arrive.

You can run the PSIVT, like the IVT, in either JNDI mode or standalone mode. JNDI
mode uses JNDI to retrieve a TopicConnectionFactory and a Topic from a JNDI
namespace. If JNDI is not used, these objects are created at runtime.

Publish/subscribe verification without JNDI

A script named PSIVTRun (PSIVTRun.bat on Windows NT) is provided to run
PSIVT. The file is in the bin subdirectory of the installation.

To run the test without JNDI, issue the following command:
PSIVTRun -nojndi [-m <qmgr>] [-bqm <broker>] [-t]

For client mode, to run the test without JNDI, issue the following command:
PSIVTRun -nojndi -client -m <qmgr> -host <hostname> [-port <port>]
[-channel <channel>] [-bqm <broker>] [-t]
Publish/subscribe IVT

where:

-nojndi means no JNDI lookup of the administered objects

qmgr is the name of the queue manager to which you wish to connect

hostname is the host on which the queue manager is running

port is the TCP/IP port on which the queue manager’s listener is running (default 1414)

channel is the client connection channel (default SYSTEM.DEF.SVRCONN)

broker is the name of the remote queue manager on which the broker is running. If this is not specified, the value used for qmgr is assumed.

-t means turn tracing on (default is off)

If the test completes successfully, output is similar to the following:

5648-C60, 5724-B41, 5655-F10 (c) Copyright IBM Corp. 2002. All Rights Reserved.

Websphere MQ classes for Java(tm) Message Service 5.300
Publish/Subscribe Installation Verification Test

Creating a Connection
Creating a TopicConnectionFactory
Creating a Session
Creating a Topic
Creating a TopicPublisher
Creating a TopicSubscriber
Creating a TextMessage
Adding text
Publishing the message to topic://MQJMS/PSIVT/Information
Waiting for a message to arrive [5 secs max]...

Got message:
JMS Message class: jms_text
  JMSType: null
  JMSDeliveryMode: 2
  JMSExpiration: 0
  JMSMessageID: ID:414d512041534341524920202020207cce883c19230020
  JMSTimestamp: 1016124933637
  JMSCorrelationID: ID:414d512041534341524920202020207cce883c09320020
  JMSDestination: topic://MQJMS/PSIVT/Information
  JMSReplyTo: null
  JMSRedelivered: false
  JMS_IBM_PutDate:20020314
  JMSXAppID:ASCARI
  JMS_IBM_Format:MQSTR
  JMS_IBM_MsgType:8
  JMSXUserID:parkiw
  JMS_IBM_PutApplType:26
  JMS_IBM_PutTime:16553367
  JMSXDeliveryCount:1
A simple text message from the MQJMS/PSIVT program
Reply string equals original string
Closing TopicSubscriber
Closing TopicPublisher
Closing Session
Closing Connection
PSIVT finished
Publish/subscribe IVT

Publish/subscribe verification with JNDI

To run the PSIVT in JNDI mode, two administered objects must be retrievable from a JNDI namespace:

- A TopicConnectionFactory bound under the name ivtTCF
- A Topic bound under the name ivtT

You can define these objects by using the WebSphere MQ JMS Administration Tool (see Chapter 5, “Using the WebSphere MQ JMS administration tool” on page 39) and using the following commands:

DEFINE TCF(ivtTCF)

This command defines the TopicConnectionFactory.

DEFINE T(ivtT) TOPIC(MQJMS/PSIVT/Information)

This command defines the Topic.

These definitions assume that a default queue manager, on which the broker is running, is available. For details on configuring these objects to use a non-default queue manager, see “Administering JMS objects” on page 43. These objects should reside in a context pointed to by the -url command-line parameter described below.

To run the test in JNDI mode, enter the following command:

PSIVTRun [ -t ] -url "<providerURL>" [ -icf <initCtxFact> ]

where:

- **-t** means turn tracing on (by default, tracing is off)
- **providerURL** is the JNDI location of the administered objects. If the default initial context factory is in use, this is an LDAP URL of the form:

  ldap://hostname.company.com/contextName

  If a file system service provider is used, (see initCtxFact below), the URL is of the form:

  file://directorySpec

  **Note:** Enclose the **providerURL** string in quotation marks (".

- **initCtxFact** is the classname of the initial context factory. The default is for an LDAP service provider, and has the value:

  com.sun.jndi.ldap.LdapCtxFactory

  If a file system service provider is used, set this parameter to:

  com.sun.jndi.fscontext.ReffSCtxFactory

If the test completes successfully, output is similar to the non-JNDI output, except that the “create” QueueConnectionFactory and Queue lines indicate retrieval of the object from JNDI.
PSIVT error recovery

If the test is not successful, the following notes may be helpful:

- If you see the following message:
  *** No broker response. Please ensure broker is running. ***

  this indicates that the broker is installed on the target queue manager, but its
  control queue contains some outstanding messages. For instructions on how to
  start it, see "Additional setup for publish/subscribe mode" on page 24

- If the following message is displayed:
  Unable to connect to queue manager: <default>

  ensure that your WebSphere MQ system has configured a default queue
  manager.

- If the following message is displayed:
  Unable to connect to queue manager: ...

  ensure that the administered TopicConnectionFactory that the PSIVT uses is
  configured with a valid queue manager name. Alternatively, if you used the
  -nojndi option, ensure that you supplied a valid queue manager (use the -m
  option).

- If the following message is displayed:
  Unable to access broker control queue on queue manager: ...
  Please ensure the broker is installed on this queue manager

  ensure that the administered TopicConnectionFactory that the PSIVT uses is
  configured with the name of the queue manager on which the broker is
  installed. If you used the -nojndi option, ensure that you supplied a queue
  manager name (use the -m option).

Running your own WebSphere MQ JMS programs

For information on writing your own WebSphere MQ JMS programs, see
Chapter 10, "Writing WebSphere MQ JMS programs" on page 189

WebSphere MQ JMS includes a utility file, runjms (runjms.bat on Windows NT), to
help you to run the supplied programs and programs that you have written.

The utility provides default locations for the trace and log files, and enables you to
decide any application runtime parameters that your application needs. The supplied
script assumes that the environment variable MQ_JAVA_INSTALL_PATH is set to
the directory in which the WebSphere MQ JMS is installed. The script also assumes
that the subdirectories trace and log within the directory pointed to by
MQ_JAVA_DATA_PATH are used for trace and log output, respectively.

Use the following command to run your application:
runjms <classname of application> [application-specific arguments]

For information on writing WebSphere MQ JMS applications and applets, see
Part 3, "Programming with WebSphere MQ JMS" on page 187
Solving problems

If a program does not complete successfully, run the installation verification program, which is described in “Running the point-to-point IVT” on page 28, and follow the advice given in the diagnostic messages.

Tracing programs

The WebSphere MQ JMS trace facility is provided to help IBM staff to diagnose customer problems.

Trace is disabled by default, because the output rapidly becomes large, and is unlikely to be of use in normal circumstances.

If you are asked to provide trace output, you can enable it by setting the Java property MQJMS TRACE_LEVEL to one of the following values:

- **on** traces WebSphere MQ JMS calls only
- **base** traces both WebSphere MQ JMS calls and the underlying WebSphere MQ base Java calls

For example:

```java
java -DMQJMS_TRACE_LEVEL=base MyJMSProg
```

To disable trace, set MQJMS TRACE_LEVEL to **off**.

By default, trace is output to a file named `mqjms.trc` in the current working directory. You can redirect it to a different directory by using the Java property MQJMS TRACE_DIR.

For example:

```java
java -DMQJMS_TRACE_LEVEL=base -DMQJMS_TRACE_DIR=/somepath/tracedir MyJMSProg
```

The `runjms` utility script sets these properties by using the environment variables MQJMS TRACE_LEVEL and MQ JAVA DATA PATH, as follows:

```bash
java -DMQJMS_TRACE_LEVEL=$MQJMS_TRACE_LEVEL -DMQJMS_TRACE_DIR=$MQ_JAVA_DATA_PATH\trace
```

This is only a suggestion, and you can change it as required.

Logging

The WebSphere MQ JMS log facility is provided to report serious problems, particularly those that may indicate configuration errors rather than programming errors. By default, log output is sent to the `System.err` stream, which usually appears on the `stderr` of the console in which the JVM is run.

You can redirect the output to a file by using a Java property that specifies the new location, for example:

```java
java -DMQJMS_LOG_DIR=/mydir/forlogs MyJMSProg
```

The utility script `runjms`, in the bin directory of the WebSphere MQ JMS installation, sets this property to:

```bash
<MQ_JAVA_DATA_PATH>/log
```
where MQ_JAVA_DATA_PATH is set, on Windows systems, to the path to your WebSphere MQ Java installation. On other platforms you need to set this environment variable.

When the log is redirected to a file, it is output in a binary form. To view the log, the utility formatLog (formatLog.bat on Windows NT) is provided, which converts the file to plain text format. The utility is stored in the bin directory of your WebSphere MQ JMS installation. Run the conversion as follows:

formatLog <inputfile> <outputfile>
Logging
Chapter 5. Using the WebSphere MQ JMS administration tool

The administration tool enables administrators to define the properties of eight types of WebSphere MQ JMS object and to store them within a JNDI namespace. Then, JMS clients can retrieve these administered objects from the namespace by using JNDI and use them.

The JMS objects that you can administer by using the tool are:
- MQQueueConnectionFactory
- MQTopicConnectionFactory
- MQQueue
- MQTopic
- MQXAQueueConnectionFactory
- MQXATopicConnectionFactory
- JMSWrapXAQueueConnectionFactory
- JMSWrapXATopicConnectionFactory

For details about these objects, refer to “Administering JMS objects” on page 43.

Note: JMSWrapXAQueueConnectionFactory and JMSWrapXATopicConnectionFactory are classes that are specific to WebSphere Application Server. They are contained in the package com.ibm.ejs.jms.mq.

The tool also allows administrators to manipulate directory namespace subcontexts within the JNDI. See “Manipulating subcontexts” on page 43.

Invoking the Administration tool

The administration tool has a command line interface. You can use this interactively, or use it to start a batch process. The interactive mode provides a command prompt where you can enter administration commands. In the batch mode, the command to start the tool includes the name of a file which contains an administration command script.

To start the tool in interactive mode, enter the command:
JMSAdmin [-t] [-v] [-cfg config_filename]

where:
-t Enables trace (default is trace off)
-v Produces verbose output (default is terse output)
-cfg config_filename The name of an alternative configuration file (see “Configuration” on page 40)

A command prompt is displayed, which indicates that the tool is ready to accept administration commands. This prompt initially appears as:
InitCtx>

indicating that the current context (that is, the JNDI context to which all naming and directory operations currently refer) is the initial context defined in the PROVIDER_URL configuration parameter (see “Configuration” on page 40).
As you traverse the directory namespace, the prompt changes to reflect this, so that the prompt always displays the current context.

To start the tool in batch mode, enter the command:

```
JMSAdmin <test.scp
```

where `test.scp` is a script file that contains administration commands (see “Administration commands” on page 42). The last command in the file must be the `END` command.

---

## Configuration

You must configure the administration tool with values for the following three properties:

**INITIAL_CONTEXT_FACTORY**
This indicates the service provider that the tool uses. There are three explicitly supported values for this property:

- `com.sun.jndi.ldap.LdapCtxFactory` (for LDAP)
- `com.sun.jndi.fscontext.RefFSContextFactory` (for file system context)
- `com.ibm.websphere.naming.WsnInitialContextFactory` (to work with WebSphere Application Server’s CosNaming repository)

On z/OS & OS/390, `com.ibm.jndi.LDAPCtxFactory` is also supported and provides access to an LDAP server. However, note that it is incompatible with `com.sun.jndi.ldap.LdapCtxFactory`, in that objects created using one `InitialContextFactory` cannot be read or modified using the other.

You can also use an `InitialContextFactory` that is not in the list above. See “Using an unlisted `InitialContextFactory`” on page 41 for more details.

**PROVIDER_URL**
This indicates the URL of the session’s initial context, the root of all JNDI operations carried out by the tool. Three forms of this property are currently supported:

- `ldap://hostname/contextname` (for LDAP)
- `file:[drive:]/pathname` (for file system context)
- `iiop://hostname[:port] [/?TargetContext=ctx]` (to access “base” WebSphere Application Server CosNaming namespace)

**SECURITY_AUTHENTICATION**
This indicates whether JNDI passes over security credentials to your service provider. This property is used only when an LDAP service provider is used. This property can currently take one of three values:

- `none` (anonymous authentication)
- `simple` (simple authentication)
- `CRAM-MD5` (CRAM-MD5 authentication mechanism)

If a valid value is not supplied, the property defaults to `none`. See “Security” on page 41 for more details about security with the administration tool.

These properties are set in a configuration file. When you invoke the tool, you can specify this configuration by using the `-cfg` command-line parameter, as described in “Invoking the Administration tool” on page 39. If you do not specify a configuration file name, the tool attempts to load the default configuration file (`JMSAdmin.config`). It looks for this file first in the current directory, and then in the
<MQ_JAVA_INSTALL_PATH>/bin directory, where "<MQ_JAVA_INSTALL_PATH>" is the path to your WebSphere MQ JMS installation.

The configuration file is a plain-text file that consists of a set of key-value pairs, separated by an "=". This is shown in the following example:

```
# Set the service provider
INITIAL_CONTEXT_FACTORY=com.sun.jndi.ldap.LdapCtxFactory
# Set the initial context
PROVIDER_URL=ldap://polaris/o=ibm_us,c=us
# Set the authentication type
SECURITY_AUTHENTICATION=none
```

(A "#" in the first column of the line indicates a comment, or a line that is not used.)

The installation comes with a sample configuration file that is called JMSAdmin.config, and is found in the <MQ_JAVA_INSTALL_PATH>/bin directory. Edit this file to suit the setup of your system.

**Using an unlisted InitialContextFactory**

You can use the administration tool to connect to JNDI contexts other than those listed in "Configuration" on page 40 by using three parameters defined in the JMSAdmin configuration file.

To use a different InitialContextFactory:

1. Set the INITIAL_CONTEXT_FACTORY property to the required class name.
2. Define the behavior of the InitialContextFactory using the USE_INITIAL_DIR_CONTEXT, NAME_PREFIX and NAME_READABILITY_MARKER properties.

The settings for these properties are described in the sample configuration file comments.

You do not need to define the three properties listed here, if you use one of the supported INITIAL_CONTEXT_FACTORY values. However, you can give them values to override the system defaults. If one or more of the three InitialContextFactory properties is omitted, the administration tool provides suitable defaults based on the values of the other properties.

**Security**

Administrators need to know about the effect of the SECURITY_AUTHENTICATION property described in "Configuration" on page 40:

- If this parameter is set to none, JNDI does not pass any security credentials to the service provider, and “anonymous authentication” is performed.
- If the parameter is set to either simple or CRAM-MD5, security credentials are passed through JNDI to the underlying service provider. These security credentials are in the form of a user distinguished name (User DN) and password.

If security credentials are required, then the user will be prompted for these when the tool initializes. This can be avoided by setting the PROVIDER_USERDN and PROVIDER_PASSWORD properties in the JMSAdmin configuration file.

**Note:** If you do not use these properties, the text typed, including the password, is echoed to the screen. This may have security implications.
Configuration

The tool does no authentication itself; the task is delegated to the LDAP server. It is the responsibility of the LDAP server administrator to set up and maintain access privileges to different parts of the directory. If authentication fails, the tool displays an appropriate error message and terminates.

More detailed information about security and JNDI is in the documentation at Sun’s Java web site (http://java.sun.com).

Configuring for WebSphere Application Server V3.5

For the administration tool (or any client application that needs to do subsequent lookups) to work with WebSphere Application Server’s CosNaming repository, you require the following configuration:

- CLASSPATH must include WebSphere Application Server’s JNDI-related jar files:
  - For WebSphere Application Server V3.5:
    `<WSAppserver>\lib\ujc.jar`
- PATH for WebSphere Application Server V3.5 must include:
  `<WSAppserver>\jdk\jre\bin`

where `<WSAppserver>` is the install path for WebSphere Application Server.

Administration commands

When the command prompt is displayed, the tool is ready to accept commands. Administration commands are generally of the following form:

verb [param]*

where verb is one of the administration verbs listed in Table 7. All valid commands consist of at least one (and only one) verb, which appears at the beginning of the command in either its standard or short form.

The parameters a verb may take depend on the verb. For example, the END verb cannot take any parameters, but the DEFINE verb may take any number of parameters from 1 to 20 or more. Details of the verbs that take at least one parameter are discussed in later sections of this chapter.

**Table 7. Administration verbs**

<table>
<thead>
<tr>
<th>Verb</th>
<th>Short form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER</td>
<td>ALT</td>
<td>Change at least one of the properties of a given administered object</td>
</tr>
<tr>
<td>DEFINE</td>
<td>DEF</td>
<td>Create and store an administered object, or create a new subcontext</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>DIS</td>
<td>Display the properties of one or more stored administered objects, or the contents of the current context</td>
</tr>
<tr>
<td>DELETE</td>
<td>DEL</td>
<td>Remove one or more administered objects from the namespace, or remove an empty subcontext</td>
</tr>
<tr>
<td>CHANGE</td>
<td>CHG</td>
<td>Alter the current context, allowing the user to traverse the directory namespace anywhere below the initial context (pending security clearance)</td>
</tr>
<tr>
<td>COPY</td>
<td>CP</td>
<td>Make a copy of a stored administered object, storing it under an alternative name</td>
</tr>
<tr>
<td>MOVE</td>
<td>MV</td>
<td>Alter the name under which an administered object is stored</td>
</tr>
</tbody>
</table>
Verb names are not case-sensitive.

Usually, to terminate commands, you press the carriage return key. However, you can override this by typing the “+” symbol directly before the carriage return. This enables you to enter multi-line commands, as shown in the following example:

```
DEFINE Q(BookingsInputQueue) +
  QMGR(QM.POLARIS.TEST) +
  QUEUE(BOOKINGS.INPUT.QUEUE) +
  PORT(1415) +
  CCSID(437)
```

Lines beginning with one of the characters *, #, or / are treated as comments, or lines that are ignored.

### Manipulating subcontexts

You can use the verbs CHANGE, DEFINE, DISPLAY and DELETE to manipulate directory namespace subcontexts. Their use is described in Table 8.

#### Table 8. Syntax and description of commands used to manipulate subcontexts

<table>
<thead>
<tr>
<th>Command syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINE CTX(ctxName)</td>
<td>Attempts to create a new child subcontext of the current context, having the name ctxName. Fails if there is a security violation, if the subcontext already exists, or if the name supplied is not valid.</td>
</tr>
<tr>
<td>DISPLAY CTX</td>
<td>Displays the contents of the current context. Administered objects are annotated with ‘a’, subcontexts with ‘[D]’. The Java type of each object is also displayed.</td>
</tr>
<tr>
<td>DELETE CTX(ctxName)</td>
<td>Attempts to delete the current context’s child context having the name ctxName. Fails if the context is not found, is non-empty, or if there is a security violation.</td>
</tr>
<tr>
<td>CHANGE CTX(ctxName)</td>
<td>Alters the current context, so that it now refers to the child context having the name ctxName. One of two special values of ctxName may be supplied: =UP which moves to the current context’s parent =INIT which moves directly to the initial context Fails if the specified context does not exist, or if there is a security violation.</td>
</tr>
</tbody>
</table>

### Administering JMS objects

This section describes the eight types of object that the administration tool can handle. It includes details about each of their configurable properties and the verbs that can manipulate them.

#### Object types

Table 9 on page 44 shows the eight types of administered objects. The Keyword column shows the strings that you can substitute for TYPE in the commands shown.
## Administering JMS objects

Table 9. The JMS object types that are handled by the administration tool

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQQueueConnectionFactory</td>
<td>QCF</td>
<td>The WebSphere MQ implementation of the JMS QueueConnectionFactory interface. This represents a factory object for creating connections in the point-to-point domain of JMS.</td>
</tr>
<tr>
<td>MQTopicConnectionFactory</td>
<td>TCF</td>
<td>The WebSphere MQ implementation of the JMS TopicConnectionFactory interface. This represents a factory object for creating connections in the publish/subscribe domain of JMS.</td>
</tr>
<tr>
<td>MQQueue</td>
<td>Q</td>
<td>The WebSphere MQ implementation of the JMS Queue interface. This represents a destination for messages in the point-to-point domain of JMS.</td>
</tr>
<tr>
<td>MQTopic</td>
<td>T</td>
<td>The WebSphere MQ implementation of the JMS Topic interface. This represents a destination for messages in the publish/subscribe domain of JMS.</td>
</tr>
<tr>
<td>MQXAQueueConnectionFactory¹</td>
<td>XAQCFC</td>
<td>The WebSphere MQ implementation of the JMS XAQueueConnectionFactory interface. This represents a factory object for creating connections in the point-to-point domain of JMS that use the XA versions of JMS classes.</td>
</tr>
<tr>
<td>MQXATopicConnectionFactory¹</td>
<td>XATCF</td>
<td>The WebSphere MQ implementation of the JMS XATopicConnectionFactory interface. This represents a factory object for creating connections in the publish/subscribe domain of JMS that use the XA versions of JMS classes.</td>
</tr>
<tr>
<td>JMSWrapXAQueueConnectionFactory²</td>
<td>WSQCF</td>
<td>The WebSphere MQ implementation of the JMS QueueConnectionFactory interface. This represents a factory object for creating connections in the point-to-point domain of JMS that use the XA versions of JMS classes with WebSphere Application Server.</td>
</tr>
<tr>
<td>JMSWrapXATopicConnectionFactory²</td>
<td>WSTCF</td>
<td>The WebSphere MQ implementation of the JMS TopicConnectionFactory interface. This represents a factory object for creating connections in the publish/subscribe domain of JMS that use the XA versions of JMS classes with WebSphere Application Server.</td>
</tr>
</tbody>
</table>

1. These classes are provided for use by vendors of application servers. They are unlikely to be directly useful to application programmers.
2. Use this style of ConnectionFactory if you wish your JMS sessions to participate in global transactions that are coordinated by WebSphere Application Server.
### Verbs used with JMS objects

You can use the verbs ALTER, DEFINE, DISPLAY, DELETE, COPY, and MOVE to manipulate administered objects in the directory namespace. Table 10 summarizes their use. Substitute TYPE with the keyword that represents the required administered object, as listed in Table 9 on page 44.

**Table 10. Syntax and description of commands used to manipulate administered objects**

<table>
<thead>
<tr>
<th>Command syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER TYPE(name) [property]*</td>
<td>Attempts to update the given administered object’s properties with the ones supplied. Fails if there is a security violation, if the specified object cannot be found, or if the new properties supplied are not valid.</td>
</tr>
<tr>
<td>DEFINE TYPE(name) [property]*</td>
<td>Attempts to create an administered object of type TYPE with the supplied properties, and tries to store it under the name name in the current context. Fails if there is a security violation, if the supplied name is not valid or already exists, or if the properties supplied are not valid.</td>
</tr>
<tr>
<td>DISPLAY TYPE(name)</td>
<td>Displays the properties of the administered object of type TYPE, bound under the name name in the current context. Fails if the object does not exist, or if there is a security violation.</td>
</tr>
<tr>
<td>DELETE TYPE(name)</td>
<td>Attempts to remove the administered object of type TYPE, having the name name, from the current context. Fails if the object does not exist, or if there is a security violation.</td>
</tr>
<tr>
<td>COPY TYPE(nameA) TYPE(nameB)</td>
<td>Makes a copy of the administered object of type TYPE, having the name nameA, naming the copy nameB. This all occurs within the scope of the current context. Fails if the object to be copied does not exist, if an object of name nameB already exists, or if there is a security violation.</td>
</tr>
<tr>
<td>MOVE TYPE(nameA) TYPE(nameB)</td>
<td>Moves (renames) the administered object of type TYPE, having the name nameA, to nameB. This all occurs within the scope of the current context. Fails if the object to be moved does not exist, if an object of name nameB already exists, or if there is a security violation.</td>
</tr>
</tbody>
</table>

### Creating objects

Objects are created and stored in a JNDI namespace using the following command syntax:

```
DEFINE TYPE(name) [property]*
```

That is, the DEFINE verb, followed by a TYPE(name) administered object reference, followed by zero or more properties (see “Properties” on page 46).

### LDAP naming considerations

To store your objects in an LDAP environment, their names must comply with certain conventions. One of these is that object and subcontext names must include a prefix, such as cn= (common name), or ou= (organizational unit).
The administration tool simplifies the use of LDAP service providers by allowing you to refer to object and context names without a prefix. If you do not supply a prefix, the tool automatically adds a default prefix to the name you supply. For LDAP this is **cn=**.

You can change the default prefix by setting the `NAME_PREFIX` property in the JMSAdmin configuration file, as described in “Using an unlisted `InitialContextFactory`” on page 41.

This is shown in the following example.

```
InitCtx> DEFINE Q(testQueue)

InitCtx> DISPLAY CTX
```

```
Contents of InitCtx

a  cn=testQueue  com.ibm.mq.jms.MQQueue

1 Object(s)
  0 Context(s)
  1 Binding(s), 1 Administered
```

Note that although the object name supplied (`testQueue`) does not have a prefix, the tool automatically adds one to ensure compliance with the LDAP naming convention. Likewise, submitting the command `DISPLAY Q(testQueue)` also causes this prefix to be added.

You may need to configure your LDAP server to store Java objects. Information to assist with this configuration is provided in Appendix C, “LDAP schema definition for storing Java objects” on page 403.

### Properties

A property consists of a name-value pair in the format:

```
PROPERTY_NAME(property_value)
```

Property names are not case-sensitive, and are restricted to the set of recognized names shown in Table 11. This table also shows the valid property values for each property.

#### Table 11. Property names and valid values

<table>
<thead>
<tr>
<th>Property</th>
<th>Short form</th>
<th>Valid values (defaults in bold)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BROKERCCDSUBQ</td>
<td>CCDSUB</td>
<td>• <code>SYSTEM.JMS.D.CC.SUBSCRIPTION.QUEUE</code>&lt;br&gt;• Any string</td>
</tr>
<tr>
<td>BROKERCCSUBQ</td>
<td>CCSUB</td>
<td>• <code>SYSTEM.JMS.ND.CC.SUBSCRIPTION.QUEUE</code>&lt;br&gt;• Any string</td>
</tr>
<tr>
<td>BROKERCONQ</td>
<td>BCON</td>
<td>Any string</td>
</tr>
<tr>
<td>BROKERDURSUBQ</td>
<td>BDSUB</td>
<td>• <code>SYSTEM.JMS.D.SUBSCRIPTION.QUEUE</code>&lt;br&gt;• Any string</td>
</tr>
<tr>
<td>BROKERPUBQ</td>
<td>BPUB</td>
<td>• <code>SYSTEM.BROKER.DEFAULT.STREAM</code>&lt;br&gt;• Any string</td>
</tr>
<tr>
<td>BROKERQMG</td>
<td>BQM</td>
<td>Any string</td>
</tr>
<tr>
<td>BROKERSUBQ</td>
<td>BSUB</td>
<td>• <code>SYSTEM.JMS.ND.SUBSCRIPTION.QUEUE</code>&lt;br&gt;• Any string</td>
</tr>
<tr>
<td>Property</td>
<td>Short form</td>
<td>Valid values (defaults in bold)</td>
</tr>
<tr>
<td>---------------</td>
<td>------------</td>
<td>---------------------------------</td>
</tr>
</tbody>
</table>
| BROKERVER     | BVER       | • **V1** - To use the WebSphere MQ broker. Also for use of the WebSphere MQ Integrator V2 or WebSphere MQ Event Broker brokers in compatibility mode.  
• **V2** - To use WebSphere MQ Integrator V2 or WebSphere MQ Event Broker brokers in native mode |
| CCSID         | CCS        | Any positive integer            |
| CHANNEL       | CHAN       | Any string                      |
| CLEANUP       | CL         | • **SAFE**  
• **ASPROP**  
• **NONE**  
• **STRONG** |
| CLEANUPINT    | CLINT      | • **60000**  
• Any positive integer |
| CLIENTID      | CID        | Any string                      |
| DESCRIPTION   | DESC       | Any string                      |
| ENCODING      | ENC        | See [The ENCODING property on page 52](#) |
| EXPIRY        | EXP        | • **APP** - Expiry may be defined by the JMS application.  
• **UNLIM** - No expiry occurs.  
• Any positive integer representing expiry in milliseconds. |
| HOSTNAME      | HOST       | • **localhost**  
• Any string |
| MSGBATCHSZ    | MBS        | • **10**  
• Any positive integer |
| MSGRETENTION  | MRET       | • **Yes** - Unwanted messages remain on the input queue  
• **No** - Unwanted messages are dealt with according to their disposition options |
| PERSISTENCE   | PER        | • **APP** - Persistence may be defined by the JMS application.  
• **QDEF** - Persistence takes the value of the queue default.  
• **PERS** - Messages are persistent.  
• **NON** - messages are non-persistent. |
| POLLINGINT    | PINT       | • **5000**  
• Any positive integer |
| PORT          |            | • **1414** (for TRANSPORT set to BND or CLIENT);  
**1506** (for TRANSPORT set to DIRECT)  
• Any positive integer |
| PRIORITY      | PRI        | • **APP** - Priority may be defined by the JMS application.  
• **QDEF** - Priority takes the value of the queue default.  
• Any integer in the range 0-9. |
| PUBACKINT     | PAI        | • **25**  
• Any positive integer |
Many of the properties are relevant only to a specific subset of the object types. Table 12 on page 49 shows for each property which object types are valid, and gives a brief description of each property. The object types are identified using keywords; refer to Table 9 on page 44 for an explanation of these.

Numbers refer to notes at the end of the table. See also “Property dependencies” on page 51, Appendix A, “Mapping between Administration tool properties and programmable properties” on page 399, shows the relationship between properties set by the tool and programmable properties.
Table 12. The valid combinations of property and object type

<table>
<thead>
<tr>
<th>Property</th>
<th>QCF</th>
<th>TCF</th>
<th>Q</th>
<th>T</th>
<th>WSQCF</th>
<th>WSTCF</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BROKERCCDSUBQ</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>The name of the queue from which durable subscription messages are retrieved for a ConnectionConsumer</td>
</tr>
<tr>
<td>BROKERCCSUBQ</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td>The name of the queue from which non-durable subscription messages are retrieved for a ConnectionConsumer</td>
</tr>
<tr>
<td>BROKERCONQ</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Broker's control queue name</td>
</tr>
<tr>
<td>BROKERDURSUBQ</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td>The name of the queue from which durable subscription messages are retrieved</td>
</tr>
<tr>
<td>BROKERPUBQ</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>The name of the broker input queue (stream queue)</td>
</tr>
<tr>
<td>BROKERQMGMR</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>The queue manager on which the broker is running</td>
</tr>
<tr>
<td>BROKERSUBQ</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>The name of the queue from which non-durable subscription messages are retrieved</td>
</tr>
<tr>
<td>BROKERVER</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>The version of the broker being used</td>
</tr>
<tr>
<td>CCSID</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>The coded-character-set-ID to be used on connections</td>
</tr>
<tr>
<td>CHANNEL</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The name of the client connection channel being used</td>
</tr>
<tr>
<td>CLEANUP</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>Cleanup Level for BROKER or MIGRATE Subscription Stores</td>
</tr>
<tr>
<td>CLEANUPINT</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>The interval between background executions of the publish/subscribe cleanup utility</td>
</tr>
<tr>
<td>CLIENTID</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td>A string identifier for the client</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>A description of the stored object</td>
</tr>
<tr>
<td>ENCODING</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>The encoding scheme used for this destination</td>
</tr>
<tr>
<td>EXPIRY</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>The period after which messages at a destination expire</td>
</tr>
<tr>
<td>HOSTNAME</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td>The name of the host on which the queue manager or WebSphere MQ Event Broker broker resides. A dotted-decimal TCP/IP address can also be used.</td>
</tr>
<tr>
<td>MSGBATCHSZ</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>The maximum number of messages to be taken from a queue in one packet when using asynchronous message delivery</td>
</tr>
<tr>
<td>MSGRETENTION</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>Whether or not the connection consumer keeps unwanted messages on the input queue</td>
</tr>
<tr>
<td>PERSISTENCE</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>The persistence of messages sent to a destination</td>
</tr>
</tbody>
</table>

Chapter 5. Using the WebSphere MQ JMS administration tool  49
**Table 12. The valid combinations of property and object type (continued)**

<table>
<thead>
<tr>
<th>Property</th>
<th>QCF</th>
<th>TCF</th>
<th>Q</th>
<th>T</th>
<th>WSQCF</th>
<th>XAQC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIORITY</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The priority for messages sent to a destination</td>
</tr>
<tr>
<td>POLLINGINT</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>The interval, in milliseconds, between scans of all receivers during asynchronous message delivery</td>
</tr>
<tr>
<td>PORT</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The port on which the queue manager or broker listens</td>
</tr>
<tr>
<td>PUBBACKINT</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The interval, in number of messages, between publish requests that require acknowledgement from the broker</td>
</tr>
<tr>
<td>QMANAGER</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>The name of the queue manager to connect to</td>
</tr>
<tr>
<td>QUEUE</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The underlying name of the queue representing this destination</td>
</tr>
<tr>
<td>RECEEXIT</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fully-qualified class name of the receive exit being used</td>
</tr>
<tr>
<td>RECEEXITINIT</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Receive exit initialization string</td>
</tr>
<tr>
<td>SECEXIT</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fully-qualified class name of the security exit being used</td>
</tr>
<tr>
<td>SECEXEXITINIT</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Security exit initialization string</td>
</tr>
<tr>
<td>SENDEXIT</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fully-qualified class name of the send exit being used</td>
</tr>
<tr>
<td>SENDEXITINIT</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Send exit initialization string</td>
</tr>
<tr>
<td>SSLCIPHERSUITE</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The cipher suite to use for SSL connection</td>
</tr>
<tr>
<td>SSLCRL</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CRL servers to check for SSL certificate revocation</td>
</tr>
<tr>
<td>SSLPEERNAME</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For SSL, a distinguished name skeleton which must match that provided by the queue manager</td>
</tr>
<tr>
<td>STATREFRESHINT</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The interval, in milliseconds, between transactions to refresh publish/subscribe status</td>
</tr>
<tr>
<td>SUBSTORE</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Where WebSphere MQ JMS should store persistent data relating to active subscriptions</td>
</tr>
<tr>
<td>SYNCPOINTALLGETS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td>Whether or not all gets should be performed under syncpoint</td>
</tr>
<tr>
<td>TARGCLIENT2</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Field indicates whether the WebSphere MQ RFH2 format is used to exchange information with target applications</td>
</tr>
<tr>
<td>TEMPMODEL</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Name of the model queue from which temporary queues are created</td>
</tr>
<tr>
<td>TOPIC</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The underlying name of the topic representing this destination</td>
</tr>
</tbody>
</table>
### Property dependencies

Some properties have dependencies on each other. This may mean that it is meaningless to supply a property unless another property is set to a particular value. The specific property groups where this can occur are

- Client properties
- Properties for connecting to WebSphere MQ Event Broker
- Exit initialization strings

#### Client properties

Some properties are only relevant to a connection with the TRANSPORENT property set to the value CLIENT. If this property is not explicitly set on a connection factory to one of the values CLIENT or DIRECT, the transport used on connections provided by the factory is WebSphere MQ Bindings. Consequently, none of the client properties on this connection factory can be configured. These are:

- HOST
- PORT
- CHANNEL
- CCSID
- RECEXIT
- RECEXITINIT
- SECEXIT
- SECEXITINIT
If you attempt to set any of these properties without setting the TRANSPORT property to CLIENT (or, for some, DIRECT; see “Properties for connecting to WebSphere MQ Event Broker”), there will be an error.

Properties for connecting to WebSphere MQ Event Broker

The only properties used with a direct connection to WebSphere MQ Event Broker are listed in note 4 associated with Table 12 on page 49.

The default values for PORT and BROKERVER are set by the definition of TRANSPORT:

1. Defining a connection factory with TRANSPORT as CLIENT sets:
   - BROKERVER to V1
   - PORT to 1414

2. Defining a connection factory with TRANSPORT as DIRECT sets:
   - BROKERVER to V2
   - PORT to 1506

If you explicitly set the value of PORT or BROKERVER, a later change to the value of TRANSPORT will not override your choices.

Exit initialization strings

You must not set any of the exit initialization strings unless the corresponding exit name has been supplied. The exit initialization properties are:

- RECEXITINIT
- SECEXITINIT
- SENDEXITINIT

For example, specifying RECEXITINIT(myString) without specifying RECEXIT(some.exit.classname) causes an error.

The ENCODING property

The valid values that the ENCODING property can take are more complex than the rest of the properties. The encoding property is constructed from three sub-properties:

integer encoding this is either normal or reversed

decimal encoding this is either normal or reversed

floating-point encoding this is IEEE normal, IEEE reversed, or z/OS.

The ENCODING is expressed as a three-character string with the following syntax:

{N|R}{N|R}{N|R|3}

In this string:
- N denotes normal
- R denotes reversed
- 3 denotes z/OS
- the first character represents integer encoding
the second character represents *decimal encoding*
the third character represents *floating-point encoding*

This provides a set of twelve possible values for the `ENCODING` property.

There is an additional value, the string `NATIVE`, which sets appropriate encoding values for the Java platform.

The following examples show valid combinations for `ENCODING`:

```
ENCODING(NNR)
ENCODING(NATIVE)
ENCODING(RR3)
```

**SSL properties**

When `TRANSPORT(CLIENT)` is specified, Secure Sockets Layer (SSL) encrypted communication can be enabled using the `SSLCIPHERSUITE` property. This property must be set to a valid CipherSuite provided by your JSSSE provider; and needs to match the CipherSpec named on the `SVRCONN` channel named by the `CHANNEL` property.

However, CipherSpecs (as specified on the `SVRCONN` channel) and CipherSuites (as specified on `ConnectionFactory` objects) use different naming schemes to represent the same SSL encryption algorithms. If a recognised CipherSpec name is specified on the `SSLCIPHERSUITE` property, JMSAdmin will issue a warning and map the CipherSpec to its equivalent CipherSuite. See Appendix H, “SSL CipherSuites supported by WebSphere MQ” on page 425 for a list of CipherSpecs recognised by WebSphere MQ and JMSAdmin.

The `SSLPEERNAME` matches the format of the SSLPEER parameter which can be set on channel definitions. It is a list of attribute name and value pairs separated by commas or semicolons. For example:

```
SSLPEERNAME(CN=QMGR.*, OU=IBM, OU=WEBSPHERE)
```

The set of names and values makes up a *distinguished name*. For more details about distinguished names and their use with WebSphere MQ, see the WebSphere MQ Security book.

The example given will cause the identifying certificate presented by the server to be checked at connect-time. For the connection to succeed, the certificate must have a Common Name beginning “QMGR.”, and must have at least two Organizational Unit names, the first of which must be “IBM” and the second “WEBSPHERE”. Checking is case-insensitive.

If `SSLPEERNAME` is not set, no such checking is performed. Note that `SSLPEERNAME` is ignored if `SSLCIPHERSUITE` is not specified.

The `SSLCRL` property specifies zero or more CRL (Certificate Revocation List) servers. Use of this property requires a JVM at Java 2 v1.4. This is a space-delimited list of entries of the form:

```
ldap://hostname:[port]
```

optionally followed by a single “/”. If `port` is omitted, the default LDAP port of 389 is assumed. At connect-time, the SSL certificate presented by the server is checked against the specified CRL servers. See the WebSphere MQ Security book for more about CRL security.
If SSLCRL is not set, no such checking is performed. Note that SSLCRL is ignored if SSLCIPHERSUITE is not specified.

Sample error conditions

This section provides examples of the error conditions that may arise during the creation of an object.

CipherSpec mapped to CipherSuite

```
InitCtx/cn=Trash> DEFINE QCF(testQCF) SSLCIPHERSUITE(RC4_MD5_US)
    WARNING: Converting CipherSpec RC4_MD5_US to
    CipherSuite SSL_RSA_WITH_RC4_128_MD5
```

Invalid property for object

```
InitCtx/cn=Trash> DEFINE QCF(testQCF) PRIORITY(4)
    Unable to create a valid object, please check the parameters supplied
    Invalid property for a QCF: PRI
```

Invalid type for property value

```
InitCtx/cn=Trash> DEFINE QCF(testQCF) CCSID(english)
    Unable to create a valid object, please check the parameters supplied
    Invalid value for CCS property: English
```

Property clash - client/bindings

```
InitCtx/cn=Trash> DEFINE QCF(testQCF) HOSTNAME(polaris.hursley.ibm.com)
    Unable to create a valid object, please check the parameters supplied
    Invalid property in this context: Client-bindings attribute clash
```

Property clash - Exit initialization

```
InitCtx/cn=Trash> DEFINE QCF(testQCF) SECEXITINIT(initStr)
    Unable to create a valid object, please check the parameters supplied
    Invalid property in this context: ExitInit string supplied without Exit string
```

Property value outside valid range

```
InitCtx/cn=Trash> DEFINE Q(testQ) PRIORITY(12)
    Unable to create a valid object, please check the parameters supplied
    Invalid value for PRI property: 12
```

Unknown property

```
InitCtx/cn=Trash> DEFINE QCF(testQCF) PIZZA(ham and mushroom)
    Unable to create a valid object, please check the parameters supplied
    Unknown property: PIZZA
```
Part 2. Programming with WebSphere MQ base Java

Chapter 6. Introduction for programmers

Why should I use the Java interface? ........................................... 57
The WebSphere MQ classes for Java interface ................. 58
Java Development Kit .......................................................... 58
WebSphere MQ classes for Java class library ................. 59

Chapter 7. Writing WebSphere MQ base Java programs

Should I write applets or applications? ......................... 61
Connection differences ..................................................... 61
Client connections .......................................................... 61
Bindings mode ................................................................. 62
Defining which connection to use .................................. 62
Example code fragments ................................................ 62
Example applet code ....................................................... 62
Example application code .............................................. 65
Operations on queue managers ...................................... 67
Setting up the WebSphere MQ environment .................... 67
Connecting to a queue manager ....................................... 68
Accessing queues and processes ..................................... 68
Handling messages .......................................................... 69
Handling errors .............................................................. 70
Getting and setting attribute values ............................... 71
Multithreaded programs .................................................. 72
Writing user exits ............................................................ 72
Connection pooling .......................................................... 73
Controlling the default connection pool ......................... 74
The default connection pool and multiple components ......... 75
Supplying a different connection pool ............................. 76
Supplying your own ConnectionManager ....................... 78
JTA/JDBC coordination via WebSphere MQ base Java ......... 79
Installation ............................................................... 79
Installation on Windows systems ................................. 79
Installation on other platforms ........................................ 80
Usage ........................................................................ 80
Known problems and limitations .................................... 81
Secure Sockets Layer (SSL) support ............................... 82
Enabling SSL ................................................................. 82
Using the distinguished name of the queue manager .......... 83
Using certificate revocation lists ..................................... 83
Supplying a customized SSL Socket Factory .................... 84
Error handling when using SSL .......................................... 85
Compiling and testing WebSphere MQ base Java programs .... 86
Running WebSphere MQ base Java applets ..................... 86
Running WebSphere MQ base Java applications ................. 86
Tracing WebSphere MQ base Java programs ..................... 86

Chapter 8. Environment-dependent behavior ...... 89
Core details ........................................................................ 89
Restrictions and variations for core classes .................... 90
MQGMO_* values .......................................................... 90
MQPMO_* values .......................................................... 90

Chapter 9. The WebSphere MQ base Java classes and interfaces ... 95
MQChannelDefinition ...................................................... 96
Variables ................................................................ 96
Constructors ............................................................... 97
MQChannelExit .............................................................. 98
Variables ................................................................ 98
Constructors ............................................................... 100
MQDistributionList ....................................................... 101
Constructors ............................................................... 101
Methods ................................................................. 101
MQDistributionListItem ................................................ 103
Variables ............................................................... 103
Constructors ............................................................... 103
MQException .............................................................. 110
Variables ............................................................... 110
Constructors ............................................................... 110
Methods ................................................................. 111
MQEnvironment .......................................................... 104
Variables ............................................................... 105
Constructors ............................................................... 108
Methods ................................................................. 108
MQMessage ............................................................... 119
Variables ............................................................... 119
Constructors ............................................................... 127
Methods ................................................................. 127
MQMessageTracker ........................................................ 137
Variables ............................................................... 137
MQPoolServices ........................................................ 139
Constructors ............................................................... 139
Methods ................................................................. 139
MQPoolServicesEvent .................................................... 140
Variables ............................................................... 140
Constructors ............................................................... 141
MQPoolToken .............................................................. 142
Constructors ............................................................... 142

© Copyright IBM Corp. 1997, 2002
<table>
<thead>
<tr>
<th>Class</th>
<th>Constructor(s)</th>
<th>Method(s)</th>
<th>Variable(s)</th>
<th>Constructor(s)</th>
<th>Method(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQProcess</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQPutMessageOptions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQQueue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQQueueManager</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQSimpleConnectionManager</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M QC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQPoolServicesEventListener</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQConnectionManager</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQReceiveExit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQSecurityExit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQSendExit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ManagedConnection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ManagedConnectionFactory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ManagedConnectionMetaData</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

WebSphere MQ Using Java
Chapter 6. Introduction for programmers

This chapter contains general information for programmers. For more detailed information about writing programs, see Chapter 7, “Writing WebSphere MQ base Java programs” on page 61.

Why should I use the Java interface?

The WebSphere MQ classes for Java programming interface makes the many benefits of Java available to you as a developer of WebSphere MQ applications:

- **The Java programming language is easy to use.**
  There is no need for header files, pointers, structures, unions, and operator overloading. Programs written in Java are easier to develop and debug than their C and C++ equivalents.

- **Java is object-oriented.**
  The object-oriented features of Java are comparable to those of C++, but there is no multiple inheritance. Instead, Java uses the concept of an interface.

- **Java is inherently distributed.**
  The Java class libraries contain a library of routines for coping with TCP/IP protocols like HTTP and FTP. Java programs can access URLs as easily as accessing a file system.

- **Java is robust.**
  Java puts a lot of emphasis on early checking for possible problems, dynamic (runtime) checking, and the elimination of situations that are error prone. Java uses a concept of references that eliminates the possibility of overwriting memory and corrupting data.

- **Java is secure.**
  Java is intended to be run in networked or distributed environments, and a lot of emphasis has been placed on security. Java programs cannot overrun their runtime stack and cannot corrupt memory outside of their process space. When Java programs are downloaded from the Internet, they cannot even read or write local files.

- **Java programs are portable.**
  There are no "implementation-dependent" aspects of the Java specification. The Java compiler generates an architecture-neutral object file format. The compiled code is executable on many processors, as long as the Java runtime system is present.

If you write your application using WebSphere MQ classes for Java, users can download the Java byte codes (called applets) for your program from the Internet. Users can then run these applets on their own machines. This means that users with access to your Web server can load and run your application with no prior installation needed on their machines.

When an update to the program is required, you update the copy on the Web server. The next time that users access the applet, they automatically receive the latest version. This can significantly reduce the costs involved in installing and updating traditional client applications where a large number of desktops are involved.
Advantages of Java

If you place your applet on a Web server that is accessible outside the corporate firewall, anyone on the Internet can download and use your application. This means that you can get messages into your WebSphere MQ system from anywhere on the internet. This opens the door to building a whole new set of Internet accessible service, support, and electronic commerce applications.

The WebSphere MQ classes for Java interface

The procedural WebSphere MQ application programming interface is built around the following verbs:

- MQBACK
- MQBEGIN
- MQCLOSE
- MQCMIT
- MQCONNX
- MQDISC
- MQGET
- MQINQ
- MQOPEN
- MQPUT
- MQPUT1
- MQSET

These verbs all take, as a parameter, a handle to the WebSphere MQ object on which they are to operate. Because Java is object-oriented, the Java programming interface turns this round. Your program consists of a set of WebSphere MQ objects, which you act upon by calling methods on those objects, as in the following example.

When you use the procedural interface, you disconnect from a queue manager by using the call MQDISC(Hconn, CompCode, Reason), where Hconn is a handle to the queue manager.

In the Java interface, the queue manager is represented by an object of class MQQueueManager. You disconnect from the queue manager by calling the disconnect() method on that class.

```java
// declare an object of type queue manager
MQQueueManager queueManager=new MQQueueManager();
...
// do something...
...
// disconnect from the queue manager
queueManager.disconnect();
```

Java Development Kit

Before you can compile any applets or applications that you write, you must have access to a Java Development Kit (JDK) for your development platform. The JDK contains all the standard Java classes, variables, constructors, and interfaces on which the WebSphere MQ classes for Java classes depend. It also contains the tools required to compile and run the applets and programs on each supported platform.

You can download IBM Developer Kits for Java from the IBM Software Download Catalog, which is available on the World Wide Web at location:

http://www.ibm.com/developerworks/java

To compile Java applications on the iSeries & AS/400 platforms, you must first install:

- The AS/400 Developer Kit for Java, 5769-JV1
- The Qshell Interpreter, OS/400 (5769-SS1) Option 30
WebSphere MQ classes for Java class library

WebSphere MQ classes for Java is a set of Java classes that enable Java applets and applications to interact with WebSphere MQ.

The following classes are provided:
- MQChannelDefinition
- MQChannelExit
- MQDistributionList
- MQDistributionListItem
- MQEnvironment
- MQException
- MQGetMessageOptions
- MQManagedObject
- MQMessage
- MQMessageTracker
- MQPoolServices
- MQPoolServicesEvent
- MQPoolToken
- MQPutMessageOptions
- MQProcess
- MQQueue
- MQQueueManager
- MQSimpleConnectionManager

The following Java interfaces are provided:
- MQC
- MQPoolServicesEventListener
- MQReceiveExit
- MQSecurityExit
- MQSendExit

Implementation of the following Java interfaces are also provided. However, these interfaces are not intended for direct use by applications:
- MQConnectionManager
- javax.resource.spi.ManagedConnection
- javax.resource.spi.ManagedConnectionFactory
- javax.resource.spi.ManagedConnectionMetaData

In Java, a package is a mechanism for grouping sets of related classes together. The WebSphere MQ classes and interfaces are shipped as a Java package called com.ibm.mq. To include the WebSphere MQ classes for Java package in your program, add the following line at the top of your source file:

```java
import com.ibm.mq.*;
```
WebSphere MQ base Java class library
Chapter 7. Writing WebSphere MQ base Java programs

To use WebSphere MQ classes for Java to access WebSphere MQ queues, you write Java programs that contain calls that put messages onto, and get messages from, WebSphere MQ queues. The programs can take the form of Java applets, Java servlets, or Java applications.

This chapter provides information to assist with writing Java applets, servlets, and applications to interact with WebSphere MQ systems. For details of individual classes, see Chapter 9, “The WebSphere MQ base Java classes and interfaces” on page 95.

Should I write applets or applications?

Whether you write applets, servlets, or applications depends on the connection that you want to use and from where you want to run the programs.

The main differences between applets and applications are:

- Applets are run with an applet viewer or in a Web browser, servlets are run in a Web application server, and applications are run standalone.
- Applets can be downloaded from a Web server to a Web browser machine, but applications and servlets are not.
- Applets run with additional security rules limiting what they can do.

See Appendix F, “Using WebSphere MQ Java in applets with Java 1.2 or later” on page 419 for more information about this.

The following general rules apply:

- If you want to run your programs from machines that do not have WebSphere MQ classes for Java installed locally, you should write applets.
- The native bindings mode of WebSphere MQ classes for Java does not support applets. Therefore, if you want to use your programs in all connection modes, including the native bindings mode, you must write servlets or applications.

Connection differences

The way you program for WebSphere MQ classes for Java has some dependencies on the connection modes you want to use.

Client connections

When WebSphere MQ classes for Java is used as a client, it is similar to the WebSphere MQ C client, but has the following differences:

- It supports only TCP/IP.
- It does not support connection tables.
- It does not read any WebSphere MQ environment variables at startup.
- Information that would be stored in a channel definition and in environment variables is stored in a class called Environment. Alternatively, this information can be passed as parameters when the connection is made.
- Error and exception conditions are written to a log specified in the MQException class. The default error destination is the Java console.
The WebSphere MQ classes for Java clients do not support the MQBEGIN verb or fast bindings.

For general information on WebSphere MQ clients, see the *WebSphere MQ Clients* book.

**Bindings mode**

The bindings mode of WebSphere MQ classes for Java differs from the client modes in the following ways:

* Most of the parameters provided by the MQEnvironment class are ignored
* The bindings support the MQBEGIN verb and fast bindings into the WebSphere MQ queue manager

**Note:** WebSphere MQ for iSeries and WebSphere MQ for z/OS do not support the use of MQBEGIN to initiate global units of work that are coordinated by the queue manager.

**Defining which connection to use**

The connection is determined by the setting of variables in the MQEnvironment class.

**MQEnvironment.properties**

This can contain the following key/value pairs:

* For client and bindings connections:
  
  MQC.TRANSPORT_PROPERTY, MQC.TRANSPORT_MQSERIES

**MQEnvironment.hostname**

Set the value of this variable follows:

* For client connections, set this to the hostname of the WebSphere MQ server to which you want to connect
* For bindings mode, set this to null

**Example code fragments**

This section includes two example code fragments; Figure 1 on page 63 and Figure 2 on page 66. Each one uses a particular connection and includes notes to describe the changes needed to use alternative connections.

**Example applet code**

The following code fragment demonstrates an applet that uses a TCP/IP connection to:

1. Connect to a queue manager
2. Put a message onto SYSTEM.DEFAULT.LOCAL.QUEUE
3. Get the message back
public class MQSample extends java.applet.Applet
{

    private String hostname = "your_hostname"; // define the name of your
    // host to connect to
    private String channel = "server_channel"; // define name of channel
    // for client to use
    // Note. assumes WebSphere MQ Server
    // is listening on the default
    // TCP/IP port of 1414

    private String qManager = "your_Q_manager"; // define name of queue
    // manager object to
    // connect to.

    private MQQueueManager qMgr; // define a queue manager object

    // When the class is called, this initialization is done first.

    public void init()
    {
        // Set up WebSphere MQ environment
        MQEnvironment.hostname = hostname; // Could have put the
        // hostname & channel
        MQEnvironment.channel = channel; // string directly here!

        MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY, //Set TCP/IP or server
                                  MQC.TRANSPORT_MQSERIES); //Connection

    } // end of init

Figure 1. WebSphere MQ classes for Java example applet (Part 1 of 3)
public void start()
{
    try {
        // Create a connection to the queue manager
        qMgr = new MQQueueManager(qManager);

        // Set up the options on the queue we wish to open...
        // Note. All WebSphere MQ Options are prefixed with MQC in Java.
        int openOptions = MQC.MQOO_INPUT_AS_Q_DEF |
            MQC.MQOO_OUTPUT;
        // Now specify the queue that we wish to open, and the open options...
        MQQueue system_default_local_queue =
            qMgr.accessQueue("SYSTEM.DEFAULT.LOCAL.QUEUE",
                openOptions);

        // Define a simple WebSphere MQ message, and write some text in UTF format..
        MQMessage hello_world = new MQMessage();
        hello_world.writeUTF("Hello World!");

        // specify the message options...
        MQPutMessageOptions pmo = new MQPutMessageOptions(); // accept the defaults,
            // same as
            // MQPMO_DEFAULT
            // constant

        // put the message on the queue
        system_default_local_queue.put(hello_world, pmo);

        // get the message back again...
        // First define a WebSphere MQ message buffer to receive the message into..
        MQMessage retrievedMessage = new MQMessage();
        retrievedMessage.messageId = hello_world.messageId;

        // Set the get message options..
        MQGetMessageOptions gmo = new MQGetMessageOptions(); // accept the defaults
            // same as
            // MQGMO_DEFAULT

        // get the message off the queue..
        system_default_local_queue.get(retrievedMessage, gmo);

        // And prove we have the message by displaying the UTF message text
        String msgText = retrievedMessage.readUTF();
        System.out.println("The message is: " + msgText);

        // Close the queue
        system_default_local_queue.close();

        // Disconnect from the queue manager
        qMgr.disconnect();
    }

    // If an error has occurred in the above, try to identify what went wrong.
    // Was it a WebSphere MQ error?
}

Figure 1. WebSphere MQ classes for Java example applet (Part 2 of 3)
Example application code

The following code fragment demonstrates a simple application that uses bindings mode to:

1. Connect to a queue manager
2. Put a message onto SYSTEM.DEFAULT.LOCAL.QUEUE
3. Get the message back again

```java
try {
    System.out.println("A WebSphere MQ error occurred : Completion code " +
        ex.completionCode +
        " Reason code " + ex.reasonCode);
}
// Was it a Java buffer space error?
try {
    System.out.println("An error occurred whilst writing to the message buffer: " + ex);
}
} // end of start
} // end of sample
```

Figure 1. WebSphere MQ classes for Java example applet (Part 3 of 3)
Example code

```java
// Licensed Materials - Property of IBM
// 5639-C34
// (c) Copyright IBM Corp. 1995, 1999
// WebSphere MQ classes for Java sample application

// This sample runs as a Java application using the command :- java MQSample

import com.ibm.mq.*; // Include the WebSphere MQ classes for Java package

public class MQSample
{
    private String qManager = "your_Q_manager"; // define name of queue manager
    private MQQueueManager qMgr; // define a queue manager object

    public static void main(String args[]) {
        new MQSample();
    }

    public MQSample() {
        try {
            // Create a connection to the queue manager
            qMgr = new MQQueueManager(qManager);
            // Set up the options on the queue we wish to open...
            // Note. All WebSphere MQ Options are prefixed with MQC in Java.
            int openOptions = MQC.MQOO_INPUT_AS_Q_DEF |
                                MQC.MQOO_OUTPUT ;
            // Now specify the queue that we wish to open,
            // and the open options...
            MQQueue system_default_local_queue =
                qMgr.accessQueue("SYSTEM.DEFAULT.LOCAL.QUEUE",
                                openOptions);

            // Define a simple WebSphere MQ message, and write some text in UTF format..
            MQMessage hello_world = new MQMessage();
            hello_world.writeUTF("Hello World!");

            // specify the message options...
            MQPutMessageOptions pmo = new MQPutMessageOptions(); // accept the // defaults,
            // same as MQPMO_DEFAULT
        }
    }
}
```

Figure 2. WebSphere MQ classes for Java example application (Part 1 of 2)
// put the message on the queue
system_default_local_queue.put(hello_world, pmo);

// get the message back again...
// First define a WebSphere MQ message buffer to receive the message into..
MQMessage retrievedMessage = new MQMessage();
retrievedMessage.messageId = hello_world.messageId;

// Set the get message options...
MQGetMessageOptions gmo = new MQGetMessageOptions(); // accept the defaults
// same as MQGMO_DEFAULT
// get the message off the queue...

system_default_local_queue.get(retrievedMessage, gmo);

// And prove we have the message by displaying the UTF message text
String msgText = retrievedMessage.readUTF();
System.out.println("The message is: " + msgText);

// Disconnect from the queue manager
qMgr.disconnect();

} // end of sample

Figure 2. WebSphere MQ classes for Java example application (Part 2 of 2)

Operations on queue managers

This section describes how to connect to, and disconnect from, a queue manager using WebSphere MQ classes for Java.

Setting up the WebSphere MQ environment

Note: This step is not necessary when using WebSphere MQ classes for Java in bindings mode. In that case, go directly to "Connecting to a queue manager" on page 68. Before you use the client connection to connect to a queue manager, you must take care to set up the MQEnvironment.

The "C" based WebSphere MQ clients rely on environment variables to control the behavior of the MQCONN call. Because Java applets have no access to environment variables, the Java programming interface includes a class MQEnvironment. This class allows you to specify the following details that are to be used during the connection attempt:
Queue manager operations

- Channel name
- Hostname
- Port number
- User ID
- Password

To specify the channel name and hostname, use the following code:

```java
MQEnvironment.hostname = "host.domain.com";
MQEnvironment.channel = "java.client.channel";
```

This is equivalent to an MQSERVER environment variable setting of:
"java.client.channel/TCP/host.domain.com".

By default, the Java clients attempt to connect to a WebSphere MQ listener at port 1414. To specify a different port, use the code:

```java
MQEnvironment.port = nnnn;
```

The user ID and password default to blanks. To specify a non-blank user ID or password, use the code:

```java
MQEnvironment.userID = "uid"; // equivalent to env var MQ_USER_ID
MQEnvironment.password = "pwd"; // equivalent to env var MQ_PASSWORD
```

Connecting to a queue manager

You are now ready to connect to a queue manager by creating a new instance of the MQQueueManager class:

```java
MQQueueManager queueManager = new MQQueueManager("qMgrName");
```

To disconnect from a queue manager, call the disconnect() method on the queue manager:

```java
queueManager.disconnect();
```

If you call the disconnect method, all open queues and processes that you have accessed through that queue manager will be closed. However, it is good programming practice to close these resources explicitly when you finish using them. To do this, use the close() method.

The commit() and backout() methods on a queue manager replace the MQCMIT and MQBACK calls that are used with the procedural interface.

Accessing queues and processes

To access queues and processes, you use the MQQueueManager class. The MQOD (object descriptor structure) is collapsed into the parameters of these methods. For example, to open a queue on a queue manager "queueManager", use the following code:

```java
MQQueue queue = queueManager.accessQueue("qName",
MQC.MQOO_OUTPUT,
"qMgrName",
"dynamicQName",
"altUserId");
```

The `options` parameter is the same as the Options parameter in the MQOPEN call.

The accessQueue method returns a new object of class MQQueue.
When you have finished using the queue, use the close() method to close it, as in the following example:
queue.close();

With WebSphere MQ classes for Java, you can also create a queue by using the MQQueue constructor. The parameters are exactly the same as for the accessQueue method, with the addition of a queue manager parameter. For example:
MQQueue queue = new MQQueue(queueManager, 
  "qName", 
  MQC.MQOO_OUTPUT, 
  "qMgrName", 
  "dynamicQName", 
  "altUserId");

Constructing a queue object in this way enables you to write your own subclasses of MQQueue.

To access a process, use the accessProcess method in place of accessQueue. This method does not have a dynamic queue name parameter, because this does not apply to processes.

The accessProcess method returns a new object of class MQProcess.

When you have finished using the process object, use the close() method to close it, as in the following example:
process.close();

With WebSphere MQ classes for Java, you can also create a process by using the MQProcess constructor. The parameters are exactly the same as for the accessProcess method, with the addition of a queue manager parameter. Constructing a process object in this way enables you to write your own subclasses of MQProcess.

Handling messages

You put messages onto queues using the put() method of the MQQueue class. You get messages from queues using the get() method of the MQQueue class. Unlike the procedural interface, where MQPUT and MQGET put and get arrays of bytes, the Java programming language puts and gets instances of the MQMessage class. The MQMessage class encapsulates the data buffer that contains the actual message data, together with all the MQMD (message descriptor) parameters that describe that message.

To build a new message, create a new instance of the MQMessage class, and use the writeXXX methods to put data into the message buffer.

When the new message instance is created, all the MQMD parameters are automatically set to their default values, as defined in the WebSphere MQ Application Programming Reference. The put() method of MQQueue also takes an instance of the MQPutMessageOptions class as a parameter. This class represents the MQPMO structure. The following example creates a message and puts it onto a queue:

```java
// Build a new message containing my age followed by my name
MQMessage myMessage = new MQMessage();
myMessage.writeInt(25); 
String name = "Wendy Ling";
```
Message handling

```java
myMessage.writeInt(name.length());
myMessage.writeBytes(name);

// Use the default put message options...
MQPutMessageOptions pmo = new MQPutMessageOptions();

// put the message!
queue.put(myMessage, pmo);
```

The `get()` method of `MQQueue` returns a new instance of `MQMessage`, which represents the message just taken from the queue. It also takes an instance of the `MQGetMessageOptions` class as a parameter. This class represents the MQGMO structure.

You do not need to specify a maximum message size, because the `get()` method automatically adjusts the size of its internal buffer to fit the incoming message. Use the `readXXX` methods of the `MQMessage` class to access the data in the returned message.

The following example shows how to get a message from a queue:

```java
// Get a message from the queue
MQMessage theMessage = new MQMessage();
MQGetMessageOptions gmo = new MQGetMessageOptions();
queue.get(theMessage, gmo); // has default values

// Extract the message data
int age = theMessage.readInt();
int strLen = theMessage.readInt();
byte[] strData = new byte[strLen];
theMessage.readFully(strData, 0, strLen);
String name = new String(strData, 0);
```

You can alter the number format that the read and write methods use by setting the `encoding` member variable.

You can alter the character set to use for reading and writing strings by setting the `characterSet` member variable.

See "MQMessage" on page 119 for more details.

Note: The `writeUTF()` method of `MQMessage` automatically encodes the length of the string as well as the Unicode bytes it contains. When your message will be read by another Java program (using `readUTF()`), this is the simplest way to send string information.

Handling errors

Methods in the Java interface do not return a completion code and reason code. Instead, they throw an exception whenever the completion code and reason code resulting from a WebSphere MQ call are not both zero. This simplifies the program logic so that you do not have to check the return codes after each call to WebSphere MQ. You can decide at which points in your program you want to deal with the possibility of failure. At these points, you can surround your code with `try` and `catch` blocks, as in the following example:

```java
try {
    myQueue.put(messageA, putMessageOptionsA);
    myQueue.put(messageB, putMessageOptionsB);
} catch (MQException ex) {
```
// This block of code is only executed if one of
// the two put methods gave rise to a non-zero
// completion code or reason code.
System.out.println("An error occurred during the put operation:" +
       " CC = " + ex.completionCode +
       " RC = " + ex.reasonCode);
System.out.println("Cause exception:" + ex.getCause() );
}

The WebSphere MQ call reason codes reported back in Java exceptions are
documented in a chapter called “Return Codes” in the WebSphere MQ Application

Sometimes the reason code does not convey all details associated with the error.
This can occur if WebSphere MQ uses services provided by another product (for
example, a JSSE implementation) which throws an Exception to WebSphere MQ. In
this case, the method MQException.getCause() will retrieve the underlying
Exception which caused the error.

Getting and setting attribute values

For many of the common attributes, the classes MQManagedObject, MQQueue,
MQProcess, and MQQueueManager contain getXXX() and setXXX() methods.
These methods allow you to get and set their attribute values. Note that for
MQQueue, the methods will work only if you specify the appropriate “inquire”
and “set” flags when you open the queue.

For less common attributes, the MQQueueManager, MQQueue, and MQProcess
classes all inherit from a class called MQManagedObject. This class defines the
inquire() and set() interfaces.

When you create a new queue manager object by using the new operator, it is
automatically opened for “inquiry”. When you use the accessProcess() method to
access a process object, that object is automatically opened for “inquiry”. When
you use the accessQueue() method to access a queue object, that object is not
automatically opened for either “inquire” or “set” operations. This is because
adding these options automatically can cause problems with some types of remote
queues. To use the inquire, set, getXXX, and setXXX methods on a queue, you
must specify the appropriate “inquire” and “set” flags in the openOptions
parameter of the accessQueue() method.

The inquire and set methods take three parameters:
• selectors array
• intAttrs array
• charAttrs array

You do not need the SelectorCount, IntAttrCount, and CharAttrLength parameters
that are found in MQINQ, because the length of an array in Java is always known.
The following example shows how to make an inquiry on a queue:

// inquire on a queue
final static int MQIA_DEF_PRIORITY = 6;
final static int MQCA_Q_DESC = 2013;
final static int MQ_Q_DESC_LENGTH = 64;

int[] selectors = new int[2];
int[] intAttrs = new int[1];
byte[] charAttrs = new byte[MQ_Q_DESC_LENGTH];

selectors[0] = MQIA_DEF_PRIORITY;
Using attribute values

```java
selectors[1] = MQCA_Q_DESC;
queue.inquire(selectors,intAttrs,charAttrs);
System.out.println("Default Priority = " + intAttrs[0]);
System.out.println("Description : " + new String(charAttrs,0));
```

**Multithreaded programs**

Multithreaded programs are hard to avoid in Java. Consider a simple program that connects to a queue manager and opens a queue at startup. The program displays a single button on the screen. When a user presses that button, the program fetches a message from the queue.

The Java runtime environment is inherently multithreaded. Therefore, your application initialization occurs in one thread, and the code that executes in response to the button press executes in a separate thread (the user interface thread).

With the “C” based WebSphere MQ client, this would cause a problem, because handles cannot be shared across multiple threads. WebSphere MQ classes for Java relaxes this constraint, allowing a queue manager object (and its associated queue and process objects) to be shared across multiple threads.

The implementation of WebSphere MQ classes for Java ensures that, for a given connection (MQQueueManager object instance), all access to the target WebSphere MQ queue manager is synchronized. Therefore, a thread wishing to issue a call to a queue manager is blocked until all other calls in progress for that connection are complete. If you require simultaneous access to the same queue manager from multiple threads within your program, create a new MQQueueManager object for each thread that requires concurrent access. (This is equivalent to issuing a separate MQCONN call for each thread.)

**Writing user exits**

WebSphere MQ classes for Java allows you to provide your own send, receive, and security exits.

To implement an exit, you define a new Java class that implements the appropriate interface. Three exit interfaces are defined in the WebSphere MQ package:

- MQSendExit
- MQReceiveExit
- MQSecurityExit

Any SSL encryption defined for a connection is performed after the send exit has been invoked. Similarly, decryption is performed before the receive or security exits are invoked.

The following sample defines a class that implements all three:

```java
class MyMQExits implements MQSendExit, MQReceiveExit, MQSecurityExit {
    // This method comes from the send exit
    public byte[] sendExit(MQChannelExit channelExitParms,
                            MQChannelDefinition channelDefParms,
                            byte agentBuffer[]) {
        // fill in the body of the send exit here
    }
}
```
// This method comes from the receive exit
public byte[] receiveExit(MQChannelExit channelExitParms,
MQChannelDefinition channelDefParms,
byte agentBuffer[])
{
    // fill in the body of the receive exit here
}

// This method comes from the security exit
public byte[] securityExit(MQChannelExit channelExitParms,
MQChannelDefinition channelDefParms,
byte agentBuffer[])
{
    // fill in the body of the security exit here
}

Each exit is passed an MQChannelExit and an MQChannelDefinition object instance. These objects represent the MQCXP and MQCD structures defined in the procedural interface.

For a Send exit, the `agentBuffer` parameter contains the data that is about to be sent. For a Receive exit or a Security exit, the `agentBuffer` parameter contains the data that has just been received. You do not need a length parameter, because the expression `agentBuffer.length` indicates the length of the array.

For the Send and Security exits, your exit code should return the byte array that you wish to send to the server. For a Receive exit, your exit code should return the modified data that you wish WebSphere MQ classes for Java to interpret.

The simplest possible exit body is:
{
    return agentBuffer;
}

If your program is to run as a downloaded Java applet, the security restrictions that apply mean that you cannot read or write any local files. If your exit needs a configuration file, you can place the file on the Web and use the java.net.URL class to download it and examine its contents.

## Connection pooling

WebSphere MQ classes for Java Version 5.2 provides additional support for applications that deal with multiple connections to WebSphere MQ queue managers. When a connection is no longer required, instead of destroying it, it can be pooled, and later reused. This can provide a substantial performance enhancement for applications and middleware that connect serially to arbitrary queue managers.

WebSphere MQ provides a default connection pool. Applications can activate or deactivate this connection pool by registering and deregistering tokens through the MQEnvironment class. If the pool is active, when WebSphere MQ base Java constructs an MQQueueManager object, it searches this default pool and reuses any suitable connection. When an MQQueueManager.disconnect() call occurs, the underlying connection is returned to the pool.
Connection pooling

Alternatively, applications can construct an MQSimpleConnectionManager connection pool for a particular use. Then, the application can either specify that pool during construction of an MQQueueManager object, or pass that pool to MQEnvironment for use as the default connection pool.

Also, WebSphere MQ base Java provides a partial implementation of the Java 2 Platform Enterprise Edition (J2EE) Connector Architecture. Applications running under a Java 2 v1.3 JVM with JAAS 1.0 (Java Authentication and Authorization Service) can provide their own connection pool by implementing the javax.resource.spi.ConnectionManager interface. Again, this interface can be specified on the MQQueueManager constructor, or specified as the default connection pool.

Controlling the default connection pool

Consider the following example application, MQApp1:

```java
import com.ibm.mq.*;
public class MQApp1
{
    public static void main(String[] args) throws MQException {
        for (int i=0; i<args.length; i++) {
            MQQueueManager qmgr=new MQQueueManager(args[i]);
            : (do something with qmgr)
            qmgr.disconnect();
        }
    }
}
```

MQApp1 takes a list of local queue managers from the command line, connects to each in turn, and performs some operation. However, when the command line lists the same queue manager many times, it is more efficient to connect only once, and to reuse that connection many times.

WebSphere MQ base Java provides a default connection pool that you can use to do this. To enable the pool, use one of the MQEnvironment.addConnectionPoolToken() methods. To disable the pool, use MQEnvironment.removeConnectionPoolToken().

The following example application, MQApp2, is functionally identical to MQApp1, but connects only once to each queue manager.

```java
import com.ibm.mq.*;
public class MQApp2
{
    public static void main(String[] args) throws MQException {
        MQPoolToken token=MQEnvironment.addConnectionPoolToken();
        for (int i=0; i<args.length; i++) {
            MQQueueManager qmgr=new MQQueueManager(args[i]);
            : (do something with qmgr)
            qmgr.disconnect();
        }
    }
}
```
The first bold line activates the default connection pool, by registering an MQPoolToken object with MQEnvironment.

The MQQueueManager constructor now searches this pool for an appropriate connection and only creates a connection to the queue manager if it cannot find an existing one. The qmgr.disconnect() call returns the connection to the pool for later reuse. These API calls are the same as the sample application MQApp1.

The second highlighted line deactivates the default connection pool, which destroys any queue manager connections stored in the pool. This is important because otherwise, the application would terminate with a number of live queue manager connections in the pool. This situation could cause errors that would appear in the queue manager logs.

The default connection pool stores a maximum of ten unused connections, and keeps unused connections active for a maximum of five minutes. The application can alter this (for details, see “Supplying a different connection pool” on page 76).

Instead of using MQEnvironment to supply an MQPoolToken, the application can construct its own:

```java
MQPoolToken token = new MQPoolToken();
MQEnvironment.addConnectionPoolToken(token);
```

Some applications or middleware vendors may provide subclasses of MQPoolToken in order to pass information to a custom connection pool. They can be constructed and passed to addConnectionPoolToken() in this way so that extra information can be passed to the connection pool.

### The default connection pool and multiple components

MQEnvironment holds a static set of registered MQPoolToken objects. To add or remove MQPoolTokens from this set, use the following methods:

- MQEnvironment.addConnectionPoolToken()
- MQEnvironment.removeConnectionPoolToken()

An application might consist of many components that exist independently and perform work using a queue manager. In such an application, each component should add an MQPoolToken to the MQEnvironment set for its lifetime.

For example, the example application MQApp3 creates ten threads and starts each one. Each thread registers its own MQPoolToken, waits for a length of time, then connects to the queue manager. After the thread disconnects, it removes its own MQPoolToken.

The default connection pool remains active while there is at least one token in the set of MQPoolTokens, so it will remain active for the duration of this application. The application does not need to keep a master object in overall control of the threads.
Connection pooling

```java
for (int i=0; i<10; i++) {
    MQApp3_Thread thread=new MQApp3_Thread(i*60000);
    thread.start();
}
```

class MQApp3_Thread extends Thread
{
    long time;

    public MQApp3_Thread(long time)
    {
        this.time=time;
    }

    public synchronized void run()
    {
        MQPoolToken token=MQEnvironment.addConnectionPoolToken();
        try {
            wait(time);
            MQQueueManager qmgr=new MQQueueManager("my.qmgr.1");
            : : (do something with qmgr)
            : : qmgr.disconnect();
        } catch (MQException mqe) {System.err.println("Error occurred!");} catch (InterruptedException ie) {} 
        MQEnvironment.removeConnectionPoolToken(token);
    }
}
```

Supplying a different connection pool

This section describes how to use the class `com.ibm.mq.MQSimpleConnectionManager` to supply a different connection pool. This class provides basic facilities for connection pooling, and applications can use this class to customize the behavior of the pool.

Once it is instantiated, an MQSimpleConnectionManager may be specified on the MQQueueManager constructor. The MQSimpleConnectionManager then manages the connection that underlies the constructed MQQueueManager. If the MQSimpleConnectionManager contains a suitable pooled connection, that connection will be reused, and it will be returned to the MQSimpleConnectionManager after an MQQueueManager.disconnect() call.

The following code fragment demonstrates this behavior:

```java
MQSimpleConnectionManager myConnMan=new MQSimpleConnectionManager();
myConnMan.setActive(MQSimpleConnectionManager.MODE_ACTIVE);
MQQueueManager qmgr=new MQQueueManager("my.qmgr.1", myConnMan);
: : (do something with qmgr)
: : qmgr.disconnect();
MQQueueManager qmgr2=new MQQueueManager("my.qmgr.1", myConnMan);
: : (do something with qmgr2)
: : qmgr2.disconnect();
myConnMan.setActive(MQSimpleConnectionManager.MODE_INACTIVE);
```
The connection that is forged during the first MQQueueManager constructor is stored in myConnMan after the qmgr.disconnect() call. The connection is then reused during the second call to the MQQueueManager constructor.

The second line enables the MQSimpleConnectionManager. The last line disables MQSimpleConnectionManager, destroying any connections held in the pool. An MQSimpleConnectionManager is, by default, in MODE_AUTO, which is described later in this section.

An MQSimpleConnectionManager allocates connections on a most-recently-used basis, and destroys connections on a least-recently-used basis. By default, a connection is destroyed if it has not been used for five minutes, or if there are more than ten unused connections in the pool. You can alter these values using:

- MQSimpleConnectionManager setTimeout()
- MQSimpleConnectionManager setHighThreshold()

It is also possible to set up an MQSimpleConnectionManager for use as the default connection pool, to be used when no Connection Manager is supplied on the MQQueueManager constructor.

The following application demonstrates this:

```java
import com.ibm.mq.*;
public class MQApp4
{
    public static void main(String[] args)
    {
        MQSimpleConnectionManager myConnMan = new MQSimpleConnectionManager();
        myConnMan.setActive(MQSimpleConnectionManager.MODE_AUTO);
        myConnMan.setTimeout(3600000);
        myConnMan.setHighThreshold(50);
        MQEnvironment.setDefaultConnectionManager(myConnMan);
        MQApp3.main(args);
    }
}
```

The bold lines set up an MQSimpleConnectionManager. This is set to:

- destroy connections that have not been used for an hour
- limit the number of unused connections held in the pool to 50
- MODE_AUTO (actually the default). This means that the pool is active only if it is the default connection manager, and there is at least one token in the set of MQPoolTokens held by MQEnvironment.

The new MQSimpleConnectionManager is then set as the default connection manager.

In the last line, the application calls MQApp3.main(). This runs a number of threads, where each thread uses WebSphere MQ independently. These threads will now use myConnMan when they forge connections.
Supplying your own ConnectionManager

Under Java 2 v1.3, with JAAS 1.0 installed, applications and middleware providers can provide alternative implementations of connection pools. WebSphere MQ base Java provides a partial implementation of the J2EE Connector Architecture. Implementations of `javax.resource.spi.ConnectionManager` can either be used as the default Connection Manager or be specified on the MQQueueManager constructor.

WebSphere MQ base Java complies with the Connection Management contract of the J2EE Connector Architecture. Please read this section in conjunction with the Connection Management contract of the J2EE Connector Architecture (refer to Sun's Web site at http://java.sun.com).

The ConnectionManager interface defines only one method:

```java
package javax.resource.spi;
public interface ConnectionManager {
    Object allocateConnection(ManagedConnectionFactory mcf,
                                ConnectionRequestInfo cxRequestInfo);
}
```

The MQQueueManager constructor calls `allocateConnection` on the appropriate ConnectionManager. It passes appropriate implementations of ManagedConnectionFactory and ConnectionRequestInfo as parameters to describe the connection required.

The ConnectionManager searches its pool for a `javax.resource.spi.ManagedConnection` object that has been created with identical ManagedConnectionFactory and ConnectionRequestInfo objects. If the ConnectionManager finds any suitable ManagedConnection objects, it creates a java.util.Set that contains the candidate ManagedConnections. Then, the ConnectionManager calls the following:

```java
ManagedConnection mc=mcf.matchManagedConnections(connectionSet, subject,
                                                   cxRequestInfo);
```

The WebSphere MQ implementation of ManagedConnectionFactory ignores the subject parameter. This method selects and returns a suitable ManagedConnection from the set, or returns null if it does not find a suitable ManagedConnection. If there is not a suitable ManagedConnection in the pool, the ConnectionManager can create one by using:

```java
ManagedConnection mc=mcf.createManagedConnection(subject, cxRequestInfo);
```

Again, the subject parameter is ignored. This method connects to a WebSphere MQ queue manager and returns an implementation of `javax.resource.spi.ManagedConnection` that represents the newly-forged connection. Once the ConnectionManager has obtained a ManagedConnection (either from the pool or freshly created), it creates a connection handle using:

```java
Object handle=mc.getConnection(subject, cxRequestInfo);
```

This connection handle can be returned from `allocateConnection()`.

A ConnectionManager should register an interest in the ManagedConnection through:

```java
mc.addConnectionEventListener()
```
The ConnectionEventListener is notified if a severe error occurs on the connection, or when MQQueueManager.disconnect() is called. When MQQueueManager.disconnect() is called, the ConnectionEventListener can do either of the following:

- reset the ManagedConnection using the mc.cleanup() call, then return the ManagedConnection to the pool
- destroy the ManagedConnection using the mc.destroy() call

If the ConnectionManager is intended to be the default ConnectionManager, it can also register an interest in the state of the MQEnvironment-managed set of MQPoolTokens. To do so, first construct an MQPoolServices object, then register an MQPoolServicesEventListener object with the MQPoolServices object:

```java
MQPoolServices mps=new MQPoolServices();
mps.addMQPoolServicesEventListener(listener);
```

The listener is notified when an MQPoolToken is added or removed from the set, or when the default ConnectionManager changes. The MQPoolServices object also provides a way to query the current size of the set of MQPoolTokens.

### JTA/JDBC coordination via WebSphere MQ base Java

WebSphere MQ base Java supports the MQQueueManager.begin() method, which allows WebSphere MQ to act as a coordinator for a database which provides a JDBC 2 compliant driver. Currently this support is available on Solaris, AIX and Windows systems with Oracle or DB/2 databases.

#### Installation

In order to use the XA-JTA support, you will need to use a special JTA switch library which is supplied. The method for using this library varies depending on whether you are using Windows systems or one of the other platforms.

**Installation on Windows systems**

On Windows systems, the new XA library is supplied as a complete DLL. The name of this DLL is jdbcxxx.dll where “xxx” indicates the database which the switch library has been compiled for. This library is in the `java/lib/jdbc` directory of your WebSphere MQ base Java installation.

**Installation on other platforms**

The switch file is supplied as an object file which you will need to link yourself using the supplied Makefile. This is required because certain libraries required by the switch library may be in different locations on different systems. Since the switch library is loaded by the queue manager which runs in a setuid environment, we cannot use the dynamic library path variable to locate these libraries. We therefore need to put the full path names to these libraries in the switch library itself.

The object files are called jdbcxxx.o where “xxx” indicates which database the object file is for. When linked, a switch file called jdbcxxx is produced, and this should be added to the qm.ini file in the same manner as the standard switch libraries.

To create the switch library, you need to go into the `java/lib/jdbc` subdirectory of your WebSphere MQ base Java installation and run make with your target database as a parameter. Currently supported targets for XA-JTA are “oracle” and “db2”. For example:
JTA/JDBC coordination

make db2

The Makefiles are set up to link against the databases and JDKs in their standard installed location. The exception to this is Oracle, which can be installed anywhere on the system and therefore the makefile makes use of Oracle’s ORACLE_HOME environment variable to link the library correctly. If your JDK is in a non-standard location, you can override the default directory with the JAVA_HOME definition:

```
make JAVA_HOME=/usr/my_jdk13 oracle
```

The above command should produce a switch file named “jdbcora”, which is used in the same way as a standard switch library, including using the same XAOpenString. Therefore, if you have previously configured an XAResourceManager in your qm.ini you should replace the “SwitchFile” line with a reference to the new JTA specific switch file. If you have not previously used an XA switch file then refer to [WebSphere MQ System Administration Guide] for the details of configuring the XAResourceManager stanza for different databases, remembering to replace the standard switch file with the Java specific one.

Once the qm.ini has been updated, the queue manager will need to be restarted. Ensure that all appropriate database environment variables have been set before calling strmqm.

Usage

The basic sequence of API calls for a user application is:

```
qMgr = new MQQueueManager("QM1")
Connection con = qMgr.getJDBCConnection( xads );
qMgr.begin()

< Perform MQ and DB operations to be grouped in a unit of work >

qMgr.commit() or qMgr.backout();
con.close()
qMgr.disconnect()
```

“xads” in the getJDBCConnection call is a database-specific implementation of the XADataSource interface which defines the details of the database to connect to. See the documentation for your database to determine how to create an appropriate XADataSource object to pass into getJDBCConnection.

You will also need to update your CLASSPATH with the appropriate database-specific jar files for performing JDBC work.

If you need to connect to multiple databases, you may call getJDBCConnection multiple times to perform the transaction across several different connections.

There are two forms of the getJDBCConnection, reflecting the two forms of XADataSource.getXAConnection:

```
public java.sql.Connection getJDBCConnection(javax.sql.XADataSource xads)
throws MQException, SQLException, Exception

public java.sql.Connection getJDBCConnection(XADataSource dataSource,
String userid, String password)
throws MQException, SQLException, Exception
```

These methods declare Exception in their throws clauses so as to avoid problems with the JVM verifier for customers who are not using the JTA functionality. The actual exception thrown is javax.transaction.xa.XAException which would require
the jta.jar file to be added to the classpath for programs which didn’t previously require it.

**Known problems and limitations**

Since this support makes calls to JDBC drivers, the implementation of those JDBC drivers can have significant impact on the system behavior. For example, the tested JDBC drivers showed divergent behavior when the database was shutdown whilst an application was running. This caused a variety of problems ranging from unexpected exceptions through crashing the JVM, and generation of reports in the WebSphere MQ error logs. However, in all cases transactional integrity is maintained. Always avoid abruptly shutting down a database while there are applications holding open connections to it.

**Oracle 8.1.7**

Calling the JDBC Connection.close() method after MQQueueManager.disconnect() generates a SQLException. Either call Connection.close() before MQQueueManager.disconnect(), or simply omit the call to Connection.close().

**DB2®**

Sometimes DB2 will return a SQL0805N error. This problem can be resolved with the following CLP command:

```
DB2 bind @db2cli.lst blocking all grant public
```

Please refer to the DB2 documentation for more information.

**Solaris and JDK 1.3**

When running on Solaris with JDK 1.3, attempting to start the queue manager from a userID other than root or mqm is likely to fail to load the jdbcora switchfile with the following message being put to the error log:

```
AMQ6175: The system could not dynamically load the library /opt/mqm/java/lib/jdbcora.
The error message was ld.so.1: amqzxma0: fatal; libverify.so: open failed:
No such file or directory. The Queue Manager will continue without this module.
```

The problem arises because of a dependency between two libraries in JDK1.3 which cannot be resolved by the dynamic linker when the invoking program has the setuid bit set (as strmqm does).

Under these circumstances, you should start the queue manager from the mqm userID; if this is not practical, an alternative is to make a symbolic link for libverify into /usr/lib.

```
ln -s /usr/j2se/jre/lib/sparc/libverify.so /usr/lib/libverify.so
```

**Solaris and multiple XAResourceManager stanzas**

When attempting to use multiple XAResourceManager stanzas on any given queue manager on Solaris, you may find the commit call will fail. This should be treated as an unsupported combination, and will not affect queue managers with a single XAResourceManager stanza.

**Windows NT**

The JDBC libraries supplied with WebSphere MQ Java (jdbcdb2.dll and jdbcora.dll) have a dependency on jvm.dll, which is supplied with the JVM. However, depending on the JVM used, this dll may be in a subdirectory that is not on the path; for example, jre/bin/classic/jvm.dll.

If jvm.dll can not be found when the queue manager starts, the queue manager produces a message like the following (this example is for DB2):
JTA/JDBC coordination

AMQ6174: The library C:\Program Files\IBM\MQSeries\Java\lib\jdbc\jdbcdb2.dll was not found. The queue manager will continue without this module.

In fact, the file not found is jvm.dll. The solution is to either copy jvm.dll to somewhere already on the path or update the path to include the location of jvm.dll.

Secure Sockets Layer (SSL) support

From WebSphere MQ V5.3, WebSphere MQ base Java clients and WebSphere MQ JMS connections using TRANSPORT(CLIENT) support Secure Sockets Layer (SSL) encryption. SSL provides communication encryption, authentication and message integrity. It is typically used to secure communications between any two peers on the public internet or within an intranet.

WebSphere MQ classes for Java uses Java Secure Socket Extension (JSSE) to handle SSL encryption, and so requires a JSSE provider. J2SE v1.4 JVMs have a JSSE provider built-in. Details of how to manage and store certificates can vary from provider to provider. For information about this, refer to your JSSE provider’s documentation.

This section assumes your JSSE provider is correctly installed and configured, and suitable certificates have been installed and made available to your JSSE provider.

Enabling SSL

SSL is supported only for client connections. To enable SSL, you must specify the CipherSuite to use when communicating with the queue manager, and this must match the CipherSpec set on the target channel. Additionally, the named CipherSuite must be supported by your JSSE provider. However, CipherSuites are distinct from CipherSpecs and so have different names. Appendix H, “SSL CipherSuites supported by WebSphere MQ” on page 425 contains a table mapping the CipherSpecs supported by WebSphere MQ to their equivalent CipherSuites as known to JSSE.

To enable SSL, specify the CipherSuite using the sslCipherSuite static member variable of MQEnvironment. The following example attaches to a SVRCONN channel named SECURE.SVRCONN.CHANNEL which has been set up to require SSL with a CipherSpec of RC4_MD5_EXPORT:

```java
MQEnvironment.hostname = "your_hostname";
MQEnvironment.channel = "SECURE.SVRCONN.CHANNEL";
MQEnvironment.sslCipherSuite = "SSL_RSA_EXPORT_WITH_RC4_40_MD5";
MQQueueManager qmgr = new MQQueueManager("your_Q_manager");
```

Note that while the channel has a CipherSpec of RC4_MD5_EXPORT, the Java application must specify a CipherSuite of SSL_RSA_EXPORT_WITH_RC4_40_MD5. For more information about CipherSpecs and CipherSuites, see the WebSphere MQ Security book. See Appendix H, “SSL CipherSuites supported by WebSphere MQ” on page 425 for a list of mappings between CipherSpecs and CipherSuites.

The sslCipherSuite property can also be set using the MQC.SSL_CIPHER_SUITE_PROPERTY in the hashtable of connection properties.

To successfully connect using SSL, the JSSE TrustStore must be set up with Certificate Authority root certificates from which the certificate presented by the queue manager can be authenticated. Similarly, if SSLClientAuth on the SVRCONN
channel has been set to MQSSL_CLIENT_AUTH_REQUIRED then the JSSE
KeyStore must contain an identifying certificate which is trusted by the queue
manager.

Using the distinguished name of the queue manager

The queue manager identifies itself using an SSL certificate, which contains a
Distinguished Name (DN). A WebSphere MQ Java client application can use this DN
to ensure that it is communicating with the correct queue manager. A DN pattern
is specified using the sslPeerName variable of MQEnvironment. For example,
setting:

MQEnvironment.sslPeerName = "CN=QMGR.*, OU=IBM, OU=WEBSPHERE";

only allows the connection to succeed if the queue manager presents a certificate
with a Common Name beginning “QMGR.”, and at least two Organizational Unit
names, the first of which must be “IBM” and the second “WEBSPHERE”.

The sslPeerName property can also be set using the
MQC.SSL_PEER_NAME_PROPERTY in the hashtable of connection properties. For
more information about distinguished names, please refer to the
WebSphere MQ Security
book.

If sslPeerName is set, connections will only succeed if it is set to a valid pattern
and the queue manager presents a matching certificate.

Using certificate revocation lists

A certificate revocation list (CRL) is a set of certificates that have been revoked,
either by the issuing Certificate Authority or by the local organization. CRLs are
typically hosted on LDAP servers. With Java 2 v1.4, a CRL server can be specified
at connect-time and the certificate presented by the queue manager will be checked
against the CRL before the connection is allowed. For more information about
certificate revocation lists and WebSphere MQ, see the
WebSphere MQ Security
book.

Note: To use a CertStore successfully with a CRL hosted on an LDAP server, make
sure that your Java Software Development Kit (SDK) is compatible with the
CRL. Some SDKs require that the CRL conforms to RFC 2587 which defines
a schema for LDAP v2. Most LDAP v3 servers use RFC 2256 instead.

The CRLs to use are specified through the java.security.cert.CertStore class. Refer to
documentation on this class for full details of how to obtain instances of CertStore.
To create a CertStore based on an LDAP server, you should first create an
LDAPCertStoreParameters instance, initialized with the server and port settings to
use. For example:

import java.security.cert.*;
CertStoreParameters csp = new LDAPCertStoreParameters("crl_server", 389);

Having created a CertStoreParameters instance, use the static constructor on
CertStore to create a CertStore of type “LDAP”:
CertStore cs = CertStore.getInstance("LDAP", csp);

Other CertStore types (for example, Collection) are also supported. Commonly
there are several CRL servers set up with identical CRL information to give
redundancy. Once you have a CertStore object for each of these CRL servers, you
should place them all in a suitable Collection. The following example shows the
CertStore objects placed in an ArrayList:
SSL support

```java
import java.util.ArrayList;
Collection crls = new ArrayList();
crls.add(cs);
```

This Collection can be set into the MQEnvironment static variable, sslCertStores, before connecting to enable CRL checking:

```java
MQEnvironment.sslCertStores = crls;
```

The certificate presented by the queue manager when a connection is being set up is validated as follows:

1. The first CertStore object in the Collection identified by sslCertStores is used to identify a CRL server.
2. An attempt is made to contact the CRL server.
3. If the attempt is successful, the server is searched for a match for the certificate.
   a. If the certificate is found to be revoked, the search process is over and the connection request will fail with reason code MQRC_SSL_CERTIFICATE_REVOKED.
   b. If the certificate is not found, the search process is over and the connection is allowed to proceed.
4. If the attempt to contact the server is unsuccessful, the next CertStore object is used to identify a CRL server and the process repeats from “An attempt is made to contact the CRL server”.

If this was the last CertStore in the Collection, or if the Collection contains no CertStore objects, the search process has failed and the connection request will fail with reason code MQRC_SSL_CERT_STORE_ERROR.

The Collection object determines the order in which CertStores are used.

The Collection of CertStores can also be set using the MQC.SSL_CERT_STORE_PROPERTY. As a convenience, this property also allows a single CertStore to be specified without it needing to be a member of a Collection.

If sslCertStores is set to null, no CRL checking will be performed. This property is ignored if sslCipherSuite is not set.

**Supplying a customized SSLSocketFactory**

Different JSSE implementations can provide different features. For example, a specialised JSSE implementation could allow configuration of a particular model of encryption hardware. Additionally, some JSSE providers allow customization of KeyStores and TrustStores by program, or allow the choice of identity certificate from the KeyStore to be altered. In JSSE, all these customizations are abstracted into a factory class, javax.net.ssl.SSLSocketFactory.

Refer to your JSSE documentation for details of how to create a customized SSLSocketFactory implementation. The details vary from provider to provider, but a typical sequence of steps might be:

1. Create an SSLContext object using a static method on SSLContext
2. Initialize this SSLContext with appropriate KeyManagers and TrustManager implementations (created from their own factory classes)
3. Create an SSLSocketFactory from the SSLContext

When you have an SSLSocketFactory object, set the MQEnvironment.sslSocketFactory to the customized factory object. For example:
javax.net.ssl.SSLSocketFactory sf = sslContext.getSocketFactory();
MQEnvironment.sslSocketFactory = sf;

WebSphere MQ classes for Java will then use this SSLSocketFactory to connect to the WebSphere MQ queue manager. This property can also be set using the MQC.SSL_SOCKET_FACTORY_PROPERTY. If sslSocketFactory is set to null, the JVM’s default SSLSocketFactory will be used. This property is ignored if sslCipherSuite is not set.

### Error handling when using SSL

The following reason codes may be issued by WebSphere MQ classes for Java when connecting to a queue manager using SSL:

- **MQRC_SSL_NOT_ALLOWED**
  The sslCipherSuite property was set, but bindings connect was used. Only client connect supports SSL.

- **MQRC_JSSE_ERROR**
  The JSSE provider reported an error which could not be handled by WebSphere MQ. This could be caused by a configuration problem with JSSE, or because the certificate presented by the queue manager could not be validated. The exception produced by JSSE can be retrieved using the getCauser() method on MQException.

- **MQRC_SSL_PEER_NAME_MISMATCH**
  The DN pattern specified in the sslPeerName property did not match the DN presented by the queue manager.

- **MQRC_SSL_PEER_NAME_ERROR**
  The DN pattern specified in the sslPeerName property was not valid.

- **MQRC_UNSUPPORTED_CIPHER_SUITE**
  The CipherSuite named in sslCipherSuite was not recognised by the JSSE provider. A full list of CipherSuites supported by the JSSE provider can be obtained by a program using the SSLSocketFactory.getSupportedCipherSuites() method. A list of CipherSuites which can be used to communicate with WebSphere MQ can be found in Appendix H, “SSL CipherSuites supported by WebSphere MQ” on page 425.

- **MQRC_SSL_CERTIFICATE_REVOKED**
  The certificate presented by the queue manager was found in a CRL specified with the sslCertStores property. Update the queue manager to use trusted certificates.

- **MQRC_SSL_CERT_STORE_ERROR**
  None of the supplied CertStores could be searched for the certificate presented by the queue manager. The MQException.getCause() method will return the error which occurred while searching the first CertStore attempted. If the causal exception is NoSuchElementException, ClassCastException or NullPointerException, check that the Collection specified on the sslCertStores property contains at least one valid CertStore object.
Compiling and testing WebSphere MQ base Java programs

Before compiling WebSphere MQ base Java programs, you must ensure that your WebSphere MQ classes for Java installation directory is in your CLASSPATH environment variable, as described in Chapter 2, “Installation” on page 7.

To compile a class “MyClass.java”, use the command:
```
javac MyClass.java
```

Running WebSphere MQ base Java applets

If you write an applet (subclass of java.applet.Applet), you must create an HTML file referencing your class before you can run it. A sample HTML file might look as follows:
```html
<html>
<body>
<applet code="MyClass.class" width=200 height=400>
</applet>
</body>
</html>
```

Run your applet either by loading this HTML file into a Java enabled Web browser, or by using the appletviewer that comes with the Java Development Kit (JDK).

To use the applet viewer, enter the command:
```
appletviewer myclass.html
```

Running WebSphere MQ base Java applications

If you write an application (a class that contains a main() method), using either the client or the bindings mode, run your program using the Java interpreter. Use the command:
```
java MyClass
```

Note: The “.class” extension is omitted from the class name.

Tracing WebSphere MQ base Java programs

WebSphere MQ base Java includes a trace facility, which you can use to produce diagnostic messages if you suspect that there might be a problem with the code. (You will normally need to use this facility only at the request of IBM service.)

Tracing is controlled by the enableTracing and disableTracing methods of the MQEnvironment class. For example:
```java
MQEnvironment.enableTracing(2); // trace at level 2
...
// these commands will be traced
MQEnvironment.disableTracing(); // turn tracing off again
```

The trace is written to the Java console (System.err).

If your program is an application, or if you run it from your local disk using the appletviewer command, you can also redirect the trace output to a file of your choice. The following code fragment shows an example of how to redirect the trace output to a file called myapp.trc:
```java
import java.io.*;

try {
    FileOutputStream traceFile = new FileOutputStream("myapp.trc");
    // do more...
    traceFile.close();
}
```

86 WebSphere MQ Using Java
traceFile = new FileOutputStream("myapp.trc");
MQEnvironment.enableTracing(2,traceFile);
}
} catch (IOException ex) {
    // couldn't open the file,
    // trace to System.err instead
    MQEnvironment.enableTracing(2);
}

There are five different levels of tracing:
1. Provides entry, exit, and exception tracing
2. Provides parameter information in addition to 1
3. Provides transmitted and received WebSphere MQ headers and data blocks in addition to 2
4. Provides transmitted and received user message data in addition to 3
5. Provides tracing of methods in the Java Virtual Machine in addition to 4

To trace methods in the Java Virtual Machine with trace level 5:
• For an application, run it by issuing the command java_g (instead of java)
• For an applet, run it by issuing the command appletviewer_g (instead of appletviewer)

Note: java_g is not supported on OS/400, but similar function is provided by using OPTION(*VERBOSE) on the RUNJVA command.
Chapter 8. Environment-dependent behavior

The WebSphere MQ classes for Java allow you to create applications that can run against different versions of WebSphere MQ and MQSeries®. This chapter describes the behavior of the Java classes dependent on these different versions.

WebSphere MQ classes for Java provides a “core” of classes, which provide consistent function and behavior in all the environments. Features outside this core will be dependent on the capability of the queue manager to which the application is connected.

Except where noted here, the behavior exhibited will be as described in the Application Programming Reference book appropriate to the queue manager.

Core details

WebSphere MQ classes for Java contains the following core set of classes, which can be used in all environments with only the minor variations listed in “Restrictions and variations for core classes” on page 90:

- MQEnvironment
- MQException
- MQGetMessageOptions
  Excluding:
  - MatchOptions
  - GroupStatus
  - SegmentStatus
  - Segmentation
- MQManagedObject
  Excluding:
  - inquire()
  - set()
- MQMessage
  Excluding:
  - groupId
  - messageFlags
  - messageSequenceNumber
  - offset
  - originalLength
- MQPoolServices
- MQPoolServicesEvent
- MQPoolServicesEventListener
- MQPoolToken
- MQPutMessageOptions
  Excluding:
  - knownDestCount
  - unknownDestCount
  - invalidDestCount
  - recordFields
- MQProcess
- MQQueue
- MQQueueManager
Core details

Excluding:
- begin()
- accessDistributionList()
- MQSimpleConnectionManager
- MQC

Notes:
1. Some constants are not included in the core (see "Restrictions and variations for core classes" for details), and you should not use them in completely portable programs.
2. Some platforms do not support all connection modes. On these platforms, you can use only the core classes and options that relate to the supported modes. (See Table 1 on page 5.)

Restrictions and variations for core classes

The core classes generally behave consistently across all environments, even if the equivalent MQI calls normally have environment differences. Normally the behavior is as if a Windows or UNIX WebSphere MQ V5.3 queue manager is used, except for the following minor restrictions and variations.

MQGMO_* values
The following MQGMO_* values are not supported by all queue managers, and their use may result in MQException being thrown from a MQQueue.get():
- MQGMO_SYNCPOINT_IF_PERSISTENT
- MQGMO_MARK_SKIP_BACKOUT
- MQGMO_BROWSE_MSG_UNDER_CURSOR
- MQGMO_LOCK
- MQGMO_UNLOCK
- MQGMO_LOGICAL_ORDER
- MQGMO_COMPLETE_MESSAGE
- MQGMO_ALL_MSGS_AVAILABLE
- MQGMO_ALL_SEGMENTS_AVAILABLE

Additionally, MQGMO_SET_SIGNAL is not supported when used from Java.

MQPMO_* values
The following MQPMO_* values are not supported by all queue managers, and their use may result in MQException being thrown from a MQQueue.put() or a MQQueueManager.put():
- MQPMO_NEW_MESSAGE_ID
- MQPMO_NEW_CORREL_ID
- MQPMO_LOGICAL_ORDER

MQPMRF_* values
These are only used when putting messages to a distribution list, and so are only supported by queue managers supporting distribution lists. For example, z/OS & OS/390 queue managers do not support distribution lists.

MQCNO_FASTPATH_BINDING
This value is ignored on queue managers which do not support it, or when using a TCP/IP client connection.
MQRO_* values

The following report options can be set but are ignored by some queue managers. This can affect applications connected to a queue manager which does honor the report options when the report message should be generated by a remote queue manager that does not. Avoid relying on these options if there is a possibility that a queue manager involved does not support them.

- MQRO_EXCEPTION_WITH_FULL_DATA
- MQRO_EXPIRATION_WITH_FULL_DATA
- MQRO_COA_WITH_FULL_DATA
- MQRO_COD_WITH_FULL_DATA
- MQRO_DISCARD_MSG

Miscellaneous differences with z/OS & OS/390

Message priority

When a message is put with a priority greater than MaxPriority, a z/OS & OS/390 queue manager will reject the put with MQCC_FAILED and MQRC_PRIORITY_ERROR. Other platforms complete the put with MQCC_WARNING and MQRC_PRIORITY_EXCEEDS_MAXIMUM, and treat the message as if it were put with MaxPriority.

BackoutCount

A z/OS & OS/390 queue manager will return a maximum BackoutCount of 255, even if the message has been backed out more than 255 times.

Default dynamic queue prefix

When connected to a z/OS & OS/390 queue manager using a bindings connection, the default dynamic queue prefix is CSQ.*. Otherwise, the default dynamic queue prefix is AMQ.*.

MQQueueManager constructor

Client connect is not supported on z/OS & OS/390. Attempting to connect with client options will result in an MQException with MQCC_FAILED and MQRC_ENVIRONMENT_ERROR. The MQQueueManager constructor may also fail with MQRC_CHAR_CONVERSION_ERROR if it fails to initialize conversion between the IBM-1047 and ISO8859-1 code pages, or MQRC_UCS2_CONVERSION_ERROR if it fails to initialize conversion between the queue manager’s code page and Unicode. If your application fails with one of these reason codes, ensure that the National Language Resources component of Language Environment is installed, and ensure that the correct conversion tables are available.

Conversion tables for Unicode are installed as part of the OS/390 C/C++ optional feature. See the OS/390: C/C++ Programming Guide (SC09-2362) for more information about enabling UCS-2 conversions.
Features outside the core

The WebSphere MQ classes for Java contain the following functions that are specifically designed to use API extensions that are not supported by all queue managers. This section describes how they behave when using a queue manager which does not support them.

MQQueueManager constructor option

The MQQueueManager constructor includes an optional integer argument. This maps onto the MQI’s MQCNO options field, and is used to switch between normal and fastpath connection. This extended form of the constructor is accepted in all environments, provided that the only options used are MQCNO_STANDARD_BINDING or MQCNO_FASTPATH_BINDING. Any other options cause the constructor to fail with MQRC_OPTIONS_ERROR. The fastpath option MQC.MQCNO_FASTPATH_BINDING is only honored when with a bindings connection to a queue manager which supports it. In other environments, it is ignored.

MQQueueManager.begin() method

This can only be used against a WebSphere MQ V5.3 queue manager on UNIX or Windows systems in bindings mode. Otherwise, it will fail with MQRC_ENVIRONMENT_ERROR. See “JTA/JDBC coordination via WebSphere MQ base Java” on page 79 for more details.

MQGetMessageOptions fields

When using a queue manager which does not support the Version 2 MQGMO structure, you must leave the following fields set to their default values:
- GroupStatus
- SegmentStatus
- Segmentation

Also, the MatchOptions field will only support MQMO_MATCH_MSG_ID and MQMO_MATCH_CORREL_ID. If you put unsupported values into these fields, the subsequent MQQueue.get() will fail with MQRC_GMO_ERROR. If the queue manager does not support the Version 2 MQGMO structure, these fields are not updated after a successful MQQueue.get().

Distribution lists

The following classes are used to create distribution lists:
- MQDistributionList
- MQDistributionListItem
- MQMessageTracker

You can create and populate MQDistributionLists and MQDistributionListItems in any environment, but not all queue managers will allow you to open an MQDistributionList. In particular, z/OS & OS/390 queue managers do not support distribution lists. Attempting to open an MQDistributionList when using such a queue manager will result in MQRC_OD_ERROR.
MQPutMessageOptions fields

Four fields in the MQPMO are rendered as the following member variables in the MQPutMessageOptions class:

- knownDestCount
- unknownDestCount
- invalidDestCount
- recordFields

These fields are primarily intended for use with distribution lists. However, a queue manager which supports distribution lists will also fill in the DestCount fields after an MQPUT to a single queue. For example, if the queue resolves to a local queue, knownDestCount is set to 1 and the other two count fields are set to 0.

If the queue manager does not support distribution lists, these values are simulated as follows:
- If the put() succeeds, unknownDestCount is set to 1, and the others are set to 0.
- If the put() fails, invalidDestCount is set to 1, and the others are set to 0.

The recordFields variable is used with distribution lists. A value may be written into recordFields at any time, regardless of the environment. It is ignored if the MQPutMessageOptions object is used on a subsequent MQQueue.put() or MQQueueManager.put(), rather than MQDistributionList.put().

MQMD fields

The following MQMD fields are largely concerned with message segmentation:

- GroupId
- MsgSeqNumber
- Offset
- MsgFlags
- OriginalLength

If an application sets any of these MQMD fields to values other than their defaults, and then does a put() or get() on a queue manager which does not support these, the put() or get() will raise an MQException with MQRC_MD_ERROR. A successful put() or get() with such a queue manager will always leave the MQMD fields set to their default values. A grouped or segmented message should not normally be sent to a Java application that runs against a queue manager that does not support message grouping and segmentation.

If a Java application attempts to get() a message from a queue manager which does not support these fields, and the physical message to be retrieved is part of a group of segmented message (that is, it has non-default values for the MQMD fields), it is retrieved without error. However, the MQMD fields in the MQMessage are not updated, the MQMessage format property is set to MQFMT_MD_EXTENSION, and the true message data is prefixed with an MQMDE structure that contains the values for the new fields.
Chapter 9. The WebSphere MQ base Java classes and interfaces

This chapter describes all the WebSphere MQ classes for Java classes and interfaces. It includes details of the variables, constructors, and methods in each class and interface.

The following classes are described:
- MQChannelDefinition
- MQChannelExit
- MQDistributionList
- MQDistributionListItem
- MQEnvironment
- MQException
- MQGetMessageOptions
- MQManagedObject
- MQMessage
- MQMessageTracker
- MQPoolServices
- MQPoolServicesEvent
- MQPoolToken
- MQPutMessageOptions
- MQProcess
- MQQueue
- MQQueueManager
- MQSimpleConnectionManager

The following interfaces are described:
- MQC
- MQPoolServicesEventListerner
- MQConnectionManager
- MQReceiveExit
- MQSecurityExit
- MQSendExit
- ManagedConnection
- ManagedConnectionFactory
- ManagedConnectionMetaData
public class MQChannelDefinition
extends Object

The MQChannelDefinition class is used to pass information concerning the connection to the queue manager to the send, receive and security exits.

Note: This class does not apply when connecting directly to WebSphere MQ in bindings mode.

Variables

channelName
public String channelName
The name of the channel through which the connection is established.

collectionName
public String connectionName
The TCP/IP hostname of the machine on which the queue manager resides.

maxMessageLength
public int maxMessageLength
The maximum length of message that can be sent to the queue manager.

queueManagerName
public String queueManagerName
The name of the queue manager to which the connection is made.

receiveUserData
public String receiveUserData
A storage area for the receive exit to use. Information placed here is preserved across invocations of the receive exit, and is also available to the send and security exits.

remotePassword
public String remotePassword
The password used to establish the connection.

remoteUserId
public String remoteUserId
The user id used to establish the connection.

securityUserData
public String securityUserData
A storage area for the security exit to use. Information placed here is preserved across invocations of the security exit, and is also available to the send and receive exits.

sendUserData
public String sendUserData
A storage area for the send exit to use. Information placed here is preserved across invocations of the send exit, and is also available to the security and receive exits.

```java
sslPeerName
public String sslPeerName
If SSL is used to encrypt data on the wire, this is set to the Distinguished Name presented by the queue manager during connection. If SSL is not used, it is left at null.
```

### Constructors

```java
MQChannelDefinition
public MQChannelDefinition()
```
public class MQChannelExit
extends Object

This class defines context information passed to the send, receive, and security exits when they are invoked. The exitResponse member variable should be set by the exit to indicate what action the WebSphere MQ Client for Java should take next.

Note: This class does not apply when connecting directly to WebSphere MQ in bindings mode.

Variables

MQXCC_CLOSE_CHANNEL
   public final static int MQXCC_CLOSE_CHANNEL

MQXCC_OK
   public final static int MQXCC_OK

MQXCC_SUPPRESS_FUNCTION
   public final static int MQXCC_SUPPRESS_FUNCTION

MQXCC_SEND_AND_REQUEST_SEC_MSG
   public final static int MQXCC_SEND_AND_REQUEST_SEC_MSG

MQXCC_SEND_SEC_MSG
   public final static int MQXCC_SEND_SEC_MSG

MQXCC_SUPPRESS_EXIT
   public final static int MQXCC_SUPPRESS_EXIT

MQXR_INIT
   public final static int MQXR_INIT

MQXR_INIT_SEC
   public final static int MQXR_INIT_SEC

MQXR_SEC_MSG
   public final static int MQXR_SEC_MSG

MQXR_TERM
   public final static int MQXR_TERM

MQXR_XMIT
   public final static int MQXR_XMIT

MQXT_CHANNEL_SEC_EXIT
   public final static int MQXT_CHANNEL_SEC_EXIT

MQXT_CHANNEL_SEND_EXIT
   public final static int MQXT_CHANNEL_SEND_EXIT

MQXT_CHANNEL_RCV_EXIT
   public final static int MQXT_CHANNEL_RCV_EXIT

capabilityFlags
   public static final int capabilityFlags
MQChannelExit

Indicates the capability of the queue manager.

Only the MQC.MQCF_DIST_LISTS flag is supported.

**exitID**

public int exitID

The type of exit that has been invoked. For an MQSecurityExit this is always MQXT_CHANNEL_SEC_EXIT. For an MQSendExit this is always MQXT_CHANNEL_SEND_EXIT, and for an MQReceiveExit this is always MQXT_CHANNEL_RCV_EXIT.

**exitReason**

public int exitReason

The reason for invoking the exit. Possible values are:

- **MQXR_INIT**
  
  Exit initialization; called after the channel connection conditions have been negotiated, but before any security flows have been sent.

- **MQXR_INIT_SEC**
  
  Indicates that the exit is to initiate the security dialog with the queue manager.

- **MQXR_SEC_MSG**
  
  Indicates to the security exit that a security message has been received from the queue manager.

- **MQXR_TERM**
  
  Exit termination; called after the disconnect flows have been sent but before the socket connection is destroyed.

- **MQXR_XMIT**
  
  For a send exit, indicates that data is to be transmitted to the queue manager.

  For a receive exit, indicates that data has been received from the queue manager.

**exitResponse**

public int exitResponse

Set by the exit to indicate the action that WebSphere MQ classes for Java should take next. Valid values are:

- **MQXCC_CLOSE_CHANNEL**
  
  Set by any exit to indicate that the connection to the queue manager should be closed.

- **MQXCC_OK**
  
  Set by the security exit to indicate that security exchanges are complete.

  Set by send exit to indicate that the returned data is to be transmitted to the queue manager.

  Set by the receive exit to indicate that the returned data is available for processing by the WebSphere MQ Client for Java.

- **MQXCC_SEND_AND_REQUEST_SEC_MSG**
  
  Set by the security exit to indicate that the returned data is to be transmitted to the queue manager, and that a response is expected from the queue manager.
MQChannelExit

**MQXCC_SEND_SEC_MSG**
Set by the security exit to indicate that the returned data is to be transmitted to the queue manager, and that no response is expected.

**MQXCC_SUPPRESS_EXIT**
Set by any exit to indicate that it should no longer be called.

**MQXCC_SUPRESS_FUNCTION**
Set by the security exit to indicate that communications with the queue manager should be shut down.

**exitUserArea**
```
public byte exitUserArea[]
```
A storage area available for the exit to use.

Any data placed in the exitUserArea is preserved by the WebSphere MQ Client for Java across exit invocations with the same exitID. (That is, the send, receive, and security exits each have their own, independent, user areas.)

**fapLevel**
```
public static final int fapLevel
```
The negotiated Format and Protocol (FAP) level.

**maxSegmentLength**
```
public int maxSegmentLength
```
The maximum length for any one transmission to a queue manager.

If the exit returns data that is to be sent to the queue manager, the length of the returned data should not exceed this value.

**Constructors**

**MQChannelExit**
```
public MQChannelExit()
```
MQDistributionList

```
java.lang.Object
   └─ com.ibm.mq.MQManagedObject
       └─ com.ibm.mq.MQDistributionList
```

public class MQDistributionList extends MQManagedObject (See page 116)

Note: You can use this class only when connected to a WebSphere MQ Version 5 (or higher) queue manager.

An MQDistributionList is created by using the MQDistributionList constructor, or by using the accessDistributionList method for MQQueueManager.

A distribution list represents a set of open queues to which messages can be sent using a single call to the put() method. (See "Distribution lists" in the WebSphere MQ Application Programming Guide)

Constructors

MQDistributionList

```
public MQDistributionList(MQQueueManager qMgr,
MQDistributionListItem[] litems,
int openOptions,
String alternateUserId)
    throws MQException
```

qMgr is the queue manager where the list is to be opened.

litems are the items to be included in the distribution list.

See "accessDistributionList" on page 159 for details of the remaining parameters.

Methods

getFirstDistributionListItem

```
public MQDistributionListItem getFirstDistributionListItem()
```

Returns the first item in the distribution list, or null if the list is empty.

getInvalidDestinationCount

```
public int getInvalidDestinationCount()
```

Returns the number of items in the distribution list that failed to open successfully.

getValidDestinationCount

```
public int getValidDestinationCount()
```

Returns the number of items in the distribution list that were opened successfully.

put

```
public synchronized void put(MQMessage message,
    MQPutMessageOptions putMessageOptions )
    throws MQException
```
MQDistributionList

Puts a message to the queues on the distribution list.

Parameters

message
An input/output parameter containing the message descriptor information and the returned message data.

putMessageOptions
Options that control the action of MQPUT. (See “MQPutMessageOptions” on page 145 for details.)

Throws MQException if the put fails.
public class MQDistributionListItem extends MQMessageTracker (See page 137).

Note: You can use this class only when connected to an WebSphere MQ Version 5 (or higher) queue manager.

An MQDistributionListItem represents a single item (queue) within a distribution list.

Variables

completionCode
public int completionCode
The completion code resulting from the last operation on this item. If this was the construction of an MQDistributionList, the completion code relates to the opening of the queue. If it was a put operation, the completion code relates to the attempt to put a message onto this queue.

The initial value is "0".

queueManagerName
public String queueManagerName
The name of the queue manager on which the queue is defined.

The initial value is "".

queueName
public String queueName
The name of a queue you want to use with a distribution list. This cannot be the name of a model queue.

The initial value is "".

reasonCode
public int reasonCode
The reason code resulting from the last operation on this item. If this was the construction of an MQDistributionList, the reason code relates to the opening of the queue. If it was a put operation, the reason code relates to the attempt to put a message onto this queue.

The initial value is "0".

Constructors

MQDistributionListItem
public MQDistributionListItem()
Construct a new MQDistributionListItem object.
**MQEnvironment**

MQEnvironment

```java
java.lang.Object
  └─ com.ibm.mq.MQEnvironment

public class MQEnvironment
  extends Object

Note: All the methods and attributes of this class apply to the WebSphere MQ classes for Java client connections, but only enableTracing, disableTracing, properties, and version_notice apply to bindings connections.

MQEnvironment contains static member variables that control the environment in which an MQQueueManager object (and its corresponding connection to WebSphere MQ) is constructed.

Values set in the MQEnvironment class take effect when the MQQueueManager constructor is called, so you should set the values in the MQEnvironment class before you construct an MQQueueManager instance.

Variables

Note: Variables marked with * do not apply when connecting directly to WebSphere MQ in bindings mode.

**CCSID***

```java
public static int CCSID
```

The CCSID used by the client.

Changing this value affects the way that the queue manager you connect to translates information in the WebSphere MQ headers. All data in WebSphere MQ headers is drawn from the invariant part of the ASCII codeset, except for the data in the applicationIdData and the putApplicationName fields of the MQMessage class. (See "MQMessage" on page 119.)

If you avoid using characters from the variant part of the ASCII codeset for these two fields, you are then safe to change the CCSID from 819 to any other ASCII codeset.

If you change the client's CCSID to be the same as that of the queue manager to which you are connecting, you gain a performance benefit at the queue manager because it does not attempt to translate the message headers.

The default value is 819.

**channel***

```java
public static String channel
```

The name of the channel to connect to on the target queue manager. You must set this member variable, or the corresponding property, before constructing an MQQueueManager instance for use in client mode.
hostname*

public static String hostname

The TCP/IP hostname of the machine on which the WebSphere MQ server resides. If the hostname is not set, and no overriding properties are set, bindings mode is used to connect to the local queue manager.

password*

public static String password

Equivalent to the WebSphere MQ environment variable MQ_PASSWORD. If a security exit is not defined for this client, the value of password is transmitted to the server and is available to the server security exit when it is invoked. The value may be used to verify the identity of the WebSphere MQ client.

The default value is "".

port*

public static int port

The port to connect to. This is the port on which the WebSphere MQ server is listening for incoming connection requests. The default value is 1414.

properties

public static java.util.Hashtable properties

A set of key/value pairs defining the WebSphere MQ environment.

This hash table allows you to set environment properties as key/value pairs rather than as individual variables.

The properties can also be passed as a hash table in a parameter on the MQQueueManager constructor. Properties passed on the constructor take precedence over values set with this properties variable, but they are otherwise interchangeable. The order of precedence of finding properties is:

1. properties parameter on MQQueueManager constructor
2. MQEnvironment.properties
3. Other MQEnvironment variables
4. Constant default values

The possible Key/value pairs are shown in the following table:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQC.CCSID_PROPERTY</td>
<td>Integer (overrides MQEnvironment.CCSID)</td>
</tr>
<tr>
<td>MQC.CHANNEL_PROPERTY</td>
<td>String (overrides MQEnvironment.channel)</td>
</tr>
<tr>
<td>MQC.CONNECT_OPTIONS_PROPERTY</td>
<td>Integer, defaults to MQC.MQCNO_NONE</td>
</tr>
<tr>
<td>MQC.HOST_NAME_PROPERTY</td>
<td>String (overrides MQEnvironment.hostname)</td>
</tr>
<tr>
<td>MQC.ORB_PROPERTY</td>
<td>org.omg.CORBA.ORB (optional)</td>
</tr>
<tr>
<td>MQC.PASSWORD_PROPERTY</td>
<td>String (overrides MQEnvironment.password)</td>
</tr>
<tr>
<td>MQC.PORT_PROPERTY</td>
<td>Integer (overrides MQEnvironment.port)</td>
</tr>
<tr>
<td>MQC.RECEIVE_EXIT_PROPERTY</td>
<td>MQReceiveExit (overrides MQEnvironment.receiveExit)</td>
</tr>
<tr>
<td>MQC.SECURITY_EXIT_PROPERTY</td>
<td>MQSecurityExit (overrides MQEnvironment.securityExit)</td>
</tr>
<tr>
<td>MQC.SEND_EXIT_PROPERTY</td>
<td>MQSendExit (overrides MQEnvironment.sendExit)</td>
</tr>
</tbody>
</table>
### MQEnvironment

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQC.SSL_CERT_STORE_PROPERTY</td>
<td>java.util.Collection, or java.security.cert.CertStore (overrides MQEnvironment.sslCertStores)</td>
</tr>
<tr>
<td>MQC.SSL_CIPHER_SUITE_PROPERTY</td>
<td>String (overrides MQEnvironment.sslCipherSuite)</td>
</tr>
<tr>
<td>MQC.SSL_PEER_NAME_PROPERTY</td>
<td>String (overrides MQEnvironment.sslPeerName)</td>
</tr>
<tr>
<td>MQC.SSL_SOCKET_FACTORY_PROPERTY</td>
<td>javax.net.ssl.SSLSocketFactory (overrides MQEnvironment.sslSocketFactory)</td>
</tr>
<tr>
<td>MQC.TRANSPORT_PROPERTY</td>
<td>MQC.TRANSPORT_MQSERIES_BINDINGS or MQC.TRANSPORT_MQSERIES_CLIENT or MQC.TRANSPORT_MQSERIES (the default, which selects bindings or client, based on the value of &quot;hostname&quot;).</td>
</tr>
<tr>
<td>MQC.USER_ID_PROPERTY</td>
<td>String (overrides MQEnvironment.userID.)</td>
</tr>
</tbody>
</table>

#### receiveExit

_public static MQReceiveExit receiveExit_

A receive exit allows you to examine, and possibly alter, data received from a queue manager. It is normally used in conjunction with a corresponding send exit at the queue manager.

To provide your own receive exit, define a class that implements the `MQReceiveExit` interface, and assign `receiveExit` to an instance of that class. Otherwise, you can leave `receiveExit` set to `null`, in which case no receive exit will be called.

See also [“MQReceiveExit” on page 174](#)

#### securityExit

_public static MQSecurityExit securityExit_

A security exit allows you to customize the security flows that occur when an attempt is made to connect to a queue manager.

To provide your own security exit, define a class that implements the `MQSecurityExit` interface, and assign `securityExit` to an instance of that class. Otherwise, you can leave `securityExit` set to `null`, in which case no security exit will be called.

See also [“MQSecurityExit” on page 176](#)

#### sendExit

_public static MQSendExit sendExit_

A send exit allows you to examine, and possibly alter, the data sent to a queue manager. It is normally used in conjunction with a corresponding receive exit at the queue manager.

To provide your own send exit, define a class that implements the `MQSendExit` interface, and assign `sendExit` to an instance of that class. Otherwise, you can leave `sendExit` set to `null`, in which case no send exit will be called.

See also [“MQSendExit” on page 178](#)
sslCertStores*

public static java.util.Collection sslCertStores

A Collection of CertStore objects used for certificate revocation checking. Use of this variable requires a JVM at Java 2 v1.4 or later. If sslCipherSuite is set, this variable can be used to ensure that the queue manager’s certificate has not become revoked. Each CertStore in the Collection represents an identical copy of the certificate revocation list (CRL). For more information on the behaviour of sslCertStores, refer to “Using certificate revocation lists” on page 83. If set to null (default), the certificate presented by the queue manager is not checked against any certificate revocation list. This variable is ignored if sslCipherSuite is null.

sslCipherSuite*

public static String sslCipherSuite

If set, SSL is enabled for the connection. The sslCipherSuite should be set to the CipherSuite name matching the CipherSpec set on the SVRCONN channel. If set to null (default), no SSL encryption is performed.

sslPeerName*

public static String sslPeerName

A distinguished name pattern. If sslCipherSuite is set, this variable can be used to ensure the correct queue manager is used. For a description of the format for this value, see “Using the distinguished name of the queue manager” on page 83. If set to null (default), no checking of the queue manager’s DN is performed. This variable is ignored if sslCipherSuite is null.

sslSocketFactory*

public static javax.net.ssl.SSLSocketFactory sslSocketFactory

The factory to use when connecting with SSL encryption. If sslCipherSuite is set, this variable can be used to customize all aspects of the SSL connection. For more information on constructing and customizing SSLSocketFactory instances, refer to your JSSE provider; for information regarding the use of this variable, refer to “Supplying a customized SSLSocketFactory” on page 84. If set to null (default) and SSL encryption is requested, the default SSLSocketFactory is used. This variable is ignored if sslCipherSuite is null.

userID*

public static String userID

Equivalent to the WebSphere MQ environment variable MQ_USER_ID. If a security exit is not defined for this client, the value of userID is transmitted to the server and will be available to the server security exit when it is invoked. The value may be used to verify the identity of the WebSphere MQ client.

The default value is "".

version_notice

public final static String version_notice

The current version of WebSphere MQ classes for Java.
MQEnvironment

Constructors

MQEnvironment
public MQEnvironment()

Methods

addConnectionPoolToken
public static void addConnectionPoolToken(MQPoolToken token)

Adds the supplied MQPoolToken to the set of tokens. A default
ConnectionManager can use this as a hint; typically, it are enabled only
while there is at least one token in the set.

Parameters:
token The MQPoolToken to add to the set of tokens.

addConnectionPoolToken
public static MQPoolToken addConnectionPoolToken()

Constructs an MQPoolToken and adds it to the set of tokens. The
MQPoolToken is returned to the application to be passed later into
removeConnectionPoolToken().

disableTracing
public static void disableTracing()

Turns off the WebSphere MQ Client for Java trace facility.

enableTracing
public static void enableTracing(int level)

Turns on the WebSphere MQ Client for Java trace facility.

Parameters
level The level of tracing required, from 1 to 5 (5 being the most
detailed).

enableTracing
public static void enableTracing(int level, OutputStream stream)

Turns on the WebSphere MQ Client for Java trace facility.

Parameters:
level The level of tracing required, from 1 to 5 (5 being the most
detailed).
stream The stream to which the trace is written.

defaultConnectionManager
public static javax.resource.spi.ConnectionManager
getDefaultConnectionManager()

getDefaultConnectionManager

Returns the default ConnectionManager. If the default ConnectionManager
is actually an MQConnectionManager, returns null.
getVersionNotice()

    public static final String getVersionNotice()

    Returns the current version of the WebSphere MQ base Java.

removeConnectionPoolToken

    public static void removeConnectionPoolToken(MQPoolToken token)

    Removes the specified MQPoolToken from the set of tokens. If that MQPoolToken is not in the set, there is no action.

    Parameters:
    
    token   The MQPoolToken to remove from the set of tokens.

setDefaultConnectionManager

    public static void setDefaultConnectionManager(MQConnectionManager cxManager)

    Sets the supplied MQConnectionManager to be the default ConnectionManager. The default ConnectionManager is used when there is no ConnectionManager specified on the MQQueueManager constructor. This method also empties the set of MQPoolTokens.

    Parameters:
    
    cxManager   The MQConnectionManager to be the default ConnectionManager.

setDefaultConnectionManager

    public static void setDefaultConnectionManager(javax.resource.spi.ConnectionManager cxManager)

    Sets the default ConnectionManager, and empties the set of MQPoolTokens. The default ConnectionManager is used when there is no ConnectionManager specified on the MQQueueManager constructor.

    This method requires a JVM at Java 2 v1.3 or later, with JAAS 1.0 or later installed.

    Parameters:
    
    cxManager   The default ConnectionManager (which implements the javax.resource.spi.ConnectionManager interface).
MQException

MQException

An MQException is thrown whenever a WebSphere MQ error occurs. You can change the output stream for the exceptions that are logged by setting the value of MQException.log. The default value is System.err. This class contains definitions of completion code and error code constants. Constants beginning MQCC_ are WebSphere MQ completion codes, and constants beginning MQRC_ are WebSphere MQ reason codes. The [WebSphere MQ Application Programming Reference](https://www-01.ibm.com/support/docview.wss?uid=swg27018246) contains a full description of these errors and their probable causes.

Variables

**completionCode**

```java
public int completionCode
```

WebSphere MQ completion code giving rise to the error. The possible values are:
- MQException.MQCC_WARNING
- MQException.MQCC_FAILED

**exceptionSource**

```java
public Object exceptionSource
```

The object instance that threw the exception. You can use this as part of your diagnostics when determining the cause of an error.

**log**

```java
public static java.io.OutputStreamWriter log
```

Stream to which exceptions are logged. (The default is System.err.) If you set this to null, no logging occurs.

**reasonCode**

```java
public int reasonCode
```

WebSphere MQ reason code describing the error. For a full explanation of the reason codes, refer to the [WebSphere MQ Application Programming Reference](https://www-01.ibm.com/support/docview.wss?uid=swg27018246).

Constructors

**MQException**

```java
public MQException(int completionCode,
                   int reasonCode,
                   Object source)
```

Construct a new MQException object.

Parameters
completionCode
   The WebSphere MQ completion code.

reasonCode
   The WebSphere MQ reason code.

source
   The object in which the error occurred.

Methods

gCause
   public Throwable getCause()

   Returns the cause of this MQException, or null if the cause is nonexistent
   or unknown. Note that this method is available on MQException even
   under JVMs before Java 2 v1.4.

initCause
   public Throwable initCause(Throwable cause)

   Initializes the cause of this MQException to the specified value. Throws
   IllegalArgumentException if cause is this MQException or
   IllegalStateException if the cause has already been set. Returns this
   MQException. Note that this method is available on MQException even
   under JVMs before Java 2 v1.4. Applications do not normally use this
   method.
MQGetMessageOptions

MQGetMessageOptions

java.lang.Object
  └─ com.ibm.mq.MQGetMessageOptions

public class MQGetMessageOptions
extends Object

This class contains options that control the behavior of MQQueue.get().

**Note:** The behavior of some of the options available in this class depends on the environment in which they are used. These elements are marked with a *. See Chapter 8, “Environment-dependent behavior” on page 89 for details.

**Variables**

**groupStatus**

public char groupStatus

This is an output field which indicates whether the retrieved message is in a group, and if it is, whether it is the last in the group. Possible values are:

MQC.MQGS_LAST_MSG_IN_GROUP
  Message is the last in the group. This is also the value returned if the group consists of only one message.

MQC.MQGS_MSG_IN_GROUP
  Message is in a group, but is not the last in the group.

MQC.MQGS_NOT_IN_GROUP
  Message is not in a group.

**matchOptions**

public int matchOptions

Selection criteria that determine which message is retrieved. The following match options can be set:

MQC.MQMO_MATCH_CORREL_ID
  Correlation id to be matched.

MQC.MQMO_MATCH_GROUP_ID
  Group id to be matched.

MQC.MQMO_MATCH_MSG_ID
  Message id to be matched.

MQC.MQMO_MATCH_MSG_SEQ_NUMBER
  Match message sequence number.

MQC.MQMO_NONE
  No matching required.

**options**

public int options

Options that control the action of MQQueue.get. Any or none of the following values can be specified. If more than one option is required, the values can be added together or combined using the bitwise OR operator.

MQC.MQGMO_ACCEPT_TRUNCATED_MSG
  Allow truncation of message data.
MQC.MQGMO_BROWSE_FIRST
Browse from start of queue.

MQC.MQGMO_BROWSE_MSG_UNDER_CURSOR*
Browse message under browse cursor.

MQC.MQGMO_BROWSE_NEXT
Browse from the current position in the queue.

MQC.MQGMO_CONVERT
Request the application data to be converted, to conform to the characterSet and encoding attributes of the MQMessage, before the data is copied into the message buffer. Because data conversion is also applied as the data is retrieved from the message buffer, applications do not usually set this option.

Using this option may cause problems when converting from single byte character sets to double byte character sets. Instead, do the conversion using the readString, readLine and writeString methods after the message has been delivered.

MQC.MQGMO_FAIL_IF_QUIESCING
Fail if the queue manager is quiescing.

MQC.MQGMO_LOCK*
Lock the message that is browsed.

MQC.MQGMO_MARK_SKIP_BACKOUT*
Allow a unit of work to be backed out without reinstating the message on the queue.

MQC.MQGMO_MSG_UNDER_CURSOR
Get message under browse cursor.

MQC.MQGMO_NONE

MQC.MQGMO_NO_SYNCPOINT
Get message without syncpoint control.

MQC.MQGMO_NO_WAIT
Return immediately if there is no suitable message.

MQC.MQGMO_SYNCPOINT
Get the message under syncpoint control; the message is marked as being unavailable to other applications, but it is deleted from the queue only when the unit of work is committed. The message is made available again if the unit of work is backed out.

MQC.MQGMO_SYNCPOINT_IF_PERSISTENT*
Get message with syncpoint control if message is persistent.

MQC.MQGMO_UNLOCK*
Unlock a previously locked message.

MQC.MQGMO_WAIT
Wait for a message to arrive.

Segmenting and grouping WebSphere MQ messages can be sent or received as a single entity, can be split into several segments for sending and receiving, and can also be linked to other messages in a group.

Each piece of data that is sent is known as a physical message, which can be a complete logical message, or a segment of a longer logical message.
Each physical message usually has a different MsgId. All the segments of a single logical message have the same groupId value and MsgSeqNumber value, but the Offset value is different for each segment. The Offset field gives the offset of the data in the physical message from the start of the logical message. The segments usually have different MsgId values, because they are individual physical messages.

Logical messages that form part of a group have the same groupId value, but each message in the group has a different MsgSeqNumber value. Messages in a group can also be segmented.

The following options can be used for dealing with segmented or grouped messages:

**MQC.MQGMO_ALL_MSGS_AVAILABLE**
Retrieve messages from a group only when all the messages in the group are available.

**MQC.MQGMO_ALL_SEGMENTS_AVAILABLE**
Retrieve the segments of a logical message only when all the segments in the group are available.

**MQC.MQGMO_COMPLETE_MSG**
Retrieve only complete logical messages.

**MQC.MQGMO_LOGICAL_ORDER**
Return messages in groups, and segments of logical messages, in logical order.

**resolvedQueueName**
public String resolvedQueueName
This is an output field that the queue manager sets to the local name of the queue from which the message was retrieved. This will be different from the name used to open the queue if an alias queue or model queue was opened.

**segmentation**
public char segmentation
This is an output field that indicates whether or not segmentation is allowed for the retrieved message is a segment of a logical message. Possible values are:

**MQC.MQSEG_INHIBITED**
Segmentation not allowed.

**MQC.MQSEG_ALLOWED**
Segmentation allowed.

**segmentStatus**
public char segmentStatus
This is an output field that indicates whether the retrieved message is a segment of a logical message. If the message is a segment, the flag indicates whether or not it is the last segment. Possible values are:

**MQC.MQSS_LAST_SEGMENT**
Message is the last segment of the logical message. This is also the value returned if the logical message consists of only one segment.

**MQC.MQSS_NOT_A_SEGMENT**
Message is not a segment.
MQC.MQSS_SEGMENT
Message is a segment, but is not the last segment of the logical message.

waitInterval
public int waitInterval
The maximum time (in milliseconds) that an MQQueue.get call waits for a suitable message to arrive (used in conjunction with MQC.MQGMO_WAIT). A value of MQC.MQWI_UNLIMITED indicates that an unlimited wait is required.

Constructors

MQGetMessageOptions
public MQGetMessageOptions()
Construct a new MQGetMessageOptions object with options set to MQC.MQGMO_NO_WAIT, a wait interval of zero, and a blank resolved queue name.
MQManagedObject

MQManagedObject

```java
java.lang.Object
    │
    └─ com.ibm.mq.MQManagedObject

public class MQManagedObject
    extends Object

MQManagedObject is a superclass for MQQueueManager, MQQueue and MQProcess. It provides the ability to inquire and set attributes of these resources.

Variables

alternateUserId
    public String alternateUserId
    The alternate user id (if any) specified when this resource was opened. Setting this attribute has no effect.

closeOptions
    public int closeOptions
    Set this attribute to control the way the resource is closed. The default value is MQC.MQCO_NONE, and this is the only permissible value for all resources other than permanent dynamic queues, and temporary dynamic queues that are being accessed by the objects that created them. For these queues, the following additional values are permissible:

MQC.MQCO_DELETE
    Delete the queue if there are no messages.

MQC.MQCO_DELETE_PURGE
    Delete the queue, purging any messages on it.

connectionReference
    public MQQueueManager connectionReference
    The queue manager to which this resource belongs. Setting this attribute has no effect.

isOpen
    public boolean isOpen
    Indicates whether this resource is currently open. This attribute is deprecated and setting it has no effect.

name
    public String name
    The name of this resource (either the name supplied on the access method, or the name allocated by the queue manager for a dynamic queue). Setting this attribute has no effect.

openOptions
    public int openOptions
    The options specified when this resource was opened. Setting this attribute has no effect.
Constructors

MQManagedObject

protected MQManagedObject()

Constructor method.

Methods

close

public synchronized void close()

throws MQException.

Close the object. No further operations against this resource are permitted after this method has been called. The behavior of the close method may be altered by setting the closeOptions attribute.

Throws MQException if the WebSphere MQ call fails.

description

public String getDescription()

Throws MQException.

Return the description of this resource as held at the queue manager.

If this method is called after the resource has been closed, an MQException is thrown.

inquire

public void inquire(int selectors[],
                     int intAttrs[],
                     byte charAttrs[])

throws MQException.

Returns an array of integers and a set of character strings containing the attributes of an object (queue, process or queue manager).

The attributes to be queried are specified in the selectors array. Refer to the [WebSphere MQ Application Programming Reference](#) for details of the permissible selectors and their corresponding integer values.

Note that many of the more common attributes can be queried using the getXXX() methods defined in MQManagedObject, MQQueue, MQQueueManager, and MQProcess.

Parameters

selectors

Integer array identifying the attributes with values to be inquired on.

intAttrs

The array in which the integer attribute values are returned. Integer attribute values are returned in the same order as the integer attribute selectors in the selectors array.
MQManagedObject

charAttrs
The buffer in which the character attributes are returned, concatenated. Character attributes are returned in the same order as the character attribute selectors in the selectors array. The length of each attribute string is fixed for each attribute.

Throws MQException if the inquire fails.

isOpen
public boolean isOpen()

Returns the value of the isOpen variable.

set
public synchronized void set(int selectors[], int intAttrs[], byte charAttrs[])

throws MQException.

Set the attributes defined in the selector's vector.

The attributes to be set are specified in the selectors array. Refer to the WebSphere MQ Application Programming Reference for details of the permissible selectors and their corresponding integer values.

Note that some queue attributes can be set using the setXXX() methods defined in MQQueue.

Parameters
selectors
Integer array identifying the attributes with values to be set.

intAttrs
The array of integer attribute values to be set. These values must be in the same order as the integer attribute selectors in the selectors array.

charAttrs
The buffer in which the character attributes to be set are concatenated. These values must be in the same order as the character attribute selectors in the selectors array. The length of each character attribute is fixed.

Throws MQException if the set fails.
MQMessage

java.lang.Object
  └─ com.ibm.mq.MQMessage

public class MQMessage
  implements DataInput, DataOutput

MQMessage represents both the message descriptor and the data for a WebSphere MQ message. There is a group of readXXX methods for reading data from a message, and a group of writeXXX methods for writing data into a message. The format of numbers and strings used by these read and write methods can be controlled by the encoding and characterSet member variables. The remaining member variables contain control information that accompanies the application message data when a message travels between sending and receiving applications. The application can set values into the member variable before putting a message to a queue and can read values after retrieving a message from a queue.

Variables

accountingToken
  public byte accountingToken[]

Part of the identity context of the message; it allows an application to cause work done as a result of the message to be appropriately charged.

The default value is "MQC.MQACT_NONE".

applicationIdData
  public String applicationIdData

Part of the identity context of the message; it is information that is defined by the application suite, and can be used to provide additional information about the message or its originator.

The default value is "".

applicationOriginData
  public String applicationOriginData

Information defined by the application that can be used to provide additional information about the origin of the message.

The default value is "".

backoutCount
  public int backoutCount

A count of the number of times the message has previously been returned by an MQQueue.get() call as part of a unit of work, and subsequently backed out.

The default value is zero.

characterSet
  public int characterSet

This specifies the coded character set identifier of character data in the application message data. The behavior of the readString, readLine and writeString methods is altered accordingly.
The default value for this field is MQC.MQCCSI_Q_MGR. If the default value is used, CharacterSet 819 (iso-8859-1/latin/ibm819) is assumed. The character set values you can use will depend upon the JVM you use.

Table 13 shows coded character set identifiers and the characterSet values to use:

<table>
<thead>
<tr>
<th>characterSet</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>ibm037</td>
</tr>
<tr>
<td>437</td>
<td>ibm437 / PC Original</td>
</tr>
<tr>
<td>500</td>
<td>ibm500</td>
</tr>
<tr>
<td>819</td>
<td>iso-8859-1 / latin1 / ibm819</td>
</tr>
<tr>
<td>1200</td>
<td>Unicode</td>
</tr>
<tr>
<td>1208</td>
<td>UTF-8</td>
</tr>
<tr>
<td>273</td>
<td>ibm273</td>
</tr>
<tr>
<td>277</td>
<td>ibm277</td>
</tr>
<tr>
<td>278</td>
<td>ibm278</td>
</tr>
<tr>
<td>280</td>
<td>ibm280</td>
</tr>
<tr>
<td>284</td>
<td>ibm284</td>
</tr>
<tr>
<td>285</td>
<td>ibm285</td>
</tr>
<tr>
<td>297</td>
<td>ibm297</td>
</tr>
<tr>
<td>420</td>
<td>ibm420</td>
</tr>
<tr>
<td>424</td>
<td>ibm424</td>
</tr>
<tr>
<td>737</td>
<td>ibm737 / PC Greek</td>
</tr>
<tr>
<td>775</td>
<td>ibm775 / PC Baltic</td>
</tr>
<tr>
<td>813</td>
<td>iso-8859-7 / greek / ibm813</td>
</tr>
<tr>
<td>838</td>
<td>ibm838</td>
</tr>
<tr>
<td>850</td>
<td>ibm850 / PC Latin 1</td>
</tr>
<tr>
<td>852</td>
<td>ibm852 / PC Latin 2</td>
</tr>
<tr>
<td>855</td>
<td>ibm855 / PC Cyrillic</td>
</tr>
<tr>
<td>856</td>
<td>ibm856</td>
</tr>
<tr>
<td>857</td>
<td>ibm857 / PC Turkish</td>
</tr>
<tr>
<td>860</td>
<td>ibm860 / PC Portuguese</td>
</tr>
<tr>
<td>861</td>
<td>ibm861 / PC Icelandic</td>
</tr>
<tr>
<td>862</td>
<td>ibm862 / PC Hebrew</td>
</tr>
<tr>
<td>863</td>
<td>ibm863 / PC Canadian French</td>
</tr>
<tr>
<td>864</td>
<td>ibm864 / PC Arabic</td>
</tr>
<tr>
<td>865</td>
<td>ibm865 / PC Nordic</td>
</tr>
<tr>
<td>866</td>
<td>ibm866 / PC Russian</td>
</tr>
<tr>
<td>868</td>
<td>ibm868</td>
</tr>
<tr>
<td>869</td>
<td>ibm869 / PC Modern Greek</td>
</tr>
<tr>
<td>870</td>
<td>ibm870</td>
</tr>
<tr>
<td>871</td>
<td>ibm871</td>
</tr>
<tr>
<td>874</td>
<td>ibm874</td>
</tr>
<tr>
<td>875</td>
<td>ibm875</td>
</tr>
<tr>
<td>912</td>
<td>iso-8859-2 / latin2 / ibm912</td>
</tr>
<tr>
<td>913</td>
<td>iso-8859-3 / latin3 / ibm913</td>
</tr>
<tr>
<td>914</td>
<td>iso-8859-4 / latin4 / ibm914</td>
</tr>
<tr>
<td>915</td>
<td>iso-8859-5 / cyrillic / ibm915</td>
</tr>
<tr>
<td>916</td>
<td>iso-8859-8 / hebrew / ibm916</td>
</tr>
<tr>
<td>918</td>
<td>ibm918</td>
</tr>
<tr>
<td>920</td>
<td>iso-8859-9 / latin5 / ibm920</td>
</tr>
<tr>
<td>921</td>
<td>ibm921</td>
</tr>
<tr>
<td>922</td>
<td>ibm922</td>
</tr>
<tr>
<td>930</td>
<td>ibm930</td>
</tr>
</tbody>
</table>
Table 13. Character set identifiers (continued)

<table>
<thead>
<tr>
<th>characterSet</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>932</td>
<td>PC Japanese</td>
</tr>
<tr>
<td>933</td>
<td>ibm933</td>
</tr>
<tr>
<td>935</td>
<td>ibm935</td>
</tr>
<tr>
<td>937</td>
<td>ibm937</td>
</tr>
<tr>
<td>939</td>
<td>ibm939</td>
</tr>
<tr>
<td>942</td>
<td>ibm942</td>
</tr>
<tr>
<td>948</td>
<td>ibm948</td>
</tr>
<tr>
<td>949</td>
<td>ibm949</td>
</tr>
<tr>
<td>950</td>
<td>ibm950 / Big 5 Traditional Chinese</td>
</tr>
<tr>
<td>954</td>
<td>EUCJIS</td>
</tr>
<tr>
<td>964</td>
<td>ibm964 / CNS 11643 Traditional Chinese</td>
</tr>
<tr>
<td>970</td>
<td>ibm970</td>
</tr>
<tr>
<td>1006</td>
<td>ibm1006</td>
</tr>
<tr>
<td>1025</td>
<td>ibm1025</td>
</tr>
<tr>
<td>1026</td>
<td>ibm1026</td>
</tr>
<tr>
<td>1089</td>
<td>iso-8859-6 / arabic / ibm1089</td>
</tr>
<tr>
<td>1097</td>
<td>ibm1097</td>
</tr>
<tr>
<td>1098</td>
<td>ibm1098</td>
</tr>
<tr>
<td>1112</td>
<td>ibm1112</td>
</tr>
<tr>
<td>1122</td>
<td>ibm1122</td>
</tr>
<tr>
<td>1123</td>
<td>ibm1123</td>
</tr>
<tr>
<td>1124</td>
<td>ibm1124</td>
</tr>
<tr>
<td>1250</td>
<td>Windows Latin 2</td>
</tr>
<tr>
<td>1251</td>
<td>Windows Cyrillic</td>
</tr>
<tr>
<td>1252</td>
<td>Windows Latin 1</td>
</tr>
<tr>
<td>1253</td>
<td>Windows Greek</td>
</tr>
<tr>
<td>1254</td>
<td>Windows Turkish</td>
</tr>
<tr>
<td>1255</td>
<td>Windows Hebrew</td>
</tr>
<tr>
<td>1256</td>
<td>Windows Arabic</td>
</tr>
<tr>
<td>1257</td>
<td>Windows Baltic</td>
</tr>
<tr>
<td>1258</td>
<td>Windows Vietnamese</td>
</tr>
<tr>
<td>1381</td>
<td>ibm1381</td>
</tr>
<tr>
<td>1383</td>
<td>ibm1383</td>
</tr>
<tr>
<td>2022</td>
<td>JIS</td>
</tr>
<tr>
<td>5601</td>
<td>ksc-5601 Korean</td>
</tr>
<tr>
<td>33722</td>
<td>ibm33722</td>
</tr>
</tbody>
</table>

**correlationId**

```java
public byte correlationId[]
```

For an MQQueue.get() call, this field specifies the correlation identifier of the message to be retrieved. Normally the queue manager returns the first message with a message identifier and correlation identifier that match those specified. The special value MQC.MQC1_NONE allows any correlation identifier to match.

For an MQQueue.put() call, this specifies the correlation identifier to use.

The default value is MQC.MQC1_NONE.

**encoding**

```java
public int encoding
```

This member variable specifies the representation used for numeric values in the application message data; this applies to binary, packed decimal, and
The behavior of the read and write methods for these numeric formats is altered accordingly.

The following encodings are defined for binary integers:

MQC.MQENC_INTEGER_NORMAL
Big-endian integers, as in Java

MQC.MQENC_INTEGER_REVERSED
Little-endian integers, as used by PCs.

The following encodings are defined for packed-decimal integers:

MQC.MQENC_DECIMAL_NORMAL
Big-endian packed-decimal, as used by z/OS.

MQC.MQENC_DECIMAL_REVERSED
Little-endian packed-decimal.

The following encodings are defined for floating-point numbers:

MQC.MQENC_FLOAT_IEEE_NORMAL
Big-endian IEEE floats, as in Java.

MQC.MQENC_FLOAT_IEEE_REVERSED
Little-endian IEEE floats, as used by PCs.

MQC.MQENC_FLOAT_S390
z/OS format floating points.

A value for the encoding field should be constructed by adding together one value from each of these three sections (or using the bitwise OR operator). The default value is:

MQC.MQENC_INTEGER_NORMAL
MQC.MQENC_DECIMAL_NORMAL
MQC.MQENC_FLOAT_IEEE_NORMAL

For convenience, this value is also represented by MQC.MQENC_NATIVE. This setting causes writeInt() to write a big-endian integer, and readInt() to read a big-endian integer. If the flag MQC.MQENC_INTEGER_REVERSED flag had been set instead, writeInt() would write a little-endian integer, and readInt() would read a little-endian integer.

Note that a loss in precision can occur when converting from IEEE format floating points to System/390® format floating points.

expiry

An expiry time expressed in tenths of a second, set by the application that puts the message. After a message’s expiry time has elapsed, it is eligible to be discarded by the queue manager. If the message specified one of the MQC.MQRO_EXPIRATION flags, a report is generated when the message is discarded.

The default value is MQC.MQEI_UNLIMITED, meaning that the message never expires.

feedback

This is used with a message of type MQC.MQMT_REPORT to indicate the nature of the report. The following feedback codes are defined by the system:
MQMessage

- MQC.MQFB_EXPIRATION
- MQC.MQFB_COA
- MQC.MQFB_COD
- MQC.MQFB.Quit
- MQC.MQFB_PAN
- MQC.MQFB_NAN
- MQC.MQFB_DATA_LENGTH_ZERO
- MQC.MQFB_DATA_LENGTH_NEGATIVE
- MQC.MQFB_DATA_LENGTH_TOO_BIG
- MQC.MQFB_BUFFER_OVERFLOW
- MQC.MQFB_LENGTH_OFF_BY_ONE
- MQC.MQFB_IIH_ERROR

Application-defined feedback values in the range
MQC.MQFB_APPL_FIRST to MQC.MQFB_APPL_LAST can also be used.

The default value of this field is MQC.MQFB_NONE, indicating that no feedback is provided.

format

public String format

A format name used by the sender of the message to indicate the nature of the data in the message to the receiver. You can use your own format names, but names beginning with the letters “MQ” have meanings that are defined by the queue manager. The queue manager built-in formats are:

MQC.MQFMT_ADMIN
Command server request/reply message.

MQC.MQFMT_COMMAND_1
Type 1 command reply message.

MQC.MQFMT_COMMAND_2
Type 2 command reply message.

MQC.MQFMT_DEADLETTER_HEADER
Dead-letter header.

MQC.MQFMT_EVENT
Event message.

MQC.MQFMT_NONE
No format name.

MQC.MQFMT_PCF
User-defined message in programmable command format.

MQC.MQFMT_STRING
Message consisting entirely of characters.

MQC.MQFMT_TRIGGER
Trigger message

MQC.MQFMT_XMIT_Q_HEADER
Transmission queue header.

The default value is MQC.MQFMT_NONE.

groupId

public byte[] groupId
MQMessage

A byte string that identifies the message group to which the physical message belongs.

The default value is "MQC.MQGI_NONE".

**messageFlags**

```java
public int messageFlags
```

Flags controlling the segmentation and status of a message.

**messageId**

```java
public byte messageId[]
```

For an MQQueue.get() call, this field specifies the message identifier of the message to be retrieved. Normally, the queue manager returns the first message with a message identifier and correlation identifier that match those specified. The special value MQC.MQMI_NONE allows any message identifier to match.

For an MQQueue.put() call, this specifies the message identifier to use. If MQC.MQMI_NONE is specified, the queue manager generates a unique message identifier when the message is put. The value of this member variable is updated after the put to indicate the message identifier that was used.

The default value is MQC.MQMI_NONE.

**messageSequenceNumber**

```java
public int messageSequenceNumber
```

The sequence number of a logical message within a group.

**messageType**

```java
public int messageType
```

Indicates the type of the message. The following values are currently defined by the system:

- MQC.MQMT_DATAGRAM
- MQC.MQMT_REPLY
- MQC.MQMT_REPORT
- MQC.MQMT_REQUEST

Application-defined values can also be used. These should be in the range MQC.MQMT_APPL_FIRST to MQC.MQMT_APPL_LAST.

The default value of this field is MQC.MQMT_DATAGRAM.

**offset**

```java
public int offset
```

In a segmented message, the offset of data in a physical message from the start of a logical message.

**originalLength**

```java
public int originalLength
```

The original length of a segmented message.

**persistence**

```java
public int persistence
```

Message persistence. The following values are defined:

- MQC.MQPER_NOT_PERSISTENT
- MQC.MQPER_PERSISTENT
- MQC.MQPER_PERSISTENCE_AS_Q_DEF
The default value is MQC.MQPER_PERSISTENCE_AS_Q_DEF, which indicates that the persistence for the message should be taken from the default persistence attribute of the destination queue.

**priority**

```java
public int priority
```

The message priority. The special value MQC.MQPRI_PRIORITY_AS_Q_DEF can also be set in outbound messages, in which case the priority for the message is taken from the default priority attribute of the destination queue.

The default value is MQC.MQPRI_PRIORITY_AS_Q_DEF.

**putApplicationName**

```java
public String putApplicationName
```

The name of the application that put the message. The default value is "".

**putApplicationType**

```java
public int putApplicationType
```

The type of application that put the message. This may be a system-defined or user-defined value. The following values are defined by the system:
- MQC.MQAT_AIX
- MQC.MQAT_CICS
- MQC.MQAT_DOS
- MQC.MQAT_IMS
- MQC.MQAT_MVS
- MQC.MQAT.OS2
- MQC.MQAT.OS400
- MQC.MQAT.QMGR
- MQC.MQAT_UNIX
- MQC.MQAT_WINDOWS
- MQC.MQAT_JAVA

The default value is the special value MQC.MQAT_NO_CONTEXT, which indicates that no context information is present in the message.

**putDateTime**

```java
public GregorianCalendar putDateTime
```

The time and date that the message was put.

**replyToQueueManagerName**

```java
public String replyToQueueManagerName
```

The name of the queue manager to which reply or report messages should be sent.

The default value is "".

If the value is "" on an MQQueue.put() call, the QueueManager fills in the value.

**replyToQueueName**

```java
public String replyToQueueName
```

The name of the message queue to which the application that issued the get request for the message should send MQC.MQMT_REPLY and MQC.MQMT_REPORT messages.

The default value is "".
MQMessage

report  public int report

A report is a message about another message. This member variable enables the application sending the original message to specify which report messages are required, whether the application message data is to be included in them, and also how the message and correlation identifiers in the report or reply are to be set. Any, all or none of the following report types can be requested:
• Exception
• Expiration
• Confirm on arrival
• Confirm on delivery

For each type, only one of the three corresponding values below should be specified, depending on whether the application message data is to be included in the report message.

Note: Values marked with ** in the following list are not supported by MVS™ queue managers and should not be used if your application is likely to access an MVS queue manager, regardless of the platform on which the application is running.

The valid values are:
• MQC.MQRO_COA
• MQC.MQRO_COA_WITH_DATA
• MQC.MQRO_COA_WITH_FULL_DATA**
• MQC.MQRO_COD
• MQC.MQRO_COD_WITH_DATA
• MQC.MQRO_COD_WITH_FULL_DATA**
• MQC.MQRO_EXCEPTION
• MQC.MQRO_EXCEPTION_WITH_DATA
• MQC.MQRO_EXCEPTION_WITH_FULL_DATA**
• MQC.MQRO_EXPIRATION
• MQC.MQRO_EXPIRATION_WITH_DATA
• MQC.MQRO_EXPIRATION_WITH_FULL_DATA**

You can specify one of the following to control how the message Id is generated for the report or reply message:
• MQC.MQRO_NEW_MSG_ID
• MQC.MQRO_PASS_MSG_ID

You can specify one of the following to control how the correlation Id of the report or reply message is to be set:
• MQC.MQRO_COPY_MSG_ID_TO_CORREL_ID
• MQC.MQRO_PASS_CORREL_ID

You can specify one of the following to control the disposition of the original message when it cannot be delivered to the destination queue:
• MQC.MQRO_DEAD_LETTER_Q
• MQC.MQRO_DISCARD_MSG**

If no report options are specified, the default is:
MQC.MQRO_NEW_MSG_ID | MQC.MQRO_COPY_MSG_ID_TO_CORREL_ID | MQC.MQRO_DEAD_LETTER_Q
MQMessage

You can specify one or both of the following to request that the receiving application send a positive action or negative action report message.

- MQRO_PAN
- MQRO_NAN

userId  public String userId

Part of the identity context of the message; it identifies the user that originated this message.

The default value is "".

Constructors

MQMessage  
public MQMessage()

Create a new message with default message descriptor information and an empty message buffer.

Methods

clearMessage  
public void clearMessage()

Throws IOException.

Discard any data in the message buffer and set the data offset back to zero.

dataLength  
public int getDataLength()

Throws MQException.

The number of bytes of message data remaining to be read.

dataOffset  
public int getDataOffset()

Throws IOException.

Return the current cursor position within the message data (the point at which read and write operations take effect).

getMessageLength  
public int getMessageLength

Throws IOException.

The number of bytes of message data in this MQMessage object.

totalMessageLength  
public int getTotalMessageLength()

The total number of bytes in the message as stored on the message queue from which this message was retrieved (or attempted to be retrieved).

When an MQQueue.get() method fails with a message-truncated error code, this method tells you the total size of the message on the queue.

See also [“MQQueue.get” on page 149](#).
MQMessage

getVersion
    public int getVersion()
    Returns the version of the structure in use.

readBoolean
    public boolean readBoolean()
    Throws IOException.
    Read a (signed) byte from the current position in the message buffer.

readChar
    public char readChar()
    Throws IOException, EOFException.
    Read a Unicode character from the current position in the message buffer.

readDecimal2
    public short readDecimal2()
    Throws IOException, EOFException.
    Read a 2-byte packed decimal number (-999.999). The behavior of this method is controlled by the value of the encoding member variable. A value of MQC.MQENC_DECIMAL_NORMAL reads a big-endian packed decimal number, and a value of MQC.MQENC_DECIMAL_REVERSED reads a little-endian packed decimal number.

readDecimal4
    public int readDecimal4()
    Throws IOException, EOFException.
    Read a 4-byte packed decimal number (-99999999.99999999). The behavior of this method is controlled by the value of the encoding member variable. A value of MQC.MQENC_DECIMAL_NORMAL reads a big-endian packed decimal number, and a value of MQC.MQENC_DECIMAL_REVERSED reads a little-endian packed decimal number.

readDecimal8
    public long readDecimal8()
    Throws IOException, EOFException.
    Read an 8-byte packed decimal number (-9999999999999999 to 9999999999999999). The behavior of this method is controlled by the encoding member variable. A value of MQC.MQENC_DECIMAL_NORMAL reads a big-endian packed decimal number, and MQC.MQENC_DECIMAL_REVERSED reads a little-endian packed decimal number.

readDouble
    public double readDouble()
    Throws IOException, EOFException.
Read a double from the current position in the message buffer. The value of the encoding member variable determines the behavior of this method.

Values of MQC.MQENC_FLOAT_IEEE_NORMAL and MQC.MQENC_FLOAT_IEEE_REVERSED read IEEE standard doubles in big-endian and little-endian formats respectively.

A value of MQC.MQENC_FLOAT_S390 reads a System/390 format floating point number.

**readFloat**

```java
public float readFloat()
```

Throws IOException, EOFException.

Read a float from the current position in the message buffer. The value of the encoding member variable determines the behavior of this method.

Values of MQC.MQENC_FLOAT_IEEE_NORMAL and MQC.MQENC_FLOAT_IEEE_REVERSED read IEEE standard floats in big-endian and little-endian formats respectively.

A value of MQC.MQENC_FLOAT_S390 reads a System/390 format floating point number.

**readFully**

```java
public void readFully(byte b[])
```

Throws Exception, EOFException.

Fill the byte array `b` with data from the message buffer.

**readFully**

```java
public void readFully(byte b[],
                      int off,
                      int len)
```

Throws IOException, EOFException.

Fill `len` elements of the byte array `b` with data from the message buffer, starting at offset `off`.

**readInt**

```java
public int readInt()
```

Throws IOException, EOFException.

Read an integer from the current position in the message buffer. The value of the encoding member variable determines the behavior of this method.

A value of MQC.MQENC_INTEGER_NORMAL reads a big-endian integer, a value of MQC.MQENC_INTEGER_REVERSED reads a little-endian integer.

**readInt2**

```java
public short readInt2()
```

Throws IOException, EOFException.
MQMessage

Synonym for readShort(), provided for cross-language WebSphere MQ API compatibility.

**readInt4**

```java
public int readInt4()
```

Throws IOException, EOFException.

Synonym for readInt(), provided for cross-language WebSphere MQ API compatibility.

**readInt8**

```java
public long readInt8()
```

Throws IOException, EOFException.

Synonym for readLong(), provided for cross-language WebSphere MQ API compatibility.

**readLine**

```java
public String readLine()
```

Throws IOException.

Converts from the codeset identified in the characterSet member variable to Unicode, and then reads in a line that has been terminated by \n, \r, \r\n, or EOF.

**readLong**

```java
public long readLong()
```

Throws IOException, EOFException.

Read a long from the current position in the message buffer. The value of the encoding member variable determines the behavior of this method.

A value of MQC.MQENC_INTEGER_NORMAL reads a big-endian long, a value of MQC.MQENC_INTEGER_REVERSED reads a little-endian long.

**readObject**

```java
public Object readObject()
```

Throws OptionalDataException, ClassNotFoundException, IOException.

Read an object from the message buffer. The class of the object, the signature of the class, and the value of the non-transient and non-static fields of the class are all read.

**readShort**

```java
public short readShort()
```

Throws IOException, EOFException.

**readString**

```java
public String readString(int length)
```

Throws IOException, EOFException.
MQMessage

Read a string in the codeset identified by the characterSet member variable, and convert it into Unicode.

Parameters:

length The number of characters to read (which may differ from the number of bytes according to the codeset, because some codesets use more than one byte per character).

readUInt2
public int readUInt2()

Throws IOException, EOFException.

Synonym for readUnsignedShort(), provided for cross-language WebSphere MQ API compatibility.

readUnsignedByte
public int readUnsignedByte()

Throws IOException, EOFException.

Read an unsigned byte from the current position in the message buffer.

readUnsignedShort
public int readUnsignedShort()

Throws IOException, EOFException.

Read an unsigned short from the current position in the message buffer. The value of the encoding member variable determines the behavior of this method.

A value of MQC.MQENC_INTEGER_NORMAL reads a big-endian unsigned short, a value of MQC.MQENC_INTEGER_REVERSED reads a little-endian unsigned short.

readUTF
public String readUTF()

Throws IOException.

Read a UTF string, prefixed by a 2-byte length field, from the current position in the message buffer.

resizeBuffer
public void resizeBuffer(int size)

Throws IOException.

A hint to the MQMessage object about the size of buffer that may be required for subsequent get operations. If the message currently contains message data, and the new size is less than the current size, the message data is truncated.

seek
public void seek(int pos)
MQMessage

Throws IOException.

Move the cursor to the absolute position in the message buffer given by pos. Subsequent reads and writes will act at this position in the buffer.

Throws EOFException if pos is outside the message data length.

setDataOffset

public void setDataOffset(int offset)

Throws IOException.

Move the cursor to the absolute position in the message buffer. This method is a synonym for seek(), and is provided for cross-language compatibility with the other WebSphere MQ APIs.

setVersion

public void setVersion(int version)

Specifies which version of the structure to use. Possible values are:

- MQC.MQMD_VERSION_1
- MQC.MQMD_VERSION_2

You should not normally need to call this method unless you wish to force the client to use a version 1 structure when connected to a queue manager that is capable of handling version 2 structures. In all other situations, the client determines the correct version of the structure to use by querying the queue manager’s capabilities.

skipBytes

public int skipBytes(int n)

Throws IOException, EOFException.

Move forward n bytes in the message buffer.

This method blocks until one of the following occurs:

- All the bytes are skipped
- The end of message buffer is detected
- An exception is thrown

Returns the number of bytes skipped, which is always n.

write

public void write(int b)

Throws IOException.

Write a byte into the message buffer at the current position.

write

public void write(byte b[])

Throws IOException.

Write an array of bytes into the message buffer at the current position.
public void write(byte b[], int off, int len)

Throws IOException.

Write a series of bytes into the message buffer at the current position. len bytes will be written, taken from offset off in the array b.

writeBoolean
public void writeBoolean(boolean v)

Throws IOException.

Write a boolean into the message buffer at the current position.

writeByte
public void writeByte(int v)

Throws IOException.

Write a byte into the message buffer at the current position.

writeBytes
public void writeBytes(String s)

Throws IOException.

Writes out the string to the message buffer as a sequence of bytes. Each character in the string is written out in sequence by discarding its high eight bits.

writeChar
public void writeChar(int v)

Throws IOException.

Write a Unicode character into the message buffer at the current position.

writeChars
public void writeChars(String s)

Throws IOException.

Write a string as a sequence of Unicode characters into the message buffer at the current position.

writeDecimal2
public void writeDecimal2(short v)

Throws IOException.

Write a 2-byte packed decimal format number into the message buffer at the current position. The value of the encoding member variable determines the behavior of this method.
MQMessage

A value of MQC.MQENC_DECIMAL_NORMAL writes a big-endian packed decimal, a value of MQC.MQENC_DECIMAL_REVERSED writes a little-endian packed decimal.

Parameters

\( v \) can be in the range -999 to 999.

writeDecimal4

```java
public void writeDecimal4(int v)
```

Throws IOException.

Write a 4-byte packed decimal format number into the message buffer at the current position. The value of the encoding member variable determines the behavior of this method.

A value of MQC.MQENC_DECIMAL_NORMAL writes a big-endian packed decimal, a value of MQC.MQENC_DECIMAL_REVERSED writes a little-endian packed decimal.

Parameters

\( v \) can be in the range -9999999 to 9999999.

writeDecimal8

```java
public void writeDecimal8(long v)
```

Throws IOException.

Write an 8-byte packed decimal format number into the message buffer at the current position. The value of the encoding member variable determines the behavior of this method.

A value of MQC.MQENC_DECIMAL_NORMAL writes a big-endian packed decimal, a value of MQC.MQENC_DECIMAL_REVERSED writes a little-endian packed decimal.

Parameters:

\( v \) can be in the range -999999999999999 to 999999999999999.

writeDouble

```java
public void writeDouble(double v)
```

Throws IOException.

Write a double into the message buffer at the current position. The value of the encoding member variable determines the behavior of this method.

Values of MQC.MQENC_FLOAT_IEEE_NORMAL and MQC.MQENC_FLOAT_IEEE_REVERSED write IEEE standard floats in Big-endian and Little-endian formats respectively.

A value of MQC.MQENC_FLOAT_S390 writes a System/390 format floating point number. Note that the range of IEEE doubles is greater than the range of S/390® double precision floating point numbers, and so very large numbers cannot be converted.
writeFloat

public void writeFloat(float v)

Throws IOException.

Write a float into the message buffer at the current position. The value of the encoding member variable determines the behavior of this method.

Values of MQC.MQENC_FLOAT_IEEE_NORMAL and MQC.MQENC_FLOAT_IEEE_REVERSED write IEEE standard floats in big-endian and little-endian formats respectively.

A value of MQC.MQENC_FLOAT_S390 will write a System/390 format floating point number.

writeInt

public void writeInt(int v)

Throws IOException.

Write an integer into the message buffer at the current position. The value of the encoding member variable determines the behavior of this method.

A value of MQC.MQENC_INTEGER_NORMAL writes a big-endian integer, a value of MQC.MQENC_INTEGER_REVERSED writes a little-endian integer.

writeInt2

public void writeInt2(int v)

Throws IOException.

Synonym for writeShort(), provided for cross-language WebSphere MQ API compatibility.

writeInt4

public void writeInt4(int v)

Throws IOException.

Synonym for writeInt(), provided for cross-language WebSphere MQ API compatibility.

writeInt8

public void writeInt8(long v)

Throws IOException.

Synonym for writeLong(), provided for cross-language WebSphere MQ API compatibility.

writeLong

public void writeLong(long v)

Throws IOException.
Write a long into the message buffer at the current position. The value of the encoding member variable determines the behavior of this method.

A value of MQC.MQENC_INTEGER_NORMAL writes a big-endian long, a value of MQC.MQENC_INTEGER_REVERSED writes a little-endian long.

writeObject

```
public void writeObject(Object obj)
```

Throws IOException.

Write the specified object to the message buffer. The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all its supertypes are all written.

writeShort

```
public void writeShort(int v)
```

Throws IOException.

Write a short into the message buffer at the current position. The value of the encoding member variable determines the behavior of this method.

A value of MQC.MQENC_INTEGER_NORMAL writes a big-endian short, a value of MQC.MQENC_INTEGER_REVERSED writes a little-endian short.

writeString

```
public void writeString(String str)
```

Throws IOException.

Write a string into the message buffer at the current position, converting it to the codeset identified by the characterSet member variable.

writeUTF

```
public void writeUTF(String str)
```

Throws IOException.

Write a UTF string, prefixed by a 2-byte length field, into the message buffer at the current position.
MQMessageTracker

```java
java.lang.Object
└─ com.ibm.mq.MQMessageTracker
```

public abstract class MQMessageTracker extends Object

**Note:** You can use this class only when connected to a WebSphere MQ Version 5 (or higher) queue manager.

This class is inherited by MQDistributionListItem (on page 103) where it is used to tailor message parameters for a given destination in a distribution list.

**Variables**

**accountingToken**

```java
public byte accountingToken[]
```

This is part of the identity context of the message. It allows an application to cause work done as a result of the message to be appropriately charged.

The default value is MQC.MQACT_NONE.

**correlationId**

```java
public byte correlationId[]
```

This specifies the correlation identifier to use when the message is put.

The default value is MQC.MQCI_NONE.

**feedback**

```java
public int feedback
```

This is used with a message of type MQC.MQMT_REPORT to indicate the nature of the report. The following feedback codes are defined by the system:

- MQC.MQFB_BUFFER_OVERFLOW
- MQC.MQFB_COA
- MQC.MQFB_COD
- MQC.MQFB_DATA_LENGTH_NEGATIVE
- MQC.MQFB_DATA_LENGTH_TOO_BIG
- MQC.MQFB_DATA_LENGTH_ZERO
- MQC.MQFB_EXPIRATION
- MQC.MQFB_IIH_ERROR
- MQC.MQFB_LENGTH_OFF_BY_ONE
- MQC.MQFB_NAN
- MQC.MQFB_NONE
- MQC.MQFB_PAN
- MQC.MQFB_QUIT

Application defined feedback values in the range MQC.MQFB_APPL_FIRST to MQC.MQFB_APPL_LAST can also be used.

The default value of this field is MQC.MQFB_NONE, indicating that no feedback is provided.

**groupId**

```java
public byte[] groupId
```
**MQMessageTracker**

A byte string that identifies the message group to which the physical message belongs.

The default value is MQC.MQGI_NONE.

**messageId**

```java
public byte messageId[]
```

This specifies the message identifier to use when the message is put. If MQC.MQMI_NONE is specified, the queue manager generates a unique message identifier when the message is put. The value of this member variable is updated after the put to indicate the message identifier that was used.

The default value is MQC.MQMI_NONE.
public class MQPoolServices extends Object

Note: Normally, applications do not use this class.

The MQPoolServices class can be used by implementations of ConnectionManager that are intended for use as the default ConnectionManager for WebSphere MQ connections.

A ConnectionManager can construct an MQPoolServices object and, through it, register a listener. This listener receives events that relate to the set of MQPoolTokens that MQEnvironment manages. The ConnectionManager can use this information to perform any necessary startup or cleanup work.

See also "MQPoolServicesEvent" on page 140 and "MQPoolServicesEventListener" on page 172.

Constructors

MQPoolServices
    public MQPoolServices()

    Construct a new MQPoolServices object.

Methods

addMQPoolServicesEventListener
    public void addMQPoolServicesEventListener
        (MQPoolServicesEventListener listener)

    Add an MQPoolServicesEventListener. The listener receives an event whenever a token is added or removed from the set of MQPoolTokens that MQEnvironment controls, or whenever the default ConnectionManager changes.

getTokenCount
    public int getTokenCount()

    Returns the number of MQPoolTokens that are currently registered with MQEnvironment.

removeMQPoolServicesEventListener
    public void removeMQPoolServicesEventListener
        (MQPoolServicesEventListener listener)

    Remove an MQPoolServicesEventListener.
MQPoolServicesEvent

Note: Normally, applications do not use this class.

An MQPoolServicesEvent is generated whenever an MQPoolToken is added to, or removed from, the set of tokens that MQEnvironment controls. An event is also generated when the default ConnectionManager is changed.

See also “MQPoolServices” on page 139 and “MQPoolServicesEventListener” on page 172.

Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_POOL_CHANGED</td>
<td>The event ID used when the default ConnectionManager changes.</td>
</tr>
<tr>
<td>ID</td>
<td>The event ID. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>TOKEN_ADDED, TOKEN_REMOVED, DEFAULT_POOL_CHANGED</td>
</tr>
<tr>
<td>TOKEN_ADDED</td>
<td>The event ID used when an MQPoolToken is added to the set.</td>
</tr>
<tr>
<td>TOKEN_REMOVED</td>
<td>The event ID used when an MQPoolToken is removed from the set.</td>
</tr>
<tr>
<td>token</td>
<td>The token. When the event ID is DEFAULT_POOL_CHANGED, this is null.</td>
</tr>
</tbody>
</table>

Constructors

MQPoolServicesEvent

public MQPoolServicesEvent(Object source, int eid)

Constructs an MQPoolServicesEvent based on the event ID.

MQPoolServicesEvent

public MQPoolServicesEvent(Object source, int eid, MQPoolToken token)

Constructs an MQPoolServicesEvent based on the event ID and the token.
Methods

**getId**

```java
public int getId()
```

Gets the event ID.

**Returns**

The event ID, with one of the following values:

- DEFAULT_POOL_CHANGED
- TOKEN_ADDED
- TOKEN_REMOVED

**getToken**

```java
public MQPoolToken getToken()
```

Returns the token that was added to, or removed from, the set. If the event ID is DEFAULT_POOL_CHANGED, this is null.
public class MQPoolToken extends Object

An MQPoolToken can be used to enable the default connection pool. MQPoolTokens are registered with the MQEnvironment class before an application component connects to WebSphere MQ. Later, they are deregistered when the component has finished using WebSphere MQ. Typically, the default ConnectionManager is active while the set of registered MQPoolTokens is not empty.

MQPoolToken provides no methods or variables. ConnectionManager providers can choose to extend MQPoolToken so that hints can be passed to the ConnectionManager.

See “MQEnvironment.addConnectionPoolToken” on page 108 and “MQEnvironment.removeConnectionPoolToken” on page 109.

Constructors

MQPoolToken

    public MQPoolToken()

    Construct a new MQPoolToken object.
MQProcess

public class MQProcess extends MQManagedObject. (See page 116.)

MQProcess provides inquire operations for WebSphere MQ processes.

Constructors

MQProcess

public MQProcess(MQQueueManager qMgr,
    String processName,
    int openOptions,
    String queueManagerName,
    String alternateUserId)
    throws MQException

Access a process on the queue manager qMgr. See accessProcess in the “MQQueueManager” on page 156 for details of the remaining parameters.

Methods

close

public synchronized void close()

Throws MQException.

Override of “MQManagedObject.close” on page 117

getApplicationId

public String getApplicationId()

A character string that identifies the application to be started. This information is for use by a trigger monitor application that processes messages on the initiation queue; the information is sent to the initiation queue as part of the trigger message.

Throws MQException if you call this method after you have closed the process.

getApplicationType

public int getApplicationType()

Throws MQException (see page 110).

This identifies the nature of the program to be started in response to the receipt of a trigger message. The application type can take any value, but the following values are recommended for standard types:

• MQC.MQAT_AIX
• MQC.MQAT_CICS
• MQC.MQAT_DOS
• MQC.MQAT_IMS
• MQC.MQAT_MVS
• MQC.MQAT_OS2
MQProcess

- MQC.MQAT_OS400
- MQC.MQAT_UNIX
- MQC.MQAT_WINDOWS
- MQC.MQAT_WINDOWS_NT
- MQC.MWQAT_USER_FIRST (lowest value for user-defined application type)
- MQC.MQAT_USER_LAST (highest value for user-defined application type)

getEnvironmentData

public String getEnvironmentData()

Throws MQException.

A string containing environment-related information pertaining to the application to be started.

getUserData

public String getUserData()

Throws MQException.

A string containing user information relevant to the application to be started.
MQPutMessageOptions

This class contains options that control the behavior of MQQueue.put().

**Variables**

- `contextReference`:
  ```
  public MQQueue ContextReference
  ```
  This is an input field which indicates the source of the context information. If the options field includes MQC.MQPMO_PASS_IDENTITY_CONTEXT, or MQC.MQPMO_PASS_ALLCONTEXT, set this field to refer to the MQQueue from which the context information should be taken.
  The initial value of this field is null.

- `invalidDestCount`:
  ```
  public int invalidDestCount
  ```
  This is an output field set by the queue manager to the number of messages that could not be sent to queues in a distribution list. The count includes queues that failed to open as well as queues that were opened successfully, but for which the put operation failed. This field is also set when opening a single queue that is not part of a distribution list.

- `knownDestCount`:
  ```
  public int knownDestCount
  ```
  This is an output field set by the queue manager to the number of messages that the current call has sent successfully to queues that resolve to local queues. This field is also set when opening a single queue that is not part of a distribution list.

- `options`:
  ```
  public int options
  ```
  Options that control the action of MQQueue.put. Any or none of the following values can be specified. If more than one option is required the values can be added together or combined using the bitwise OR operator.

  - **MQC.MQPMO_DEFAULT_CONTEXT**
    Associate default context with the message.

  - **MQC.MQPMO_FAIL_IF_QUIESCING**
    Fail if the queue manager is quiescing.

  - **MQC.MQPMO_LOGICAL_ORDER**
    Put logical messages and segments in message groups into their logical order.
MQPutMessageOptions

MQC.MQPMO_NEW_CORREL_ID*
   Generate a new correlation id for each sent message.

MQC.MQPMO_NEW_MSG_ID*
   Generate a new message id for each sent message.

MQC.MQPMO_NONE
   No options specified. Do not use in conjunction with other options.

MQC.MQPMO_NO_CONTEXT
   No context is to be associated with the message.

MQC.MQPMO_NO_SYNCPOINT
   Put a message without syncpoint control. Note that, if the
   syncpoint control option is not specified, a default of ‘no
   syncpoint’ is assumed. This applies for all supported platforms,
   including z/OS.

MQC.MQPMO_PASS_ALL_CONTEXT
   Pass all context from an input queue handle.

MQC.MQPMO_PASS_IDENTITY_CONTEXT
   Pass identity context from an input queue handle.

MQC.MQPMO_SET_ALL_CONTEXT
   Set all context from the application.

MQC.MQPMO_SET_IDENTITY_CONTEXT
   Set identity context from the application.

MQC.MQPMO_SYNCPOINT
   Put a message with syncpoint control. The message is not visible
   outside the unit of work until the unit of work is committed. If the
   unit of work is backed out, the message is deleted.

recordFields *

public int recordFields
   Flags indicating which fields are to be customized on a per-queue basis
   when putting a message to a distribution list. One or more of the following
   flags can be specified:

MQC.MQPMRF_ACCOUNTING_TOKEN
   Use the accountingToken attribute in the MQDistributionListItem.

MQC.MQPMRF_CORREL_ID
   Use the correlationId attribute in the MQDistributionListItem.

MQC.MQPMRF_FEEDBACK
   Use the feedback attribute in the MQDistributionListItem.

MQC.MQPMRF_GROUP_ID
   Use the groupId attribute in the MQDistributionListItem.

MQC.MQPMRF_MSG_ID
   Use the messageId attribute in the MQDistributionListItem.

   The special value MQC.MQPMRF_NONE indicates that no fields are to be
   customized.

resolvedQueueManagerName

public String resolvedQueueManagerName
   This is an output field set by the queue manager to the name of the queue
   manager that owns the queue specified by the remote queue name. This
resolvedQueueName

`public String resolvedQueueName`

This is an output field that is set by the queue manager to the name of the queue on which the message is placed. This may be different from the name used to open the queue if the opened queue was an alias or model queue.

unknownDestCount *

`public int unknownDestCount`

This is an output field set by the queue manager to the number of messages that the current call has sent successfully to queues that resolve to remote queues. This field is also set when opening a single queue that is not part of a distribution list.

Constructors

**MQPutMessageOptions**

`public MQPutMessageOptions()`

Construct a new MQPutMessageOptions object with no options set, and a blank resolvedQueueName and resolvedQueueManagerName.
MQQueue

MQQueue

```
java.lang.Object
    └─ com.ibm.mq.MQManagedObject
        └─ com.ibm.mq.MQQueue
```

```java
public class MQQueue
    extends MQManagedObject. (See page 116.)

MQQueue provides inquire, set, put, and get operations for WebSphere MQ
queues. The inquire and set capabilities are inherited from MQ.MQManagedObject.

See also “MQQueueManager.accessQueue” on page 161

Constructors

MQQueue

```java
public MQQueue(MQQueueManager qMgr, String queueName, int openOptions,
                String queueManagerName, String dynamicQueueName,
                String alternateUserId )
    throws MQException
```

Access a queue on the queue manager qMgr.

See “MQQueueManager.accessQueue” on page 161 for details of the
remaining parameters.

Methods

close

```java
public synchronized void close()
```

Throws MQException.

Override of “MQManagedObject.close” on page 117

get

```java
public synchronized void get(MQMessage message,
                             MQGetMessageOptions getMessageOptions,
                             int MaxMsgSize)
```

Throws MQException.

Retrieves a message from the queue, up to a maximum specified message
size.

This method takes an MQMessage object as a parameter. It uses some of
the fields in the object as input parameters - in particular the messageId
and correlationId, so it is important to ensure that these are set as required.
(See “Message” on page 305)

If the get fails, the MQMessage object is unchanged. If it succeeds the
message descriptor (member variables) and message data portions of the
MQMessage are completely replaced with the message descriptor and
message data from the incoming message.
Note that all calls to WebSphere MQ from a given MQQueueManager are synchronous. Therefore, if you perform a get with wait, all other threads using the same MQQueueManager are blocked from making further WebSphere MQ calls until the get completes. If you need multiple threads to access WebSphere MQ simultaneously, each thread must create its own MQQueueManager object.

**Parameters**

*message*

An input/output parameter containing the message descriptor information and the returned message data.

*getMessageOptions*

Options controlling the action of the get. (See “MQGetMessageOptions” on page 112.)

Using option MQC.MQGMO_CONVERT may result in an exception with reason code MQException.MQRC_CONVERTED_STRING_TOO_BIG when converting from single byte character codes to double byte codes. In this case, the message will be copied into the buffer but will remain encoded using its original character set.

*MaxMsgSize*

The largest message this call will be able to receive. If the message on the queue is larger than this size, one of two things can occur:

1. If the MQC.MQGMO_ACCEPT_TRUNCATED_MSG flag is set in the options member variable of the MQGetMessageOptions object, the message is filled with as much of the message data as will fit in the specified buffer size, and an exception is thrown with completion code MQException.MQCC_WARNING and reason code MQException.MQRC_TRUNCATED_MSG_ACCEPTED.

2. If the MQC.MQGMO_ACCEPT_TRUNCATED_MSG flag is not set, the message is left on the queue and an MQException is raised with completion code MQException.MQCC_WARNING and reason code MQException.MQRC_TRUNCATED_MSG_FAILED.

Throws MQException if the get fails.

```java
public synchronized void get(MQMessage message,
MQGetMessageOptions getMessageOptions)

Throws MQException.
```

Retrieves a message from the queue, regardless of the size of the message. For large messages, the get method may have to issue two calls to WebSphere MQ on your behalf, one to establish the required buffer size and one to get the message data itself.

This method takes an MQMessage object as a parameter. It uses some of the fields in the object as input parameters - in particular the messageId and correlationId, so it is important to ensure that these are set as required. (See “Message” on page 305)
If the get fails, the MQMessage object is unchanged. If it succeeds, the message descriptor (member variables) and message data portions of the MQMessage are completely replaced with the message descriptor and message data from the incoming message.

Note that all calls to WebSphere MQ from a given MQQueueManager are synchronous. Therefore, if you perform a get with wait, all other threads using the same MQQueueManager are blocked from making further WebSphere MQ calls until the get completes. If you need multiple threads to access WebSphere MQ simultaneously, each thread must create its own MQQueueManager object.

**Parameters**

*message*

An input/output parameter containing the message descriptor information and the returned message data.

*getMessageOptions*

Options controlling the action of the get. (See "MQGetMessageOptions" on page 112 for details.)

Throws MQException if the get fails.

**get**

```
public synchronized void get(MQMessage message)
```

This is a simplified version of the get method previously described.

**Parameters**

*MQMessage*

An input/output parameter containing the message descriptor information and the returned message data.

This method uses a default instance of MQGetMessageOptions to do the get. The message option used is MQGMO_NOWAIT.

**getCreationDateTime**

```
public GregorianCalendar getCreationDateTime()
```

Throws MQException.

The date and time that this queue was created.

**getQueueType**

```
public int getQueueType()
```

Throws MQException

**Returns**

The type of this queue with one of the following values:

- MQC.MQQT_ALIAS
- MQC.MQQT_LOCAL
- MQC.MQQT_REMOTE
- MQC.MQQT_CLUSTER

**getCurrentDepth**

```
public int getCurrentDepth()
```
Throws MQException.

Get the number of messages currently on the queue. This value is incremented during a put call, and during backout of a get call. It is decremented during a non-browse get and during backout of a put call.

**getDefinitionType**

```java
public int getDefinitionType()
```

Throws MQException.

Indicates how the queue was defined.

**Returns**

One of the following:

- MQ.C.MQQDT_PREDEFINED
- MQ.C.MQQDT_PERMANENT_DYNAMIC
- MQ.C.MQQDT_TEMPORARY_DYNAMIC

**getInhibitGet**

```java
public int getInhibitGet()
```

Throws MQException.

Indicates whether or not get operations are allowed for this queue.

**Returns**

The possible values are:

- MQ.C.MQQA_GET_INHIBITED
- MQ.C.MQQA_GET_ALLOWED

**getInhibitPut**

```java
public int getInhibitPut()
```

Throws MQException.

Indicates whether or not put operations are allowed for this queue.

**Returns**

One of the following:

- MQ.C.MQQA_PUT_INHIBITED
- MQ.C.MQQA_PUT_ALLOWED

**getMaximumDepth**

```java
public int getMaximumDepth()
```

Throws MQException.

The maximum number of messages that can exist on the queue at any one time. An attempt to put a message to a queue that already contains this many messages fails with reason code MQException.MQRC_Q_FULL.

**getMaximumMessageLength**

```java
public int getMaximumMessageLength()
```

Throws MQException.
This is the maximum length of the application data that can exist in each message on this queue. An attempt to put a message larger than this value fails with reason code MQException.MQRC_MSG_TOO_BIG_FOR_Q.

**getOpenInputCount**

```java
public int getOpenInputCount()
```

Throws MQException.

The number of handles that are currently valid for removing messages from the queue. This is the total number of such handles known to the local queue manager, not just those created by the WebSphere MQ classes for Java (using accessQueue).

**getOpenOutputCount**

```java
public int getOpenOutputCount()
```

Throws MQException.

The number of handles that are currently valid for adding messages to the queue. This is the total number of such handles known to the local queue manager, not just those created by the WebSphere MQ classes for Java (using accessQueue).

**getShareability**

```java
public int getShareability()
```

Throws MQException.

Indicates whether the queue can be opened for input multiple times.

**Returns**

One of the following:

- MQC.MQQA_SHAREABLE
- MQC.MQQA_NOT_SHAREABLE

**getTriggerControl**

```java
public int getTriggerControl()
```

Throws MQException.

Indicates whether or not trigger messages are written to an initiation queue, in order to cause an application to be started to service the queue.

**Returns**

The possible values are:

- MQC.MQTC_OFF
- MQC.MQTC_ON

**getTriggerData**

```java
public String getTriggerData()
```

Throws MQException.

The free-format data that the queue manager inserts into the trigger message when a message arriving on this queue causes a trigger message to be written to the initiation queue.

**getTriggerDepth**
public int getTriggerDepth()

Throws MQException.

The number of messages that have to be on the queue before a trigger message is written when trigger type is set to MQC.MQTT_DEPTH.

getTriggerMessagePriority
public int getTriggerMessagePriority()

Throws MQException.

This is the message priority below which messages do not contribute to the generation of trigger messages (that is, the queue manager ignores these messages when deciding whether a trigger should be generated). A value of zero causes all messages to contribute to the generation of trigger messages.

getTriggerType
public int getTriggerType()

Throws MQException.

The conditions under which trigger messages are written as a result of messages arriving on this queue.

Returns
The possible values are:
- MQC.MQTT_NONE
- MQC.MQTT_FIRST
- MQC.MQTT_EVERY
- MQC.MQTT_DEPTH

put
public synchronized void put(MQMessage message,
MQPutMessageOptions putMessageOptions)

Throws MQException.

Places a message onto the queue.

Note: For simplicity and performance, if you want to put just a single message to a queue, use the put() method on your MQQueueManager object. For this you do not need to have an MQQueue object. See “MQQueueManager.put” on page 167.

This method takes an MQMessage object as a parameter. The message descriptor properties of this object may be altered as a result of this method. The values they have immediately after the completion of this method are the values that were put onto the WebSphere MQ queue.

Modifications to the MQMessage object after the put has completed do not affect the actual message on the WebSphere MQ queue.

A put updates the messageId and correlationId. This must be considered when making further calls to put/get using the same MQMessage object. Also, calling put does not clear the message data, so:
MQQueue

msg.writeString("a");
q.put(msg,pmo);
msg.writeString("b");
q.put(msg,pmo);

puts two messages. The first contains “a” and the second “ab”.

**Parameters**

*message*
Message Buffer containing the Message Descriptor data and message to be sent.

*putMessageOptions*
Options controlling the action of the put. (See “MQPutMessageOptions” on page 145)

Throws MQException if the put fails.

**put**

`public synchronized void put(MQMessage message)`

This is a simplified version of the put method previously described.

**Parameters**

*MQMessage*
Message Buffer containing the Message Descriptor data and message to be sent.

This method uses a default instance of MQPutMessageOptions to do the put.

**setInhibitGet**

`public void setInhibitGet(int inhibit)`

Throws MQException.

Controls whether or not get operations are allowed for this queue. The permissible values are:
- MQC.MQQA_GET_INHIBITED
- MQC.MQQA_GET_ALLOWED

**setInhibitPut**

`public void setInhibitPut(int inhibit)`

Throws MQException.

Controls whether or not put operations are allowed for this queue. The permissible values are:
- MQC.MQQA_PUT_INHIBITED
- MQC.MQQA_PUT_ALLOWED

**setTriggerControl**

`public void setTriggerControl(int trigger)`

Throws MQException.
Controls whether or not trigger messages are written to an initiation queue, in order to cause an application to be started to service the queue. The permissible values are:
• MQC.MQTC_OFF
• MQC.MQTC_ON

**setTriggerData**

```java
public void setTriggerData(String data)
```

Throws MQException.

Sets the free-format data that the queue manager inserts into the trigger message when a message arriving on this queue causes a trigger message to be written to the initiation queue. The maximum permissible length of the string is given by MQC.MQ_TRIGGER_DATA_LENGTH.

**setTriggerDepth**

```java
public void setTriggerDepth(int depth)
```

Throws MQException.

Sets the number of messages that have to be on the queue before a trigger message is written when trigger type is set to MQC.MQTT_DEPTH.

**setTriggerMessagePriority**

```java
public void setTriggerMessagePriority(int priority)
```

Throws MQException.

Sets the message priority below which messages do not contribute to the generation of trigger messages (that is, the queue manager ignores these messages when deciding whether a trigger should be generated). A value of zero causes all messages to contribute to the generation of trigger messages.

**setTriggerType**

```java
public void setTriggerType(int type)
```

Throws MQException.

Sets the conditions under which trigger messages are written as a result of messages arriving on this queue. The possible values are:
• MQC.MQTT_NONE
• MQC.MQTT_FIRST
• MQC.MQTT_EVERY
• MQC.MQTT_DEPTH
public class MQQueueManager extends MQManagedObject. (See page 116.)

Note: The behavior of some of the options available in this class depends on the environment in which they are used. These elements are marked with an asterisk (*). See Chapter 8, “Environment-dependent behavior” on page 89 for details.

Variables

isConnected
	public boolean isConnected

True if the connection to the queue manager is still open.

Constructors

MQQueueManager
	public MQQueueManager(String queueManagerName)

Throws MQException.

Create a connection to the named queue manager.

Note: When using WebSphere MQ classes for Java, the hostname, channel name and port to use during the connection request are specified in the MQEnvironment class. This must be done before calling this constructor.

The following example shows a connection to a queue manager MYQM, running on a machine with hostname fred.mq.com.

```java
MQEnvironment.hostname = "fred.mq.com";  // host to connect to
MQEnvironment.port    = 1414;            // port to connect to.
                        // If I don't set this, // it defaults to 1414
                        // (the default WebSphere MQ port)
MQEnvironment.channel = "channel.name"; // the CASE-SENSITIVE
                      // name of the
                      // SVR CONN channel on
                      // the queue manager
MQQueueManager qMgr    = new MQQueueManager("MYQM");
```

If the queue manager name is left blank (null or ""), a connection is made to the default queue manager.

See also "MQEnvironment” on page 104
MQQueueManager

public MQQueueManager(String queueManagerName,
MQConnectionManager cxManager)

Throws MQException.

This constructor connects to the specified Queue Manager, using the
properties in MQEnvironment. The specified MQConnectionManager
manages the connection.

MQQueueManager

public MQQueueManager(String queueManagerName,
ConnectionManager cxManager)

Throws MQException.

This constructor connects to the specified Queue Manager, using the
properties in MQEnvironment. The specified ConnectionManager manages
the connection.

This constructor requires a JVM at at least Java 2 v1.3, with at least JAAS
1.0 installed.

MQQueueManager

public MQQueueManager(String queueManagerName,
int options)

Throws MQException.

This version of the constructor is intended for use only in bindings mode
and it uses the extended connection API (MQCONNX) to connect to the
queue manager. The options parameter allows you to choose fast or normal
bindings. Possible values are:
• MQC.MQCNO_FASTPATH_BINDING for fast bindings *.
• MQC.MQCNO_STANDARD_BINDING for normal bindings.

MQQueueManager

public MQQueueManager(String queueManagerName,
int options,
MQConnectionManager cxManager)

Throws MQException.

This constructor performs an MQCONNX, passing the supplied options.
The specified MQConnectionManager manages the connection.

MQQueueManager

public MQQueueManager(String queueManagerName,
int options,
ConnectionManager cxManager)

Throws MQException.

This constructor performs an MQCONNX, passing the supplied options.
The specified ConnectionManager manages the connection.

This constructor requires a JVM at at least Java 2 v1.3, with at least JAAS
1.0 installed.
MQQueueManager

public MQQueueManager(String queueManagerName,
java.util.Hashtable properties)

The properties parameter takes a series of key/value pairs that describe
the WebSphere MQ environment for this particular queue manager. These
properties, where specified, override the values set by the MQEnvironment
class, and allow the individual properties to be set on a queue manager by
queue manager basis. See “MQEnvironment.properties” on page 105.

MQQueueManager

public MQQueueManager(String queueManagerName,
Hashtable properties,
MQConnectionManager cxManager)

Throws MQException.

This constructor connects to the named Queue Manager, using the
supplied Hashtable of properties to override those in MQEnvironment. The
specified MQConnectionManager manages the connection.

MQQueueManager

public MQQueueManager(String queueManagerName,
Hashtable properties,
ConnectionManager cxManager)

Throws MQException.

This constructor connects to the named Queue Manager, using the
supplied Hashtable of properties to override those in MQEnvironment. The
specified ConnectionManager manages the connection.

This constructor requires a JVM at at least Java 2 v1.3, with at least JAAS
1.0 installed.
accessDistributionList

```java
public synchronized MQDistributionList accessDistributionList(
    MQDistributionListItem[] litems, int openOptions,
    String alternateUserId
)
```

Throws MQException.

**Parameters**

- **litems**  
  The items to be included in the distribution list.

- **openOptions**  
  Options that control the opening of the distribution list.

- **alternateUserId**  
  If MQOO_ALTERNATE_USER_AUTHORITY is specified in the openOptions parameter, this parameter specifies the alternate user identifier that is used to check the authorization for the open. If MQOO_ALTERNATE_USER_AUTHORITY is not specified, this parameter can be left blank (or null).

**Returns**

A newly created MQDistributionList which is open and ready for put operations.

Throws MQException if the open fails.

See also [“MQQueueManager.accessQueue” on page 161](#).

accessDistributionList

This is a simplified version of the AccessDistributionList method previously described.

```java
public synchronized MQDistributionList accessDistributionList(
    MQDistributionListItem[] litems, int openOptions,
)
```

**Parameters**

- **litems**  
  The items to be included in the distribution list.

- **openOptions**  
  Options that control the opening of the distribution list.

See [“accessDistributionList”](#) for details of the parameters.

**alternateUserId** is set to "".
accessProcess

    public synchronized MQProcess accessProcess
    {
        String processName,
        int openOptions,
        String queueManagerName,
        String alternateUserId
    }

Throws MQException.

Establishes access to a WebSphere MQ process on this queue manager to inquire about the process attributes.

Parameters

processName
    Name of process to open.

openOptions
    Options that control the opening of the process. Inquire is automatically added to the options specified, so there is no need to specify it explicitly.

Valid options are:

MQC.MQOO_ALTERNATE_USER_AUTHORITY
    Validate with the specified user id

MQC.MQOO_FAIL_IF_QUIESCING
    Fail if the queue manager is quiescing

If more than one option is required, the values can be added together or combined using the bitwise OR operator. See the WebSphere MQ Application Programming Reference for a fuller description of these options.

queueManagerName
    Name of the queue manager on which the process is defined. Applications should leave this parameter blank or null.

alternateUserId
    If MQOO_ALTERNATE_USER_AUTHORITY is specified in the openOptions parameter, this parameter specifies the alternate user identifier that is used to check the authorization for the open. If MQOO_ALTERNATE_USER_AUTHORITY is not specified, this parameter can be left blank (or null).

accessProcess
    This is a simplified version of the AccessProcess method previously described.

    public synchronized MQProcess accessProcess
    {
        String processName,
        int openOptions
    }
Parameters

processName
The name of the process to open.

openOptions
Options that control the opening of the process.

See “accessProcess” on page 160 for details of the options.

queueManagerName and alternateUserId are set to "".

accessQueue

public synchronized MQQueue accessQueue
(String queueName, int openOptions,
String queueManagerName,
String dynamicQueueName,
String alternateUserId)

Throws MQException.

Establishes access to a WebSphere MQ queue on this queue manager to get or browse messages, put messages, inquire about the attributes of the queue or set the attributes of the queue.

If the queue named is a model queue, then a dynamic local queue is created. The name of the created queue can be determined by inspecting the name attribute of the returned MQQueue object.

Parameters

queueName
Name of queue to open.

openOptions
Options that control the opening of the queue. Valid options are:

MQC.MQOO_ALTERNATE_USER_AUTHORITY
Validate with the specified user identifier.

MQC.MQOO_BIND_AS_Q_DEF
Use default binding for queue.

MQC.MQOO_BIND_NOT_FIXED
Do not bind to a specific destination.

MQC.MQOO_BIND_ON_OPEN
Bind handle to destination when queue is opened.

MQC.MQOO_BROWSE
Open to browse message.

MQC.MQOO_FAIL_IF_QUIESCING
Fail if the queue manager is quiescing.

MQC.MQOO_INPUT_AS_Q_DEF
Open to get messages using queue-defined default.

MQC.MQOO_INPUT_SHARED
Open to get messages with shared access.
MQQueueManager

MQC.MQOO_INPUT_EXCLUSIVE
Open to get messages with exclusive access.

MQC.MQOO_INQUIRE
Open for inquiry - required if you wish to query properties.

MQC.MQOO_OUTPUT
Open to put messages.

MQC.MQOO_PASS_ALLCONTEXT
Allow all context to be passed.

MQC.MQOO_PASS_IDENTITYCONTEXT
Allow identity context to be passed.

MQC.MQOO_SAVE_ALL_CONTEXT
Save context when message retrieved\(^*\).

MQC.MQOO_SET
Open to set attributes.

MQC.MQOO_SET_ALL_CONTEXT
Allows all context to be set.

MQC.MQOO_SET_IDENTITY_CONTEXT
 Allows identity context to be set.

If more than one option is required, the values can be added together or combined using the bitwise OR operator. See [WebSphere MQ Application Programming Reference](#) for a fuller description of these options.

queueManagerName
Name of the queue manager on which the queue is defined. A name which is entirely blank, or which is null, denotes the queue manager to which this MQQueueManager object is connected.

dynamicQueueName
This parameter is ignored unless queueName specifies the name of a model queue. If it does, this parameter specifies the name of the dynamic queue to be created. A blank or null name is not valid if queueName specifies the name of a model queue. If the last non-blank character in the name is an asterisk (*), the queue manager replaces the asterisk with a string of characters that guarantees that the name generated for the queue is unique on this queue manager.

alternateUserId
If MQOO_ALTERNATE_USER_AUTHORITY is specified in the openOptions parameter, this parameter specifies the alternate user identifier that is used to check the authorization for the open. If MQOO_ALTERNATE_USER_AUTHORITY is not specified, this parameter can be left blank (or null).

Returns
MQQueue that has been successfully opened.

Throws MQException if the open fails.
accessQueue

public synchronized MQQueue accessQueue
(
        String queueName,
        int openOptions
)

Throws MQException if you call this method after disconnecting from the queue manager.

Parameters

queueName
    Name of queue to open

openOptions
    Options that control the opening of the queue

See "MQQueueManager.accessQueue" on page 161 for details of the parameters.

For this version of the method, queueManagerName, dynamicQueueName, and alternateUserId are set to "".

Returns

MQProcess that has been successfully opened.

backout

public synchronized void backout()

Throws MQException.

Calling this method indicates to the queue manager that all the message gets and puts that have occurred since the last syncpoint are to be backed out. Messages put as part of a unit of work (with the MQC.MQPMO_SYNCPOINT flag set in the options field of MQPutMessageOptions) are deleted; messages retrieved as part of a unit of work (with the MQC.MQGMO_SYNCPOINT flag set in the options field of MQGetMessageOptions) are reinstated on the queue.

See also the description of the "commit" method.

begin* (bindings connection only)

public synchronized void begin()

Throws MQException.

This method is supported only by the WebSphere MQ classes for Java in bindings mode and it signals to the queue manager that a new unit of work is starting.

Do not use this method for applications that use local one-phase transactions.
MQQueueManager

commit

public synchronized void commit()

Throws MQException.

Calling this method indicates to the queue manager that the application has reached a syncpoint, and that all of the message gets and puts that have occurred since the last syncpoint are to be made permanent. Messages put as part of a unit of work (with the MQC.MQPMO_SYNCPOINT flag set in the options field of MQPutMessageOptions) are made available to other applications. Messages retrieved as part of a unit of work (with the MQC.MQGMO_SYNCPOINT flag set in the options field of MQGetMessageOptions) are deleted.

See also the description of the “backout” method.

disconnect

public synchronized void disconnect()

Throws MQException.

Terminates the connection to the queue manager. All open queues and processes accessed by this queue manager are closed, and hence become unusable. When you have disconnected from a queue manager the only way to reconnect is to create a new MQQueueManager object.

Normally, any work performed as part of a unit of work is committed. However, if this connection is managed by a ConnectionManager, rather than an MQConnectionManager, the unit of work might be rolled back.

getCharacterSet

public int getCharacterSet()

Throws MQException.

Returns the CCSID (Coded Character Set Identifier) of the queue manager’s codeset. This defines the character set used by the queue manager for all character string fields in the application programming interface.

Throws MQException if you call this method after disconnecting from the queue manager.

getCommandInputQueueName

public String getCommandInputQueueName()

Throws MQException.

Returns the name of the command input queue defined on the queue manager. This is a queue to which applications can send commands, if authorized to do so.

Throws MQException if you call this method after disconnecting from the queue manager.
getCommandLevel

public int getCommandLevel()

Throws MQException.

Indicates the level of system control commands supported by the queue manager. The set of system control commands that correspond to a particular command level varies according to the architecture of the platform on which the queue manager is running. See the WebSphere MQ documentation for your platform for further details.

Throws MQException if you call this method after disconnecting from the queue manager.

Returns

One of the MQC.MQCMDL_LEVEL_xxx constants

getDistributionListCapable

public boolean getDistributionListCapable()

Indicates whether the queue manager supports distribution lists.

getJDBCConnection

public java.sql.Connection getJDBCConnection(XADataSource dataSource,
  String userid, String password)
  throws MQException, SQLException, Exception

Returns a Connection object for use with the JTA-JDBC support.

This method declares Exception in its throws clause so as to avoid problems with the JVM verifier for customers who are not using the JTA functionality. The actual exception thrown is javax.transaction.xa.XAException which would require the jta.jar file to be added to the classpath for programs which didn’t previously require it.

Parameters

dataSource

A database-specific implementation of the XADataSource interface which defines the details of the database to connect to. See the documentation for your database to determine how to create an appropriate XADataSource object to pass into getJDBCConnection.

userid

The userid to use for this connection to the database. This is passed to the underlying XADataSource.getXAConnection method.

password

The password to use for this connection to the database. This is passed to the underlying XADataSource.getXAConnection method.
getJDBCConnection

public java.sql.Connection getJDBCConnection(javax.sql.XADataSource xads)
throws MQException, SQLException, Exception

Returns a Connection object for use with the JTA-JDBC support.

This method declares Exception in its throws clause so as to avoid problems with the JVM verifier for customers who are not using the JTA functionality. The actual exception thrown is javax.transaction.xa.XAException which would require the jta.jar file to be added to the classpath for programs which didn’t previously require it.

Parameters

xads A database-specific implementation of the XADataSource interface which defines the details of the database to connect to. See the documentation for your database to determine how to create an appropriate XADataSource object to pass into getJDBCConnection.

getMaximumMessageLength

public int getMaximumMessageLength()

Throws MQException.

Returns the maximum length of a message (in bytes) that can be handled by the queue manager. No queue can be defined with a maximum message length greater than this.

Throws MQException if you call this method after disconnecting from the queue manager.

getMaximumPriority

public int getMaximumPriority()

Throws MQException.

Returns the maximum message priority supported by the queue manager. Priorities range from zero (lowest) to this value.

Throws MQException if you call this method after disconnecting from the queue manager.

getSyncpointAvailability

public int getSyncpointAvailability()

Throws MQException.

Indicates whether the queue manager supports units of work and syncpointing with the MQQueue.get and MQQueue.put methods.
Returns

- MQC.MQSP_AVAILABLE if syncpointing is available.
- MQC.MQSP_NOT_AVAILABLE if syncpointing is not available.

Throws MQException if you call this method after disconnecting from the queue manager.

isConnected

```java
public boolean isConnected()
```

Returns the value of the isConnected variable.

put

```java
public synchronized void put(String qName,
    String qmName,
    MQMessage msg,
    MQPutMessageOptions pmo,
    String altUserId)
    throws MQException
```

Places a single message onto a queue without having to create a MQQueue object first.

The qName (queue name) and qmName (queue manager name) parameters identify where the message is placed. If the queue is a model queue then an MQException is thrown.

In other respects, this method behaves like the put method on the MQQueue object. It is an implementation of the MQPUT1 MQI call. See “MQQueue.put” on page 153.

Parameters

- **qName** The name of the queue onto which the message should be placed.
- **qmName** The name of the queue manager on which the queue is defined.
- **msg** The message to send.
- **pmo** Options controlling the actions of the put. See “MQPutMessageOptions” on page 145 for more details.
- **altUserId** Specifies an alternative user identifier used to check authorization when placing the message on a queue.

```java
public synchronized void put(String qName,
    String qmName,
    MQMessage msg,
    MQPutMessageOptions pmo)
    throws MQException
```

Places a single message onto a queue without having to create a MQQueue object first.

This version of the method allows you to omit the altUserId parameter. See the fully-specified method (MQQueueManager.put) for details of the parameters.
public synchronized void put(String qName,
                       String qmName,
                       MQMessage msg)
                       throws MQException

Places a single message onto a queue without having to create a MQQueue object first.

This version of the method allows you to omit the put message options (pmo) and altUserid parameters. See the fully-specified method (“MQQueueManager.put” on page 167) for details of the parameters.

public synchronized void put(String qName,
                       MQMessage msg,
                       MQPutMessageOptions pmo)
                       throws MQException

Places a single message onto a queue without having to create a MQQueue object first.

This version of the method allows you to omit the qmName and altUserid parameters. See the fully-specified method (“MQQueueManager.put” on page 167) for details of the parameters.

public synchronized void put(String qName,
                       MQMessage msg)
                       throws MQException

Places a single message onto a queue without having to create a MQQueue object first.

This version of the method allows you to omit the qmName, put message options (pmo), and altUserid parameters. See the fully-specified method (“MQQueueManager.put” on page 167) for details of the parameters.
MQSimpleConnectionManager

An MQSimpleConnectionManager provides basic connection pooling functionality. You can use an MQSimpleConnectionManager either as the default Connection Manager, or as a parameter to an MQQueueManager constructor. When an MQQueueManager is constructed, the most-recently-used connection in the pool is used.

Connections are destroyed (by a separate thread) when they are unused for a specified period, or when there are more than a specified number of unused connections in the pool. You can specify the timeout period and the maximum number of unused connections.

**Variables**

- **MODE_ACTIVE**
  
  ```java
  public static final int MODE_ACTIVE
  ```

  See “setActive” on page 170.

- **MODE_AUTO**
  
  ```java
  public static final int MODE_AUTO
  ```

  See “setActive” on page 170.

- **MODE_INACTIVE**
  
  ```java
  public static final int MODE_INACTIVE
  ```

  See “setActive” on page 170.

**Constructors**

```java
MQSimpleConnectionManager
public MQSimpleConnectionManager()
```

Constructs an MQSimpleConnectionManager.

**Methods**

- **getActive**
  
  ```java
  public int getActive()
  ```

  Gets the mode of the connection pool.

  **Returns**
  The current active mode of the connection pool, with one of the following values (see “setActive” on page 170):
  
  - MODE_ACTIVE
  - MODE_AUTO
  - MODE_INACTIVE

- **getHighThreshold**
  
  ```java
  public int getHighThreshold ()
  ```

  Returns the HighThreshold value.

- **getTimeout**
  
  ```java
  public long getTimeout()
  ```
MQSimpleConnectionManager

Returns the Timeout value.

**setActive**

```java
public void setActive(int mode)
```

Sets the active mode of the connection pool.

**Parameters**

- **mode**  
  The required active mode of the connection pool. Valid values are:

  - **MODE_ACTIVE**  
    The connection pool is always active. When MQQueueManager.disconnect() is called, the underlying connection is pooled, and potentially reused the next time that an MQQueueManager object is constructed. Connections will be destroyed by a separate thread if they are unused for longer than the Timeout period, or if the size of the pool exceeds HighThreshold.

  - **MODE_AUTO**  
    The connection pool is active while the Connection Manager is the default Connection Manager and there is at least one token in the set of MQPoolTokens held by MQEnvironment. This is the default mode.

  - **MODE_INACTIVE**  
    The connection pool is always inactive. When this mode is entered, the pool of connections to WebSphere MQ is cleared. When MQQueueManager.disconnect() is called, the connection that underlies any active MQQueueManager object is destroyed.

**setHighThreshold**

```java
public void setHighThreshold(int threshold)
```

Sets the HighThreshold. If the number of unused connections in the pool exceeds this value, the oldest unused connection in the pool is destroyed.

**Parameters**

- **threshold**  
  The maximum number of unused connections in the pool.

**setTimeout**

```java
public void setTimeout(long timeout)
```

Sets the Timeout value, where connections that remain unused for this length of time are destroyed by a separate thread.

**Parameters**

- **timeout**  
  The value of the timeout in milliseconds.
public interface MQC
extends Object

The MQC interface defines all the constants used by the WebSphere MQ Java
programming interface (except for completion code constants and error code
constants). To refer to one of these constants from within your programs, prefix the
constant name with “MQC.”. For example, you can set the close options for a
queue as follows:
MQQueue queue;
...
queue.closeOptions = MQC.MQCO_DELETE; // delete the
    // queue when
    // it is closed
...

A full description of these constants is in the WebSphere MQ Application
Programming Reference

Completion code and error code constants are defined in the MQException class.
See “MQException” on page 110
public interface MQPoolServicesEventListener extends Object

Note: Normally, applications do not use this interface.

MQPoolServicesEventListener is for implementation by providers of default ConnectionManagers. When an MQPoolServicesEventListener is registered with an MQPoolServices object, the event listener receives an event whenever an MQPoolToken is added to, or removed from, the set of MQPoolTokens that MQEnvironment manages. It also receives an event whenever the default ConnectionManager changes.

See also “MQPoolServices” on page 139 and “MQPoolServicesEvent” on page 140.

Methods

defaultConnectionManagerChanged
public void defaultConnectionManagerChanged(MQPoolServicesEvent event)

Called when the default ConnectionManager is set. The set of MQPoolTokens will have been cleared.

tokenAdded
public void tokenAdded(MQPoolServicesEvent event)

Called when an MQPoolToken is added to the set.

tokenRemoved
public void tokenRemoved(MQPoolServicesEvent event)

Called when an MQPoolToken is removed from the set.
MQConnectionManager

This is a private interface that cannot be implemented by applications. WebSphere MQ classes for Java supplies an implementation of this interface (MQSimpleConnectionManager), which you can specify on the MQQueueManager constructor, or through MQEnvironment.setDefaultConnectionManager.

See "MQSimpleConnectionManager" on page 169.

Applications or middleware that wish to provide their own ConnectionManager should implement javax.resource.spi.ConnectionManager. This requires Java 2 v1.3 with JAAS 1.0 installed.
MQReceiveExit

MQReceiveExit

public interface MQReceiveExit
extends Object

The receive exit interface allows you to examine and possibly alter the data received from the queue manager by the WebSphere MQ classes for Java.

Note: This interface does not apply when connecting directly to WebSphere MQ in bindings mode.

To provide your own receive exit, define a class that implements this interface. Create a new instance of your class and assign the MQEnvironment.receiveExit variable to it before constructing your MQQueueManager object. For example:

```java
// in MyReceiveExit.java
class MyReceiveExit implements MQReceiveExit {
    // you must provide an implementation
    // of the receiveExit method
    public byte[] receiveExit(MQChannelExit channelExitParms,
                              MQChannelDefinition channelDefinition,
                              byte[] agentBuffer)
    {
        // your exit code goes here...
    }
}
// in your main program...
MQEnvironment.receiveExit = new MyReceiveExit();
... // other initialization
MQQueueManager qMgr = new MQQueueManager("");
```

Methods

**receiveExit**

```java
public abstract byte[] receiveExit(MQChannelExit channelExitParms,
                                   MQChannelDefinition channelDefinition,
                                   byte[] agentBuffer)
```

The receive exit method that your class must provide. This method will be invoked whenever the WebSphere MQ classes for Java receives some data from the queue manager.

**Parameters**

- **channelExitParms**
  Contains information regarding the context in which the exit is being invoked. The exitResponse member variable is an output parameter that you use to tell the WebSphere MQ classes for Java what action to take next. See "MQChannelExit" on page 98 for further details.

- **channelDefinition**
  Contains details of the channel through which all communications with the queue manager take place.

- **agentBuffer**
  If the channelExitParms.exitReason is MQChannelExit.MQXR_XMIT, agentBuffer contains the data received from the queue manager; otherwise agentBuffer is null.
Returns

If the exit response code (in channelExitParms) is set so that the WebSphere MQ classes for Java can now process the data (MQXCC_OK), your receive exit method must return the data to be processed. The simplest receive exit, therefore, consists of the single line “return agentBuffer;”.

See also:
- “MQC” on page 171
- “MQChannelDefinition” on page 96
public interface MQSecurityExit
extends Object

The security exit interface allows you to customize the security flows that occur when an attempt is made to connect to a queue manager.

Note: This interface does not apply when connecting directly to WebSphere MQ in bindings mode.

To provide your own security exit, define a class that implements this interface. Create a new instance of your class and assign the MQEnvironment.securityExit variable to it before constructing your MQQueueManager object. For example:

```java
// in MySecurityExit.java
class MySecurityExit implements MQSecurityExit {
    // you must provide an implementation
    // of the securityExit method
    public byte[] securityExit(MQChannelExit channelExitParms,
                               MQChannelDefinition channelDefinition,
                               byte[] agentBuffer) {
        // your exit code goes here...
    }
}

// in your main program...
MQEnvironment.securityExit = new MySecurityExit();
... // other initialization
MQQueueManager qMgr = new MQQueueManager("");
```

**Methods**

**securityExit**

```java
public abstract byte[] securityExit(MQChannelExit channelExitParms,
MQChannelDefinition channelDefinition,
byte[] agentBuffer)
```

The security exit method that your class must provide.

**Parameters**

**channelExitParms**
Contains information regarding the context in which the exit is being invoked. The exitResponse member variable is an output parameter that you use to tell the WebSphere MQ Client for Java what action to take next. See the “MQChannelExit” on page 98 for further details.

**channelDefinition**
Contains details of the channel through which all communications with the queue manager take place.

**agentBuffer**
If the channelExitParms.exitReason is MQChannelExit.MQXR_SEC_MSG, agentBuffer contains the security message received from the queue manager; otherwise agentBuffer is null.
Returns

If the exit response code (in channelExitParms) is set so that a message is to be transmitted to the queue manager, your security exit method must return the data to be transmitted.

See also:
- “MQC” on page 171
- “MQChannelDefinition” on page 96
MQSendExit

public interface MQSendExit extends Object

The send exit interface allows you to examine and possibly alter the data sent to the queue manager by the WebSphere MQ Client for Java.

Note: This interface does not apply when connecting directly to WebSphere MQ in bindings mode.

To provide your own send exit, define a class that implements this interface. Create a new instance of your class and assign the MQEnvironment.sendExit variable to it before constructing your MQQueueManager object. For example:

```java
// in MySendExit.java
class MySendExit implements MQSendExit {
  // you must provide an implementation of the sendExit method
  public byte[] sendExit(MQChannelExit channelExitParms, MQChannelDefinition channelDefinition, byte[] agentBuffer) {
    // your exit code goes here...
  }
}
```

Methods

sendExit

```java
public abstract byte[] sendExit(MQChannelExit channelExitParms, MQChannelDefinition channelDefinition, byte[] agentBuffer[])
```

The send exit method that your class must provide. This method is invoked whenever the WebSphere MQ classes for Java wishes to transmit some data to the queue manager.

Parameters

channelExitParms
Contains information regarding the context in which the exit is being invoked. The exitResponse member variable is an output parameter that you use to tell the WebSphere MQ classes for Java what action to take next. See "MQChannelExit" on page 98 for further details.

channelDefinition
Contains details of the channel through which all communications with the queue manager take place.

agentBuffer
If the channelExitParms.exitReason is MQChannelExit.MQXR_XMIT, agentBuffer contains the data to be transmitted to the queue manager; otherwise agentBuffer is null.
Returns

If the exit response code (in channelExitParms) is set so that a message is to be transmitted to the queue manager (MQXCC_OK), your send exit method must return the data to be transmitted. The simplest send exit, therefore, consists of the single line "return agentBuffer;".

See also:
- "MQC" on page 171
- "MQChannelDefinition" on page 96
ManagedConnection

public interface javax.resource.spi.ManagedConnection

Note: Normally, applications do not use this class; it is intended for use by implementations of ConnectionManager.

WebSphere MQ classes for Java provides an implementation of ManagedConnection that is returned from ManagedConnectionFactory.createManagedConnection. This object represents a connection to a WebSphere MQ Queue Manager. For more details about this interface, see the J2EE Connector Architecture specification (refer to Sun’s Web site at http://java.sun.com).

Methods

addConnectionEventListener

public void addConnectionEventListener(ConnectionEventListener listener)

Add a ConnectionEventListener to the ManagedConnection instance.

The listener is notified if a severe error occurs on the ManagedConnection, or when MQQueueManager.disconnect() is called on a connection handle that is associated with this ManagedConnection. The listener is not notified about local transaction events (see “getLocalTransaction” on page 181).

associateConnection

public void associateConnection(Object connection)

Throws ResourceException.

WebSphere MQ classes for Java does not currently support this method. A javax.resource.NotSupportedException is thrown.

cleanup

public void cleanup()

Throws ResourceException.

Closes all open connection handles, and resets the physical connection to an initial state ready to be pooled. Any pending local transaction is rolled back. For more details, see “getLocalTransaction” on page 181.

destroy

public void destroy()

Throws ResourceException.

Destroys the physical connection to the WebSphere MQ Queue Manager. Any pending local transaction is committed. For more details, see “getLocalTransaction” on page 181.

cgetConnection

public Object getConnection(javax.security.auth.Subject subject,
                ConnectionRequestInfo cxRequestInfo)

Throws ResourceException.
ManagedConnection

Creates a new connection handle for the physical connection represented by the ManagedConnection object. For WebSphere MQ classes for Java, this returns an MQQueueManager object. The ConnectionManager normally returns this object from allocateConnection.

The subject parameter is ignored. If the cxRequestInfo parameter is not suitable, a ResourceException is thrown. Multiple connection handles can be used simultaneously for each single ManagedConnection.

getLocalTransaction

public LocalTransaction getLocalTransaction()

Throws ResourceException.

WebSphere MQ classes for Java does not currently support this method. A javax.resource.NotSupportedException is thrown.

Currently, a ConnectionManager cannot manage the WebSphere MQ local transaction, and registered ConnectionEventListeners are not informed about events relating to the local transaction. When cleanup() occurs, any ongoing unit of work is rolled back. When destroy() occurs, any ongoing unit of work is committed.

Existing API behavior is that an ongoing unit of work is committed at MQQueueManager.disconnect(). This existing behavior is preserved only when an MQConnectionManager (rather than a ConnectionManager) manages the connection.

getLogWriter

public java.io.PrintWriter getLogWriter()

Throws ResourceException.

Returns the log writer for this ManagedConnection.

WebSphere MQ classes for Java does not currently use the log writer. See “MQException.log” on page 110 for more information about logging.

getMetaData

public ManagedConnectionMetaData getMetaData()

Throws ResourceException.

Gets the meta data information for the underlying Queue Manager. See “ManagedConnectionMetaData” on page 185.

getXAResource

public javax.transaction.xa.XAResource getXAResource()

Throws ResourceException.

WebSphere MQ classes for Java does not currently support this method. A javax.resource.NotSupportedException is thrown.

removeConnectionEventListener

public void removeConnectionEventListener(ConnectionEventListener listener)

Removes a registered ConnectionEventListener.
ManagedConnection

setLogWriter

    public void setLogWriter(java.io.PrintWriter out)

    Throws ResourceException.

Sets the log writer for this ManagedConnection. When a
ManagedConnection is created, it inherits the log writer from its
ManagedConnectionFactory.

WebSphere MQ classes for Java does not currently use the log writer. See
"MQException.log" on page 110 for more information about logging.
ManagedConnectionFactory

public interface javax.resource.spi.ManagedConnectionFactory

Note: Normally, applications do not use this class; it is intended for use by implementations of ConnectionManager.

WebSphere MQ classes for Java provides an implementation of this interface to ConnectionManagers. A ManagedConnectionFactory is used to construct ManagedConnections, and to select suitable ManagedConnections from a set of candidates. For more details about this interface, see the J2EE Connector Architecture specification (refer to Sun’s Web site at http://java.sun.com).

Methods

createConnectionFactory

public Object createConnectionFactory()

Throws ResourceException.

WebSphere MQ classes for Java does not currently support the createConnectionFactory methods. This method throws a javax.resource.NotSupportedException.

createConnectionFactory

public Object createConnectionFactory(ConnectionManager cxManager)

Throws ResourceException.

WebSphere MQ classes for Java does not currently support the createConnectionFactory methods. This method throws a javax.resource.NotSupportedException.

createManagedConnection

public ManagedConnection createManagedConnection

(javax.security.auth.Subject subject,
 ConnectionRequestInfo cxRequestInfo)

Throws ResourceException.

Creates a new physical connection to a WebSphere MQ Queue Manager, and returns a ManagedConnection object that represents this connection. WebSphere MQ ignores the subject parameter.

equals

public boolean equals(Object other)

Checks whether this ManagedConnectionFactory is equal to another ManagedConnectionFactory. Returns true if both ManagedConnectionFactoryes describe the same target Queue Manager.

getLogWriter

public java.io.PrintWriter getLogWriter()

Throws ResourceException.

Returns the log writer for this ManagedConnectionFactory.
ManagedConnectionFactory

WebSphere MQ classes for Java does not currently use the log writer. See "MQException.log" on page 110 for more information about logging.

hashCode

public int hashCode()

Returns the hashCode for this ManagedConnectionFactory.

matchManagedConnection

public ManagedConnection matchManagedConnection

&space;&space; &space;java.util.Set connectionSet,
&space;&space; &space;javax.security.auth.Subject subject,
&space;&space; &space;ConnectionRequestInfo cxRequestInfo)

Throws ResourceException.

Searches the supplied set of candidate ManagedConnections for an appropriate ManagedConnection. Returns either null, or a suitable ManagedConnection from the set that meets the criteria for connection.

setLogWriter

public void setLogWriter(java.io.PrintWriter out)

Throws ResourceException.

Sets the log writer for this ManagedConnectionFactory. When a ManagedConnection is created, it inherits the log writer from its ManagedConnectionFactory.

WebSphere MQ classes for Java does not currently use the log writer. See "MQException.log" on page 110 for more information about logging.
ManagedConnectionMetaData

public interface javax.resource.spi.ManagedConnectionMetaData

Note: Normally, applications do not use this interface; it is intended for use by implementations of ConnectionManager.

A ConnectionManager can use this interface to retrieve meta data that is related to an underlying physical connection to a Queue Manager. An implementation of this interface is returned from ManagedConnection.getMetaData(). For more details about this interface, see the J2EE Connector Architecture specification (refer to Sun’s Web site at http://java.sun.com).

Methods

getEISProductName

public String getEISProductName()

Throws ResourceException.

Returns “IBM WebSphere MQ”.

getMaxConnections

public int getMaxConnections()

Throws ResourceException.

Returns 0.

getProductVersion

public String getProductVersion()

Throws ResourceException.

Returns a string that describes the command level of the WebSphere MQ Queue Manager to which the ManagedConnection is connected.

getUserName

public String getUserName()

Throws ResourceException.

If the ManagedConnection represents a Client connection to a Queue Manager, this returns the user ID used for the connection. Otherwise, it returns an empty string.
ManagedConnectionMetaData
Part 3. Programming with WebSphere MQ JMS

Chapter 10. Writing WebSphere MQ JMS programs .......................... 189
The JMS model ........................................ 189
Building a connection .................................. 190
   Retrieving the factory from JNDI .......................... 190
   Using the factory to create a connection ................. 191
   Creating factories at runtime ............................ 191
   Starting the connection ................................ 192
   Choosing client or bindings transport .................... 192
Obtaining a session .................................... 193
Sending a message ..................................... 193
   Setting properties with the 'set' method ................. 195
   Message types ....................................... 195
Receiving a message .................................... 196
   Message selectors ................................... 196
   Asynchronous delivery ................................ 197
Closing down ............................................ 197
   Java Virtual Machine hangs at shutdown ................. 198
Handling errors ....................................... 198
   Exception listener ................................... 198
User Exits .............................................. 198
   Using Secure Sockets Layer (SSL) ......................... 199
      SSL administrative properties ......................... 199
      SSL_CIPHER_SUITES object property ................. 199
      SSL_CERTSTORES object property ..................... 200
      SSL_SOCKET_FACTORY ................................. 201

Chapter 11. Programming publish/subscribe applications ................. 203
   Introduction ........................................ 203
   Getting started with WebSphere MQ JMS and publish/subscribe .......... 203
      Choosing a broker ................................... 203
      Setting up the broker to run the WebSphere MQ JMS .......... 204
      Connecting to your broker via WebSphere MQ .............. 204
      Connecting to your broker directly ..................... 204
Writing a simple publish/subscribe application connecting through WebSphere MQ .................. 205
   Import required packages .............................. 207
   Obtain or create JMS objects ........................... 207
   Publish messages .................................... 209
   Receive subscriptions ................................ 209
   Close down unwanted resources ......................... 209
   TopicConnectionFactory administered objects .............. 209
   Topic administered objects ............................ 210
Using topics .......................................... 210
   Topic names ......................................... 210
   Creating topics at runtime ............................. 212
Subscriber options .................................... 213
   Creating non-durable subscribers ................. 214
   Creating durable subscribers ......................... 214
   Using message selectors .............................. 214
   Suppressing local publications ......................... 214
   Combining the subscriber options ..................... 215
   Configuring the base subscriber queue ................. 215
   Default configuration ............................... 215
   Configuring non-durable subscribers ................... 216
   Configuring durable subscribers ....................... 216
   Subscription stores .................................. 217
   Migration and coexistence considerations ............... 219
   Solving publish/subscribe problems ..................... 219
   Incomplete publish/subscribe close down ................. 219
   Subscriber cleanup utility ............................ 220
   Manual cleanup ...................................... 222
   Cleanup from within a program ......................... 223
   Handling broker reports ............................... 223

Chapter 12. JMS messages .................................. 225
   Message selectors ................................... 225
   Mapping JMS messages onto WebSphere MQ messages .......... 229
      The MQRFH2 header ................................ 230
      JMS fields and properties with corresponding MQMD fields .......... 233
      Mapping JMS fields onto WebSphere MQ fields (outgoing messages) .......... 234
      Mapping JMS header fields at send() or publish() .................. 236
      Mapping JMS property fields .......................... 237
      Mapping JMS provider-specific fields .................... 237
      Mapping WebSphere MQ fields onto JMS fields (incoming messages) .......... 239
   Mapping JMS to a native WebSphere MQ application .................. 240
   Message body ....................................... 241

Chapter 13. WebSphere MQ JMS Application Server Facilities .............. 243
   ASF classes and functions ............................ 243
   ConnectionConsumer ................................ 243
   Planning an application ............................... 244
      General principles for point-to-point messaging ............ 244
      General principles for publish/subscribe messaging .......... 245
      Handling poison messages ............................ 246
      Removing messages from the queue ....................... 247
      Error handling .................................... 248
      Recovering from error conditions ...................... 248
      Reason and feedback codes ........................... 249
   Application server sample code ........................ 249
      MyServerSession.java ................................ 251
      MyServerSessionPool.java ............................. 251
      MessageListenerFactory.java ......................... 252
   Examples of ASF use .................................. 253
      Load1.java ......................................... 253
      CountingMessageListenerFactory.java ................... 254
      ASFClient1.java .................................... 255
      Load2.java ......................................... 256

© Copyright IBM Corp. 1997, 2002
Chapter 14. JMS interfaces and classes

Sun Java Message Service classes and interfaces

WebSphere MQ JMS classes

BytesMessage

Methods

Cleanup *

WebSphere MQ constructor.

Methods

Connection

Methods

ConnectionConsumer

Methods

ConnectionFactory

WebSphere MQ constructor.

Methods

ConnectionMetaData

WebSphere MQ constructor.

Methods

DeliveryMode

Fields

Destination

WebSphere MQ constructors

Methods

ExceptionListener

Methods

MapMessage

Methods

Message

Fields

Methods

MessageConsumer

Methods

MessageListener

Methods

MessageProducer

WebSphere MQ constructors

Methods

MQQueueEnumeration *

Methods

ObjectMessage

Methods

Queue

WebSphere MQ constructors

Methods

QueueBrowser

Methods

QueueConnection

Methods

QueueConnectionFactory

WebSphere MQ constructor.

Methods

QueueReceiver

Methods

QueueRequestor

Methods

Constructors.

Methods

QueueSender

Methods

QueueSession

Methods

Session

Fields

Methods

StreamMessage

Methods

TemporaryQueue

Methods

TemporaryTopic

WebSphere MQ constructor.

Methods

TextMessage

Methods

Topic

WebSphere MQ constructor.

Methods

TopicConnection

Methods

TopicConnectionFactory

WebSphere MQ constructor.

Methods

TopicPublisher

Methods

TopicRequestor

Constructors.

Methods

TopicSession

WebSphere MQ constructor.

Methods

TopicSubscriber

Methods

XAConnection

Methods

XAConnectionFactory

WebSphere MQ constructor.

Methods

XAQueueConnection

Methods

XAQueueConnectionFactory

Methods

XAQueueSession

Methods

XASession

Methods

XATopicConnection

Methods

XATopicConnectionFactory

Methods

XATopicSession

Methods
Chapter 10. Writing WebSphere MQ JMS programs

This chapter provides information to help with writing WebSphere MQ JMS applications. It gives a brief introduction to the JMS model, and detailed information on programming some common tasks that application programs are likely to need to perform.

The JMS model

JMS defines a generic view of a message passing service. It is important to understand this view, and how it maps onto the underlying WebSphere MQ transport.

The generic JMS model is based around the following interfaces that are defined in Sun’s `javax.jms` package:

**Connection**
Provides access to the underlying transport, and is used to create Sessions.

**Session**
Provides a context for producing and consuming messages, including the methods used to create MessageProducers and MessageConsumers.

**MessageProducer**
Used to send messages.

**MessageConsumer**
Used to receive messages.

Note that a Connection is thread safe, but Sessions, MessageProducers, and MessageConsumers are not. The recommended strategy is to use one Session per application thread.

In WebSphere MQ terms:

**Connection**
Provides a scope for temporary queues. Also, it provides a place to hold the parameters that control how to connect to WebSphere MQ. Examples of these parameters are the name of the queue manager, and the name of the remote host if you use the WebSphere MQ Java client connectivity.

**Session**
Contains an HCONN and therefore defines a transactional scope.

**MessageProducer and MessageConsumer**
Contain an HOBJ that defines a particular queue for writing to, or reading from.

Note that normal WebSphere MQ rules apply:

- Only a single operation can be in progress per HCONN at any given time. Therefore, the MessageProducers or MessageConsumers associated with a Session cannot be called concurrently. This is consistent with the JMS restriction of a single thread per Session.
- PUTs can use remote queues, but GETs can only be applied to queues on the local queue manager.
The generic JMS interfaces are subclassed into more specific versions for “Point-to-Point” and “Publish/Subscribe” behavior.

The point-to-point versions are:
- QueueConnection
- QueueSession
- QueueSender
- QueueReceiver

A key idea in JMS is that it is possible, and strongly recommended, to write application programs that use only references to the interfaces in `javax.jms`. All vendor-specific information is encapsulated in implementations of:
- QueueConnectionFactory
- TopicConnectionFactory
- Queue
- Topic

These are known as “administered objects”, that is, objects that can be built using a vendor-supplied administration tool and can be stored in a JNDI namespace. A JMS application can retrieve these objects from the namespace and use them without needing to know which vendor provided the implementation.

Building a connection

Connections are not created directly, but are built using a connection factory. Factory objects can be stored in a JNDI namespace, thus insulating the JMS application from provider-specific information. Details of how to create and store factory objects are in Chapter 5, “Using the WebSphere MQ JMS administration tool” on page 39.

If you do not have a JNDI namespace available, see “Creating factories at runtime” on page 191.

Retrieving the factory from JNDI

To retrieve an object from a JNDI namespace, an initial context must be set up, as shown in this fragment taken from the IVTRun sample file:

```java
import javax.jms.*;
import javax.naming.*;
import javax.naming.directory.*;

java.util.Hashtable environment = new java.util.Hashtable();
environment.put(Context.INITIAL_CONTEXT_FACTORY, icf);
environment.put(Context.PROVIDER_URL, url);
Context ctx = new InitialDirContext( environment );
```

where:
- icf defines a factory class for the initial context
- url defines a context specific URL

For more details about JNDI usage, see Sun’s JNDI documentation.
Building a connection

**Note:** Some combinations of the JNDI packages and LDAP service providers can result in an LDAP error 84. To resolve the problem, insert the following line before the call to InitialDirContext.

```java
environment.put(Context.REFERRAL, "throw");
```

Once an initial context is obtained, objects are retrieved from the namespace by using the lookup() method. The following code retrieves a QueueConnectionFactory named ivtQCF from an LDAP-based namespace:

```java
QueueConnectionFactory factory = (QueueConnectionFactory)ctx.lookup("cn=ivtQCF");
```

**Using the factory to create a connection**

The createQueueConnection() method on the factory object is used to create a ‘Connection’, as shown in the following code:

```java
QueueConnection connection = factory.createQueueConnection();
```

**Creating factories at runtime**

If a JNDI namespace is not available, it is possible to create factory objects at runtime. However, using this method reduces the portability of the JMS application because it requires references to WebSphere MQ specific classes.

The following code creates a QueueConnectionFactory with all default settings:

```java
factory = new com.ibm.mq.jms.MQQueueConnectionFactory();
```

(You can omit the com.ibm.mq.jms. prefix if you import the com.ibm.mq.jms package instead.)

A connection created from the above factory uses the Java bindings to connect to the default queue manager on the local machine. The set methods shown in Table 14 on page 192 can be used to customize the factory with WebSphere MQ specific information.

The only way to create a TopicConnectionFactory object at runtime is to construct it using the MQTopicConnectionFactory constructor. For example:

```java
MQTopicConnectionFactory fact = new MQTopicConnectionFactory();
```

This will create a default TopicConnectionFactory object with the bindings transportType and all other default settings.

It is possible to change the transportType for the TopicConnectionFactory using its setTransportType() method. For example:

```java
fact.setTransportType(JMSC.MQJMS_TP_BINDINGS_MQ);  // Bindings mode
fact.setTransportType(JMSC.MQJMS_TP_CLIENT_MQ_TCP); // Client mode
fact.setTransportType(JMSC.MQJMS_TP_DIRECT_TCP);   // Direct TCP/IP mode
```

The full JMS TopicConnectionFactory interface has been implemented. Refer to “TopicConnectionFactory” on page 367 for more details. Note that certain combinations of property settings are not valid for TopicConnectionFactory objects. See “Properties” on page 46 for more details.
Building a connection

Starting the connection
The JMS specification defines that connections should be created in the ‘stopped’ state. Until the connection starts, MessageConsumers that are associated with the connection cannot receive any messages. To start the connection, issue the following command:

```
connection.start();
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setCCSID(int)</td>
<td>Used to set the MQEnvironment.CCSID property</td>
</tr>
<tr>
<td>setChannel(String)</td>
<td>The name of the channel for a client connection</td>
</tr>
<tr>
<td>setHostName(String)</td>
<td>The name of the host for a client connection</td>
</tr>
<tr>
<td>setPort(int)</td>
<td>The port for a client connection</td>
</tr>
<tr>
<td>setQueueManager(String)</td>
<td>The name of the queue manager</td>
</tr>
<tr>
<td>setTemporaryModel(String)</td>
<td>The name of a model queue used to generate a temporary destination as a result of a call to QueueSession.createTemporaryQueue(). We recommend that this is the name of a temporary dynamic queue, rather than a permanent dynamic queue.</td>
</tr>
<tr>
<td>setTransportType(int)</td>
<td>Specify how to connect to WebSphere MQ. The options currently available are:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JMSC is in the package com.ibm.mq.jms</td>
</tr>
<tr>
<td>setReceiveExit(String)</td>
<td>These methods exist to allow the use of the send, receive and security exits provided by the underlying WebSphere MQ Classes for Java. The set*Exit methods take the name of a class that implements the relevant exit methods. (See the WebSphere MQ product documentation for details.)</td>
</tr>
<tr>
<td>setSecurityExit(String)</td>
<td>Also, the class must implement a constructor with a single String parameter. This string provides any initialization data that may be required by the exit, and is set to the value provided in the corresponding set*ExitInit method.</td>
</tr>
<tr>
<td>setSendExit(String)</td>
<td></td>
</tr>
<tr>
<td>setReceiveExitInit(String)</td>
<td></td>
</tr>
<tr>
<td>setSecurityExitInit(String)</td>
<td></td>
</tr>
<tr>
<td>setSendExitInit(String)</td>
<td></td>
</tr>
</tbody>
</table>

Choosing client or bindings transport
WebSphere MQ JMS can communicate with WebSphere MQ using either the client or bindings transports. If you use the Java bindings, the JMS application and the WebSphere MQ queue manager must be located on the same machine. If you use the client, the queue manager can be on a different machine to the application.

The contents of the connection factory object determine which transport to use. Chapter 5, “Using the WebSphere MQ JMS administration tool” on page 39 describes how to define a factory object for use with client or bindings transport.

The following code fragment illustrates how you can define the transport within an application:

---

2. However, client transport is not supported on the z/OS & OS/390 platform.
String HOSTNAME = "machine1";
String QMGRNAME = "machine1.QM1";
String CHANNEL = "SYSTEM.DEF.SVRCONN";

factory = new MQQueueConnectionFactory();
factory.setTransportType(JMSC.MQJMS_TP_CLIENT_MQ_TCPIP);
factory.setQueueManager(QMGRNAME);
factory.setHostName(HOSTNAME);
factory.setChannel(CHANNEL);

### Obtaining a session

Once a connection is made, use the createQueueSession method on the QueueConnection to obtain a session.

The method takes two parameters:
1. A boolean that determines whether the session is “transacted” or “non-transacted”.
2. A parameter that determines the “acknowledge” mode.

The simplest case is that of the “non-transacted” session with AUTO_ACKNOWLEDGE, as shown in the following code fragment:

```java
QueueSession session;
boolean transacted = false;
session = connection.createQueueSession(transacted,
                      Session.AUTO_ACKNOWLEDGE);
```

**Note:** A connection is thread safe, but sessions (and objects that are created from them) are not. The recommended practice for multi-threaded applications is to use a separate session for each thread.

### Sending a message

Messages are sent using a MessageProducer. For point-to-point this is a QueueSender that is created using the createSender method on QueueSession. A QueueSender is normally created for a specific queue, so that all messages sent using that sender are sent to the same destination. The destination is specified using a Queue object. Queue objects can be either created at runtime, or built and stored in a JNDI namespace.

Queue objects are retrieved from JNDI in the following way:

```java
Queue ioQueue;
ioQueue = (Queue)ctx.lookup( qLookup );
```

WebSphere MQ JMS provides an implementation of Queue in com.ibm.mq.jms.MQQueue. It contains properties that control the details of WebSphere MQ specific behavior, but in many cases it is possible to use the default values. JMS defines a standard way to specify the destination that minimizes the WebSphere MQ specific code in the application. This mechanism uses the QueueSession.createQueue method, which takes a string parameter describing the destination. The string itself is still in a vendor-specific format, but this is a more flexible approach than directly referencing the vendor classes.

WebSphere MQ JMS accepts two forms for the string parameter of createQueue().
- The first is the name of the WebSphere MQ queue, as illustrated in the following fragment taken from the IVTRun program in the samples directory:
public static final String QUEUE = "SYSTEM.DEFAULT.LOCAL.QUEUE" ;

ioQueue = session.createQueue( QUEUE );

- The second, and more powerful, form is based on “uniform resource identifiers” (URI). This form allows you to specify remote queues (queues on a queue manager other than the one to which you are connected). It also allows you to set the other properties contained in a com.ibm.mq.jms.MQQueue object.

The URI for a queue begins with the sequence queue://, followed by the name of the queue manager on which the queue resides. This is followed by a further “/”, the name of the queue, and optionally, a list of name-value pairs that set the remaining Queue properties. For example, the URI equivalent of the previous example is:

ioQueue = session.createQueue("queue:///SYSTEM.DEFAULT.LOCAL.QUEUE");

Note that the name of the queue manager is omitted. This is interpreted as the queue manager to which the owning QueueConnection is connected at the time when the Queue object is used.

The following example connects to queue Q1 on queue manager HOST1.QM1, and causes all messages to be sent as non-persistent and priority 5:

ioQueue = session.createQueue("queue://HOST1.QM1/Q1?persistence=1&priority=5");

Table 15 lists the names that can be used in the name-value part of the URI. A disadvantage of this format is that it does not support symbolic names for the values, so where appropriate, the table also indicates ‘special’ values. Note that these special values may be subject to change. (See “Setting properties with the ‘set’ method” on page 195 for an alternative method to set properties.)

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>expiry</td>
<td>Lifetime of the message in milliseconds</td>
<td>0 for unlimited, positive integers for timeout (ms)</td>
</tr>
<tr>
<td>priority</td>
<td>Priority of the message</td>
<td>0 through 9, -1=QDEF, -2=APP</td>
</tr>
<tr>
<td>persistence</td>
<td>Whether the message should be 'hardened' to disk</td>
<td>1=non-persistent, 2=persistent, -1=QDEF, -2=APP</td>
</tr>
<tr>
<td>CCSID</td>
<td>Character set of the destination</td>
<td>integers - valid values listed in base WebSphere MQ documentation</td>
</tr>
<tr>
<td>targetClient</td>
<td>Whether the receiving application is JMS compliant or not</td>
<td>0=JMS, 1=MQ</td>
</tr>
<tr>
<td>encoding</td>
<td>How to represent numeric fields</td>
<td>An integer value as described in the base WebSphere MQ documentation</td>
</tr>
<tr>
<td>QDEF</td>
<td>- a special value that means the property should be determined by the configuration of the WebSphere MQ queue.</td>
<td></td>
</tr>
<tr>
<td>APP</td>
<td>- a special value that means the JMS application can control this property.</td>
<td></td>
</tr>
</tbody>
</table>

Once the Queue object is obtained (either using createQueue as above or from JNDI), it must be passed into the createSender method to create a QueueSender:

QueueSender queueSender = session.createSender(ioQueue);
The resulting `queueSender` object is used to send messages by using the `send` method:

```java
queueSender.send(outMessage);
```

### Setting properties with the ‘set’ method

You can set Queue properties by first creating an instance of `com.ibm.mq.jms.MQQueue` using the default constructor. Then you can fill in the required values by using public `set` methods. This method means that you can use symbolic names for the property values. However, because these values are vendor-specific, and are embedded in the code, the applications become less portable.

The following code fragment shows the setting of a queue property with a `set` method.

```java
com.ibm.mq.jms.MQQueue q1 = new com.ibm.mq.jms.MQQueue();
q1.setBaseQueueManagerName("HOST1.QM1");
q1.setBaseQueueName("Q1");
q1.setPersistence(DeliveryMode.NON_PERSISTENT);
q1.setPriority(5);
```

Table 16 shows the symbolic property values that are supplied with WebSphere MQ JMS for use with the `set` methods.

Table 16. Symbolic values for queue properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Admin tool keyword</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>expiry</td>
<td>UNLIM</td>
<td>JMSC.MQJMS_EXP_UNLIMITED</td>
</tr>
<tr>
<td></td>
<td>APP</td>
<td>JMSC.MQJMS_EXP_APP</td>
</tr>
<tr>
<td>priority</td>
<td>APP</td>
<td>JMSC.MQJMS_PRI_APP</td>
</tr>
<tr>
<td></td>
<td>QDEF</td>
<td>JMSC.MQJMS_PRI_QDEF</td>
</tr>
<tr>
<td>persistence</td>
<td>APP</td>
<td>JMSC.MQJMS_PER_APP</td>
</tr>
<tr>
<td></td>
<td>QDEF</td>
<td>JMSC.MQJMS_PER_QDEF</td>
</tr>
<tr>
<td></td>
<td>PERS</td>
<td>JMSC.MQJMS_PER_PER</td>
</tr>
<tr>
<td></td>
<td>NON</td>
<td>JMSC.MQJMS_PER_NON</td>
</tr>
<tr>
<td>targetClient</td>
<td>JMS</td>
<td>JMSC.MQJMS_CLIENT_JMS_COMPLIANT</td>
</tr>
<tr>
<td></td>
<td>MQ</td>
<td>JMSC.MQJMS_CLIENT_NONJMS_MQ</td>
</tr>
<tr>
<td>encoding</td>
<td>Integer(N)</td>
<td>JMSC.MQJMS_ENCODING_INTEGER_NORMAL</td>
</tr>
<tr>
<td></td>
<td>Integer(R)</td>
<td>JMSC.MQJMS_ENCODING_INTEGER_REVERSED</td>
</tr>
<tr>
<td></td>
<td>Decimal(N)</td>
<td>JMSC.MQJMS_ENCODING_DECIMAL_NORMAL</td>
</tr>
<tr>
<td></td>
<td>Decimal(R)</td>
<td>JMSC.MQJMS_ENCODING_DECIMAL_REVERSED</td>
</tr>
<tr>
<td></td>
<td>Float(N)</td>
<td>JMSC.MQJMS_ENCODING_FLOAT_IEEE_NORMAL</td>
</tr>
<tr>
<td></td>
<td>Float(R)</td>
<td>JMSC.MQJMS_ENCODING_FLOAT_IEEE_REVERSED</td>
</tr>
<tr>
<td></td>
<td>Native</td>
<td>JMSC.MQJMS_ENCODING_NATIVE</td>
</tr>
</tbody>
</table>

See “The ENCODING property” on page 52 for a discussion on encoding.

### Message types

JMS provides several message types, each of which embodies some knowledge of its content. To avoid referencing the vendor-specific class names for the message types, methods are provided on the `Session` object for message creation.

In the sample program, a text message is created in the following manner:

```java
```
**Sending a message**

```java
System.out.println( "Creating a TextMessage" );
TextMessage outMessage = session.createTextMessage();
System.out.println("Adding Text");
outMessage.setText(outString);
```

The message types that can be used are:
- BytesMessage
- MapMessage
- ObjectMessage
- StreamMessage
- TextMessage

Details of these types are in [Chapter 14, “JMS interfaces and classes” on page 261](#).

**Receiving a message**

Messages are received by using a QueueReceiver. This is created from a Session by using the `createReceiver()` method. This method takes a Queue parameter that defines where the messages are received from. See “Sending a message” on page 193 for details of how to create a Queue object.

The sample program creates a receiver and reads back the test message with the following code:

```java
QueueReceiver queueReceiver = session.createReceiver(ioQueue);
Message inMessage = queueReceiver.receive(1000);
```

The parameter in the receive call is a timeout in milliseconds. This parameter defines how long the method should wait if there is no message available immediately. You can omit this parameter, in which case, the call blocks indefinitely. If you do not want any delay, use the `receiveNoWait()` method.

The receive methods return a message of the appropriate type. For example, if a `TextMessage` is put on a queue, when the message is received, the object that is returned is an instance of `TextMessage`.

To extract the content from the body of the message, it is necessary to cast from the generic Message class (which is the declared return type of the receive methods) to the more specific subclass, such as `TextMessage`. If the received message type is not known, you can use the “`instanceof`” operator to determine which type it is. It is good practice always to test the message class before casting, so that unexpected errors can be handled gracefully.

The following code illustrates the use of “`instanceof`”, and extraction of the content from a `TextMessage`:

```java
if (inMessage instanceof TextMessage) {
    String replyString = ((TextMessage) inMessage).getText();
    .
    .
} else {
    // Print error message if Message was not a TextMessage.
    System.out.println("Reply message was not a TextMessage");
}
```

**Message selectors**

JMS provides a mechanism to select a subset of the messages on a queue so that this subset is returned by a receive call. When creating a QueueReceiver, a string
can be provided that contains an SQL (Structured Query Language) expression to determine which messages to retrieve. The selector can refer to fields in the JMS message header as well as fields in the message properties (these are effectively application-defined header fields). Details of the header field names, as well as the syntax for the SQL selector, are in Chapter 12, “JMS messages” on page 225.

The following example shows how to select for a user-defined property named myProp:

```java
queueReceiver = session.createReceiver(ioQueue, "myProp = 'blue'");
```

**Note:** The JMS specification does not permit the selector associated with a receiver to be changed. Once a receiver is created, the selector is fixed for the lifetime of that receiver. This means that if you require different selectors, you must create new receivers.

### Asynchronous delivery

An alternative to making calls to `QueueReceiver.receive()` is to register a method that is called automatically when a suitable message is available. The following fragment illustrates the mechanism:

```java
import javax.jms.*;

public class MyClass implements MessageListener
{
    // The method that will be called by JMS when a message
    // is available.
    public void onMessage(Message message)
    {
        System.out.println("message is "+message);

        // application specific processing here
        ...
        ...
    }
}

// In Main program (possibly of some other class)
MyClass listener = new MyClass();
queueReceiver.setMessageListener(listener);

// main program can now continue with other application specific
// behavior.
```

**Note:** Use of asynchronous delivery with a `QueueReceiver` marks the entire Session as asynchronous. It is an error to make an explicit call to the `receive` methods of a `QueueReceiver` that is associated with a Session that is using asynchronous delivery.

### Closing down

Garbage collection alone cannot release all WebSphere MQ resources in a timely manner. This is especially true if the application needs to create many short-lived JMS objects at the Session level or lower. It is therefore important to call the `close()` methods of the various classes (QueueConnection, QueueSession, QueueSender, and QueueReceiver) when the resources are no longer required.
Java Virtual Machine hangs at shutdown

If an application using WebSphere MQ JMS finishes without calling Connection.close(), some JVMs will appear to hang. If this problem occurs, either edit the application to include a call to Connection.close(), or terminate the JVM by using the Ctrl-C keys.

Handling errors

Any runtime errors in a JMS application are reported by exceptions. The majority of methods in JMS throw JMSExceptions to indicate errors. It is good programming practice to catch these exceptions and display them on a suitable output.

Unlike normal Java Exceptions, a JMSException may contain a further exception embedded in it. For JMS, this can be a valuable way to pass important detail from the underlying transport. In the case of WebSphere MQ JMS, when WebSphere MQ raises an MQException, this exception is usually included as the embedded exception in a JMSException.

The implementation of JMSException does not include the embedded exception in the output of its toString() method. Therefore, it is necessary to check explicitly for an embedded exception and print it out, as shown in the following fragment:

```java
try {
    // code which may throw a JMSException
} catch (JMSException je) {
    System.err.println("caught "+je);
    Exception e = je.getLinkedException();
    if (e != null) {
        System.err.println("linked exception: "+e);
    }
}
```

Exception listener

For asynchronous message delivery, the application code cannot catch exceptions raised by failures to receive messages. This is because the application code does not make explicit calls to receive() methods. To cope with this situation, it is possible to register an ExceptionListener, which is an instance of a class that implements the onException() method. When a serious error occurs, this method is called with the JMSException passed as its only parameter. Further details are in Sun’s JMS documentation.

User Exits

WebSphere MQ JMS allows you to code and use implementations of the WebSphere MQ base Java send, receive and security exits. For WebSphere MQ JMS, you must ensure that your exit has a constructor that takes a single string argument. See the description of exit-related set methods in Table 14 on page 192 and “Property dependencies” on page 51.
Using Secure Sockets Layer (SSL)

From WebSphere MQ V5.3, WebSphere MQ base Java clients and WebSphere MQ JMS connections using TRANSPORT(CLIENT) support Secure Sockets Layer (SSL) encryption. SSL provides communication encryption, authentication and message integrity. It is typically used to secure communications between any two peers on the public internet or within an intranet.

WebSphere MQ classes for Java uses Java Secure Socket Extension (JSSE) to handle SSL encryption, and so requires a JSSE provider. J2SE v1.4 JVMs have a JSSE provider built-in. Details of how to manage and store certificates can vary from provider to provider. For information about this, refer to your JSSE provider’s documentation.

This section assumes your JSSE provider is correctly installed and configured, and suitable certificates have been installed and made available to your JSSE provider.

SSL administrative properties

SSLCIPHERSUITE object property
To enable SSL encryption on a ConnectionFactory, use JMSAdmin to set the SSLCIPHERSUITE property to a CipherSuite supported by your JSSE provider. This must match the CipherSpec set on the target channel. However, CipherSuites are distinct from CipherSpecs and so have different names. Appendix H, “SSL CipherSuites supported by WebSphere MQ” on page 425 contains a table mapping the CipherSpecs supported by WebSphere MQ to their equivalent CipherSuites as known to JSSE. Additionally, the named CipherSuite must be supported by your JSSE provider. For more information about CipherSpecs and CipherSuites with WebSphere MQ, see the WebSphere MQ Security book.

For example, to set a QueueConnectionFactory to connect to an SSL-enabled SVRCONN channel using a CipherSpec of RC4_MD5_EXPORT, issue the following command to JMSAdmin:

```
ALTER QCF(my.qcf) SSLCIPHERSUITE(SSL_RSA_EXPORT_WITH_RC4_40_MD5)
```

This can also be set from a program, using the setSSLCipherSuite() method on MQConnectionFactory.

For convenience, if a CipherSpec is specified on the SSLCIPHERSUITE property, JMSAdmin will attempt to map the CipherSpec to an appropriate CipherSuite and issue a warning. Note that this attempt to map is not made if the property is specified by a program.

SSLPEERNAME object property
A JMS application can ensure that it has connected to the correct queue manager, by specifying a distinguished name (DN) pattern. The connection will only succeed if the queue manager presents a DN which matches the pattern. For more details of the format of this pattern, refer to the WebSphere MQ Script (MQSC) Command Reference book or the WebSphere MQ Security book.

The DN is set using the SSLPEERNAME property of ConnectionFactory. For example, the following JMSAdmin command sets the ConnectionFactory to expect the queue manager to identify itself with a Common Name beginning “QMGR.” with at least two Organizational Unit names, the first of which must be “IBM” and the second “WEBSPHERE”:

```
ALTER QCF(my.qcf) SSLPEERNAME(CN=QMGR.*, OU=IBM, OU=WEBSPHERE)
```
Note that checking is case-insensitive, and that semicolons can be used in place of the commas. This can also be set from a program, using the setSSLPeerName() method on MQConnectionFactory. If this property is not set, no checking is performed on the Distinguished Name supplied by the queue manager. This property is ignored if no CipherSuite is set.

**SSLCERTSTORES object property**

It is common to use a certificate revocation list (CRL) to manage revocation of certificates that have become untrusted. These are typically hosted on LDAP servers; and JMS allows an LDAP server to be specified for CRL checking under Java 2 v1.4 or later. The following JMSAdmin example directs JMS to use a CRL hosted on an LDAP server named crl1.ibm.com:

ALTER QCF(my.qcf) SSLCRL(ldap://crl1.ibm.com)

**Note:** To use a CertStore successfully with a CRL hosted on an LDAP server, make sure that your Java Software Development Kit (SDK) is compatible with the CRL. Some SDKs require that the CRL conforms to RFC 2587 which defines a schema for LDAP v2. Most LDAP v3 servers use RFC 2256 instead.

If your LDAP server is not running on the default port of 389, the port can be specified by appending a colon and the port number to the hostname. If the certificate presented by the queue manager is present in the CRL hosted on crl1.ibm.com, the connection will not complete. To avoid single-point-of-failure, JMS allows multiple LDAP servers to be supplied, by supplying a space-delimited list of ldap servers. For example:

ALTER QCF(my.qcf) SSLCRL(ldap://crl1.ibm.com ldap://crl2.ibm.com)

When multiple LDAP servers are specified, JMS will try each one in turn until it finds a server with which it can successfully verify the queue manager’s certificate. Each server must contain identical information.

A string of this format can be supplied by a program on the MQConnectionFactory.setSSLCertStores() method. Alternatively, the application can create one or more java.security.cert.CertStore objects and place these in a suitable Collection object, and supply this Collection to the setSSLCertStores() method. In this way, the application can customize CRL checking. Please refer to your JSSE documentation for details on constructing and using CertStore objects.

The certificate presented by the queue manager when a connection is being set up is validated as follows:

1. The first CertStore object in the Collection identified by sslCertStores is used to identify a CRL server.
2. An attempt is made to contact the CRL server.
3. If the attempt is successful, the server is searched for a match for the certificate.
   a. If the certificate is found to be revoked, the search process is over and the connection request will fail with reason code MQRC_SSL_CERTIFICATE_REVOKED.
   b. If the certificate is not found, the search process is over and the connection is allowed to proceed.
4. If the attempt to contact the server is unsuccessful, the next CertStore object is used to identify a CRL server and the process repeats from “An attempt is made to contact the CRL server.”
Using SSL

If this was the last CertStore in the Collection, or if the Collection contains no CertStore objects, the search process has failed and the connection request will fail with reason code MQRC_SSL_CERT_STORE_ERROR.

The Collection object determines the order in which CertStores are used.

If your application uses setSSLCertStores() to set a Collection of CertStore objects, the MQConnectionFactory can no longer be bound into a JNDI namespace. Attempting to do so will cause an Exception. If the sslCertStores property is not set, no revocation checking is performed on the certificate provided by the queue manager. This property is ignored if no CipherSuite is set.

SSLSocketFactory

You might want to customize other aspects of the SSL connection for an application. For example, you might want to initialize cryptographic hardware; or change the KeyStore and TrustStore in use. To do this, the application should first create a javax.net.ssl.SSLSocketFactory instance customized accordingly. Please refer to your JSSE documentation for information on how to do this as the customizable features vary from provider to provider. Once a suitable SSLSocketFactory has been obtained, use the MQConnectionFactory.setSSLSocketFactory() method to configure JMS to use the customized SSLSocketFactory.

If your application uses setSSLSocketFactory() to set a customized SSLSocketFactory, the MQConnectionFactory can no longer be bound into a JNDI namespace. Attempting to do so will cause an Exception. If this property is not set, the default SSLSocketFactory will be used; please refer to your JSSE documentation for details on the behavior of the default SSLSocketFactory. This property is ignored if no CipherSuite is set.

Important: Do not assume that use of the SSL properties will ensure security when the ConnectionFactory will be retrieved from a JNDI namespace which is not itself secure. Specifically, the standard LDAP implementation of JNDI is not secure; it is possible for an attacker to spoof the LDAP server and hence fool a JMS application into connecting to the wrong server without noticing. With suitable security arrangements in place, other implementations of JNDI (such as the fscontext implementation) are secure.
Using SSL
Chapter 11. Programming publish/subscribe applications

You can write applications with WebSphere MQ JMS using two programming models:
- Point to point
- Publish/subscribe

This section considers publish/subscribe and how publish/subscribe messaging is implemented in WebSphere MQ JMS.

Introduction

With publish/subscribe messaging, one message producer can send messages to many message consumers at one time. The message producer need know nothing about the consumers receiving its messages, it only needs to know about the common destination. Similarly, the message consumers only need to know about the common destination. This common destination is called a topic.

A message producer that sends messages to a topic is a publisher and a message consumer that receives messages from a topic is a subscriber.

A message consumer will receive messages on all topics to which it has subscribed. To receive messages from a topic, a message consumer must first subscribe to that topic. All messages sent to a topic are forwarded to all the message consumers subscribed to that topic at that time. Each consumer receives its own copy of each message.

JMS clients can establish durable subscriptions that allow consumers to disconnect and later reconnect and collect messages published while they were disconnected.

The connection between messages issued by publishers and the subscribers is made, in WebSphere MQ, by the publish/subscribe broker. The broker (sometimes referred to as the message broker) has a record of all the subscribers registered to a topic. When a message is published to a topic, the broker manages the forwarding of that message to the topic’s subscribers.

To run a WebSphere MQ JMS publish/subscribe application, you must be able to connect to a message broker.

Getting started with WebSphere MQ JMS and publish/subscribe

Choosing a broker

WebSphere MQ offers a choice of three brokers:
- The MQSeries Publish/Subscribe broker is a publish/subscribe broker using WebSphere MQ and a SupportPac. The SupportPac MA0C is available for download from http://www.ibm.com/software/ts/mqseries/txppacs
- WebSphere MQ Integrator provides a broker that can be run in one of two modes. Compatibility mode, which provides a broker of equivalent functionality
to the WebSphere MQ broker; and native mode, which provides additional functionality. Currently, WebSphere MQ JMS can connect to WebSphere MQ Integrator in compatibility mode only.

- WebSphere MQ Event Broker provides a broker that can be connected to in two different ways:

  **Using message queues and WebSphere MQ**
  With this connection, you can run the broker in either compatibility mode or native mode.

  **Directly using a TCP/IP socket**
  With this connection, you can only run the broker in native mode. Also there is no support for:
  - Persistent messages
  - Transacted messages
  - Durable subscriptions

### Setting up the broker to run the WebSphere MQ JMS

Broker setup is dependent on the broker you intend to use and how you intend to use it. Each broker provides its own documentation describing installation and setup. However, for convenience and because of WebSphere MQ JMS requirements, some setup instructions are given here.

#### Connecting to your broker via WebSphere MQ

This section applies to the MQSeries Publish/Subscribe broker and the broker in WebSphere MQ Integrator. It also applies to the WebSphere MQ Event Broker broker when you choose to connect to it via WebSphere MQ.

Each broker requires its own queue manager. Please refer to the broker’s documentation regarding installation and set up.

For the WebSphere MQ JMS publish/subscribe implementation to work correctly, a number of system queues must be created on the queue manager on which the broker is running. These message queues must be created on each queue manager for each broker you wish to run WebSphere MQ JMS. WebSphere MQ JMS provides a script that creates these queues (see “Create the WebSphere MQ JMS system queues” on page 25).

Run the script to create the system queues. If you are using the MQSeries Publish/Subscribe broker, your broker is now fully configured. To check that the broker is correctly configured, run the publish/subscribe verification as described in “Publish/subscribe verification without JNDI” on page 32.

If you are using the broker provided by WebSphere MQ Integrator or WebSphere MQ Event Broker, you must configure a publish/subscribe message flow in the broker for messages to be correctly routed. The method for creating the required message flow is similar in both cases. Refer to Appendix D, “Connecting with other products” on page 409 for details.

#### Connecting to your broker directly

This is only possible when you use the broker provided in WebSphere MQ Event Broker. Because the connection to this broker is made directly, no system queues are required. However, you must set up a publish/subscribe message flow in the broker for messages to be correctly routed. Refer to Appendix D, “Connecting with other products” on page 409 for details.
Writing a simple publish/subscribe application connecting through WebSphere MQ

This section provides a walkthrough of a simple WebSphere MQ JMS application.

Here is the complete example. Individual sections are discussed after.

```java
/**
 * Basic pub/sub example
 *
 * A TopicConnectionFactory object is retrieved from LDAP; this is used
 * to create a TopicConnection. The TopicConnection is used to create
 * a TopicSession, which creates two publishers and two subscribers.
 * Both publishers subscribe to a topic; both subscribers then receive.
 */

import javax.jms.*; // JMS interfaces
import javax.naming.*; // Used for JNDI lookup of
import javax.naming.directory.*; // administered objects
import java.io.*; // Java IO classes
import java.util.*; // Java Util classes

class PubSubSample {
    // using LDAP
    String icf = "com.sun.jndi.ldap.LdapCtxFactory"; // initial context factory
    String url = "ldap://server.company.com/o=company_us,c=us"; //url
    private String tcfLookup = "cn=testTCF"; // TopicConnectionFactory (TCF) lookup
    private String tLookup = "cn=testT"; // topic lookup

    // Pub/Sub objects used by this program
    private TopicConnectionFactory fact = null;
    private Topic topic = null;

    public static void main(String args[]) {
        // Initialise JNDI properties
        Hashtable env = new Hashtable();
        env.put( Context.INITIAL_CONTEXT_FACTORY, icf );
        env.put( Context.PROVIDER_URL, url );
        env.put( Context.REFERRAL, "throw" );

        Context ctx = null;
        try {
            System.out.print("Initialising JNDI... ");
            ctx = new InitialDirContext( env );
            System.out.println("Done!");
        } catch (NamingException nx) {
            System.out.println("ERROR: "+nx);
            System.exit(-1);
        }

        // LOOKUP TCF
        try {
            System.out.print("Obtaining TCF from JNDI... ");
            fact = (TopicConnectionFactory)ctx.lookup( tcfLookup );
            System.out.println("Done!");
        } catch (NamingException nx) {
            System.out.println("ERROR: "+nx);
            System.exit(-1);
        }

        // LOOKUP TOPIC
        try {
            System.out.print("Obtaining topic T from JNDI... ");
        }
    }
}
```

Chapter 11. Programming publish/subscribe applications  205
Writing publish/subscribe application

```java
    topic = (Topic)ctx.lookup('tLookup');
    System.out.println("Done!");
} catch (NamingException nx) {
    System.out.println("ERROR: " + nx);
    System.exit(-1);
}

try {
    ctx.close();
} catch (NamingException nx) {
    // Just ignore an exception on closing the context
}

try {
    // create connection
    TopicConnection conn = fact.createTopicConnection();
    // start connection
    conn.start();

    // session
    TopicSession sess = conn.createTopicSession(false,
        Session.AUTO_ACKNOWLEDGE);

    // Create a topic dynamically
    Topic t = sess.createTopic("myTopic");

    // publisher
    TopicPublisher pub = sess.createPublisher(t);
    // subscriber
    TopicSubscriber sub = sess.createSubscriber(t);
    // publisher
    TopicPublisher pubA = sess.createPublisher(topic);
    // subscriber
    TopicSubscriber subA = sess.createSubscriber(topic);

    // publish "hello world"
    TextMessage hello = sess.createTextMessage();
    hello.setText("Hello World");
    pub.publish(hello);
    hello.setText("Hello World 2");
    pubA.publish(hello);

    // receive message
    TextMessage m = (TextMessage)sub.receive();
    System.out.println("Message Text = " + m.getText());
    m = (TextMessage)subA.receive();
    System.out.println("Message Text = " + m.getText());

    // close publishers and subscribers
    pub.close();
    pubA.close();
    sub.close();
    subA.close();

    // close session and connection
    sess.close();
    conn.close();

    System.exit(0);
}

  } catch (JMSException je) {
    System.out.println("ERROR: " + je);
    System.out.println("LinkedInException: " +
```
Import required packages

The import statements for an application using WebSphere MQ classes for Java Message Service should include at least the following:

```java
import javax.jms.*;  // JMS interfaces
import javax.naming.*;  // Used for JNDI lookup of
import javax.naming.directory.*;  // administered objects
```

Obtain or create JMS objects

The next step is to obtain or create a number of JMS objects:

1. Obtain a TopicConnectionFactory
2. Create a TopicConnection
3. Create a TopicSession
4. Obtain a Topic from JNDI
5. Create TopicPublishers and TopicSubscribers

Many of these processes are similar to those that are used for point-to-point, as shown in the following:

**Obtain a TopicConnectionFactory**

The preferred way to do this is to use JNDI lookup, so that portability of the application code is maintained. The following code initializes a JNDI context:

```java
String icf = "com.sun.jndi.ldap.LdapCtxFactory";  // initial context factory
String url = "ldap://server.company.com/o=company_us,c=us";  // url

// Initialise JNDI properties
Java.util.Hashtable env = new Hashtable();
env.put( Context.INITIAL_CONTEXT_FACTORY, icf );
env.put( Context.PROVIDER_URL, url );
env.put( Context.REFERRAL, "throw" );

Context ctx = null;
try {
    System.out.print( "Initialising JNDI... ");
    ctx = new InitialDirContext( env );
    System.out.println( "Done!" );
} catch ( NamingException nx ) {
    System.out.println( "ERROR: " + nx);
    System.exit(-1);
}
```

**Note:** The icf and url variables need to be changed to suit your installation and your JNDI service provider.

The properties required by JNDI initialization are in a hashtable, which is passed to the InitialDirContext constructor. If this connection fails, an exception is thrown to indicate that the administered objects required later in the application are not available.

Now obtain a TopicConnectionFactory by using a lookup key that the administrator has defined:
Writing publish/subscribe application

    // LOOKUP TCF
    try {
        System.out.print( "Obtaining TCF from JNDI... ");
        fact = (TopicConnectionFactory)ctx.lookup( tcfLookup );
        System.out.println( "Done!" );
    } catch ( NamingException nx ) {
        System.out.println( "ERROR: " + nx );
        System.exit(-1);
    }

If a JNDI namespace is not available, you can create a TopicConnectionFactory at runtime. You create a new com.ibm.mq.jms.MQTopicConnectionFactory as described in "Creating factories at runtime" on page 191.

Create a TopicConnection
This is created from the TopicConnectionFactory object. Connections are always initialized in a stop state and must be started with the following code:

    // create connection
    TopicConnection conn = fact.createTopicConnection();
    // start connection
    conn.start();

Create a TopicSession
This is created by using the TopicConnection. This method takes two parameters; one to signify whether the session is transacted, and one to specify the acknowledgement mode:

    TopicSession sess = conn.createTopicSession(false,
        Session.AUTO_ACKNOWLEDGE);

Obtain a Topic
This object can be obtained from JNDI, for use with TopicPublishers and TopicSubscribers that are created later. The following code retrieves a Topic:

    Topic topic = null;
    try {
        System.out.print( "Obtaining topic T from JNDI... ");
        topic = (Topic)ctx.lookup( tLookup );
        System.out.println( "Done!" );
    } catch ( NamingException nx ) {
        System.out.println( "ERROR: " + nx );
        System.exit(-1);
    }

If a JNDI namespace is not available, you can create a Topic at runtime, as described in "Creating topics at runtime" on page 212.

The following code creates a Topic at runtime in this example:

    // topic
    Topic t = sess.createTopic("myTopic");

Create consumers and producers of publications
Depending on the nature of the JMS client application that you write, a subscriber, a publisher, or both must be created. Use the createPublisher and createSubscriber methods as follows:

    // publisher
    TopicPublisher pub = sess.createPublisher(t);
    // subscriber
    TopicSubscriber sub = sess.createSubscriber(t);
    // publisher
Writing publish/subscribe application

```java
TopicPublisher pubA = sess.createPublisher(topic);
// subscriber
TopicSubscriber subA = sess.createSubscriber(topic);
```

### Publish messages

The `TopicPublisher` object, `pub`, is used to publish messages, rather like a `QueueSender` is used in the point-to-point domain. The following fragment creates a `TextMessage` by using the session, and then publishes the message:

```java
// publish "hello world"
TextMessage hello = sess.createTextMessage();
hello.setText("Hello World");
pub.publish(hello);
hello.setText("Hello World 2");
pubA.publish(hello);
```

### Receive subscriptions

Subscribers must be able to read the subscriptions that are delivered to them, as in the following code:

```java
// receive message
TextMessage m = (TextMessage) sub.receive();
System.out.println("Message Text = \" + m.getText());
m = (TextMessage) subA.receive();
System.out.println("Message Text = \" + m.getText());
```

This fragment of code performs a "get-with-wait", which means that the receive call will block until a message is available. Alternative versions of the receive call are available (such as "receiveNoWait"). For details, see "TopicSubscriber" on page 382.

### Close down unwanted resources

It is important to free up all the resources used by the application when it terminates. Use the `close()` method on objects that can be closed (publishers, subscribers, sessions, and connections):

```java
// close publishers and subscribers
pub.close();
pubA.close();
sub.close();
subA.close();
sess.close();
```

```
// close session and connection
sess.close();
conn.close();
```

### TopicConnectionFactory administered objects

In the example, the `TopicConnectionFactory` object is obtained from JNDI name space. The `TopicConnectionFactory` in this case is an administered object that has been created and administered using the `JMSAdmin` tool. Use this method of obtaining `TopicConnectionFactory` objects because it ensures code portability.

The `TopicConnectionFactory` in the example is `testTCF` in `JMSAdmin`. `testTCF` must be created in `JMSAdmin` **before** running the application. A `Topic` must also be created in `JMSAdmin`; see "Topic administered objects" on page 210.

To create a `TopicConnectionFactory` object, invoke the `JMSAdmin` tool, as described in "Invoking the Administration tool" on page 39, and execute one of the following commands, depending on the type of connection you wish to make to the broker:
Writing publish/subscribe application

Bindings connection

```
InitCtx> def tcf(testTCF) transport(bind)
```

or, because this is the default transport type for TopicConnectionFactory objects,

```
InitCtx> def tcf(testTCF)
```

This will create a TopicConnectionFactory with default settings for bindings transport, connecting to the default queue manager.

Client connection

```
InitCtx> def tcf(testTCF) transport(client)
```

This will create a TopicConnectionFactory with default settings for the client transport type, connecting to localhost, on port 1414, using channel SYSTEM.DEF.SVRCONN.

Direct TCP/IP connection to WebSphere MQ Event Broker

```
InitCtx> def tcf(testTCF) transport(direct)
```

This will create a TopicConnectionFactory to make direct connections to a WebSphere MQ Event Broker broker, connecting to localhost on port 1506.

**Topic administered objects**

In the example, one of the Topic objects has been obtained from JNDI name space. This Topic is an administered object that has been created and administered in the JMSAdmin tool. Use this method of obtaining Topic objects because it ensures code portability.

To run the example application above, the Topic called testT must have been created in JMSAdmin before running the application.

To create a Topic object, invoke the JMSAdmin tool, as described in "Invoking the Administration tool" on page 39 and execute one of the following commands, depending on the type of connection you wish to make to the broker:

**Compatibility mode, or MQSeries Publish/Subscribe (SupportPac MA0C)**

```
InitCtx> def t(testT) bver(V1) topic(test/topic)
```

**Native mode, or direct to WebSphere MQ Event Broker**

```
InitCtx> def t(testT) bver(V2) topic(test/topic)
```

---

**Using topics**

This section discusses the use of JMS Topic objects in WebSphere MQ classes for Java Message Service applications.

**Topic names**

This section describes the use of topic names within WebSphere MQ classes for Java Message Service.

**Note:** The JMS specification does not specify exact details about the use and maintenance of topic hierarchies. Therefore, this area can vary from one provider to the next.
Topic names in WebSphere MQ JMS are arranged in a tree-like hierarchy, an example of which is shown in Figure 3.

In a topic name, levels in the tree are separated by the “/” character. This means that the “Signings” node in Figure 3 is identified by the topic name: Sport/Football/Spurs/Signings

A powerful feature of the topic system in WebSphere MQ classes for Java Message Service is the use of wildcards. These allow subscribers to subscribe to more than one topic at a time. Different brokers use different wildcard characters and different rules for their substitution. Use the broker version property of the topic (BROKERVER) to define which type of wildcards apply.

**Note:** The broker version of a topic must match the broker version of the topic connection factory you are using.

**Broker Version 1 wildcards**

The “*” wildcard matches zero or more characters, while the “?” wildcard matches a single character.

If a subscriber subscribes to the topic represented by the following topic name:

Sport/Football/*/Results

it receives publications on topics including:

- Sport/Football/Spurs/Results
- Sport/Football/Arsenal/Results

If the subscription topic is:

Sport/Football/Spurs/*

it receives publications on topics including:

- Sport/Football/Spurs/Results
- Sport/Football/Spurs/Signings

If the subscription topic is:

Sport/Football/*/ Results

it receives publications on topics including:
Using topics

- Sport/Football/Arsenal/Results
- Sport/Football/Spurs/Results
- Sport/Football/Spurs/Signings

Broker Version 2 wildcards

The "#" wildcard matches multiple levels in a topic, the "+" wildcard matches a single level.

These wildcards can only be used to stand for complete levels within a topic; that is they can only be preceded by "/" or by start-of-string, and they can only be followed by "/" or by end-of-string.

If a subscriber subscribes to the topic represented by the following topic name:
Sport/Football/+/Results

it receives publications on topics including:
- Sport/Football/Spurs/Results
- Sport/Football/Arsenal/Results

If a subscriber subscribes to the topic represented by the following topic name:
Sport/#/Results

it receives publications on topics including:
- Sport/Football/Spurs/Results
- Sport/Football/Arsenal/Results

Although
Sport/Football/Spur?/Results

works with broker version 1, there is no equivalent for broker version 2 which does not support single character substitutions.

There is no need to administer the topic hierarchies that you use on the broker-side of your system explicitly. When the first publisher or subscriber on a given topic comes into existence, the broker automatically creates the state of the topics currently being published on, and subscribed to.

Note: A publisher cannot publish on a topic whose name contains wildcards.

Creating topics at runtime

There are four ways to create Topic objects at runtime:

1. Construct a topic by using the one-argument MQTopic constructor
2. Construct a topic by using the default MQTopic constructor, and then call the setBaseTopicName(..) method
3. Use the session’s createTopic(..) method
4. Use the session’s createTemporaryTopic() method

Method 1: Using MQTopic(..)

This method requires a reference to the WebSphere MQ implementation of the JMS Topic interface, and therefore renders the code non-portable.
The constructor takes one argument, which should be a uniform resource identifier (URI). For WebSphere MQ classes for Java Message Service Topics, this should be of the form:

```
topic://TopicName[?property=value[&property=value]*]
```

For further details on URIs and the permitted name-value pairs, see “Sending a message” on page 193.

The following code creates a topic for non-persistent, priority 5 messages:

```
// Create a Topic using the one-argument MQTopic constructor
String tSpec = "Sport/Football/Spurs/Results?persistence=1&priority=5";
Topic rtTopic = new MQTopic( "topic://" + tSpec );
```

**Method 2: Using MQTopic(), then setBaseTopicName(..)**

This method uses the default MQTopic constructor, and therefore renders the code non-portable.

After the object is created, set the baseTopicName property by using the setBaseTopicName method, passing in the required topic name.

**Note:** The topic name used here is the non-URI form, and cannot include name-value pairs. Set these by using the “set” methods, as described in “Setting properties with the ‘set’ method” on page 195. The following code uses this method to create a topic:

```
// Create a Topic using the default MQTopic constructor
Topic rtTopic = new MQTopic();

// Set the object properties using the setter methods
((MQTopic)rtTopic).setBaseTopicName( "Sport/Football/Spurs/Results" );
((MQTopic)rtTopic).setPersistence(1);
((MQTopic)rtTopic).setPriority(5);
```

**Method 3: Using session.createTopic(..)**

A Topic object may also be created by using the createTopic method of TopicSession, which takes a topic URI as follows:

```
// Create a Topic using the session factory method
Topic rtTopic = session.createTopic( "topic://Sport/Football/Spurs/Results" );
```

Although the createTopic method is in the JMS specification, the format of the string argument is vendor-specific. Therefore, using this method may make your code non-portable.

**Method 4: Using session.createTemporaryTopic()**

A TemporaryTopic is a Topic that may be consumed only by subscribers that are created by the same TopicConnection. A TemporaryTopic is created as follows:

```
// Create a TemporaryTopic using the session factory method
Topic rtTopic = session.createTemporaryTopic();
```

**Subscriber options**

There are a number of different ways to use JMS subscribers. This section describes some examples of their use.

JMS provides two types of subscribers:

**Non-durable subscribers**

These subscribers receive messages on their chosen topic, only if the messages are published while the subscriber is active.
Subscriber options

Durable subscribers
These subscribers receive all the messages published on a topic, including those that are published while the subscriber is inactive.

Creating non-durable subscribers
The subscriber created in "Create consumers and producers of publications" on page 208 is non-durable and is created with the following code:

```java
// Create a subscriber, subscribing on the given topic
TopicSubscriber sub = session.createSubscriber( topic );
```

Creating durable subscribers
Durable subscribers cannot be configured with a direct connection to WebSphere MQ Event Broker.

Creating a durable subscriber is very similar to creating a non-durable subscriber, but you must also provide a name that uniquely identifies the subscriber:

```java
// Create a durable subscriber, supplying a uniquely-identifying name
TopicSubscriber sub = session.createDurableSubscriber( topic, "D_SUB_000001" );
```

Non-durable subscribers automatically deregister themselves when their close() method is called (or when they fall out of scope). However, if you wish to terminate a durable subscription, you must explicitly notify the system. To do this, use the session's unsubscribe() method and pass in the unique name that created the subscriber:

```java
// Unsubscribe the durable subscriber created above
session.unsubscribe( "D_SUB_000001" );
```

A durable subscriber is created at the queue manager specified in the MQTopicConnectionFactory queue manager parameter. If there is a subsequent attempt to create a durable subscriber with the same name at a different queue manager, a new and completely independent durable subscriber is returned.

Using message selectors
You can use message selectors to filter out messages that do not satisfy given criteria. For details about message selectors, see "Message selectors" on page 196.

Message selectors are associated with a subscriber as follows:

```java
// Associate a message selector with a non-durable subscriber
String selector = "company = 'IBM'";
TopicSubscriber sub = session.createSubscriber( topic, selector, false );
```

Suppressing local publications
It is possible to create a subscriber that ignores publications that are published on the subscriber's own connection. Set the third parameter of the createSubscriber call to true, as follows:

```java
// Create a non-durable subscriber with the noLocal option set
TopicSubscriber sub = session.createSubscriber( topic, null, true );
```
Combining the subscriber options

You can combine the subscriber variations, so that you can create a durable subscriber that applies a selector and ignores local publications, if you wish to. The following code fragment shows the use of the combined options:

```
// Create a durable, noLocal subscriber with a selector applied
String selector = "company = 'IBM'";
TopicSubscriber sub = session.createDurableSubscriber( topic, "D_SUB_000001",
selector, true );
```

Configuring the base subscriber queue

Subscriber queues cannot be configured for a direct connection to WebSphere MQ Event Broker.

With WebSphere MQ JMS from V5.2, there are two ways in which you can configure subscribers:

- Multiple queue approach
  Each subscriber has an exclusive queue assigned to it, from which it retrieves all its messages. JMS creates a new queue for each subscriber. This is the only approach available with WebSphere MQ JMS V1.1.

- Shared queue approach
  A subscriber uses a shared queue, from which it, and other subscribers, retrieve their messages. This approach requires only one queue to serve multiple subscribers. This is the default approach used with WebSphere MQ JMS from V5.2.

From WebSphere MQ JMS V5.2, you can choose which approach to use, and configure which queues to use.

In general, the shared queue approach gives a modest performance advantage. For systems with a high throughput, there are also large architectural and administrative advantages, because of the significant reduction in the number of queues required.

In some situations, there are still good reasons for using the multiple queue approach:

- The theoretical physical capacity for message storage is greater.
  A WebSphere MQ queue cannot hold more than 640000 messages, and in the shared queue approach, this must be divided between all the subscribers that share the queue. This issue is more significant for durable subscribers, because the lifetime of a durable subscriber is usually much longer than that of a non-durable subscriber. Therefore, more messages might accumulate for a durable subscriber.

- External administration of subscription queues is easier.
  For certain application types, administrators may wish to monitor the state and depth of particular subscriber queues. This task is much simpler when there is one to one mapping between a subscriber and a queue.

Default configuration

The default configuration uses the following shared subscription queues:

- SYSTEM.JMS.ND.SUBSCRIBER.QUEUE for non-durable subscriptions
- SYSTEM.JMS.D.SUBSCRIBER.QUEUE for durable subscriptions

These are created for you when you run the MQJMS_PSQ.MQSC script.
Subscriber options

If required, you can specify alternative physical queues. You can also change the configuration to use the multiple queue approach.

Configuring non-durable subscribers
You can set the non-durable subscriber queue name property in either of the following ways:

- Use the WebSphere MQ JMS administration tool (for JNDI retrieved objects) to set the BROKERSUBQ property
- Use the setBrokerSubQueue() method in your program

For non-durable subscriptions, the queue name you provide should start with the following characters:

```
SYSTEM.JMS.ND.
```

To select a shared queue approach, specify an explicit queue name, where the named queue is the one to use for the shared queue. The queue that you specify must already physically exist before you create the subscription.

To select the multiple queue approach, specify a queue name that ends with the * character. Subsequently, each subscriber that is created with this queue name creates an appropriate dynamic queue, for exclusive use by that particular subscriber. MQ JMS uses its own internal model queue to create such queues. Therefore, with the multiple queue approach, all required queues are created dynamically.

When you use the multiple queue approach, you cannot specify an explicit queue name. However, you can specify the queue prefix. This enables you to create different subscriber queue domains. For example, you could use:

```
SYSTEM.JMS.ND.MYDOMAIN.*
```

The characters that precede the * character are used as the prefix, so that all dynamic queues that are associated with this subscription will have queue names that start with SYSTEM.JMS.ND.MYDOMAIN.

Configuring durable subscribers
As discussed earlier, there may still be good reasons to use the multiple queue approach for durable subscriptions. Durable subscriptions are likely to have a longer life span, so it is possible that a large number of un-retrieved messages could accumulate on the queue.

Therefore, the durable subscriber queue name property is set in the Topic object (that is, at a more manageable level than TopicConnectionFactory). This enables you to specify a number of different subscriber queue names, without needing to re-create multiple objects starting from the TopicConnectionFactory.

You can set the durable subscriber queue name in either of the following ways:

- Use the WebSphere MQ JMS administration tool (for JNDI retrieved objects) to set the BROKERDURSUBQ property
- Use the setBrokerDurSubQueue() method in your program:

```
// Set the MQTopic durable subscriber queue name using
// the multi-queue approach
sportTopic.setBrokerDurSubQueue("SYSTEM.JMS.D.FOOTBALL.*");
```

Once the Topic object is initialized, it is passed into the TopicSession createDurableSubscriber() method to create the specified subscription:
// Create a durable subscriber using our earlier Topic
TopicSubscriber sub = new session.createDurableSubscriber
(sportsTopic, "D_SUB_SPORT_001");

For durable subscriptions, the queue name you provide should start with the following characters:
SYSTEM.JMS.D.

To select a shared queue approach, specify an explicit queue name, where the named queue is the one to use for the shared queue. The queue that you specify must already physically exist before you create the subscription.

To select the multiple queue approach, specify a queue name that ends with the * character. Subsequently, each subscriber that is created with this queue name creates an appropriate dynamic queue, for exclusive use by that particular subscriber. MQ JMS uses its own internal model queue to create such queues. Therefore, with the multiple queue approach, all required queues are created dynamically.

When you use the multiple queue approach, you cannot specify an explicit queue name. However, you can specify the queue prefix. This enables you to create different subscriber queue domains. For example, you could use:
SYSTEM.JMS.D.MYDOMAIN.*

The characters that precede the * character are used as the prefix, so that all dynamic queues that are associated with this subscription will have queue names that start with SYSTEM.JMS.D.MYDOMAIN.

Note that you can not change the queue used by a durable subscriber. To do so, for example to move from the multiple queue approach to the single queue approach, you should first delete the old subscriber (using the unsubscribe() method) and create the subscription again from new. This will remove any unconsumed messages on the durable subscription.

**Subscription stores**

There is no subscription store with a direct connection to WebSphere MQ Event Broker.

WebSphere MQ JMS maintains a persistent store of subscription information, used to resume durable subscriptions and cleanup after failed non-durable subscribers. Before WebSphere MQ JMS 5.2.1, this information was stored using messages on system queues on the application’s local queue manager; from WebSphere MQ JMS 5.2.1 this information can be managed by the publish/subscribe broker. This new subscription store requires recent levels of both queue manager and publish/subscribe broker. It is designed to provide improved resilience.

Please see the readme provided with WebSphere MQ JMS for information about suitable levels of queue manager and broker.

The choice of subscription store is based on the SUBSTORE property of the TopicConnectionFactory. This takes one of three values, QUEUE, BROKER or MIGRATE.

**SUBSTORE(QUEUE)**

Subscription information is stored on SYSTEM.JMS.ADMIN.QUEUE and SYSTEM.JMS.PS.STATUS.QUEUE on the local queue manager.
Subscriber options

WebSphere MQ JMS maintains an extra connection for a long-running transaction used to detect failed subscribers. There is a connection to each queue manager in use. In a busy system, this may cause the queue manager logs to fill up and this will result in errors from both the queue manager and its applications.

If you experience these problems, the system administrator can add extra log files or datasets to the queue manager. Alternatively, the STATREFRESHINT property on the TopicConnectionFactory can be reduced. This will cause the long-running transaction to be refreshed more frequently, allowing the logs to reset themselves.

After a non-durable subscriber has failed, information is left on the two SYSTEM.JMS queues which allows a later JMS application to cleanup after the failed subscriber. See “Subscriber cleanup utility” on page 220 for more information.

After failure of a non-durable subscriber, messages will continue to be delivered to the subscriber until a later JMS application is executed.

SUBSTORE(QUEUE) is provided for compatibility with versions of WebSphere MQ JMS before 5.2.1.

SUBSTORE(BROKER)

Subscription information is stored by the publish/subscribe broker used by the application. This option requires recent levels of queue manager and publish/subscribe broker. Please see the readme provided with WebSphere MQ JMS for information about suitable levels of Queue Manager and broker.

When a non-durable subscriber fails, the subscription is deregistered from the broker as soon as possible. The broker adds a response to this deregistration onto the SYSTEM.JMS.REPORT.QUEUE, which is used to cleanup after the failed subscriber. With SUBSTORE(BROKER), a separate cleanup thread is run regularly in the background of each JMS publish/subscribe application.

Cleanup is run once every 10 minutes by default, but you can change this using the CLEANUPINT property on the TopicConnectionFactory. The actions performed by cleanup can be customised using the CLEANUP property on the TopicConnectionFactory.

See “Subscriber cleanup utility” on page 220 for more information about the different behaviours of cleanup with SUBSTORE(BROKER).

SUBSTORE(MIGRATE)

MIGRATE is the default value for SUBSTORE.

This option dynamically selects the Queue-based or broker-based subscription store based on the levels of queue manager and publish/subscribe broker installed. If both queue manager and broker are capable of supporting SUBSTORE(BROKER) then this will behave as SUBSTORE(BROKER); otherwise it will behave as SUBSTORE(QUEUE). Additionally, SUBSTORE(MIGRATE) will transfer durable subscription information from the Queue-based subscription store to the broker-based store.

This provides an easy migration path from older versions of WebSphere MQ JMS, WebSphere MQ and publish/subscribe broker. This migration occurs the first time the durable subscription is opened when both Queue Manager and broker are capable of supporting the broker-based store.
Migration and coexistence considerations
Except when SUBSTORE(MIGRATE) is used, the two subscription stores are entirely independent.

A durable subscription will be created in the store specified by the TopicConnectionFactory. If there is a subsequent attempt to create a durable subscriber with the same name and ClientID but with the other subscription store, a new durable subscription will be created.

When a connection uses SUBSTORE(MIGRATE), subscription information will automatically be transferred from the Queue-based subscription store to the broker-based subscription store when createDurableSubscriber() is called. If a durable subscription with matching name and ClientID already exists in the broker-based subscription store, this migration will not be able to complete and an exception will be thrown from createDurableSubscriber().

Hence, once a subscription has been migrated, it is important that the subscription is not accessed from an application using an older version of WebSphere MQ JMS, or an application running with SUBSTORE(QUEUE). This would create a subscription in the Queue-based subscription store, and prevent an application running with SUBSTORE(MIGRATE) from using the subscription.

To recover from this situation, either use SUBSTORE(BROKER) from your application or unsubscribe from the subscription with SUBSTORE(QUEUE).

In order for SUBSTORE(BROKER) to function, or for SUBSTORE(MIGRATE) to use the broker-based subscription store, suitable versions of both Queue Manager and broker need to be available to WebSphere MQ JMS. Please refer to the readme for information regarding suitable levels.

Solving publish/subscribe problems
This section describes some problems that can occur when you develop JMS client applications that use the publish/subscribe domain. Note that this section discusses problems that are specific to the publish/subscribe domain. Refer to “Handling errors” on page 198 and “Solving problems” on page 36 for more general troubleshooting guidance.

Incomplete publish/subscribe close down
It is important that JMS client applications surrender all external resources when they terminate. To do this, call the close() method on all objects that can be closed once they are no longer required. For the publish/subscribe domain, these objects are:

- TopicConnection
- TopicSession
- TopicPublisher
- TopicSubscriber

The WebSphere MQ classes for Java Message Service implementation eases this task through the use of a “cascading close”. With this process, a call to “close” on a
TopicConnection results in calls to “close” on each of the TopicSessions it created. This in turn results in calls to “close” on all TopicSubscribers and TopicPublishers the sessions created.

Therefore, to ensure the proper release of external resources, it is important to call connection.close() for each of the connections that an application creates.

There are some circumstances where this “close” procedure may not complete. These include:

- Loss of a WebSphere MQ client connection
- Unexpected application termination

In these circumstances, the close() is not called, and external resources remain open on the terminated application’s behalf. The main consequences of this are:

**Broker state inconsistency**

The WebSphere MQ Message Broker may well contain registration information for subscribers and publishers that no longer exist. This means that the broker may continue forwarding messages to subscribers that will never receive them.

**Subscriber messages and queues remain**

Part of the subscriber deregistration procedure is the removal of subscriber messages. If appropriate, the underlying WebSphere MQ queue that was used to receive subscriptions is also removed. If normal closure has not occurred, these messages and queues remain. If there is broker state inconsistency, the queues continue to fill up with messages that will never be read.

Additionally, if the broker resides on a queue manager other than the application’s local queue manager, correct operation of WebSphere MQ JMS depends on the communication channels between the two queue managers. If these channels fail for any reason, problems such as the above can occur until the channels restart. When diagnosing problems relating to channels, be careful not to lose WebSphere MQ JMS control messages on the transmission queue.

**Subscriber cleanup utility**

To avoid the problems associated with non-graceful closure of subscriber objects, WebSphere MQ JMS includes a subscriber cleanup utility which attempts to detect any earlier WebSphere MQ JMS publish/subscribe problems that could have resulted from other applications. This utility runs transparently in the background and should not affect other WebSphere MQ JMS operations. If a large number of problems are detected against a given queue manager, some performance degradation may be observed while resources are cleaned up.

**Note:** You should still close all subscriber objects gracefully whenever possible, to avoid a buildup of subscriber problems.

The exact behavior of the utility depends on the subscription store in use:

**Subscriber Cleanup with SUBSTORE(QUEUE)**

When using the queue-based subscription store, Cleanup runs on a queue manager when the first TopicConnection to use that physical queue manager initializes.
If all the TopicConnections on a given queue manager become closed, when the next TopicConnection initializes for that queue manager, the utility runs again.

The Cleanup utility uses information found on the SYSTEM.JMS.ADMIN.QUEUE and SYSTEM.JMS.PS.STATUS.QUEUE to detect previously failed subscribers. If any are found, it cleans up associated resources by deregistering the subscriber from the broker, and cleaning up any unconsumed messages or temporary queues associated with the subscription.

**Subscriber Cleanup with SUBSTORE(BROKER)**

With the broker-based subscription store, Cleanup runs regularly on a background thread while there is an open TopicConnection to a particular physical queue manager. One instance of the Cleanup thread is created for each physical queue manager to which a TopicConnection exists within the JVM.

The Cleanup utility uses information found on the SYSTEM.JMS.REPORT.QUEUE (typically responses from the publish/subscribe broker) to remove unconsumed messages and temporary queues associated with a failed subscriber. It can be a few seconds after the subscriber fails before the Cleanup routine is able to remove the messages and queues.

Two properties on the TopicConnectionFactory control behavior of this Cleanup thread: CLEANUP and CLEANUPINT. CLEANUPINT determines how often (in milliseconds) Cleanup is executed against any given queue manager. CLEANUP takes one of four possible values:

**CLEANUP(SAFE)**

This is the default value.

The Cleanup thread will attempt to remove unconsumed subscription messages, or temporary queues, for failed subscriptions. This mode of Cleanup will not interfere with the operation of other JMS applications.

**CLEANUP(STRONG)**

The Cleanup thread will perform as CLEANUP(SAFE), but will also clear the SYSTEM.JMS.REPORT.QUEUE of any unrecognised messages.

This mode of Cleanup may interfere with the operation of JMS applications running with later versions of WebSphere MQ JMS. Hence, it is important that if multiple JMS applications are using the same queue manager, but using different versions of WebSphere MQ JMS, only clients using the most recent version of WebSphere MQ JMS use this option.

**CLEANUP(NONE)**

In this special mode, no Cleanup is performed; and no Cleanup thread exists. Additionally, if the application is using the single-queue approach, unconsumed messages can be left on the queue.

This option can be useful if the application is distant from the Queue Manager, and especially if it only publishes rather than subscribes. However, some application should perform Cleanup on the queue manager to deal with any unconsumed messages - this
could be a JMS application with CLEANUP(SAFE) or CLEANUP(STRONG), or the manual Cleanup utility described in "Manual cleanup".

**CLEANUP(ASPROP)**

The style of Cleanup to use is determined by the system property `com.ibm.mq.jms.cleanup`, which is queried at JVM startup.

This property can be set on the java command-line using the `-D` option, and should be set to NONE, SAFE or STRONG. Any other value causes an exception. If not set, the property defaults to SAFE. This allows easy JVM-wide change to the Cleanup level without needing to update every TopicConnectionFactory used by the system. This may be useful for applications or application servers which use multiple TopicConnectionFactory objects.

Where multiple TopicConnections exist within a JVM, against the same queue manager, but with differing values for CLEANUP and CLEANUPINT, the following rules are used to determine behavior:

1. A TopicConnection with CLEANUP(NONE) will not attempt to clean up immediately after its subscription has closed. However, if another TopicConnection is using SAFE or STRONG Cleanup, the Cleanup thread will eventually clean up after the subscription.
2. If any TopicConnection is using STRONG Cleanup, then the cleanup thread will operate at STRONG level. Otherwise, if any TopicConnection is using SAFE Cleanup, then the cleanup thread will operate at SAFE level. Otherwise, there will be no cleanup thread.
3. The smallest value for CLEANUPINT of those TopicConnections with SAFE or STRONG Cleanup will be used.

**Manual cleanup**

If you use the broker-based subscription store, Cleanup can be operated manually from the command-line. The syntax for bindings attach is:

```
Cleanup [-m <qmgr>] [-r <interval>] [SAFE | STRONG | FORCE | NONDUR] [-t]
```

or, for client attach:

```
Cleanup -client [-m <qmgr>] -host <hostname> [-port <port>] [-channel <channel>] [-r <interval>] [SAFE | STRONG | FORCE | NONDUR] [-t]
```

Where:

- `qmgr`, `hostname`, `port` and `channel` determine connection settings for the queue manager to clean up
- `-r` sets the interval between executions of Cleanup, in minutes. If not set, Cleanup will be performed once.
- `-t` enables tracing, to the `mqjms.trc` file
- SAFE, STRONG, FORCE and NONDUR set the Cleanup level, as follows:
  - SAFE and STRONG Cleanup behave like the CLEANUP(SAFE) and CLEANUP(STRONG) options discussed in "Subscriber cleanup utility" on page 220
– FORCE Cleanup behaves like STRONG Cleanup. However, while STRONG Cleanup will leave messages on the SYSTEM.JMS.REPORT.QUEUE which could not be processed, FORCE Cleanup will remove all messages even if it encounters an error during processing.

This is a dangerous option which can leave an inconsistent state between the queue manager and the broker. It can not be run while any WebSphere MQ JMS publish/subscribe application is running against the queue manager; doing so will cause the Cleanup utility to abort.

– NONDUR behaves like FORCE Cleanup. After clearing the SYSTEM.JMS.REPORT.QUEUE, it attempts to remove any remaining unconsumed messages sent to non-durable subscribers. If the queue manager’s command server is running, on any queue beginning SYSTEM.JMS.ND.* messages will be cleared and the queue itself may be deleted. Otherwise, only SYSTEM.JMS.ND.SUBSCRIBER.QUEUE and SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE will be cleared of messages.

### Cleanup from within a program

You can use a programming interface to the Cleanup routines for use with SUBSTORE(BROKER), through the class com.ibm.mq.jms.Cleanup. Instances of this class have getter/setter methods for each of the connection properties; and also for the cleanup level and interval.

It exposes two methods:

- cleanup() - executes Cleanup once
- run() - runs Cleanup at intervals determined by the properties of the Cleanup object

This class allows complete customization of publish/subscribe Cleanup, but it is intended for use by system administration programs rather than application programs.

For more details, refer to "Cleanup *" on page 274.

### Handling broker reports

The WebSphere MQ JMS implementation uses report messages from the broker to confirm registration and deregistration commands. These reports are normally consumed by the WebSphere MQ classes for Java Message Service implementation, but under some error conditions, they may remain on the queue. These messages are sent to the SYSTEM.JMS.REPORT.QUEUE queue on the local queue manager.

A Java application, PSReportDump, is supplied with WebSphere MQ classes for Java Message Service, which dumps the contents of this queue in plain text format. The information can then be analyzed, either by the user, or by IBM support staff. You can also use the application to clear the queue of messages after a problem is diagnosed or fixed.

The compiled form of the tool is installed in the `<MQ_JAVA_INSTALL_PATH>/bin` directory. To invoke the tool, change to this directory, then use the following command:

```
java PSReportDump [-m queueManager] [-clear]
```

where:
Publish/subscribe problems

-\texttt{-m queueManager}
  The name of the queue manager to use

-\texttt{-clear}
  Clear the queue of messages after dumping its contents

\textbf{Attention:} Do not use this option if you are using the Broker-based subscription store. Instead, run the manual cleanup utility at FORCE level.

Output is sent to the screen, or you can redirect it to a file.
Chapter 12. JMS messages

JMS Messages are composed of the following parts:

**Header**

All messages support the same set of header fields. Header fields contain values that are used by both clients and providers to identify and route messages.

**Properties**

Each message contains a built-in facility to support application-defined property values. Properties provide an efficient mechanism to filter application-defined messages.

**Body**

JMS defines several types of message body which cover the majority of messaging styles currently in use.

JMS defines five types of message body:

- **Stream**
  a stream of Java primitive values. It is filled and read sequentially.

- **Map**
  a set of name-value pairs, where names are Strings and values are Java primitive types. The entries can be accessed sequentially or randomly by name. The order of the entries is undefined.

- **Text**
  a message containing a java.util.String.

- **Object**
  a message that contains a Serializable java object

- **Bytes**
  a stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.

The JMSCorrelationID header field is used to link one message with another. It typically links a reply message with its requesting message. JMSCorrelationID can hold a provider-specific message ID, an application-specific String, or a provider-native byte[] value.

**Message selectors**

A Message contains a built-in facility to support application-defined property values. In effect, this provides a mechanism to add application-specific header fields to a message. Properties allow an application, via message selectors, to have a JMS provider select or filter messages on its behalf, using application-specific criteria. Application-defined properties must obey the following rules:

- Property names must obey the rules for a message selector identifier.
- Property values can be boolean, byte, short, int, long, float, double, and string.
- The following name prefixes are reserved: JMSX, JMS_.

Property values are set before sending a message. When a client receives a message, the message properties are read-only. If a client attempts to set properties at this point, a MessageNotWritableException is thrown. If clearProperties is called, the properties can now be both read from, and written to.

A property value may duplicate a value in a message's body, or it may not. JMS does not define a policy for what should or should not be made into a property.
Message selectors

However, application developers should note that JMS providers will probably handle data in a message’s body more efficiently than data in a message’s properties. For best performance, applications should only use message properties when they need to customize a message’s header. The primary reason for doing this is to support customized message selection.

A JMS message selector allows a client to specify the messages that it is interested in by using the message header. Only messages whose headers match the selector are delivered.

Message selectors cannot reference message body values.

A message selector matches a message when the selector evaluates to true when the message’s header field and property values are substituted for their corresponding identifiers in the selector.

A message selector is a String, whose syntax is based on a subset of the SQL92 conditional expression syntax. The order in which a message selector is evaluated is from left to right within a precedence level. You can use parentheses to change this order. Predefined selector literals and operator names are written here in upper case; however, they are not case-sensitive.

A selector can contain:

- Literals
  - A string literal is enclosed in single quotes. A doubled single quote represents a single quote. Examples are ‘literal’ and ‘literal’\’s’. Like Java string literals, these use the Unicode character encoding.
  - An exact numeric literal is a numeric value without a decimal point, such as 57, -957, +62. Numbers in the range of Java long are supported.
  - An approximate numeric literal is a numeric value in scientific notation, such as 7E3 or -57.9E2, or a numeric value with a decimal, such as 7.0, -95.7, or +6.2. Numbers in the range of Java double are supported.
  - The boolean literals TRUE and FALSE.

- Identifiers:
  - An identifier is an unlimited length sequence of Java letters and Java digits, the first of which must be a Java letter. A letter is any character for which the method Character.isJavaLetter returns true. This includes “_” and “$”. A letter or digit is any character for which the method Character.isJavaLetterOrDigit returns true.
  - Identifiers cannot be the names NULL, TRUE, or FALSE.
  - Identifiers cannot be NOT, AND, OR, BETWEEN, LIKE, IN, and IS.
  - Identifiers are either header field references or property references.
  - Identifiers are case-sensitive.
  - Message header field references are restricted to:
    - JMSDeliveryMode
    - JMSPriority
    - JMSMessageID
    - JMSTimestamp
    - JMSCorrelationID
    - JMSType
Message selectors

JMSMessageID, JMSTimestamp, JMSCorrelationID, and JMSType values may be null, and if so, are treated as a NULL value.
- Any name beginning with “JMSX” is a JMS-defined property name.
- Any name beginning with “JMS_” is a provider-specific property name.
- Any name that does not begin with “JMS” is an application-specific property name. If there is a reference to a property that does not exist in a message, its value is NULL. If it does exist, its value is the corresponding property value.

- White space is the same as it is defined for Java: space, horizontal tab, form feed, and line terminator.

- Expressions:
  - A selector is a conditional expression. A selector that evaluates to true does match, and a selector that evaluates to false or unknown does not match.
  - Arithmetic expressions are composed of themselves, arithmetic operations, identifiers (whose value is treated as a numeric literal), and numeric literals.
  - Conditional expressions are composed of themselves, comparison operations, and logical operations.

- Standard bracketing (), to set the order in which expressions are evaluated, is supported.

- Logical operators in precedence order: NOT, AND, OR.

- Comparison operators: =, >, >=, <, <=, <> (not equal).
  - Only values of the same type can be compared. One exception is that it is valid to compare exact numeric values and approximate numeric values. (The type conversion required is defined by the rules of Java numeric promotion.) If there is an attempt to compare different types, the selector is always false.
  - String and boolean comparison is restricted to = and <>. Two strings are equal if, and only if, they contain the same sequence of characters.

- Arithmetic operators in precedence order:
  - +, - unary.
  - *, /, multiplication, and division.
  - +, -, addition, and subtraction.
  - Arithmetic operations on a NULL value are not supported. If they are attempted, the complete selector is always false.
  - Arithmetic operations must use Java numeric promotion.

- arithmetic-expr1 [NOT] BETWEEN arithmetic-expr2 and arithmetic-expr3 comparison operator:
  - age BETWEEN 15 and 19 is equivalent to age >= 15 AND age <= 19.
  - age NOT BETWEEN 15 and 19 is equivalent to age < 15 OR age > 19.
  - If any of the exprs of a BETWEEN operation are NULL, the value of the operation is false. If any of the exprs of a NOT BETWEEN operation are NULL, the value of the operation is true.

- identifier [NOT] IN (string-literal1, string-literal2,...) comparison operator where identifier has a String or NULL value.
  - Country IN (‘UK’, ‘US’, ‘France’) is true for ‘UK’ and false for ‘Peru’. It is equivalent to the expression (Country = ‘UK’) OR (Country = ‘US’) OR (Country = ‘France’).
  - Country NOT IN (‘UK’, ‘US’, ‘France’) is false for ‘UK’ and true for ‘Peru’. It is equivalent to the expression NOT ((Country = ‘UK’) OR (Country = ‘US’) OR (Country = ‘France’)).
Message selectors

- If the identifier of an IN or NOT IN operation is NULL, the value of the operation is unknown.

- identifier [NOT] LIKE pattern-value [ESCAPE escape-character] comparison operator, where identifier has a String value, pattern-value is a string literal, where '_' stands for any single character and '%' stands for any sequence of characters (including the empty sequence). All other characters stand for themselves. The optional escape-character is a single character string literal, whose character is used to escape the special meaning of the '_' and '%' in pattern-value.
  - phone LIKE '12%3' is true for '123' '12993' and false for '1234'.
  - word LIKE 'l_se' is true for 'lose' and false for 'loose'.
  - underscored LIKE '\_%' ESCAPE '\' is true for '_foo' and false for 'bar'.
  - phone NOT LIKE '12%3' is false for '123' '12993' and true for '1234'.

- If the identifier of a LIKE or NOT LIKE operation is NULL, the value of the operation is unknown.

- identifier IS NULL comparison operator tests for a null header field value, or a missing property value.
  - prop_name IS NULL.

- identifier IS NOT NULL comparison operator tests for the existence of a non-null header field value or a property value.
  - prop_name IS NOT NULL.

The following message selector selects messages with a message type of car, color of blue, and weight greater than 2500 lbs:

"JMSType = 'car' AND color = 'blue' AND weight > 2500"

As noted above, property values may be NULL. The evaluation of selector expressions that contain NULL values is defined by SQL 92 NULL semantics. The following is a brief description of these semantics:

- SQL treats a NULL value as unknown.
- Comparison or arithmetic with an unknown value always yields an unknown value.
- The IS NULL and IS NOT NULL operators convert an unknown value into the respective TRUE and FALSE values.

Although SQL supports fixed decimal comparison and arithmetic, JMS message selectors do not. This is why exact numeric literals are restricted to those without a decimal. It is also why there are numerics with a decimal as an alternate representation for an approximate numeric value.

SQL comments are not supported.
Mapping JMS messages onto WebSphere MQ messages

This section describes how the JMS message structure that is described in the first part of this chapter is mapped onto a WebSphere MQ message. It is of interest to programmers who wish to transmit messages between JMS and traditional WebSphere MQ applications. It is also of interest to people who wish to manipulate messages transmitted between two JMS applications - for example, in a message broker implementation.

This section does not apply when you use a direct connection to WebSphere MQ Event Broker.

WebSphere MQ messages are composed of three components:
- The WebSphere MQ Message Descriptor (MQMD)
- An WebSphere MQ MQRFH2 header
- The message body.

The MQRFH2 is optional, and its inclusion in an outgoing message is governed by a flag in the JMS Destination class. You can set this flag using the WebSphere MQ JMS administration tool. Because the MQRFH2 carries JMS-specific information, always include it in the message when the sender knows that the receiving destination is a JMS application. Normally, omit the MQRFH2 when sending a message directly to a non-JMS application (WebSphere MQ Native application). This is because such an application does not expect an MQRFH2 in its WebSphere MQ message. Figure 4 shows how the structure of a JMS message is transformed to a WebSphere MQ message and back again:

![Diagram showing mapping and copying of JMS to WebSphere MQ messages](image)

Figure 4. How messages are transformed between JMS and WebSphere MQ using the MQRFH2 header

The structures are transformed in two ways:

**Mapping**

Where the MQMD includes a field that is equivalent to the JMS field, the JMS field is mapped onto the MQMD field. Additional MQMD fields are exposed as JMS properties, because a JMS application may need to get or set these fields when communicating with a non-JMS application.

**Copying**

Where there is no MQMD equivalent, a JMS header field or property is passed, possibly transformed, as a field inside the MQRFH2.
The MQRFH2 header

This section describes the MQRFH Version 2 header, which carries JMS-specific data that is associated with the message content. The MQRFH2 Version 2 is an extensible header, and can also carry additional information that is not directly associated with JMS. However, but this section covers only its use by JMS.

There are two parts of the header, a fixed portion, and a variable portion.

Fixed portion

The fixed portion is modelled on the “standard” WebSphere MQ header pattern and consists of the following fields:

**StrucId (MQCHAR4)**
Structure identifier.
Must be MQRFH_STRUC_ID (value: “RFH ”) (initial value).
MQRFH_STRUC_ID_ARRAY (value: “R”, “F”, “H”, “”) is also defined in the usual way.

**Version (MQLONG)**
Structure version number.
Must be MQRFH_VERSION_2 (value: 2) (initial value).

**StrucLength (MQLONG)**
Total length of MQRFH2, including the NameValueData fields.
The value set into StrucLength must be a multiple of 4 (the data in the NameValueData fields may be padded with space characters to achieve this).

**Encoding (MQLONG)**
Data encoding.
Encoding of any numeric data in the portion of the message following the MQRFH2 (the next header, or the message data following this header).

**CodedCharSetId (MQLONG)**
Coded character set identifier.
Representation of any character data in the portion of the message following the MQRFH2 (the next header, or the message data following this header).

**Format (MQCHAR8)**
Format name.
Format name for the portion of the message following the MQRFH2.

**Flags (MQLONG)**
Flags.
MQRFH_NO_FLAGS = 0. No flags set.

**NameValueCCSId (MQLONG)**
The coded character set identifier (CCSID) for the NameValueData character strings contained in this header. The NameValueData may be coded in a character set that differs from the other character strings that are contained in the header (StrucID and Format).
Mapping JMS messages

If the NameValueCCSID is a 2-byte Unicode CCSID (1200, 13488, or 17584), the byte order of the Unicode is the same as the byte ordering of the numeric fields in the MQRFH2. (For example, Version, StrucLength, NameValueCCSID itself.)

The NameValueCCSID may take only values from the following table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200</td>
<td>UCS2 open-ended</td>
</tr>
<tr>
<td>1208</td>
<td>UTF8</td>
</tr>
<tr>
<td>13488</td>
<td>UCS2 2.0 subset</td>
</tr>
<tr>
<td>17584</td>
<td>UCS2 2.1 subset (includes Euro symbol)</td>
</tr>
</tbody>
</table>

Variable Portion

The variable portion follows the fixed portion. The variable portion contains a variable number of MQRFH2 Folders. Each folder contains a variable number of elements or properties. Folders group together related properties. The MQRFH2 headers created by JMS can contain up to three folders:

The <mcd> folder

This contains properties that describe the “shape” or “format” of the message. For example the Msd property identifies the message as being Text, Bytes, Stream, Map, Object, or “Null”. This folder is always present in a JMS MQRFH2.

The <jms> folder

This is used to transport JMS header fields, and JMSX properties that cannot be fully expressed in the MQMD. This folder is always present in a JMS MQRFH2.

The <usr> folder

This is used to transport any application-defined properties associated with the message. This folder is only present if the application has set some application-defined properties.

Table 18 shows a full list of property names.

<table>
<thead>
<tr>
<th>JMS Field Name</th>
<th>Java type</th>
<th>MQRFH2 Folder name</th>
<th>Property name</th>
<th>Type/values</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSDestination</td>
<td>Destination</td>
<td>jms</td>
<td>Dst</td>
<td>string</td>
</tr>
<tr>
<td>JMSExpiration</td>
<td>long</td>
<td>jms</td>
<td>Exp</td>
<td>i8</td>
</tr>
<tr>
<td>JMSPriority</td>
<td>int</td>
<td>jms</td>
<td>Pri</td>
<td>i4</td>
</tr>
<tr>
<td>JMSDeliveryMode</td>
<td>int</td>
<td>jms</td>
<td>Dlv</td>
<td>i4</td>
</tr>
<tr>
<td>JMSCorrelationID</td>
<td>String</td>
<td>jms</td>
<td>Cid</td>
<td>string</td>
</tr>
<tr>
<td>JMSReplyTo</td>
<td>Destination</td>
<td>jms</td>
<td>Rto</td>
<td>string</td>
</tr>
<tr>
<td>JMSTimestamp</td>
<td>long</td>
<td>jms</td>
<td>Tms</td>
<td>i8</td>
</tr>
<tr>
<td>JMSType</td>
<td>String</td>
<td>mcd</td>
<td>Type, Set, Fmt</td>
<td>string</td>
</tr>
<tr>
<td>JMSXGroupID</td>
<td>String</td>
<td>jms</td>
<td>Gid</td>
<td>string</td>
</tr>
<tr>
<td>JMSXGroupSeq</td>
<td>int</td>
<td>jms</td>
<td>Seq</td>
<td>i4</td>
</tr>
</tbody>
</table>
Mapping JMS messages

Table 18. MQRFH2 folders and properties used by JMS (continued)

<table>
<thead>
<tr>
<th>JMS Field Name</th>
<th>Java type</th>
<th>MQRFH2 Folder name</th>
<th>Property name</th>
<th>Type/values</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx (User Defined)</td>
<td>Any</td>
<td>usr</td>
<td>xxx</td>
<td>any</td>
</tr>
<tr>
<td>mcd</td>
<td></td>
<td>Msd</td>
<td></td>
<td>jms_none, jms_text, jms_bytes, jms_map, jms_stream, jms_object</td>
</tr>
</tbody>
</table>

The syntax used to express the properties in the variable portion is as follows:

**NameValueLength (MQLONG)**
Length in bytes of the NameValueData string that immediately follows this length field (it does not include its own length). The value set into NameValueLength is always a multiple of 4 (the NameValueData field is padded with space characters to achieve this).

**NameValueData (MQCHARn)**
A single character string, whose length in bytes is given by the preceding NameValueLength field. It contains a “folder” holding a sequence of “properties”. Each property is a “name/type/value” triplet, contained within an XML element whose name is the folder name, as follows:

```xml
<foldername> triplet1 triplet2 ..... tripletn </foldername>
```

The closing `</foldername>` tag can be followed by spaces as padding characters. Each triplet is encoded using an XML-like syntax:

```xml
<name dt='datatype'>value</name>
```

The `dt='datatype'` element is optional and is omitted for many properties, because their datatype is predefined. If it is included, one or more space characters must be included before the `dt=` tag.

- `name` is the name of the property - see [Table 18 on page 231](#)
- `datatype` must match, after folding, one of the literal Datatype values in [Table 19](#)
- `value` is a string representation of the value to be conveyed, using the Definitions in [Table 19](#)

A null value is encoded using the following syntax:

```xml
<name dt="DataType" xsi:nil="true"></name>
```

Note that `xsi:nil="false"` should not be used.

Table 19. Property datatype values and definitions

<table>
<thead>
<tr>
<th>Datatype value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Any sequence of characters excluding &lt; and &amp;</td>
</tr>
<tr>
<td>boolean</td>
<td>The character 0 or 1 (1 = &quot;true&quot;)</td>
</tr>
</tbody>
</table>
Table 19. Property datatype values and definitions (continued)

<table>
<thead>
<tr>
<th>Datatype value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin.hex</td>
<td>Hexadecimal digits representing octets</td>
</tr>
<tr>
<td>i1</td>
<td>A number, expressed using digits 0..9, with optional sign (no fractions or exponent). Must lie in the range -128 to 127 inclusive</td>
</tr>
<tr>
<td>i2</td>
<td>A number, expressed using digits 0..9, with optional sign (no fractions or exponent). Must lie in the range -32768 to 32767 inclusive</td>
</tr>
<tr>
<td>i4</td>
<td>A number, expressed using digits 0..9, with optional sign (no fractions or exponent). Must lie in the range -2147483648 to 2147483647 inclusive</td>
</tr>
<tr>
<td>i8</td>
<td>A number, expressed using digits 0..9, with optional sign (no fractions or exponent). Must lie in the range -9223372036854775808 to 92233720368547750807 inclusive</td>
</tr>
<tr>
<td>int</td>
<td>A number, expressed using digits 0..9, with optional sign (no fractions or exponent). Must lie in the same range as 'i8'. This can be used in place of one of the 'i*' types if the sender does not wish to associate a particular precision with the property</td>
</tr>
<tr>
<td>r4</td>
<td>Floating point number, magnitude &lt;= 3.40282347E+38, &gt;= 1.175E-37 expressed using digits 0..9, optional sign, optional fractional digits, optional exponent</td>
</tr>
<tr>
<td>r8</td>
<td>Floating point number, magnitude &lt;= 1.7976931348623E+308, &gt;= 2.225E-307 expressed using digits 0..9, optional sign, optional fractional digits, optional exponent</td>
</tr>
</tbody>
</table>

A string value may contain spaces. You must use the following escape sequences in a string value:

- `&amp;` for the & character
- `&lt;` for the < character

You can use the following escape sequences, but they are not required:

- `&gt;` for the > character
- `&apos;` for the ' character
- `&quot;` for the " character

JMS fields and properties with corresponding MQMD fields

Table 20 lists the JMS header fields and Table 21 on page 234 lists the JMS properties that are mapped directly to MQMD fields. Table 22 on page 234 lists the provider specific properties and the MQMD fields that they are mapped to.

Table 20. JMS header fields mapping to MQMD fields

<table>
<thead>
<tr>
<th>JMS header field</th>
<th>Java type</th>
<th>MQMD field</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSDeliveryMode</td>
<td>int</td>
<td>Persistence</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMSExpiration</td>
<td>long</td>
<td>Expiry</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMSPriority</td>
<td>int</td>
<td>Priority</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMSMessageID</td>
<td>String</td>
<td>MessageID</td>
<td>MQBYTE24</td>
</tr>
<tr>
<td>JMSTimestamp</td>
<td>long</td>
<td>PutDate</td>
<td>MQCHAR8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PutTime</td>
<td>MQCHAR8</td>
</tr>
<tr>
<td>JMSCorrelationID</td>
<td>String</td>
<td>CorrelId</td>
<td>MQBYTE24</td>
</tr>
</tbody>
</table>
Mapping JMS messages

Table 21. JMS properties mapping to MQMD fields

<table>
<thead>
<tr>
<th>JMS property</th>
<th>Java type</th>
<th>MQMD field</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSXUserID</td>
<td>String</td>
<td>UserIdentifier</td>
<td>MQCHAR12</td>
</tr>
<tr>
<td>JMSXAppID</td>
<td>String</td>
<td>PutApplName</td>
<td>MQCHAR28</td>
</tr>
<tr>
<td>JMSXDeliveryCount</td>
<td>int</td>
<td>BackoutCount</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMSXGroupID</td>
<td>String</td>
<td>GroupId</td>
<td>MQBYTE24</td>
</tr>
<tr>
<td>JMSXGroupSeq</td>
<td>int</td>
<td>MsgSeqNumber</td>
<td>MQLONG</td>
</tr>
</tbody>
</table>

Table 22. JMS provider specific properties mapping to MQMD fields

<table>
<thead>
<tr>
<th>JMS provider specific property</th>
<th>Java type</th>
<th>MQMD field</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMS_IBM_Report_Exception</td>
<td>int</td>
<td>Report</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_Report_Expiration</td>
<td>int</td>
<td>Report</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_Report_COA</td>
<td>int</td>
<td>Report</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_Report_COD</td>
<td>int</td>
<td>Report</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_Report_PAN</td>
<td>int</td>
<td>Report</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_Report_NAN</td>
<td>int</td>
<td>Report</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_Report_Pass_Msg_ID</td>
<td>int</td>
<td>Report</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_Report_Pass_Correl_ID</td>
<td>int</td>
<td>Report</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_Report_Discard_Msg</td>
<td>int</td>
<td>Report</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_MsgType</td>
<td>int</td>
<td>MsgType</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_Feedback</td>
<td>int</td>
<td>Feedback</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_Format</td>
<td>String</td>
<td>Format</td>
<td>MQCHAR8</td>
</tr>
<tr>
<td>JMS_IBM_PutApplType</td>
<td>int</td>
<td>PutApplType</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_Encoding</td>
<td>int</td>
<td>Encoding</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_Character_Set</td>
<td>String</td>
<td>CodedCharacterSetId</td>
<td>MQLONG</td>
</tr>
<tr>
<td>JMS_IBM_PutDate</td>
<td>String</td>
<td>PutDate</td>
<td>MQCHAR8</td>
</tr>
<tr>
<td>JMS_IBM_PutTime</td>
<td>String</td>
<td>PutTime</td>
<td>MQCHAR8</td>
</tr>
<tr>
<td>JMS_IBM_Last_Msg_In_Group</td>
<td>boolean</td>
<td>MsgFlags</td>
<td>MQLONG</td>
</tr>
</tbody>
</table>

Mapping JMS fields onto WebSphere MQ fields (outgoing messages)

Table 23 on page 235 shows how the JMS header fields are mapped into MQMD/RFH2 fields at send() or publish() time. Table 24 on page 235 shows how JMS properties and Table 25 on page 235 shows how JMS provider specific properties are mapped to MQMD fields at send() or publish() time.

For fields marked “Set by Message Object”, the value transmitted is the value held in the JMS message immediately before the send() or publish() operation. The value in the JMS Message is left unchanged by the operation.

For fields marked “Set by Send Method”, a value is assigned when the send() or publish() is performed (any value held in the JMS Message is ignored). The value in the JMS message is updated to show the value used.
Fields marked as “Receive-only” are not transmitted and are left unchanged in the message by `send()` or `publish()`.

### Table 23. Outgoing message field mapping

<table>
<thead>
<tr>
<th>JMS header field name</th>
<th>MQMD field used for transmission</th>
<th>Header</th>
<th>Set by</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSDestination</td>
<td></td>
<td>MQRFH2</td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSDeliveryMode</td>
<td>Persistence</td>
<td>MQRFH2</td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSExpiration</td>
<td>Expiry</td>
<td>MQRFH2</td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSPriority</td>
<td>Priority</td>
<td>MQRFH2</td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSMessageID</td>
<td>MessageID</td>
<td></td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSTimestamp</td>
<td>PutDate/PutTime</td>
<td></td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSReplyTo</td>
<td>ReplyToQ/ReplyToQMgr</td>
<td>MQRFH2</td>
<td>Message Object</td>
</tr>
<tr>
<td>JMSType</td>
<td></td>
<td>MQRFH2</td>
<td>Message Object</td>
</tr>
<tr>
<td>JMSRedelivered</td>
<td></td>
<td>MQRFH2</td>
<td>Receive-only</td>
</tr>
</tbody>
</table>

### Table 24. Outgoing message JMS property mapping

<table>
<thead>
<tr>
<th>JMS property name</th>
<th>MQMD field used for transmission</th>
<th>Header</th>
<th>Set by</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSXUserID</td>
<td>UserIdentifier</td>
<td></td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSXAppID</td>
<td>PutApplName</td>
<td></td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSXDeliveryCount</td>
<td></td>
<td></td>
<td>Receive-only</td>
</tr>
<tr>
<td>JMSXGroupID</td>
<td>GroupId</td>
<td>MQRFH2</td>
<td>Message Object</td>
</tr>
<tr>
<td>JMSXGroupSeq</td>
<td>MsgSeqNumber</td>
<td>MQRFH2</td>
<td>Message Object</td>
</tr>
</tbody>
</table>

### Table 25. Outgoing message JMS provider specific property mapping

<table>
<thead>
<tr>
<th>JMS provider specific property name</th>
<th>MQMD field used for transmission</th>
<th>Header</th>
<th>Set by</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMS_IBM_Report_Exception</td>
<td>Report</td>
<td></td>
<td>Message Object</td>
</tr>
<tr>
<td>JMS_IBM_Report_Expiration</td>
<td>Report</td>
<td></td>
<td>Message Object</td>
</tr>
<tr>
<td>JMS_IBM_Report_COA/COD</td>
<td>Report</td>
<td></td>
<td>Message Object</td>
</tr>
<tr>
<td>JMS_IBM_Report_NAN/PAN</td>
<td>Report</td>
<td></td>
<td>Message Object</td>
</tr>
<tr>
<td>JMS_IBM_Report_Pass_Correl_ID</td>
<td>Report</td>
<td></td>
<td>Message Object</td>
</tr>
<tr>
<td>JMS_IBM_Report_Discard_Msg</td>
<td>Report</td>
<td></td>
<td>Message Object</td>
</tr>
<tr>
<td>JMS_IBM_MsgType</td>
<td>MsgType</td>
<td></td>
<td>Message Object</td>
</tr>
<tr>
<td>JMS_IBM_Feedback</td>
<td>Feedback</td>
<td></td>
<td>Message Object</td>
</tr>
<tr>
<td>JMS_IBM_Format</td>
<td>Format</td>
<td></td>
<td>Message Object</td>
</tr>
<tr>
<td>JMS_IBM_PutApplType</td>
<td>PutApplType</td>
<td></td>
<td>Send Method</td>
</tr>
<tr>
<td>JMS_IBM_Encoding</td>
<td>Encoding</td>
<td></td>
<td>Message Object</td>
</tr>
<tr>
<td>JMS_IBM_Character_Set</td>
<td>CodedCharacterSetId</td>
<td></td>
<td>Message Object</td>
</tr>
<tr>
<td>JMS_IBM_PutDate</td>
<td>PutDate</td>
<td></td>
<td>Send Method</td>
</tr>
<tr>
<td>JMS_IBM_PutTime</td>
<td>PutTime</td>
<td></td>
<td>Send Method</td>
</tr>
</tbody>
</table>
### Mapping JMS messages

Table 25. Outgoing message JMS provider specific property mapping (continued)

<table>
<thead>
<tr>
<th>JMS provider specific property name</th>
<th>MQMD field used for transmission</th>
<th>Header</th>
<th>Set by</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMS_IBM_Last_Msg_In_Group</td>
<td>MsgFlags</td>
<td></td>
<td>Message Object</td>
</tr>
</tbody>
</table>

#### Mapping JMS header fields at send() or publish()

The following notes relate to the mapping of JMS fields at send() or publish():

- **JMS Destination to MQRFH2:** This is stored as a string that serializes the salient characteristics of the destination object, so that a receiving JMS can reconstitute an equivalent destination object. The MQRFH2 field is encoded as URI (see "uniform resource identifiers" on page 194 for details of the URI notation).

- **JMSReplyTo to MQMD ReplyToQ, ReplyToQMgr, MQRFH2:** The Queue and QueueManager name are copied to the MQMD ReplyToQ and ReplyToQMgr fields respectively. The destination extension information (other “useful” details that are kept in the Destination Object) is copied into the MQRFH2 field. The MQRFH2 field is encoded as URI (see "uniform resource identifiers" on page 194 for details of the URI notation).

- **JMSDeliveryMode to MQMD Persistence:** The JMSDeliveryMode value is set by the send() or publish() Method or MessageProducer, unless the Destination Object overrides it. The JMSDeliveryMode value is mapped to the MQMD Persistence field as follows:
  - JMS value PERSISTENT is equivalent to MQPER_PERSISTENT
  - JMS value NON_PERSISTENT is equivalent to MQPER_NOT_PERSISTENT

  If the MQQueue persistence property is not set to JMSC.MQJMS_PER_QDEF the delivery mode value is also encoded in the MQRFH2.

- **JMSExpiration to/from MQMD Expiry, MQRFH2:** JMSExpiration stores the time to expire (the sum of the current time and the time to live), whereas MQMD stores the time to live. Also, JMSExpiration is in milliseconds, but MQMD.expiry is in centiseconds.
  - If the send() method sets an unlimited time to live, MQMD Expiry is set to MQEI_UNLIMITED, and no JMSExpiration is encoded in the MQRFH2.
  - If the send() method sets a time to live that is less than 214748364.7 seconds (about 7 years), the time to live is stored in MQMD. Expiry and the expiration time (in milliseconds) is encoded as an i8 value in the MQRFH2.
  - If the send() method sets a time to live greater than 214748364.7 seconds, MQMD.Expiry is set to MQEI_UNLIMITED. The true expiration time in milliseconds is encoded as an i8 value in the MQRFH2.

- **JMSPriority to MQMD Priority:** Directly map JMSPriority value (0-9) onto MQMD priority value (0-9). If JMSPriority is set to a non-default value, the priority level is also encoded in the MQRFH2.

- **JMSMessageID from MQMD MessageID:** All messages sent from JMS have unique message identifiers assigned by WebSphere MQ. The value assigned is returned in the MQMD messageId field after the MQPUT call, and is passed back to the application in the JMSMessageID field. The WebSphere MQ messageID is a 24-byte binary value, whereas the JMSMessageID is a String. The JMSMessageID is composed of the binary messageID value converted to a sequence of 48 hexadecimal characters, prefixed with the characters "ID:". JMS provides a hint that can be set to disable the production of message identifiers. This hint is ignored, and a unique identifier is assigned in all cases. Any value that is set into the JMSMessageId field before a send() is overwritten.
Mapping JMS messages

• JMSTimestamp to MQRFH2: During a send, the JMSTimestamp field is set according to the JVM’s clock. This value is set into the MQRFH2. Any value that is set into the JMSTimestamp field before a send() is overwritten. See also the JMS_IBM_PutDate and JMS_IBM_PutTime properties.

• JMSType to MQRFH2: This string is set into the MQRFH2 mcd.Type field. If it is in URI format, it may also affect mcd.Set and mcd.Fmt fields. See also Appendix D, “Connecting with other products” on page 409.

• JMSCorrelationID to MQMD CorrelId, MQRFH2: The JMSCorrelationID can hold one of the following:
  – A provider specific message ID: This is a message identifier from a message previously sent or received, and so should be a string of 48 hexadecimal digits that are prefixed with “ID:”. The prefix is removed, the remaining characters are converted into binary, and then they are set into the MQMD CorrelId field. No CorrelId value is encoded in the MQRFH2.
  – A provider-native byte[] value: The value is copied into the MQMD CorrelId field - padded with nulls, or truncated to 24 bytes if necessary. No CorrelId value is encoded in the MQRFH2.
  – An application-specific String: The value is copied into the MQRFH2. The first 24 bytes of the string, in UTF8 format, are written into the MQMD CorrelId.

Mapping JMS property fields
These notes refer to the mapping of JMS property fields in WebSphere MQ messages:

• JMSXUserID from MQMD UserIdentifier: JMSXUserID is set on return from send call.

• JMSXAppID from MQMD PutApplName: JMSXAppID is set on return from send call.

• JMSXGroupID to MQRFH2 (point-to-point): For point-to-point messages, the JMSXGroupID is copied into the MQMD GroupID field. If the JMSXGroupID starts with the prefix “ID:”, it is converted into binary. Otherwise, it is encoded as a UTF8 string. The value is padded or truncated if necessary to a length of 24 bytes. The MQMF_MSG_IN_GROUP flag is set.

• JMSXGroupID to MQRFH2 (publish/subscribe): For publish/subscribe messages, the JMSXGroupID is copied into the MQRFH2 as a string.

• JMSXGroupSeq MQMD MsgSeqNumber (point-to-point): For point-to-point messages, the JMSXGroupSeq is copied into the MQMD MsgSeqNumber field. The MQMF_MSG_IN_GROUP flag is set.

• JMSXGroupSeq MQMD MsgSeqNumber (publish/subscribe): For publish/subscribe messages, the JMSXGroupSeq is copied into the MQRFH2 as an i4.

Mapping JMS provider-specific fields
The following notes refer to the mapping of JMS Provider specific fields into WebSphere MQ messages:

• JMS_IBM_Report_<name> to MQMD Report: A JMS application can set the MQMD Report options, using the following JMS_IBM_Report_XXX properties. The single MQMD is mapped to several JMS_IBM_Report_XXX properties. The application should set the value of these properties to the standard WebSphere MQ MQRO_ constants (included in com.ibm.mq.MQC). So, for example, to request COD with full Data, the application should set JMS_IBM_Report_COD to the value MQC.MQRO_COD_WITH_FULL_DATA.
Mapping JMS messages

MQRO_EXCEPTION or
MQRO_EXCEPTION_WITH_DATA or
MQRO_EXCEPTION_WITH_FULL_DATA

JMS_IBM_Report_Expiration

MQRO_EXPIRATION or
MQRO_EXPIRATION_WITH_DATA or
MQRO_EXPIRATION_WITH_FULL_DATA

JMS_IBM_Report_COA

MQRO_COA or
MQRO_COA_WITH_DATA or
MQRO_COA_WITH_FULL_DATA

JMS_IBM_Report_COD

MQRO_COD or
MQRO_COD_WITH_DATA or
MQRO_COD_WITH_FULL_DATA

JMS_IBM_Report_PAN

MQRO_PAN

JMS_IBM_Report_NAN

MQRO_NAN

JMS_IBM_Report_Pass_Msg_ID

MQRO_PASS_MSG_ID

JMS_IBM_Report_Pass_Correl_ID

MQRO_PASS_CORREL_ID

JMS_IBM_Report_Discard_Msg

MQRO_DISCARD_MSG

- **JMS_IBM_MsgType to MQMD MsgType:** Value maps directly onto MQMD MsgType. If the application has not set an explicit value of JMS_IBM_MsgType, then a default value is used. This default value is determined as follows:
  - If JMSReplyTo is set to a WebSphere MQ queue destination, MsgType is set to the value MQMT_REQUEST
  - If JMSReplyTo is not set, or is set to anything other than a WebSphere MQ queue destination, MsgType is set to the value MQMT_DATAGRAM

- **JMS_IBM_Feedback to MQMD Feedback:** Value maps directly onto MQMD Feedback.

- **JMS_IBM_Format to MQMD Format:** Value maps directly onto MQMD Format.

- **JMS_IBM_Encoding to MQMD CodedCharacterSetId:** If set, this property overrides the numeric encoding of the Destination Queue or Topic.

- **JMS_IBM_Character_Set to MQMD CodedCharacterSetId:** If set, this property overrides the coded character set property of the Destination Queue or Topic.

- **JMS_IBM_PutDate from MQMD PutDate:** The value of this property is set, during send, directly from the PutDate field in the MQMD. Any value that is set into the JMS_IBM_PutDate property before a send is overwritten. This field is a String of eight characters, in the WebSphere MQ Date format of “YYYYMMDD”. This property can be used in conjunction with the JMS_IBM_PutTime property to determine the time the message was put according to the queue manager.

- **JMS_IBM_PutTime from MQMD PutTime:** The value of this property is set, during send, directly from the PutTime field in the MQMD. Any value that is set
into the JMS.ibm.PutTime property before a send is overwritten. This field is a String of eight characters, in the WebSphere MQ Time format of “HHMMSSSTH”. This property can be used in conjunction with the JMS.ibm.PutDate property to determine the time the message was put according to the queue manager.

- **JMS.ibm.Last_Msg_In_Group to MQMD MsgFlags:** For Point-to-Point messaging, this boolean value maps to the MQMF_LAST_MSG_IN_GROUP flag in the MQMD MsgFlags field. It is normally used in conjunction with the JMSXGroupID and JMSXGroupSeq properties to indicate to a legacy WebSphere MQ application that this is the last message in a group. This property is ignored for publish/subscribe messaging.

### Mapping WebSphere MQ fields onto JMS fields (incoming messages)

Table 26 shows how JMS header fields and Table 27 shows how JMS property fields are mapped into MQMD/MQRFH2 fields at send() or publish() time. Table 28 on page 240 shows how JMS provider specific properties are mapped.

#### Table 26. Incoming message JMS header field mapping

<table>
<thead>
<tr>
<th>JMS header field name</th>
<th>MQMD field retrieved from</th>
<th>MQRFH2 field retrieved from</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSDestination</td>
<td>jms.Dst</td>
<td></td>
</tr>
<tr>
<td>JMSDeliveryMode</td>
<td>jms.Dlv</td>
<td></td>
</tr>
<tr>
<td>JMSExpiration</td>
<td>jms.Exp</td>
<td></td>
</tr>
<tr>
<td>JMSPriority</td>
<td>Priority</td>
<td></td>
</tr>
<tr>
<td>JMSMessageID</td>
<td>MessageID</td>
<td></td>
</tr>
<tr>
<td>JMSTimestamp</td>
<td>PutDate</td>
<td>jms.Tms</td>
</tr>
<tr>
<td>JMSCorrelationID</td>
<td>CorrelId</td>
<td>jms.Cid</td>
</tr>
<tr>
<td>JMSReplyTo</td>
<td>ReplyTo</td>
<td>jms.Rto</td>
</tr>
<tr>
<td>JMSType</td>
<td>MCD.Type, MCD.Set, MCD.Fmt</td>
<td></td>
</tr>
<tr>
<td>JMSRedelivered</td>
<td>BackoutCount</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
1. For properties that can have values retrieved from the MQRFH2 or the MQMD, if both are available, the setting in the MQRFH2 is used.

#### Table 27. Incoming message property mapping

<table>
<thead>
<tr>
<th>JMS property name</th>
<th>MQMD field retrieved from</th>
<th>MQRFH2 field retrieved from</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSXUserID</td>
<td>UserIdentifier</td>
<td></td>
</tr>
<tr>
<td>JMSXAppID</td>
<td>PutAppName</td>
<td></td>
</tr>
<tr>
<td>JMSXDeliveryCount</td>
<td>BackoutCount</td>
<td></td>
</tr>
<tr>
<td>JMSXGroupID</td>
<td>GroupId</td>
<td>jms.Gid</td>
</tr>
<tr>
<td>JMSXGroupSeq</td>
<td>MsgSeqNumber</td>
<td>jms.Seq</td>
</tr>
</tbody>
</table>
Mapping JMS messages

Table 27. Incoming message property mapping (continued)

<table>
<thead>
<tr>
<th>JMS property name</th>
<th>MQMD field retrieved from</th>
<th>MQRFH2 field retrieved from</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMS_IBM_Report_Exception</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_Expiration</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_COA</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_COD</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_PAN</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_NAN</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_Pass_Msg_ID</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_Pass_Correl_ID</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_Discard_Msg</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_MsgType</td>
<td>(MsgType)</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Feedback</td>
<td>Feedback</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Format</td>
<td>Format</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_PutAppType</td>
<td>PutAppType</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Encoding[1]</td>
<td>Encoding</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Character_Set[1]</td>
<td>CodedCharacterSetId</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_PutDate</td>
<td>PutDate</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_PutTime</td>
<td>PutTime</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Last_Msg_In_Group</td>
<td>MsgFlags</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. For properties that can have values retrieved from the MQRFH2 or the MQMD, if both are available, the setting in the MQRFH2 is used.

Table 28. Incoming message provider specific JMS property mapping

<table>
<thead>
<tr>
<th>JMS property name</th>
<th>MQMD field retrieved from</th>
<th>MQRFH2 field retrieved from</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMS_IBM_Report_Exception</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_Expiration</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_COA</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_COD</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_PAN</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_NAN</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_Pass_Msg_ID</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_Pass_Correl_ID</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Report_Discard_Msg</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_MsgType</td>
<td>(MsgType)</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Feedback</td>
<td>Feedback</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Format</td>
<td>Format</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_PutAppType</td>
<td>PutAppType</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Encoding[1]</td>
<td>Encoding</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Character_Set[1]</td>
<td>CodedCharacterSetId</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_PutDate</td>
<td>PutDate</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_PutTime</td>
<td>PutTime</td>
<td></td>
</tr>
<tr>
<td>JMS_IBM_Last_Msg_In_Group</td>
<td>MsgFlags</td>
<td></td>
</tr>
</tbody>
</table>

1. Only set if the incoming message is a Bytes Message.

Mapping JMS to a native WebSphere MQ application

This section describes what happens if you send a message from a JMS Client application to a traditional WebSphere MQ application which has no knowledge of MQRFH2 headers. Figure 5 on page 241 is a diagram of the mapping.

The administrator indicates that the JMS Client is communicating with such an application by setting the WebSphere MQ Destination’s TargetClient value to JMSC.MQJMS_CLIENT_NONJMS_MQ. This indicates that no MQRFH2 field is to be produced. Note that if this is not done, the receiving application must be able to handle the MQRFH2 field.

The mapping from JMS to MQMD targeted at a native WebSphere MQ application is the same as mapping from JMS to MQMD targeted at a true JMS client. If JMS receives a WebSphere MQ message with the MQMD Format field set to other than MQFMT_RFH2, we know that we are receiving data from a non-JMS application. If the Format is MQFMT_STRING, the message is received as a JMS Text Message.
Mapping JMS messages

Otherwise, it is received as a JMS Bytes message. Because there is no MQRFH2, only those JMS properties that are transmitted in the MQMD can be restored.

Figure 5. How JMS messages are transformed to WebSphere MQ messages (no MQRFH2 header)

Message body

This section discusses the encoding of the message body itself. The encoding depends on the type of JMS message:

ObjectMessage

is an object serialized by the Java Runtime in the normal way.

TextMessage

is an encoded string. For an outgoing message, the string is encoded in the character set given by the Destination object. This defaults to UTF8 encoding (the UTF8 encoding starts with the first character of the message - there is no length field at the start). It is, however, possible to specify any other character set supported by WebSphere MQ Java. Such character sets are used mainly when you send a message to a non-JMS application.

If the character set is a double-byte set (including UTF16), the Destination object’s integer encoding specification determines the order of the bytes.

An incoming message is interpreted using the character set and encoding that are specified in the message itself. These specifications are in the rightmost WebSphere MQ header (or MQMD if there are no headers). For JMS messages, the rightmost header will usually be the MQRFH2.

BytesMessage

is, by default, a sequence of bytes as defined by the JMS 1.0.2 specification, and associated Java documentation.

For an outgoing message that was assembled by the application itself, the Destination object’s encoding property may be used to override the encodings of integer and floating point fields contained in the message. For example, you can request that floating point values are stored in S/390 rather than IEEE format).

An incoming message is interpreted using the numeric encoding specified in the message itself. This specification is in the rightmost WebSphere MQ header (or MQMD if there are no headers). For JMS messages, the rightmost header will usually be the MQRFH2.

If a BytesMessage is received, and is resent without modification, its body is transmitted byte for byte, as it was received. The Destination object’s encoding property has no effect on the body. The only string-like entity that can be sent explicitly in a BytesMessage is a UTF8 string. This is
Mapping JMS messages

encoded in Java UTF8 format, and starts with a 2-byte length field. The Destination object’s character set property has no effect on the encoding of an outgoing BytesMessage. The character set value in an incoming WebSphere MQ message has no effect on the interpretation of that message as a JMS BytesMessage.

Non-Java applications are unlikely to recognize the Java UTF8 encoding. Therefore, for a JMS application to send a BytesMessage that contains text data, the application itself must convert its strings to byte arrays, and write these byte arrays into the BytesMessage.

MapMessage

is a string containing a set of XML name/type/value triplets, encoded as:

<map>
<elementName1 dt='datatype'>value</elementName1>
<elementName2 dt='datatype'>value</elementName2>.....
</map>

where:

datatype can take one of the values described in Table 19 on page 232
string is the default datatype, so dt='string' is omitted.

The character set used to encode or interpret the XML string that makes up the MapMessage body is determined following the rules that apply to a TextMessage.

StreamMessage

is like a map, but without element names:

<stream>
<elt dt='datatype'>value</elt>
<elt dt='datatype'>value</elt>.....</stream>

Every element is sent using the same tagname (elt). The default type is string, so dt='string' is omitted for string elements.

The character set used to encode or interpret the XML string that makes up the StreamMessage body is determined following the rules that apply to a TextMessage.

The MQRFH2.format field is set as follows:

MQFMT_NONE
for ObjectMessage, BytesMessage, or messages with no body.

MQFMT_STRING
for TextMessage, StreamMessage, or MapMessage.
Chapter 13. WebSphere MQ JMS Application Server Facilities

WebSphere MQ JMS V5.2 supports the Application Server Facilities (ASF) that are specified in the Java Message Service 1.0.2 specification (see Sun’s Java Web site at http://java.sun.com). This specification identifies three roles within this programming model:

- **The JMS provider** supplies ConnectionConsumer and advanced Session functionality.
- **The application server** supplies ServerSessionPool and ServerSession functionality.
- **The client application** uses the functionality that the JMS provider and application server supply.

This chapter does not apply if you use a direct connection to WebSphere MQ Event Broker.

The following sections contain details about how WebSphere MQ JMS implements ASF:

- **“ASF classes and functions”** describes how WebSphere MQ JMS implements the ConnectionConsumer class and advanced functionality in the Session class.
- **“Application server sample code”** on page 249 describes the sample ServerSessionPool and ServerSession code that is supplied with WebSphere MQ JMS.
- **“Examples of ASF use”** on page 253 describes supplied ASF samples and examples of ASF use from the perspective of a client application.

**Note:** The Java Message Service 1.0.2 specification for ASF also describes JMS support for distributed transactions using the X/Open XA protocol. For details of the XA support that WebSphere MQ JMS provides, see Appendix E, “JMS JTA/XA interface with WebSphere Application Server V4” on page 413.

ASF classes and functions

WebSphere MQ JMS implements the ConnectionConsumer class and advanced functionality in the Session class. For details, see:

- “MQPoolServices” on page 139
- “MQPoolServicesEvent” on page 140
- “MQPoolToken” on page 142
- “MQPoolServicesEventListener” on page 172
- “ConnectionConsumer” on page 282
- “QueueConnection” on page 332
- “Session” on page 345
- “TopicConnection” on page 364

**ConnectionConsumer**

The JMS specification enables an application server to integrate closely with a JMS implementation by using the ConnectionConsumer interface. This feature provides concurrent processing of messages. Typically, an application server creates a pool
of threads, and the JMS implementation makes messages available to these threads. A JMS-aware application server can use this feature to provide high-level messaging functionality, such as message processing beans.

Normal applications do not use the ConnectionConsumer, but expert JMS clients might use it. For such clients, the ConnectionConsumer provides a high-performance method to deliver messages concurrently to a pool of threads. When a message arrives on a queue or a topic, JMS selects a thread from the pool and delivers a batch of messages to it. To do this, JMS runs an associated MessageListener's `onMessage()` method.

You can achieve the same effect by constructing multiple Session and MessageConsumer objects, each with a registered MessageListener. However, the ConnectionConsumer provides better performance, less use of resources, and greater flexibility. In particular, fewer Session objects are required.

To help you develop applications that use ConnectionConsumers, WebSphere MQ JMS provides a fully-functioning example implementation of a pool. You can use this implementation without any changes, or adapt it to suit the specific needs of the application.

### Planning an application

**General principles for point-to-point messaging**

When an application creates a ConnectionConsumer from a QueueConnection object, it specifies a JMS Queue object and a selector string. The ConnectionConsumer then begins to receive messages (or, more accurately, to provide messages to Sessions in the associated ServerSessionPool). Messages arrive on the queue, and if they match the selector, they are delivered to Sessions in the associated ServerSessionPool.

In WebSphere MQ terms, the Queue object refers to either a QLOCAL or a QALIAS on the local queue manager. If it is a QALIAS, that QALIAS must refer to a QLOCAL. The fully resolved WebSphere MQ QLOCAL is known as the underlying QLOCAL. A ConnectionConsumer is said to be active if it is not closed and its parent QueueConnection is started.

It is possible for multiple ConnectionConsumers, each with different selectors, to run against the same underlying QLOCAL. To maintain performance, unwanted messages should not accumulate on the queue. Unwanted messages are those for which no active ConnectionConsumer has a matching selector. You can set the QueueConnectionFactory so that these unwanted messages are removed from the queue (for details, see "Removing messages from the queue" on page 247). You can set this behavior in one of two ways:

- Use the JMS Administration tool to set the QueueConnectionFactory to MRET(NO).
- In your program, use: 
  ```java
  MQQueueConnectionFactory.setMessageRetention(JMSC.MQJMS_MRET_NO)
  ```

If you do not change this setting, the default is to retain such unwanted messages on the queue.

It is possible that ConnectionConsumers that target the same underlying QLOCAL could be created from multiple QueueConnection objects. However, for performance reasons, we recommend that multiple JVMs do not create ConnectionConsumers against the same underlying QLOCAL.
When you set up the WebSphere MQ queue manager, consider the following points:

- The underlying QLOCAL must be enabled for shared input. To do this, use the following MQSC command:
  ```sql
  ALTER QLOCAL(your.qlocal.name) SHARE GET(ENABLED)
  ```

- Your queue manager must have an enabled dead-letter queue. If a ConnectionConsumer experiences a problem when it puts a message on the dead-letter queue, message delivery from the underlying QLOCAL stops. To define a dead-letter queue, use:
  ```sql
  ALTER QMGR DEADQ(your.dead.letter.queue.name)
  ```

- The user that runs the ConnectionConsumer must have authority to perform MQOPEN with MQOO_SAVE_ALL_CONTEXT and MQOO_PASS_ALL_CONTEXT. For details, see the WebSphere MQ documentation for your specific platform.

- If unwanted messages are left on the queue, they degrade the system performance. Therefore, plan your message selectors so that between them, the ConnectionConsumers will remove all messages from the queue.

For details about MQSC commands, see [WebSphere MQ Script (MQSC) Command Reference](#).

### General principles for publish/subscribe messaging

When an application creates a ConnectionConsumer from a TopicConnection object, it specifies a Topic object and a selector string. The ConnectionConsumer then begins to receive messages on that Topic that match the selector.

Alternatively, an application can create a durable ConnectionConsumer that is associated with a specific name. This ConnectionConsumer receives messages that have been published on the Topic since the durable ConnectionConsumer was last active. It receives all such messages on the Topic that match the selector.

For non-durable subscriptions, a separate queue is used for ConnectionConsumer subscriptions. The CCSUB configurable option on the TopicConnectionFactory specifies the queue to use. Normally, the CCSUB should specify a single queue for use by all ConnectionConsumers that use the same TopicConnectionFactory. However, it is possible to make each ConnectionConsumer generate a temporary queue by specifying a queue name prefix followed by a `*`.

For durable subscriptions, the CCDSUB property of the Topic specifies the queue to use. Again, this may be a queue that already exists or a queue name prefix followed by a `*`. If you specify a queue that already exists, all durable ConnectionConsumers that subscribe to the Topic use this queue. If you specify a queue name prefix followed by a `*`, a queue is generated the first time that a durable ConnectionConsumer is created with a given name. This queue is reused later when a durable ConnectionConsumer is created with the same name.

When you set up the WebSphere MQ queue manager, consider the following points:

- Your queue manager must have an enabled dead-letter queue. If a ConnectionConsumer experiences a problem when it puts a message on the dead-letter queue, message delivery from the underlying QLOCAL stops. To define a dead-letter queue, use:
  ```sql
  ALTER QMGR DEADQ(your.dead.letter.queue.name)
  ```

- The user that runs the ConnectionConsumer must have authority to perform MQOPEN with MQOO_SAVE_ALL_CONTEXT and
MQOO_PASS_ALL_CONTEXT. For details, see the WebSphere MQ documentation for your specific platform.

- You can optimize performance for an individual ConnectionConsumer by creating a separate, dedicated, queue for it. This is at the cost of extra resource usage.

**Handling poison messages**

Sometimes, a badly-formatted message arrives on a queue. Such a message might make the receiving application fail and back out the receipt of the message. In this situation, such a message might be received, then returned to the queue, repeatedly. These messages are known as poison messages. The ConnectionConsumer must be able to detect poison messages and reroute them to an alternative destination.

When an application uses ConnectionConsumers, the circumstances in which a message is backed out depend on the Session that the application server provides:

- When the Session is non-transacted, with AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE, a message is backed out only after a system error, or if the application terminates unexpectedly.
- When the Session is non-transacted with CLIENT_ACKNOWLEDGE, unacknowledged messages can be backed out by the application server calling Session.recover(). Typically, the client implementation of MessageListener or the application server calls Message.acknowledge(). Message.acknowledge() acknowledges all messages delivered on the session so far.
- When the Session is transacted, typically, the application server commits the Session. If the application server detects an error, it may choose to back out one or more messages.
- If the application server supplies an XASession, messages are committed or backed out depending on a distributed transaction. The application server takes responsibility for completing the transaction.

The WebSphere MQ queue manager keeps a record of the number of times that each message has been backed out. When this number reaches a configurable threshold, the ConnectionConsumer requeues the message on a named Backout Queue. If this requeue fails for any reason, the message is removed from the queue and either requeued to the dead-letter queue, or discarded. See “Removing messages from the queue” on page 247 for more details.

On most platforms, the threshold and requeue queue are properties of the WebSphere MQ QLOCAL. For point-to-point messaging, this should be the underlying QLOCAL. For publish/subscribe messaging, this is the CCSUB queue defined on the TopicConnectionFactory, or the CCDSUB queue defined on the Topic. To set the threshold and requeue Queue properties, issue the following MQSC command:

```
ALTER QLOCAL(your.queue.name) BOTHRESH(threshold) BOQUEUE(your.requeue.queue.name)
```

For publish/subscribe messaging, if your system creates a dynamic queue for each subscription, these settings are obtained from the WebSphere MQ JMS model queue. To alter these settings, you can use:

```
ALTER QMODEL(SYSTEM.JMS.MODEL.QUEUE) BOTHRESH(threshold) BOQUEUE(your.requeue.queue.name)
```

If the threshold is zero, poison message handling is disabled, and poison messages will remain on the input queue. Otherwise, when the backout count reaches the
threshold, the message is sent to the named requeue queue. If the backout count reaches the threshold, but the message cannot go to the requeue queue, the message is sent to the dead-letter queue or discarded. This situation occurs if the requeue queue is not defined, or if the ConnectionConsumer cannot send the message to the requeue queue. On some platforms, you cannot specify the threshold and requeue queue properties. On these platforms, messages are sent to the dead-letter queue, or discarded, when the backout count reaches 20.

See “Removing messages from the queue” for further details.

Removing messages from the queue

When an application uses ConnectionConsumers, JMS might need to remove messages from the queue in a number of situations:

Badly formatted message
A message may arrive that JMS cannot parse.

Poison message
A message may reach the backout threshold, but the ConnectionConsumer fails to requeue it on the backout queue.

No interested ConnectionConsumer
For point-to-point messaging, when the QueueConnectionFactory is set so that it does not retain unwanted messages, a message arrives that is unwanted by any of the ConnectionConsumers.

In these situations, the ConnectionConsumer attempts to remove the message from the queue. The disposition options in the report field of the message’s MQMD set the exact behavior. These options are:

MQRO_DEAD_LETTER_Q
The message is requeued to the queue manager’s dead-letter queue. This is the default.

MQRO_DISCARD_MSG
The message is discarded.

The ConnectionConsumer also generates a report message, and this also depends on the report field of the message’s MQMD. This message is sent to the message’s ReplyToQ on the ReplyToQmgr. If there is an error while the report message is being sent, the message is sent to the dead-letter queue instead. The exception report options in the report field of the message’s MQMD set details of the report message. These options are:

MQRO_EXCEPTION
A report message is generated that contains the MQMD of the original message. It does not contain any message body data.

MQRO_EXCEPTION_WITH_DATA
A report message is generated that contains the MQMD, any MQ headers, and 100 bytes of body data.

MQRO_EXCEPTION_WITH_FULL_DATA
A report message is generated that contains all data from the original message.

default
No report message is generated.

When report messages are generated, the following options are honoured:
• MQRO_NEW_MSG_ID
If a ConnectionConsumer cannot follow the disposition options, or the exception report options, in the message’s MQMD, its action depends on the persistence of the message. If the message is non-persistent, the message is discarded, and no report message is generated. If the message is persistent, delivery of all messages from the QLOCAL stops.

Therefore, it is important to define a dead-letter queue, and to check it regularly to ensure that no problems occur. Particularly, ensure that the dead-letter queue does not reach its maximum depth, and that its maximum message size is large enough for all messages.

When a message is requeued to the dead-letter queue, it is preceded by a WebSphere MQ dead-letter header (MQDLH). See *WebSphere MQ Application Programming Reference* for details about the format of the MQDLH. You can identify messages that a ConnectionConsumer has placed on the dead-letter queue, or report messages that a ConnectionConsumer has generated, by the following fields:

- `PutApplType` is `MQAT_JAVA (0x1C)`
- `PutApplName` is “MQ JMS ConnectionConsumer”

These fields are in the MQDLH of messages on the dead-letter queue, and the MQMD of report messages. The feedback field of the MQMD, and the Reason field of the MQDLH, contain a code describing the error. For details about these codes, see “Error handling”. Other fields are as described in the *WebSphere MQ Application Programming Reference*.

### Error handling

#### Recovering from error conditions

If a ConnectionConsumer experiences a serious error, message delivery to all ConnectionConsumers with an interest in the same QLOCAL stops. Typically, this occurs if the ConnectionConsumer cannot requeue a message to the dead-letter queue, or it experiences an error when reading messages from the QLOCAL.

When this occurs, the application and application server are notified in the following way:

- Any ExceptionListener that is registered with the affected Connection is notified.

You can use these to identify the cause of the problem. In some cases, the system administrator must intervene to resolve the problem.

There are two ways in which an application can recover from these error conditions:

- Call `close()` on all affected ConnectionConsumers. The application can create new ConnectionConsumers only after all affected ConnectionConsumers are closed and any system problems are resolved.
- Call `stop()` on all affected Connections. Once all Connections are stopped and any system problems are resolved, the application should be able to `start()` all Connections successfully.
Reason and feedback codes
To determine the cause of an error, you can use:

- The feedback code in any report messages
- The reason code in the MQDLH of any messages in the dead-letter queue

ConnectionConsumers generate the following reason codes.

MQRC_BACKOUT_THRESHOLD_REACHED (0x93A; 2362)

| Cause | The message reaches the Backout Threshold defined on the QLOCAL, but no Backout Queue is defined.
|       | On platforms where you cannot define the Backout Queue, the message reaches the JMS-defined backout threshold of 20.
| Action | If this is not wanted, define the Backout Queue for the relevant QLOCAL. Also look for the cause of the multiple backouts.

MQRC_MSG_NOT_MATCHED (0x93B; 2363)

| Cause | In point-to-point messaging, there is a message that does not match any of the selectors for the ConnectionConsumers monitoring the queue. To maintain performance, the message is requeued to the dead-letter queue.
| Action | To avoid this situation, ensure that ConnectionConsumers using the queue provide a set of selectors that deal with all messages, or set the QueueConnectionFactory to retain messages. Alternatively, investigate the source of the message.

MQRC_JMS_FORMAT_ERROR (0x93C; 2364)

| Cause | JMS cannot interpret the message on the queue.
| Action | Investigate the origin of the message. JMS usually delivers messages of an unexpected format as a BytesMessage or TextMessage. Occasionally, this fails if the message is very badly formatted.

Other codes that appear in these fields are caused by a failed attempt to requeue the message to a Backout Queue. In this situation, the code describes the reason that the requeue failed. To diagnose the cause of these errors, refer to the WebSphere MQ Application Programming Reference.

If the report message cannot be put on the ReplyToQ, it is put on the dead-letter queue. In this situation, the feedback field of the MQMD is filled in as described above. The reason field in the MQDLH explains why the report message could not be placed on the ReplyToQ.

Application server sample code

Figure 6 on page 250 summarizes the principles of ServerSessionPool and ServerSession functionality.
1. The ConnectionConsumers get message references from the queue.
2. Each ConnectionConsumer selects specific message references.
3. The ConnectionConsumer buffer holds the selected message references.
4. The ConnectionConsumer requests one or more ServerSessions from the ServerSessionPool.
5. ServerSessions are allocated from the ServerSessionPool.
6. The ConnectionConsumer assigns message references to the ServerSessions and starts the ServerSession threads running.
7. Each ServerSession retrieves its referenced messages from the queue. It passes them to the onMessage method from the MessageListener that is associated with the JMS Session.
8. After it completes its processing, the ServerSession is returned to the pool.

Figure 6. ServerSessionPool and ServerSession functionality
Normally, the application server supplies ServerSessionPool and ServerSession functionality. However, WebSphere MQ JMS is supplied with a simple implementation of these interfaces, with program source. These samples are in the following directory, where <install_dir> is the installation directory for WebSphere MQ JMS:

<install_dir>/samples/jms/asf

These samples enable you to use the WebSphere MQ JMS ASF in a standalone environment (that is, you do not need a suitable application server). Also, they provide examples of how to implement these interfaces and take advantage of the WebSphere MQ JMS ASF. These examples are intended to aid both WebSphere MQ JMS users, and vendors of other application servers.

**MyServerSession.java**

This class implements the `javax.jms.ServerSession` interface. Its basic function is to associate a thread with a JMS Session. Instances of this class are pooled by a ServerSessionPool (see “MyServerSessionPool.java”). As a ServerSession, it must implement the following two methods:

- `getSession()`, which returns the JMS Session this associated with this ServerSession
- `start()`, which starts this ServerSession’s thread and results in the JMS Session’s `run()` method being invoked

MyServerSession also implements the `Runnable` interface. Therefore, the creation of the ServerSession’s thread can be based on this class, and does not need a separate class.

The class uses a `wait()`-`notify()` mechanism that is based on the values of two boolean flags, ready and quit. This mechanism means that the ServerSession creates and starts its associated thread during its construction. However, it does not automatically execute the body of the `run()` method. The body of the `run()` method is executed only when the ready flag is set to true by the `start()` method. The ASF calls the `start()` method when it is necessary to deliver messages to the associated JMS Session.

For delivery, the `run()` method of the JMS Session is called. The WebSphere MQ JMS ASF will have already loaded the `run()` method with messages.

After delivery completes, the ready flag is reset to false, and the owning ServerSessionPool is notified that delivery is complete. The ServerSession then remains in a wait state until either the `start()` method is called again, or the `close()` method is invoked and ends this ServerSession’s thread.

**MyServerSessionPool.java**

This class implements the `javax.jms.ServerSessionPool` interface, and exists to create and control access to a pool of ServerSessions.

In this simple implementation, the pool consists of a static array of ServerSession objects that are created during the construction of the pool. The following four parameters are passed into the constructor:

- `javax.jms.Connection connection`
  The connection used to create JMS Sessions.
- `int capacity`
Application server sample code

The size of the array of MyServerSession objects.

- **int ackMode**
  
The required acknowledge mode of the JMS Sessions.

- **MessageListenerFactory mlf**
  
The MessageListenerFactory that creates the message listener that is supplied to the JMS Sessions. See MessageListenerFactory.java.

The pool’s constructor uses these parameters to create an array of MyServerSession objects. The supplied connection is used to create JMS Sessions of the given acknowledge mode and correct domain (QueueSessions for point-to-point and TopicSessions for publish/subscribe). The Sessions are supplied with a message listener. Finally, the ServerSession objects, based on the JMS Sessions, are created.

This sample implementation is a static model. That is, all the ServerSessions in the pool are created when the pool is created, and after this, the pool cannot grow or shrink. This approach is just for simplicity. It is possible for a ServerSessionPool to use a sophisticated algorithm to create ServerSessions dynamically, as needed.

MyServerSessionPool keeps a record of which ServerSessions are currently in use by maintaining an array of boolean values called inUse. These booleans are all initialized to false. When the getServerSession method is invoked and requests a ServerSession from the pool, the inUse array is searched for the first false value. When one is found, the boolean is set to true and the corresponding ServerSession is returned. If there are no false values in the inUse array, the getServerSession method must wait() until notification occurs.

Notification occurs in either of the following circumstances:

- The pool’s close() method is called, indicating that the pool should be shut down.

- A ServerSession that is currently in use completes its workload and calls the serverSessionFinished method. The serverSessionFinished method returns the ServerSession to the pool, and sets the corresponding inUse flag to false. The ServerSession then becomes eligible for re-use.

MessageListenerFactory.java

In this sample, a message listener factory object is associated with each ServerSessionPool instance. The MessageListenerFactory class represents a very simple interface that is used to obtain an instance of a class that implements the javax.jms.MessageListener interface. The class contains a single method:

```
javax.jms.MessageListener createMessageListener();
```

An implementation of this interface is supplied when the ServerSessionPool is constructed. This object is used to create message listeners for the individual JMS Sessions that back up the ServerSessions in the pool. This architecture means that each separate implementation of the MessageListenerFactory interface must have its own ServerSessionPool.

WebSphere MQ JMS includes a sample MessageListenerFactory implementation, which is discussed in CountingMessageListenerFactory.java on page 254.
Examples of ASF use

There is a set of classes, with their source, in the directory <install_dir>/samples/jms/asf (where <install_dir> is the installation directory for WebSphere MQ JMS). These classes use the WebSphere MQ JMS application server facilities that are described in “ASF classes and functions” on page 243 within the sample standalone application server environment that is described in “Application server sample code” on page 249.

These samples provide examples of ASF use from the perspective of a client application:

- A simple point-to-point example uses:
  - ASFClient1.java
  - Load1.java
  - CountingMessageListenerFactory.java

- A more complex point-to-point example uses:
  - ASFClient2.java
  - Load2.java
  - CountingMessageListenerFactory.java
  - LoggingMessageListenerFactory.java

- A simple publish/subscribe example uses:
  - ASFClient3.java
  - TopicLoad.java
  - CountingMessageListenerFactory.java

- A more complex publish/subscribe example uses:
  - ASFClient4.java
  - TopicLoad.java
  - CountingMessageListenerFactory.java
  - LoggingMessageListenerFactory.java

- A publish/subscribe example using a durable ConnectionConsumer uses:
  - ASFClient5.java
  - TopicLoad.java

The following sections describe each class in turn.

Load1.java

This class is a simple generic JMS application that loads a given queue with a number of messages, then terminates. It can either retrieve the required administered objects from a JNDI namespace, or create them explicitly, using the WebSphere MQ JMS classes that implement these interfaces. The administered objects that are required are a QueueConnectionFactory and a Queue. You can use the command line options to set the number of messages with which to load the queue, and the sleep time between individual message puts.

This application has two versions of the command line syntax.

For use with JNDI, the syntax is:

```
java Load1 [-icf jndiICF] [-url jndiURL] [-qcfLookup qcfLookup]
            [-qLookup qLookup] [-sleep sleepTime] [-msgs numMsgs]
```
Examples of ASF use

For use without JNDI, the syntax is:

```
java Load1 -nojndi [-qm qMgrName] [-q qName]
    [-sleep sleepTime] [-msgs numMsgs]
```

Table 29 describes the parameters and gives their defaults.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>jndiICF</td>
<td>Initial context factory class used for JNDI</td>
<td>com.sun.jndi.ldap.LdapCtxFactory</td>
</tr>
<tr>
<td>jndiURL</td>
<td>Provider URL used for JNDI</td>
<td>ldap://localhost/o=ibm,c=us</td>
</tr>
<tr>
<td>qcfLookup</td>
<td>JNDI lookup key used for QueueConnectionFactory</td>
<td>cn=qcf</td>
</tr>
<tr>
<td>qLookup</td>
<td>JNDI lookup key used for Queue</td>
<td>cn=q</td>
</tr>
<tr>
<td>qMgrName</td>
<td>Name of queue manager to connect to</td>
<td>&quot;&quot; (use the default queue manager)</td>
</tr>
<tr>
<td>qName</td>
<td>Name of queue to load</td>
<td>SYSTEM.DEFAULT.LOCAL.QUEUE</td>
</tr>
<tr>
<td>sleepTime</td>
<td>Time (in milliseconds) to pause between message puts</td>
<td>0 (no pause)</td>
</tr>
<tr>
<td>numMsgs</td>
<td>Number of messages to put</td>
<td>1000</td>
</tr>
</tbody>
</table>

If there are any errors, an error message is displayed, and the application terminates.

You can use this application to simulate message load on a WebSphere MQ queue. In turn, this message load could trigger the ASF-enabled applications described in the following sections. The messages put to the queue are simple JMS TextMessage objects. These objects do not contain user-defined message properties, which could be useful to make use of different message listeners. The source code is supplied so that you can modify this load application if necessary.

**CountingMessageListenerFactory.java**

This file contains definitions for two classes:

- CountingMessageListener
- CountingMessageListenerFactory

CountingMessageListener is a very simple implementation of the `javax.jms.MessageListener` interface. It keeps a record of the number of times its `onMessage` method has been invoked, but does nothing with the messages it is passed.

CountingMessageListenerFactory is the factory class for CountingMessageListener. It is an implementation of the `MessageListenerFactory` interface described in "MessageListenerFactory.java" on page 252. This factory keeps a record of all the message listeners that it produces. It also includes a method, `printStats()`, which displays usage statistics for each of these listeners.
This application acts as a client of the WebSphere MQ JMS ASF. It sets up a single ConnectionConsumer to consume the messages in a single WebSphere MQ queue. It displays throughput statistics for each message listener that is used, and terminates after one minute.

The application can either retrieve the required administered objects from a JNDI namespace, or create them explicitly, using the WebSphere MQ JMS classes that implement these interfaces. The administered objects that are required are a QueueConnectionFactory and a Queue.

This application has two versions of the command line syntax:

For use with JNDI, the syntax is:

```
```

For use without JNDI, the syntax is:

```
java ASFClient1 -nojndi [-qm qMgrName] [-q qName] [-poolSize poolSize] [-batchSize batchSize]
```

Table 30 describes the parameters and gives their defaults.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>jndiICF</td>
<td>Initial context factory class used for JNDI</td>
<td>com.sun.jndi.ldap.LdapCtxFactory</td>
</tr>
<tr>
<td>jndiURL</td>
<td>Provider URL used for JNDI</td>
<td>ldap://localhost/o=ibm,c=us</td>
</tr>
<tr>
<td>qcfLookup</td>
<td>JNDI lookup key used for QueueConnectionFactory</td>
<td>cn=qcf</td>
</tr>
<tr>
<td>qLookup</td>
<td>JNDI lookup key used for Queue</td>
<td>cn=q</td>
</tr>
<tr>
<td>qMgrName</td>
<td>Name of queue manager to connect to</td>
<td>&quot;&quot; (use the default queue manager)</td>
</tr>
<tr>
<td>qName</td>
<td>Name of queue to consume from</td>
<td>SYSTEM.DEFAULT.LOCAL.QUEUE</td>
</tr>
<tr>
<td>poolSize</td>
<td>The number of ServerSessions created in the ServerSessionPool being used</td>
<td>5</td>
</tr>
<tr>
<td>batchSize</td>
<td>The maximum number of message that can be assigned to a ServerSession at a time</td>
<td>10</td>
</tr>
</tbody>
</table>

The application obtains a QueueConnection from the QueueConnectionFactory.

A ServerSessionPool, in the form of a MyServerSessionPool, is constructed using:
- the QueueConnection that was created previously
- the required poolSize
- an acknowledge mode, AUTO_ACKNOWLEDGE
- an instance of a CountingMessageListenerFactory, as described in "CountingMessageListenerFactory.java" on page 254
Then, the connection’s `createConnectionConsumer` method is invoked, passing in:

- the Queue that was obtained earlier
- a null message selector (indicating that all messages should be accepted)
- the `ServerSessionPool` that was just created
- the `batchSize` that is required

The consumption of messages is then started through invocation of the connection’s `start()` method.

The client application displays throughput statistics for each message listener that is used, displaying statistics every 10 seconds. After one minute, the connection is closed, the server session pool is stopped, and the application terminates.

**Load2.java**

This class is a JMS application that loads a given queue with a number of messages, then terminates, in a similar way to `Load1.java`. The command line syntax is also similar to that for `Load1.java` (substitute `Load2` for `Load1` in the syntax). For details, see "`Load1.java` on page 253".

The difference is that each message contains a user property called value, which takes a randomly selected integer value between 0 and 100. This property means that you can apply message selectors to the messages. Consequently, the messages can be shared between the two consumers that are created in the client application described in "`ASFClient2.java`".

**LoggingMessageListenerFactory.java**

This file contains definitions for two classes:

- `LoggingMessageListener`
- `LoggingMessageListenerFactory`

`LoggingMessageListener` is an implementation of the `javax.jms.MessageListener` interface. It takes the messages that are passed to it and writes an entry to the log file. The default log file is `.ASFClient2.log`. You can inspect this file and check the messages that are sent to the connection consumer that is using this message listener.

`LoggingMessageListenerFactory` is the factory class for `LoggingMessageListener`. It is an implementation of the `MessageListenerFactory` interface described in "`MessageListenerFactory.java` on page 252".

**ASFClient2.java**

`ASFClient2.java` is a slightly more complicated client application than `ASFClient1.java`. It creates two `ConnectionConsumers` that feed off the same queue, but that apply different message selectors. The application uses a `CountingMessageListenerFactory` for one consumer, and a `LoggingMessageListenerFactory` for the other. Use of two different message listener factories means that each consumer must have its own server session pool.

The application displays statistics that relate to one `ConnectionConsumer` on screen, and writes statistics that relate to the other `ConnectionConsumer` to a log file.
The command line syntax is similar to that for "ASFClient1.java" on page 255 (substitute ASFClient2 for ASFClient1 in the syntax). Each of the two server session pools contains the number of ServerSessions set by the poolSize parameter.

There should be an uneven distribution of messages. The messages loaded onto the source queue by Load2 contain a user property, where the value should be between 0 and 100, evenly and randomly distributed. The message selector value>75 is applied to highConnectionConsumer, and the message selector value≤75 is applied to normalConnectionConsumer. The highConnectionConsumer’s messages (approximately 25% of the total load) are sent to a LoggingMessageListener. The normalConnectionConsumer’s messages (approximately 75% of the total load) are sent to a CountingMessageListener.

When the client application runs, statistics that relate to the normalConnectionConsumer, and its associated CountingMessageListenerFactories, are printed to screen every 10 seconds. Statistics that relate to the highConnectionConsumer, and its associated LoggingMessageListenerFactories, are written to the log file.

You can inspect the screen and the log file to see the real destination of the messages. Add the totals for each of the CountingMessageListeners. As long as the client application does not terminate before all the messages are consumed, this should account for approximately 75% of the load. The number of log file entries should account for the remainder of the load. (If the client application terminates before all the messages are consumed, you can increase the application timeout.)

**TopicLoad.java**

This class is a JMS application that is a publish/subscribe version of the Load2 queue loader described in "Load2.java" on page 256. It publishes the required number of messages under the given topic, then it terminates. Each message contains a user property called value, which takes a randomly selected integer value between 0 and 100.

To use this application, ensure that the broker is running and that the required setup is complete. For details, see "Additional setup for publish/subscribe mode" on page 24.

This application has two versions of the command line syntax.

For use with JNDI, the syntax is:
```
```

For use without JNDI, the syntax is:
```
java TopicLoad -nojndi [-qm qMgrName] [-t tName] [-sleep sleepTime] [-msgs numMsgs]
```

Table 31 describes the parameters and gives their defaults.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>jndiICF</td>
<td>Initial context factory class used for JNDI</td>
<td>com.sun.jndi.ldap.LdapCtxFactory</td>
</tr>
<tr>
<td>jndiURL</td>
<td>Provider URL used for JNDI</td>
<td>ldap://localhost/o=ibm,c=us</td>
</tr>
</tbody>
</table>
Table 31. *TopicLoad* parameters and defaults (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcfLookup</td>
<td>JNDI lookup key used for TopicConnectionFactory</td>
<td>cn=tcf</td>
</tr>
<tr>
<td>tLookup</td>
<td>JNDI lookup key used for Topic</td>
<td>cn=t</td>
</tr>
<tr>
<td>qMgrName</td>
<td>Name of queue manager to connect to, and broker queue manager to publish messages to</td>
<td>&quot;&quot; (use the default queue manager)</td>
</tr>
<tr>
<td>tName</td>
<td>Name of topic to publish to MQJMS/ASF/TopicLoad</td>
<td></td>
</tr>
<tr>
<td>sleepTime</td>
<td>Time (in milliseconds) to pause between message puts</td>
<td>0 (no pause)</td>
</tr>
<tr>
<td>numMsgs</td>
<td>Number of messages to put</td>
<td>200</td>
</tr>
</tbody>
</table>

If there are any errors, an error message is displayed, and the application terminates.

**ASFClient3.java**

ASFClient3.java is a client application that is a publish/subscribe version of "ASFClient1.java" on page 255. It sets up a single ConnectionConsumer to consume the messages published on a single Topic. It displays throughput statistics for each message listener that is used, and terminates after one minute.

This application has two versions of the command line syntax.

For use with JNDI, the syntax is:

```
java ASFClient3 [-icf jndiICF] [-url jndiURL] [-tcfLookup tcfLookup]
  [-tLookup tLookup] [-poolsize poolSize] [-batchsize batchSize]
```

For use without JNDI, the syntax is:

```
java ASFClient3 -nojndi [-qm qMgrName] [-t tName]
  [-poolsize poolSize] [-batchsize batchSize]
```

Table 32 describes the parameters and gives their defaults.

Table 32. *ASFClient3* parameters and defaults

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>jndiICF</td>
<td>Initial context factory class used for JNDI</td>
<td>com.sun.jndi.ldap.LdapCtxFactory</td>
</tr>
<tr>
<td>jndiURL</td>
<td>Provider URL used for JNDI</td>
<td>ldap://localhost/o=ibm,c=us</td>
</tr>
<tr>
<td>tcfLookup</td>
<td>JNDI lookup key used for TopicConnectionFactory</td>
<td>cn=tcf</td>
</tr>
<tr>
<td>tLookup</td>
<td>JNDI lookup key used for Topic</td>
<td>cn=t</td>
</tr>
<tr>
<td>qMgrName</td>
<td>Name of queue manager to connect to, and broker queue manager to publish messages to</td>
<td>&quot;&quot; (use the default queue manager)</td>
</tr>
<tr>
<td>tName</td>
<td>Name of topic to consume from MQJMS/ASF/TopicLoad</td>
<td></td>
</tr>
<tr>
<td>poolSize</td>
<td>The number of ServerSessions created in the ServerSessionPool being used</td>
<td>5</td>
</tr>
</tbody>
</table>
Like ASFClient1, the client application displays throughput statistics for each message listener that is used, displaying statistics every 10 seconds. After one minute, the connection is closed, the server session pool is stopped, and the application terminates.

**ASFClient4.java**

ASFClient4.java is a more complex publish/subscribe client application. It creates three ConnectionConsumers that all feed off the same topic, but each one applies different message selectors.

The first two consumers use ‘high’ and ‘normal’ message selectors, in the same way as the application "ASFClient2.java" on page 256. The third consumer does not use any message selector. The application uses two CountingMessageListenerFactories for the two selector-based consumers, and a LoggingMessageListenerFactory for the third consumer. Because the application uses different message listener factories, each consumer must have its own server session pool.

The application displays statistics that relate to the two selector-based consumers on screen. It writes statistics that relate to the third ConnectionConsumer to a log file.

The command line syntax is similar to that for "ASFClient3.java" on page 258 (substitute ASFClient4 for ASFClient3 in the syntax). Each of the three server session pools contains the number of ServerSessions set by the poolSize parameter.

When the client application runs, statistics that relate to the normalConnectionConsumer and the highConnectionConsumer, and their associated CountingMessageListenerFactories, are printed to screen every 10 seconds. Statistics that relate to the third ConnectionConsumer, and its associated LoggingMessageListenerFactories, are written to the log file.

You can inspect the screen and the log file to see the real destination of the messages. Add the totals for each of the CountingMessageListeners and inspect the number of log file entries.

The distribution of messages should be different from the distribution obtained by a point-to-point version of the same application (ASFClient2.java). This is because, in the publish/subscribe domain, each consumer of a topic obtains its own copy of each message published on that topic. In this application, for a given topic load, the ‘high’ and ‘normal’ consumers will receive approximately 25% and 75% of the load, respectively. The third consumer will still receive 100% of the load. Therefore, the total number of messages received is greater than 100% of the load originally published on the topic.
Examples of ASF use

**ASFClient5.java**

This sample exercises the durable publish/subscribe ConnectionConsumer functionality in WebSphere MQ JMS.

You invoke it with the same command-line options as the ASFClient4 sample, and, as with the other samples, the TopicLoad sample application can be used to trigger the consumer that is created. For details of TopicLoad, see “TopicLoad.java” on page 257.

When invoked, ASFClient5 displays a menu of three options:

1. Create/reactivate a durable ConnectionConsumer
2. Unsubscribe a durable ConnectionConsumer
3. Exit

If option 1 is chosen, and this is the first time this sample has been run, then a brand new durable ConnectionConsumer will be created using the given name. It will then display one minute’s worth of throughput statistics, rather like the other samples, before closing the connection and terminating.

Having created a durable consumer, messages published on the topic in question will continue to arrive at the consumer’s destination even though the consumer is inactive.

This can be confirmed by running ASFClient5 again, and again selecting option 1. This will reactivate the named durable consumer, and the statistics displayed will show that any relevant messages published during the period of inactivity were subsequently delivered to the consumer.

If you run ASFClient5 once again and select option 2, this will unsubscribe the named durable ConnectionConsumer and discard any outstanding messages delivered to it. Do this to ensure that the broker does not continue to deliver unwanted messages.
Chapter 14. JMS interfaces and classes

WebSphere MQ classes for Java Message Service consists of a number of Java classes and interfaces that are based on the Sun javax.jms package of interfaces and classes. Clients should be written using the Sun interfaces and classes that are listed below, and that are described in detail in the following sections. The names of the WebSphere MQ objects that implement the Sun interfaces and classes have a prefix of “MQ” (unless stated otherwise in the object description). The descriptions include details about any deviations of the WebSphere MQ objects from the standard JMS definitions. These deviations are marked with ‘*’.

Sun Java Message Service classes and interfaces

The following tables list the JMS objects contained in the package javax.jms. Interfaces marked with ‘*’ are implemented by applications. Interfaces marked with ‘**’ are implemented by application servers.

Table 33. Summary of interfaces in package javax.jms

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BytesMessage</td>
<td>A BytesMessage is used to send a message containing a stream of uninterpreted bytes.</td>
</tr>
<tr>
<td>Connection</td>
<td>A JMS Connection is a client’s active connection to its JMS provider.</td>
</tr>
<tr>
<td>ConnectionConsumer</td>
<td>For application servers, Connections provide a special facility for creating a ConnectionConsumer.</td>
</tr>
<tr>
<td>ConnectionFactory</td>
<td>AConnectionFactory encapsulates a set of connection configuration parameters that an administrator has defined.</td>
</tr>
<tr>
<td>ConnectionMetaData</td>
<td>ConnectionMetaData provides information that describes the Connection.</td>
</tr>
<tr>
<td>DeliveryMode</td>
<td>Delivery modes supported by JMS.</td>
</tr>
<tr>
<td>Destination</td>
<td>The parent interface for Queue and Topic.</td>
</tr>
<tr>
<td>ExceptionListener*</td>
<td>An exception listener is used to receive exceptions thrown by Connections asynchronous delivery threads.</td>
</tr>
<tr>
<td>MapMessage</td>
<td>A MapMessage is used to send a set of name-value pairs where names are Strings and values are Java primitive types.</td>
</tr>
<tr>
<td>Message</td>
<td>The Message interface is the root interface of all JMS messages.</td>
</tr>
<tr>
<td>MessageConsumer</td>
<td>The parent interface for all message consumers.</td>
</tr>
<tr>
<td>MessageListener*</td>
<td>A MessageListener is used to receive asynchronously delivered messages.</td>
</tr>
<tr>
<td>MessageProducer</td>
<td>A client uses a MessageProducer to send messages to a Destination.</td>
</tr>
<tr>
<td>ObjectMessage</td>
<td>An ObjectMessage is used to send a message that contains a serializable Java object.</td>
</tr>
<tr>
<td>Queue</td>
<td>A Queue object encapsulates a provider-specific queue name.</td>
</tr>
</tbody>
</table>
Table 33. Summary of interfaces in package javax.jms (continued)

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueueBrowser</td>
<td>A client uses a QueueBrowser to look at messages on a queue without removing them.</td>
</tr>
<tr>
<td>QueueConnection</td>
<td>A QueueConnection is an active connection to a JMS point-to-point provider.</td>
</tr>
<tr>
<td>QueueConnectionFactory</td>
<td>A client uses a QueueConnectionFactory to create QueueConnections with a JMS point-to-point provider.</td>
</tr>
<tr>
<td>QueueReceiver</td>
<td>A client uses a QueueReceiver to receive messages that have been delivered to a queue.</td>
</tr>
<tr>
<td>QueueSender</td>
<td>A client uses a QueueSender to send messages to a queue.</td>
</tr>
<tr>
<td>QueueSession</td>
<td>A QueueSession provides methods to create QueueReceivers, QueueSenders, QueueBrowsers and TemporaryQueues.</td>
</tr>
<tr>
<td>ServerSession **</td>
<td>A ServerSession is an object implemented by an application server.</td>
</tr>
<tr>
<td>ServerSessionPool **</td>
<td>A ServerSessionPool is an object implemented by an application server to provide a pool of ServerSessions for processing the messages of a ConnectionConsumer.</td>
</tr>
<tr>
<td>Session</td>
<td>A JMS Session is a single threaded context for producing and consuming messages.</td>
</tr>
<tr>
<td>StreamMessage</td>
<td>A StreamMessage is used to send a stream of Java primitives.</td>
</tr>
<tr>
<td>TemporaryQueue</td>
<td>A TemporaryQueue is a unique Queue object created for the duration of a QueueConnection.</td>
</tr>
<tr>
<td>TemporaryTopic</td>
<td>A TemporaryTopic is a unique Topic object created for the duration of a TopicConnection.</td>
</tr>
<tr>
<td>TextMessage</td>
<td>A TextMessage is used to send a message containing a java.lang.String.</td>
</tr>
<tr>
<td>Topic</td>
<td>A Topic object encapsulates a provider-specific topic name.</td>
</tr>
<tr>
<td>TopicConnection</td>
<td>A TopicConnection is an active connection to a JMS Pub/Sub provider.</td>
</tr>
<tr>
<td>TopicConnectionFactory</td>
<td>A client uses a TopicConnectionFactory to create TopicConnections with a JMS Publish/Subscribe provider.</td>
</tr>
<tr>
<td>TopicPublisher</td>
<td>A client uses a TopicPublisher for publishing messages on a topic.</td>
</tr>
<tr>
<td>TopicSession</td>
<td>A TopicSession provides methods to create TopicPublishers, TopicSubscribers and TemporaryTopics.</td>
</tr>
<tr>
<td>TopicSubscriber</td>
<td>A client uses a TopicSubscriber to receive messages that have been published to a topic.</td>
</tr>
<tr>
<td>XAConnection</td>
<td>XAConnection extends the capability of Connection by providing an XASession.</td>
</tr>
<tr>
<td>XAConnectionFactory</td>
<td>Some application servers provide support for grouping Java Transaction Service (JTS)-capable resource use into a distributed transaction.</td>
</tr>
<tr>
<td>XAQueueConnection</td>
<td>XAQueueConnection provides the same create options as QueueConnection.</td>
</tr>
</tbody>
</table>
Table 33. Summary of interfaces in package javax.jms (continued)

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAQueueConnectionFactory</td>
<td>An XAQueueConnectionFactory provides the same create options as a QueueConnectionFactory.</td>
</tr>
<tr>
<td>XAQueueSession</td>
<td>An XAQueueSession provides a regular QueueSession which can be used to create Queue Receivers, Queue Senders and Queue Browsers.</td>
</tr>
<tr>
<td>XASession</td>
<td>XASession extends the capability of Session by adding access to a JMS provider’s support for the Java Transaction API (JTA).</td>
</tr>
<tr>
<td>XATopicConnection</td>
<td>An XATopicConnection provides the same create options as TopicConnection.</td>
</tr>
<tr>
<td>XATopicConnectionFactory</td>
<td>An XATopicConnectionFactory provides the same create options as TopicConnectionFactory.</td>
</tr>
<tr>
<td>XATopicSession</td>
<td>An XATopicSession provides a regular TopicSession which can be used to create Topic Subscribers and Topic Publishers.</td>
</tr>
</tbody>
</table>

Table 34. Summary of classes in package javax.jms

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueueRequestor</td>
<td>JMS provides a QueueRequestor helper class to simplify making service requests.</td>
</tr>
<tr>
<td>TopicRequestor</td>
<td>JMS provides a TopicRequestor helper class to simplify making service requests.</td>
</tr>
</tbody>
</table>
WebSphere MQ JMS classes

Two packages contain the WebSphere MQ classes for Java Message Service that implement the Sun interfaces. Table 35 shows the interfaces implemented by classes in the `com.ibm.mq.jms` package and Table 36 on page 265 those implemented by classes in the `com.ibm.jms` package.

Usually, do not use the implementation classes directly, but instead program to the JMS interfaces. Note that many of the interfaces are not applicable when running a publish/subscribe application on a direct connection to the IBM WebSphere MQ Event Broker broker. Where the names of implementation classes are listed, provider-specific methods are documented in this chapter.

Table 35. Summary of classes in package `com.ibm.mq.jms`

<table>
<thead>
<tr>
<th>JMS interface</th>
<th>Client or bindings implementation</th>
<th>Direct connection to WebSphere MQ Event Broker implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cleanup</td>
<td>MQConnection</td>
<td>Y</td>
</tr>
<tr>
<td>Connection</td>
<td>MQConnectionConsumer</td>
<td></td>
</tr>
<tr>
<td>ConnectionConsumer</td>
<td>MQConnectionConsumer</td>
<td></td>
</tr>
<tr>
<td>ConnectionFactory</td>
<td>MQConnectionFactory</td>
<td></td>
</tr>
<tr>
<td>ConnectionMetaData</td>
<td>MQConnectionMetaData</td>
<td>Y</td>
</tr>
<tr>
<td>Destination</td>
<td>MQDestination</td>
<td></td>
</tr>
<tr>
<td>MessageConsumer</td>
<td>MQMessageConsumer</td>
<td></td>
</tr>
<tr>
<td>MessageProducer</td>
<td>MQMessageProducer</td>
<td></td>
</tr>
<tr>
<td>Queue</td>
<td>MQQueue</td>
<td></td>
</tr>
<tr>
<td>QueueBrowser</td>
<td>MQQueueBrowser</td>
<td></td>
</tr>
<tr>
<td>QueueConnection</td>
<td>MQQueueConnection</td>
<td></td>
</tr>
<tr>
<td>QueueConnectionFactory</td>
<td>MQQueueConnectionFactory</td>
<td></td>
</tr>
<tr>
<td>MQQueueEnumeration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QueueReceiver</td>
<td>MQQueueReceiver</td>
<td></td>
</tr>
<tr>
<td>QueueSender</td>
<td>MQQueueSender</td>
<td></td>
</tr>
<tr>
<td>QueueSession</td>
<td>MQQueueSession</td>
<td></td>
</tr>
<tr>
<td>Session</td>
<td>MQSession</td>
<td>Y</td>
</tr>
<tr>
<td>TemporaryQueue</td>
<td>MQTemporaryQueue</td>
<td></td>
</tr>
<tr>
<td>TemporaryTopic</td>
<td>MQTemporaryTopic</td>
<td>Y</td>
</tr>
<tr>
<td>Topic</td>
<td>MQTopic</td>
<td>Y</td>
</tr>
<tr>
<td>TopicConnection</td>
<td>MQTopicConnection</td>
<td>Y</td>
</tr>
<tr>
<td>TopicConnectionFactory</td>
<td>MQTopicConnectionFactory</td>
<td>Y</td>
</tr>
<tr>
<td>TopicPublisher</td>
<td>MQTopicPublisher</td>
<td>Y</td>
</tr>
<tr>
<td>TopicSession</td>
<td>MQTopicSession</td>
<td>Y</td>
</tr>
<tr>
<td>TopicSubscriber</td>
<td>MQTopicSubscriber</td>
<td>Y</td>
</tr>
<tr>
<td>XAConnection</td>
<td>MQXAConnection</td>
<td></td>
</tr>
<tr>
<td>XAConnectionFactory</td>
<td>MQXAConnectionFactory</td>
<td></td>
</tr>
<tr>
<td>XAQueueConnection</td>
<td>MQXAQueueConnection</td>
<td></td>
</tr>
</tbody>
</table>
Table 35. Summary of classes in package com.ibm.mq.jms (continued)

<table>
<thead>
<tr>
<th>JMS interface</th>
<th>Client or bindings implementation</th>
<th>Direct connection to WebSphere MQ Event Broker implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAQueueConnectionFactory</td>
<td>MQXAQueueConnectionFactory</td>
<td></td>
</tr>
<tr>
<td>XAQueueSession</td>
<td>MQXAQueueSession</td>
<td></td>
</tr>
<tr>
<td>XASession</td>
<td>MQXAASession</td>
<td></td>
</tr>
<tr>
<td>XATopicConnectionFactory</td>
<td>MQXATopicConnectionFactory</td>
<td></td>
</tr>
<tr>
<td>XATopicSession</td>
<td>MQXATopicSession</td>
<td></td>
</tr>
</tbody>
</table>

Table 36. Summary of classes in package com.ibm.jms

<table>
<thead>
<tr>
<th>JMS interface</th>
<th>Client or bindings implementation</th>
<th>Direct connection to WebSphere MQ Event Broker implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BytesMessage</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>MapMessage</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Message</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ObjectMessage</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>StreamMessage</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>TextMessage</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

A sample implementation of the following JMS interfaces is supplied in this release of WebSphere MQ classes for Java Message Service.

- ServerSession
- ServerSessionPool

See "Application server sample code" on page 249
A BytesMessage is used to send a message containing a stream of uninterpreted bytes. It inherits Message and adds a bytes message body. The receiver of the message supplies the interpretation of the bytes.

Note: This message type is for client encoding of existing message formats. If possible, one of the other self-defining message types should be used instead.

See also: MapMessage, Message, ObjectMessage, StreamMessage, and TextMessage.

Methods

readBoolean

```java
public boolean readBoolean() throws JMSException
```

Read a boolean from the bytes message.

**Returns:** the boolean value read.

**Throws:**

- MessageNotReadableException - if the message is in write-only mode.
- JMSEception - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if it is the end of the message bytes.

readByte

```java
public byte readByte() throws JMSException
```

Read a signed 8-bit value from the bytes message.

**Returns:** the next byte from the bytes message as a signed 8-bit byte.

**Throws:**

- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSEception - if JMS fails to read the message because of an internal JMS error.
BytesMessage

readBytes

public int readBytes(byte[] value) throws JMSException

Read a byte array from the bytes message. If there are sufficient bytes remaining in the stream the entire buffer is filled, if not, the buffer is partially filled.

Parameters:
value - the buffer into which the data is read.

Returns:
the total number of bytes read into the buffer, or -1 if there is no more data because the end of the bytes has been reached.

Throws:
• MessageNotReadableException - if the message is in write-only mode.
• JMSException - if JMS fails to read the message because of an internal JMS error.

readBytes

public int readBytes(byte[] value, int length) throws JMSException

Read a portion of the bytes message.

Parameters:
value - the buffer into which the data is read.
length - the number of bytes to read.

Returns:
the total number of bytes read into the buffer, or -1 if there is no more data because the end of the bytes has been reached.

Throws:
• MessageNotReadableException - if the message is in write-only mode.
• IndexOutOfBoundsException - if length is negative, or is less than the length of the array value
• JMSException - if JMS fails to read the message because of an internal JMS error.

readChar

public char readChar() throws JMSException

Read a Unicode character value from the bytes message.

Returns:
the next two bytes from the bytes message as a Unicode character.

Throws:
• MessageNotReadableException - if the message is in write-only mode.
• MessageEOFException - if it is the end of the message bytes.
• JMSException - if JMS fails to read the message because of an internal JMS error.
BytesMessage

readDouble

public double readDouble() throws JMSException

Read a double from the bytes message.

Returns:
the next eight bytes from the bytes message, interpreted as a double.

Throws:
- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

readFloat

public float readFloat() throws JMSException

Read a float from the bytes message.

Returns:
the next four bytes from the bytes message, interpreted as a float.

Throws:
- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

readInt

public int readInt() throws JMSException

Read a signed 32-bit integer from the bytes message.

Returns:
the next four bytes from the bytes message, interpreted as an int.

Throws:
- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

readLong

public long readLong() throws JMSException

Read a signed 64-bit integer from the bytes message.

Returns:
the next eight bytes from the bytes message, interpreted as a long.

Throws:
- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
**BytesMessage**

- JMSException - if JMS fails to read the message because of an internal JMS error.

**readShort**

```java
public short readShort() throws JMSException
```

Read a signed 16-bit number from the bytes message.

**Returns:**

- the next two bytes from the bytes message, interpreted as a signed 16-bit number.

**Throws:**

- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

**readUnsignedByte**

```java
public int readUnsignedByte() throws JMSException
```

Read an unsigned 8-bit number from the bytes message.

**Returns:**

- the next byte from the bytes message, interpreted as an unsigned 8-bit number.

**Throws:**

- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

**readUnsignedShort**

```java
public int readUnsignedShort() throws JMSException
```

Read an unsigned 16-bit number from the bytes message.

**Returns:**

- the next two bytes from the bytes message, interpreted as an unsigned 16-bit integer.

**Throws:**

- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.
BytesMessage

readUTF

public java.lang.String readUTF() throws JMSException

Read in a string that has been encoded using a modified UTF-8 format from the bytes message. The first two bytes are interpreted as a 2-byte length field.

Returns:

a Unicode string from the bytes message.

Throws:

- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

reset

public void reset() throws JMSException

Put the message body in read-only mode, and reposition the bytes of bytes to the beginning.

Throws:

- JMSException - if JMS fails to reset the message because of an internal JMS error.
- MessageFormatException - if message has an incorrect format

writeBoolean

public void writeBoolean(boolean value) throws JMSException

Write a boolean to the bytes message as a 1-byte value. The value true is written out as the value (byte)1; the value false is written out as the value (byte)0.

Parameters:

value - the boolean value to be written.

Throws:

- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.

writeByte

public void writeByte(byte value) throws JMSException

Write out a byte to the bytes message as a 1-byte value.

Parameters:

value - the byte value to be written.

Throws:

- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.
writeBytes

public void writeBytes(byte[] value) throws JMSException

Write a byte array to the bytes message.

Parameters:
value - the byte array to be written.

Throws:
- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.

writeBytes

public void writeBytes(byte[] value, int length) throws JMSException

Write a portion of a byte array to the bytes message.

Parameters:
- value - the byte array value to be written.
- offset - the initial offset within the byte array.
- length - the number of bytes to use.

Throws:
- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.

writeChar

public void writeChar(char value) throws JMSException

Write a char to the bytes message as a 2-byte value, high byte first.

Parameters:
value - the char value to be written.

Throws:
- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.

writeDouble

public void writeDouble(double value) throws JMSException

Convert the double argument to a long using doubleToLongBits method in class Double, and then write that long value to the bytes message as an 8-byte quantity.

Parameters:
value - the double value to be written.

Throws:
- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.
BytesMessage

writeFloat

```java
public void writeFloat(float value) throws JMSException
```

Convert the float argument to an int using `Float.floatToIntBits` method in class `Float`, and then write that int value to the bytes message as a 4-byte quantity.

**Parameters:**
value - the float value to be written.

**Throws:**
- `MessageNotWriteableException` - if message in read-only mode.
- `JMSException` - if JMS fails to write the message because of an internal JMS error.

writeInt

```java
public void writeInt(int value) throws JMSException
```

Write an int to the bytes message as four bytes.

**Parameters:**
value - the int to be written.

**Throws:**
- `MessageNotWriteableException` - if message in read-only mode.
- `JMSException` - if JMS fails to write the message because of an internal JMS error.

writeLong

```java
public void writeLong(long value) throws JMSException
```

Write a long to the bytes message as eight bytes,

**Parameters:**
value - the long to be written.

**Throws:**
- `MessageNotWriteableException` - if message in read-only mode.
- `JMSException` - if JMS fails to write the message because of an internal JMS error.

writeObject

```java
public void writeObject(java.lang.Object value) throws JMSException
```

Write a Java object to the bytes message.

**Note:** This method only works for the primitive object types (such as `Integer`, `Double`, and `Long`), Strings, and byte arrays.

**Parameters:**
value - the Java object to be written.

**Throws:**
- `MessageNotWriteableException` - if message in read-only mode.
- `MessageFormatException` - if object is not a valid type.
- `JMSException` - if JMS fails to write the message because of an internal JMS error.
writeShort

public void writeShort(short value) throws JMSException

Write a short to the bytes message as two bytes.

Parameters:
value - the short to be written.

Throws:
• MessageNotWriteableException - if message in read-only mode.
• JMSException - if JMS fails to write the message because of an internal JMS error.

writeUTF

public void writeUTF(java.lang.String value)
    throws JMSException

Write a string to the bytes message using UTF-8 encoding in a machine-independent manner. The UTF-8 string written to the buffer starts with a 2-byte length field.

Parameters:
value - the String value to be written.

Throws:
• MessageNotWriteableException - if message in read-only mode.
• JMSException - if JMS fails to write the message because of an internal JMS error.
public class Cleanup
implements Runnable

WebSphere MQ class: Cleanup

Cleanup contains utilities for dealing with broken non-durable subscriptions using the SUBSTATE(BROKER) option. It is not applicable if you use a direct connection to WebSphere MQ Event Broker.

See also: ConnectionFactory.

WebSphere MQ constructor

Cleanup
public Cleanup()
Default constructor.

Cleanup
public Cleanup(MQTopicConnectionFactory mqtcf) throws JMSException
Constructor which copies property values from the supplied MQTopicConnectionFactory.

Methods

cleanup
public void cleanup() throws JMSException
This method executes Cleanup once. If cleanupLevel is NONE, it throws an IllegalStateException.

getCCSID
public int getCCSID()
Get the character set of the queue manager.

getchannel
public String getChannel()
For client only, get the channel that was used.

getCleanupInterval
public long getCleanupInterval()
Retrieve the cleanup interval.
getCleanupLevel
public int getCleanupLevel()

Retrieve the cleanup level.

getExceptionListener
public ExceptionListener getExceptionListener()

Returns the ExceptionListener.

getHostName
public String getHostName()

Retrieve the name of the host.

getPort
public int getPort()

For client connections, get the port number.

getQueueManager
public String getQueueManager()

Get the name of the queue manager.

getReceiveExit
public String getReceiveExit()

Get the name of the receive exit class.

getReceiveExitInit
public String getReceiveExitInit()

Get the initialization string that was passed to the receive exit class.

getSecurityExit
public String getSecurityExit()

Get the name of the security exit class.

getSecurityExitInit
public String getSecurityExitInit()

Get the security exit initialization string.

getSendExit
public String getSendExit()

Get the name of the send exit class.
Cleanup

getSendExitInit

```java
public String getSendExitInit()

Get the send exit initialization string.
```

getTransportType

```java
public int getTransportType()

Retrieve the transport type.
```

isRunning

```java
public boolean isRunning()

Returns true if the run() method is currently active.
```

main

```java
public static void main(String args[])
    throws java.io.UnsupportedEncodingException

    Allows the utility to be invoked from a command line. For details of the
    invocation options and parameters, see "Manual cleanup" on page 222.
```

run

```java
public void run()

Method to call to run this utility in the background at intervals, as
determined by the cleanupLevel and cleanupInterval properties.
```

setCCSID

```java
public void setCCSID(int x) throws JMSException

Set the character set to be used when connecting to the queue manager. See
Table 13 on page 120 for a list of allowed values. We recommend that you
use the default value (819) for most situations.
```

setChannel

```java
public void setChannel(String x) throws JMSException

For client only, set the channel to use.
```

setCleanupInterval

```java
public void setCleanupInterval(long interval) throws JMSException

Set the cleanupInterval.

Parameters:

- interval - length of time in milliseconds between runs of the cleanup utility

Throws:

  JMSException - if interval is less than 0
```
setCleanupLevel

`public void setCleanupLevel(int level) throws JMSException`

Set the cleanup level to use. It can be one of:
- JMSC.MQJMS_CLEANUP_NONE
- JMSC.MQJMS_CLEANUP_SAFE
- JMSC.MQJMS_CLEANUP_STRONG
- JMSC.MQJMS_CLEANUP_FORCE
- JMSC.MQJMS_CLEANUP_NONDUR

setExceptionListener

`public void setExceptionListener(ExceptionListener el)`

Sets the ExceptionListener. If set, the ExceptionListener receives any exceptions caused during the run() method. Shortly after issuing the exception to the ExceptionListener, Cleanup terminates.

setHostName

`public void setHostName(String hostname)`

For client connections, the name of the host to connect to.

setPort

`public void setPort(int port) throws JMSException`

Set the port for a client connection.

Parameters:
- `port` - the new value to use.

Throws:
- `JMSException` if the port is negative.

setQueueManager

`public void setQueueManager(String x) throws JMSException`

Set the name of the queue manager to connect to.

setReceiveExit

`public void setReceiveExit(String receiveExit)`

The name of a class that implements a receive exit.

setReceiveExitInit

`public void setReceiveExitInit(String x)`

Initialization string that is passed to the constructor of the receive exit class.
Cleanup

**setSecurityExit**

```java
public void setSecurityExit(String securityExit)
```

The name of a class that implements a security exit.

**setSecurityExitInit**

```java
public void setSecurityExitInit(String x)
```

Initialization string that is passed to the security exit constructor.

**setSendExit**

```java
public void setSendExit(String sendExit)
```

The name of a class that implements a send exit.

**setSendExitInit**

```java
public void setSendExitInit(String x)
```

Initialization string that is passed to the constructor of send exit.

**setTransportType**

```java
public void setTransportType(int x) throws JMSException
```

Set the transport type to use. It can be one of:

- JMSC.MQJMS_TP_BINDINGS_MQ
- JMSC.MQJMS_TP_CLIENT_MQ_TCPIP

**stop**

```java
public void stop()
```

Stops any currently running cleanup thread. Returns when cleanup has finished. Does nothing if cleanup is not running.
public interface Connection
Subinterfaces: QueueConnection, TopicConnection, XAQueueConnection, and XATopicConnection

WebSphere MQ class: MQConnection

java.lang.Object

A JMS Connection is a client's active connection to its JMS provider.

See also: QueueConnection, TopicConnection, XAQueueConnection, and XATopicConnection

Methods

close

public void close() throws JMSException

Because a provider may allocate some resources outside the JVM on behalf of a Connection, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough. There is no need to close the sessions, producers, and consumers of a closed connection.

Closing a connection causes any of its sessions' in-process transactions to be rolled back. In the case where a session's work is coordinated by an external transaction manager, when using XASession, a session's commit and rollback methods are not used and the result of a closed session's work is determined later by a transaction manager. Closing a connection does NOT force an acknowledge of client acknowledged sessions.

WebSphere MQ JMS keeps a pool of WebSphere MQ hConns available for use by Sessions. Under some circumstances, Connection.close() clears this pool. If an application uses multiple Connections sequentially, it is possible to force the pool to remain active between JMS Connections. To do this, register an MQPoolToken with com.ibm.mq.MQEnvironment for the lifetime of your JMS application. For details, see "Connection pooling" on page 73 and "MQEnvironment" on page 104.

Throws:

JMSException - if the JMS implementation fails to close the connection because of an internal error. Examples are a failure to release resources or to close a socket connection.
getClientID

```java
public java.lang.String getClientID() throws JMSException
```

Get the client identifier for this connection. The client identifier can either be preconfigured by the administrator in a ConnectionFactory, or assigned by calling setClientId.

**Returns:**
the unique client identifier.

**Throws:**
JMSException - if the JMS implementation fails to return the client ID for this Connection because of an internal error.

getExceptionListener

```java
public ExceptionListener getExceptionListener() throws JMSException
```

Get the ExceptionListener for this Connection.

**Returns:**
The ExceptionListener for this Connection

**Throws:**
JMSException - general exception if the JMS implementation fails to get the Exception listener for this Connection.

getMetaData

```java
public ConnectionMetaData getMetaData() throws JMSException
```

Get the metadata for this connection.

**Returns:**
the connection metadata.

**Throws:**
JMSException - general exception if the JMS implementation fails to get the Connection metadata for this Connection.

See also:
ConnectionMetaData

setClientID

```java
public void setClientID(java.lang.String clientID) throws JMSException
```

Set the client identifier for this connection.

**Note:** The client identifier is ignored for point-to-point connections.

---

**WebSphere MQ Event Broker note**
This method always throws an IllegalStateException when you make a direct connection to WebSphere MQ Event Broker.

**Parameters:**
clientID - the unique client identifier.

**Throws:**
Connection

- JMSException - if the JMS implementation fails to set the client ID for this Connection because of an internal error.
- InvalidClientIDException - if the JMS client specifies a non valid or duplicate client id.
- IllegalStateException - if attempting to set a connection’s client identifier at the wrong time, or if it has been configured administratively.

setExceptionListener

```
public void setExceptionListener(ExceptionListener listener)
```

Throws: JMSException - general exception if the JMS implementation fails to set the Exception listener for this Connection.

start

```
public void start() throws JMSException
```

Start (or restart) a Connection’s delivery of incoming messages. Starting a started session is ignored.

Throws: JMSException - if the JMS implementation fails to start the message delivery because of an internal error.

See also: stop

stop

```
public void stop() throws JMSException
```

Used to stop a Connection’s delivery of incoming messages temporarily. It can be restarted using its start method. When stopped, delivery to all the Connection’s message consumers is inhibited. Synchronous receives are blocked, and messages are not delivered to message listeners.

Stopping a session has no affect on its ability to send messages. Stopping a stopped session is ignored.

Throws: JMSException - if the JMS implementation fails to stop the message delivery because of an internal error.

See also: start

Chapter 14. JMS interfaces and classes 281
ConnectionConsumer

public interface ConnectionConsumer

WebSphere MQ class: MQConnectionConsumer

java.lang.Object

+----com.ibm.mq.jms.MQConnectionConsumer

For application servers, Connections provide a special facility to create a ConnectionConsumer. A Destination and a Property Selector specify the messages that it is to consume. Also, a ConnectionConsumer must be given a ServerSessionPool to use to process its messages.

See also: QueueConnection, and TopicConnection.

Methods

close()

public void close() throws JMSException

Because a provider may allocate some resources outside of the JVM on behalf of a ConnectionConsumer, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Throws:

JMSException - if a JMS implementation fails to release resources on behalf of ConnectionConsumer, or if it fails to close the connection consumer.

getServerSessionPool()

public ServerSessionPool getServerSessionPool()

throws JMSException

Get the server session associated with this connection consumer.

Returns:

the server session pool used by this connection consumer.

Throws:

JMSException - if a JMS implementation fails to get the server session pool associated with this connection consumer because of an internal error.
ConnectionFactory

public interface ConnectionFactory
Subinterfaces: QueueConnectionFactory, TopicConnectionFactory, XAQueueConnectionFactory, and XATopicConnectionFactory

WebSphere MQ class: MQConnectionFactory

java.lang.Object
  +----com.ibm.mq.jms.MQConnectionFactory

A ConnectionFactory encapsulates a set of connection configuration parameters that has been defined by an administrator. A client uses it to create a Connection with a JMS provider.

WebSphere MQ Event Broker note
For direct connections to WebSphere MQ Event Broker, properties accessed by methods marked with a “§” are ignored.

See also: QueueConnectionFactory, TopicConnectionFactory, XAQueueConnectionFactory, and XATopicConnectionFactory

WebSphere MQ constructor

MQConnectionFactory
  public MQConnectionFactory()

Methods

getCCSID * §
  public int getCCSID()
  Get the character set of the queue manager.

getChannel * §
  public String getChannel()
  For client only, get the channel that was used.

getClientId *
  public String getClientId()
  Get the client Identifier that is used for all connections that are created using this ConnectionFactory.
ConnectionFactory

getDescription *
    public String getDescription()

Retrieve the object description.

getHostName *
    public String getHostName()

Retrieve the name of the host.

getMsgBatchSize *
    public int getMsgBatchSize()

Return current value of this property.

getPollingInterval *
    public int getPollingInterval()

Return current value of this property.

getPort *
    public int getPort()

For client connections or direct TCP/IP connection to WebSphere MQ
Event Broker, get the port number.

getQueueManager *
    public String getQueueManager()

Get the name of the queue manager.

getReceiveExit *
    public String getReceiveExit()

Get the name of the receive exit class.

getReceiveExitInit *
    public String getReceiveExitInit()

Get the initialization string that was passed to the receive exit class.

getSecurityExit *
    public String getSecurityExit()

Get the name of the security exit class.

getSecurityExitInit *
    public String getSecurityExitInit()

Get the security exit initialization string.
getSendExit * §

public String getSendExit()

Get the name of the send exit class.

getSendExitInit * §

public String getSendExitInit()

Get the send exit initialization string.

getSSLCertStores * §

public java.util.Collection getSSLCertStores()

Returns a Collection of CertStore objects. If setSSLCertStores() was used to set a Collection of CertStore objects, the value returned from getSSLCertStores() will be a copy of the original Collection. If setSSLCertStores() was used to set a String detailing a list of LDAP URIs, this method returns a Collection of CertStore objects representing the LDAP CRLs.

getSSLCertStoresAsString * §

public String getSSLCertStoresAsString()
throws JMSException

Returns the String of LDAP URIs, set with setSSLCertStores. Throws JMSException if a Collection of CertStores was set.

getSSLCipherSuite * §

public String getSSLCipherSuite()

Returns the CipherSuite used for SSL encryption.

getSSLPeerName * §

public String getSSLPeerName()

Returns the distinguished name pattern used to validate the queue manager.

getSSLSocketFactory * §

public javax.net.ssl.SSLSocketFactory getSSLSocketFactory()

Returns the SSLSocketFactory used with SSL encryption.

getSyncpointAllGets * §

public boolean getSyncpointAllGets()

Return current value of this property.

getTransportType *

public int getTransportType()

Retrieve the transport type.
ConnectionFactory

**getUseConnectionPooling** *§*

```java
public boolean getUseConnectionPooling()
```

Return current value of this property.

**setCCSID** *§*

```java
public void setCCSID(int x) throws JMSException
```

Set the character set to be used when connecting to the queue manager. See Table 13 on page 120 for a list of allowed values. We recommend that you use the default value (819) for most situations.

**setChannel** *§*

```java
public void setChannel(String x) throws JMSException
```

For client only, set the channel to use.

**setClientId** *

```java
public void setClientId(String x)
```

Sets the client Identifier to be used for all connections created using this Connection.

---

**WebSphere MQ Event Broker note**

This method always throws an IllegalStateException when you make a direct connection to WebSphere MQ Event Broker.

**setDescription** *

```java
public void setDescription(String x)
```

A short description of the object.

**setHostName** *

```java
public void setHostName(String hostname)
```

For client connections or direct TCP/IP connections to WebSphere MQ Event Broker, the name of the host to connect to.

**setMsgBatchSize** *§*

```java
public void setMsgBatchSize(int x)
```

Set the maximum number of messages to be taken at once when using asynchronous delivery.

**setPollingInterval** *§*

```java
public void setPollingInterval(int x)
```

Set the interval between scans of all receivers during asynchronous message delivery. The value is a number of milliseconds.
ConnectionFactory

setPort *

public void setPort(int port) throws JMSException

Set the port for a client connection or direct TCP/IP connection to WebSphere MQ Event Broker.

Parameters:
   port - the new value to use.

Throws:
   JMSException if the port is negative.

setQueueManager * §

public void setQueueManager(String x) throws JMSException

Set the name of the queue manager to connect to.

setReceiveExit * §

public void setReceiveExit(String receiveExit)

The name of a class that implements a receive exit.

setReceiveExitInit * §

public void setReceiveExitInit(String x)

Initialization string that is passed to the constructor of the receive exit class.

setSecurityExit * §

public void setSecurityExit(String securityExit)

The name of a class that implements a security exit.

setSecurityExitInit * §

public void setSecurityExitInit(String x)

Initialization string that is passed to the security exit constructor.

setSendExit * §

public void setSendExit(String sendExit)

The name of a class that implements a send exit.

setSendExitInit * §

public void setSendExitInit(String x)

Initialization string that is passed to the constructor of send exit.

setSSLCertStores * §

public void setSSLCertStores(java.util.Collection stores)

Provides a Collection of CertStore objects used for CRL checking. The certificate provided by the queue manager is checked against one of the CertStore objects contained within the Collection; if the certificate is found, the connection attempt will fail. At connect-time, each CertStore in the Collection is tried in turn until one is successfully used to verify the queue manager’s certificate. If set to null (default), no checking of the queue manager’s certificate is performed. This property is ignored if sslCipherSuite is null. Use of this property requires Java 2 v1.4.
If CertStores are specified using this method, the MQConnectionFactory cannot be bound into a JNDI namespace. Attempting to do so will result in an exception.

**Note:** To use a CertStore successfully with a CRL hosted on an LDAP server, make sure that your Java Software Development Kit (SDK) is compatible with the CRL. Some SDKs require that the CRL conforms to RFC 2587 which defines a schema for LDAP v2. Most LDAP v3 servers use RFC 2256 instead.

```java
public void setSSLCertStores(String storeSpec)
    throws JMSException
```

Specifies a list of LDAP servers used for CRL checking. This string must consist of a sequence of space-delimited LDAP URIs of the form `ldap://host[:port]`. If no port is specified, the LDAP default of 389 is assumed. The certificate provided by the queue manager is checked against one of the listed LDAP CRL servers; if found, the connection will fail. Each LDAP server is tried in turn until one is successfully used to verify the queue manager’s certificate. If set to null (default), no checking of the queue manager’s certificate is performed. Throws JMSException if the supplied list of LDAP URIs is not valid. This property is ignored if sslCipherSuite is null. Use of this property requires Java 2 v1.4.

**Note:** To use a CertStore successfully with a CRL hosted on an LDAP server, make sure that your Java Software Development Kit (SDK) is compatible with the CRL. Some SDKs require that the CRL conforms to RFC 2587 which defines a schema for LDAP v2. Most LDAP v3 servers use RFC 2256 instead.

```java
public void setSSLCipherSuite(String cipherSuite)
```

This should be set to the CipherSuite matching the CipherSpec set on the SVRCONN channel. If set to null (default), no SSL encryption is performed. See Appendix H, “SSL CipherSuites supported by WebSphere MQ” on page 425 for a list of CipherSuites and their associated CipherSpecs.

```java
public void setSSLPeerName(String peerName)
    throws JMSException
```

Sets sslPeerName to a distinguished name pattern. If sslCipherSuite is set, this variable can be used to ensure the correct queue manager is used. For a description of the format for this value, see “Using the distinguished name of the queue manager” on page 83. The distinguished name provided by the queue manager must match this pattern, otherwise the connection attempt will fail. If set to null (default), no checking of the queue manager’s DN is performed. Throws JMSException if the supplied pattern is not valid. This property is ignored if sslCipherSuite is null.
public void setSSLSocketFactory(javax.net.ssl.SSLSocketFactory sf)

Sets the SSLSocketFactory for use with SSL encryption. This can be used to
customize all aspects of SSL encryption. For more information on
constructing and customizing SSLSocketFactory instances, refer to your
JSSE provider’s documentation. If set to null (default), the JSSE default
SSLSocketFactory is used when SSL encryption is requested. This property
is ignored if sslCipherSuite is null.

If a custom SSLSocketFactory is specified, the MQConnectionFactory
cannot be bound into a JNDI namespace. Attempting to do so will result in
an exception.

public void setSyncpointAllGets(boolean x)

Choose whether or not all GET operations will be done within a syncpoint.
The default setting for this property is false. This allows GET operations
not under transaction management to perform more quickly.

public void setTransportType(int x) throws JMSException

Set the transport type to use. It can be one of:
JMSC.MQJMS_TP_BINDINGS_MQ
JMSC.MQJMS_TP_CLIENT_MQ_TCPIP
JMSC.MQJMS_TP_DIRECT_TCPIP

public void setUseConnectionPooling(boolean x)

Choose whether or not to use connection pooling.
public interface ConnectionMetaData

WebSphere MQ class: MQConnectionMetaData

java.lang.Object

ConnectionMetaData provides information that describes the connection.

WebSphere MQ constructor

MQConnectionMetaData

    public MQConnectionMetaData()

Methods

getJMSMajorVersion

    public int getJMSMajorVersion() throws JMSException

    Get the JMS major version number.

    Returns:
    the JMS major version number.

    Throws:
    JMSException - if an internal error occurs in JMS implementation during the metadata retrieval.

getJMSMinorVersion

    public int getJMSMinorVersion() throws JMSException

    Get the JMS minor version number.

    Returns:
    the JMS minor version number.

    Throws:
    JMSException - if an internal error occurs in JMS implementation during the metadata retrieval.

getJMSProviderName

    public java.lang.String getJMSProviderName() throws JMSException

    Get the JMS provider name.

    Returns:
    the JMS provider name.

    Throws:
    JMSException - if an internal error occurs in JMS implementation during the metadata retrieval.

getJMSVersion

    public java.lang.String getJMSVersion() throws JMSException

    Get the JMS version.
Returns:
the JMS version.

Throws:
JMSException - if an internal error occurs in JMS implementation during the metadata retrieval.

getJMSXPropertyNames
public java.util.Enumeration getJMSXPropertyNames() throws JMSException

Get an enumeration of the names of the JMSX Properties supported by this connection.

Returns:
an Enumeration of JMSX PropertyNames.

Throws:
JMSException - if an internal error occurs in JMS implementation during the property names retrieval.

getProviderMajorVersion
public int getProviderMajorVersion() throws JMSException

Get the JMS provider major version number.

Returns:
the JMS provider major version number.

Throws:
JMSException - if an internal error occurs in JMS implementation during the metadata retrieval.

getProviderMinorVersion
public int getProviderMinorVersion() throws JMSException

Get the JMS provider minor version number.

Returns:
the JMS provider minor version number.

Throws:
JMSException - if an internal error occurs in JMS implementation during the metadata retrieval.

getProviderVersion
public java.lang.String getProviderVersion() throws JMSException

Get the JMS provider version.

Returns:
the JMS provider version.

Throws:
JMSException - if an internal error occurs in JMS implementation during the metadata retrieval.

toString *
pUBLIC String toString()

Overrides:
toString in class Object.
public interface DeliveryMode

Delivery modes supported by JMS.

Fields

NON_PERSISTENT
    public static final int NON_PERSISTENT

    This is the lowest overhead delivery mode because it does not require that
    the message be logged to stable storage.

PERSISTENT
    public static final int PERSISTENT

    This mode instructs the JMS provider to log the message to stable storage
    as part of the client’s send operation.
public interface Destination

Subinterfaces: Queue, TemporaryQueue, TemporaryTopic, and Topic

WebSphere MQ class: MQDestination

java.lang.Object

+----com.ibm.mq.jms.MQDestination

The Destination object encapsulates provider-specific addresses.

See also: Queue, TemporaryQueue, TemporaryTopic, and Topic

WebSphere MQ constructors

MQDestination

public MQDestination()

Methods

getCCSID *

public int getCCSID()

Get the name of the character set that is used by this destination.

getDescription *

public String getDescription()

Get the description of the object.

getEncoding *

public int getEncoding()

Get the encoding that is used for this destination.

getExpiry *

public int getExpiry()

Get the value of the expiry for this destination.

getPersistence *

public int getPersistence()

Get the value of the persistence for this destination.

getPriority *

public int getPriority()

Get the override priority value.

getTargetClient *

public int getTargetClient()

Get JMS compliance indicator flag.

setCCSID *

public void setCCSID(int x) throws JMSException

Chapter 14. JMS interfaces and classes 293
Destination

Character set to be used to encode text strings in messages sent to this destination. See Table 13 on page 120 for a list of allowed values. The default value is 1208 (UTF8).

setDescription *

public void setDescription(String x)

A short description of the object.

setEncoding *

public void setEncoding(int x) throws JMSException

Specifies the encoding to be used for numeric fields in messages sent to this destination. See Table 13 on page 120 for a list of allowed values.

setExpiry *

public void setExpiry(int expiry) throws JMSException

Used to override the expiry of all messages sent to this destination.

setPersistence *

public void setPersistence(int persistence) throws JMSException

Used to override the persistence of all messages sent to this destination.

setPriority *

public void setPriority(int priority) throws JMSException

Used to override the priority of all messages sent to this destination.

setTargetClient *

public void setTargetClient(int targetClient) throws JMSException

Flag to indicate whether or not the remote application is JMS compliant.
public interface ExceptionListener

If a JMS provider detects a serious problem with a Connection, it will inform the Connection’s ExceptionListener if one has been registered. It does this by calling the listener’s onException() method, passing it a JMSException that describes the problem.

This allows a client to be asynchronously notified of a problem. Some Connections only consume messages so they would have no other way to learn their Connection has failed.

Exceptions are delivered when:
• There is a failure in receiving an asynchronous message
• A message throws a runtime exception

Methods

onException

public void onException(JMSException exception)

Notify user of a JMS exception.

Parameters:

exception - the JMS exception. These are exceptions that result from asynchronous message delivery. Typically, they indicate a problem with receiving a message from the queue manager, or possibly an internal error in the JMS implementation.
MapMessage

public interface MapMessage
extends Message

WebSphere MQ class: JMSMapMessage

java.lang.Object
   +----com.ibm.jms.JMSMessage
       +----com.ibm.jms.JMSMapMessage

A MapMessage is used to send a set of name-value pairs where names are Strings and values are Java primitive types. The entries can be accessed sequentially or randomly by name. The order of the entries is undefined.

See also: BytesMessage, Message, ObjectMessage, StreamMessage, and TextMessage

Methods

getBoolean

public boolean getBoolean(java.lang.String name)
   throws JMSException

Return the boolean value with the given name.

Parameters:
   name - the name of the boolean

Returns:
   the boolean value with the given name.

Throws:
   • JMSEception - if JMS fails to read the message because of an internal JMS error.
   • MessageFormatException - if this type conversion is not valid.

getByte

public byte getByte(java.lang.String name)
   throws JMSException

Return the byte value with the given name.

Parameters:
   name - the name of the byte.

Returns:
   the byte value with the given name.

Throws:
   • JMSEception - if JMS fails to read the message because of an internal JMS error.
   • MessageFormatException - if this type conversion is not valid.
MapMessage

getBytes

public byte[] getBytes(java.lang.String name) throws JMSException

Return the byte array value with the given name.

Parameters:
name - the name of the byte array.

Returns:
a copy of the byte array value with the given name. If there is no item by this name, a null value is returned.

Throws:
• JMSException - if JMS fails to read the message because of an internal JMS error.
• MessageFormatException - if this type conversion is not valid.

gerChar

public char getChar(java.lang.String name) throws JMSException

Return the Unicode character value with the given name.

Parameters:
name - the name of the Unicode character.

Returns:
the Unicode character value with the given name.

Throws:
• JMSException - if JMS fails to read the message because of an internal JMS error.
• MessageFormatException - if this type conversion is not valid.

gerDouble

public double getDouble(java.lang.String name) throws JMSException

Return the double value with the given name.

Parameters:
name - the name of the double.

Returns:
the double value with the given name.

Throws:
• JMSException - if JMS fails to read the message because of an internal JMS error.
• MessageFormatException - if this type conversion is not valid.
MapMessage

**getFloat**

```java
public float getFloat(java.lang.String name) throws JMSException
```

Return the float value with the given name.

**Parameters:**
- name - the name of the float.

**Returns:**
- the float value with the given name.

**Throws:**
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageFormatException - if this type conversion is not valid.

**getInt**

```java
public int getInt(java.lang.String name) throws JMSException
```

Return the integer value with the given name.

**Parameters:**
- name - the name of the integer.

**Returns:**
- the integer value with the given name.

**Throws:**
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageFormatException - if this type conversion is not valid.

**getLong**

```java
public long getLong(java.lang.String name) throws JMSException
```

Return the long value with the given name.

**Parameters:**
- name - the name of the long.

**Returns:**
- the long value with the given name.

**Throws:**
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageFormatException - if this type conversion is not valid.
MapMessage

getMapNames

public java.util.Enumeration getMapNames() throws JMSException

Return an Enumeration of all the Map message’s names.

Returns:
an enumeration of all the names in this Map message.

Throws:
JMSException - if JMS fails to read the message because of an internal JMS error.

getObject

public java.lang.Object getObject(java.lang.String name) throws JMSException

Return the Java object value with the given name. This method returns in object format, a value that has been stored in the Map either using the setObject method call, or the equivalent primitive set method.

Parameters:
name - the name of the Java object.

Returns:
a copy of the Java object value with the given name, in object format (if it is set as an int, then a Integer is returned). If there is no item by this name, a null value is returned.

Throws:
JMSException - if JMS fails to read the message because of an internal JMS error.

getShort

public short getShort(java.lang.String name) throws JMSException

Return the short value with the given name.

Parameters:
name - the name of the short.

Returns:
the short value with the given name.

Throws:
JMSException - if JMS fails to read the message because of an internal JMS error.

getString

public java.lang.String getString(java.lang.String name) throws JMSException

Return the String value with the given name.

Parameters:
name - the name of the String.

Returns:
the String value with the given name. If there is no item by this name, a null value is returned.

Throws:
MapMessage

- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageFormatException - if this type conversion is not valid.

**itemExists**

```java
public boolean itemExists(java.lang.String name)
throws JMSException
```

Check if an item exists in this MapMessage.

**Parameters:**
- name - the name of the item to test.

**Returns:**
- true if the item does exist.

**Throws:**
- JMSException - if a JMS error occurs.

**setBoolean**

```java
public void setBoolean(java.lang.String name, boolean value)
throws JMSException
```

Set a boolean value with the given name into the Map.

**Parameters:**
- name - the name of the boolean.
- value - the boolean value to set in the Map.

**Throws:**
- JMSException - if JMS fails to write message due to some internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

**setByte**

```java
public void setByte(java.lang.String name, byte value)
throws JMSException
```

Set a byte value with the given name into the Map.

**Parameters:**
- name - the name of the byte.
- value - the byte value to set in the Map.

**Throws:**
- JMSException - if JMS fails to write message due to some internal JMS error
- MessageNotWriteableException - if the message is in read-only mode.
MapMessage

setBytes

public void setBytes(java.lang.String name, byte[] value) throws JMSException

Set a byte array value with the given name into the Map.

Parameters:

- name - the name of the byte array.
- value - the byte array value to set in the Map.

The array is copied, so the value in the map is not altered by subsequent modifications to the array.

Throws:

- JMSException - if JMS fails to write message due to some internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

setBytes

public void setBytes(java.lang.String name, byte[] value, int offset, int length) throws JMSException

Set a portion of the byte array value with the given name into the Map.

The array is copied, so the value in the map is not altered by subsequent modifications to the array.

Parameters:

- name - the name of the byte array.
- value - the byte array value to set in the Map.
- offset - the initial offset within the byte array.
- length - the number of bytes to be copied.

Throws:

- JMSException - if JMS fails to write message due to some internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

setChar

public void setChar(java.lang.String name, char value) throws JMSException

Set a Unicode character value with the given name into the Map.

Parameters:

- name - the name of the Unicode character.
- value - the Unicode character value to set in the Map.

Throws:

- JMSException - if JMS fails to write message due to some internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.
MapMessage

**setDouble**

```java
public void setDouble(java.lang.String name,
                      double value) throws JMSException
```

Set a double value with the given name into the Map.

**Parameters:**
- name - the name of the double.
- value - the double value to set in the Map.

**Throws:**
- JMSException - if JMS fails to write message due to some internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

**setFloat**

```java
public void setFloat(java.lang.String name,
                     float value) throws JMSException
```

Set a float value with the given name into the Map.

**Parameters:**
- name - the name of the float.
- value - the float value to set in the Map.

**Throws:**
- JMSException - if JMS fails to write message due to some internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

**setInt**

```java
public void setInt(java.lang.String name,
                   int value) throws JMSException
```

Set an integer value with the given name into the Map.

**Parameters:**
- name - the name of the integer.
- value - the integer value to set in the Map.

**Throws:**
- JMSException - if JMS fails to write message due to some internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.
setLong

```java
public void setLong(java.lang.String name,
                      long value) throws JMSException
```

Set a long value with the given name into the Map.

**Parameters:**
- name - the name of the long.
- value - the long value to set in the Map.

**Throws:**
- JMSException - if JMS fails to write message due to some internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

setObject

```java
public void setObject(java.lang.String name,
                       java.lang.Object value) throws JMSException
```

Set a Java object value with the given name into the Map. This method only works for object primitive types (Integer, Double, Long, for example), Strings and byte arrays.

**Parameters:**
- name - the name of the Java object.
- value - the Java object value to set in the Map.

**Throws:**
- JMSException - if JMS fails to write message due to some internal JMS error.
- MessageFormatException - if object is not valid.
- MessageNotWriteableException - if the message is in read-only mode.

setShort

```java
public void setShort(java.lang.String name,
                      short value) throws JMSException
```

Set a short value with the given name into the Map.

**Parameters:**
- name - the name of the short.
- value - the short value to set in the Map.

**Throws:**
- JMSException - if JMS fails to write message due to some internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.
MapMessage

**setString**

```java
public void setString(java.lang.String name,
                    java.lang.String value)
  throws JMSException
```

Set a String value with the given name into the Map.

**Parameters:**

- name - the name of the String.
- value - the String value to set in the Map.

**Throws:**

- JMSException - if JMS fails to write message due to some internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.
public interface Message

WebSphere MQ class: JMSMessage

java.lang.Object
  +-----com.ibm.jms.MQJMSMessage

The Message interface is the root interface of all JMS messages. It defines the JMS header and the acknowledge method used for all messages.

Fields

DEFAULT_DELIVERY_MODE
  public static final int DEFAULT_DELIVERY_MODE

The default delivery mode value.

DEFAULT_PRIORITY
  public static final int DEFAULT_PRIORITY

The default priority value.

DEFAULT_TIME_TO_LIVE
  public static final long DEFAULT_TIME_TO_LIVE

The default time to live value.

Methods

acknowledge
  public void acknowledge() throws JMSException

Acknowledge this and all previous messages received by the session.

Throws:
  JMSException - if JMS fails to acknowledge because of an internal JMS error.

clearBody
  public void clearBody() throws JMSException

Clear out the message body. All other parts of the message are left untouched.

Throws:
  JMSException - if JMS fails to because of an internal JMS error.
clearProperties

public void clearProperties() throws JMSException

Clear a message’s properties. The header fields and message body are not cleared.

Throws:
   JMSException - if JMS fails to clear JMS message properties because of an internal JMS error.

getBooleanProperty

public boolean getBooleanProperty(java.lang.String name) throws JMSException

Return the boolean property value with the given name.

Parameters:
   name - the name of the boolean property.

Returns:
   the boolean property value with the given name.

Throws:
   • JMSException - if JMS fails to get the property because of an internal JMS error.
   • MessageFormatException - if this type conversion is not valid.

getByteProperty

public byte getByteProperty(java.lang.String name) throws JMSException

Return the byte property value with the given name.

Parameters:
   name - the name of the byte property.

Returns:
   the byte property value with the given name.

Throws:
   • JMSException - if JMS fails to get the property because of an internal JMS error.
   • MessageFormatException - if this type conversion is not valid.

getDoubleProperty

public double getDoubleProperty(java.lang.String name) throws JMSException

Return the double property value with the given name.

Parameters:
   name - the name of the double property.

Returns:
   the double property value with the given name.

Throws:
   • JMSException - if JMS fails to get the property because of an internal JMS error.
   • MessageFormatException - if this type conversion is not valid.
getFloatProperty

```java
public float getFloatProperty(java.lang.String name)
    throws JMSException
```

Return the float property value with the given name.

**Parameters:**
name - the name of the float property.

**Returns:**
the float property value with the given name.

**Throws:**
- JMSException - if JMS fails to get the property because of an internal JMS error.
- MessageFormatException - if this type conversion is not valid.

getIntProperty

```java
public int getIntProperty(java.lang.String name)
    throws JMSException
```

Return the integer property value with the given name.

**Parameters:**
name - the name of the integer property.

**Returns:**
the integer property value with the given name.

**Throws:**
- JMSException - if JMS fails to get the property because of an internal JMS error.
- MessageFormatException - if this type conversion is not valid.

getJMSCorrelationID

```java
public java.lang.String getJMSCorrelationID()
    throws JMSException
```

Get the correlation ID for the message.

**Returns:**
the correlation ID of a message as a String.

**Throws:**
JMSException - if JMS fails to get the correlation ID because of an internal JMS error.

**See also:**
setJMSCorrelationID(), getJMSCorrelationIDAsBytes(), setJMSCorrelationIDAsBytes()
**Message**

*getJMSCorrelationIDAsBytes*

```java
public byte[] getJMSCorrelationIDAsBytes()
```

Get the correlation ID as an array of bytes for the message.

**Returns:**
the correlation ID of a message as an array of bytes.

**Throws:**
JMSException - if JMS fails to get correlation ID because of an internal JMS error.

**See also:**
setJMSCorrelationID(), getJMSCorrelationID(), setJMSCorrelationIDAsBytes()

*getJMSDeliveryMode*

```java
public int getJMSDeliveryMode()
```

Get the delivery mode for this message.

**Returns:**
the delivery mode of this message.

**Throws:**
JMSException - if JMS fails to get JMS DeliveryMode because of an internal JMS error.

**See also:**
setJMSDeliveryMode(), DeliveryMode

*getJMSDestination*

```java
public Destination getJMSDestination()
```

Get the destination for this message.

**Returns:**
the destination of this message.

**Throws:**
JMSException - if JMS fails to get JMS Destination because of an internal JMS error.

**See also:**
setJMSDestination()

*getJMSExpiration*

```java
public long getJMSExpiration()
```

Get the message’s expiration value.

**Returns:**
the time the message expires. It is the sum of the time-to-live value specified by the client, and the GMT at the time of the send.

**Throws:**
JMSException - if JMS fails to get JMS message expiration because of an internal JMS error.

**See also:**
setJMSExpiration()
getJMSMessageID

public java.lang.String getJMSMessageID() throws JMSException

Get the message ID.

Returns:
the message ID.

Throws:
JMSException - if JMS fails to get the message ID because of an internal JMS error.

See also:
setJMSMessageID()

getJMSPriority

public int getJMSPriority() throws JMSException

Get the message priority.

Returns:
the message priority.

Throws:
JMSException - if JMS fails to get JMS message priority because of an internal JMS error.

See also:
setJMSPriority() for priority levels

getJMSRedelivered

public boolean getJMSRedelivered() throws JMSException

Get an indication of whether this message is being redelivered.

If a client receives a message with the redelivered indicator set, it is likely, but not guaranteed, that this message was delivered to the client earlier but the client did not acknowledge its receipt at that earlier time.

Returns:
set to true if this message is being redelivered.

Throws:
JMSException - if JMS fails to get JMS Redelivered flag because of an internal JMS error.

See also:
setJMSRedelivered()

getJMSReplyTo

public Destination getJMSReplyTo() throws JMSException

Get where a reply to this message should be sent.

Returns:
where to send a response to this message

Throws:
JMSException - if JMS fails to get ReplyTo Destination because of an internal JMS error.
Message

See also:
setJMSReplyTo()

getJMSTimestamp
public long getJMSTimestamp() throws JMSException

Get the message timestamp.

Returns:
the message timestamp.

Throws:
JMSException - if JMS fails to get the Timestamp because of an internal JMS error.

See also:
setJMSTimestamp()

getJMSType
public java.lang.String getJMSType() throws JMSException

Get the message type.

Returns:
the message type.

Throws:
JMSException - if JMS fails to get JMS message type because of an internal JMS error.

See also:
setJMSType()

getLongProperty
public long getLongProperty(java.lang.String name) throws JMSException

Return the long property value with the given name.

Parameters:
name - the name of the long property.

Returns:
the long property value with the given name.

Throws:
- JMSException - if JMS fails to get the property because of an internal JMS error.
- MessageFormatException - if this type conversion is not valid.

ggetObjectProperty
public java.lang.Object getObjectProperty(java.lang.String name) throws JMSException

Return the Java object property value with the given name.

Parameters:
name - the name of the Java object property.

Returns:
the Java object property value with the given name, in object format (for example, if it set as an int, an Integer is returned). If there is no property by this name, a null value is returned.
Throws:
   JMSException - if JMS fails to get the property because of an internal JMS error.

getPropertyNames
   public java.util.Enumeration getPropertyNames()
   throws JMSException

Return an Enumeration of all the property names.

Returns:
   an enumeration of all the names of property values.

Throws:
   JMSException - if JMS fails to get the property names because of an internal JMS error.

getShortProperty
   public short getShortProperty(java.lang.String name)
   throws JMSException

Return the short property value with the given name.

Parameters:
   name - the name of the short property.

Returns:
   the short property value with the given name.

Throws:
   • JMSException - if JMS fails to get the property because of an internal JMS error.
   • MessageFormatException - if this type conversion is not valid.

getStringProperty
   public java.lang.String getStringProperty(java.lang.String name)
   throws JMSException

Return the String property value with the given name.

Parameters:
   name - the name of the String property

Returns:
   the String property value with the given name. If there is no property by this name, a null value is returned.

Throws:
   • JMSException - if JMS fails to get the property because of an internal JMS error.
   • MessageFormatException - if this type conversion is not valid.
propertyExists

public boolean propertyExists(java.lang.String name)
    throws JMSException

Check if a property value exists.

Parameters:
    name - the name of the property to test.

Returns:
    true if the property does exist.

Throws:
    JMSException - if JMS fails to check whether a property exists because of an internal JMS error.

setBooleanProperty

public void setBooleanProperty(java.lang.String name,
                      boolean value) throws JMSException

Set a boolean property value with the given name into the Message.

Parameters:
    • name - the name of the boolean property.
    • value - the boolean property value to set in the Message.

Throws:
    • JMSException - if JMS fails to set Property because of an internal JMS error.
    • MessageNotWriteableException - if the properties are read-only.

setByteProperty

public void setByteProperty(java.lang.String name,
                      byte value) throws JMSException

Set a byte property value with the given name into the Message.

Parameters:
    • name - the name of the byte property.
    • value - the byte property value to set in the Message.

Throws:
    • JMSException - if JMS fails to set Property because of an internal JMS error.
    • MessageNotWriteableException - if the properties are read-only.

setDoubleProperty

public void setDoubleProperty(java.lang.String name,
                      double value) throws JMSException

Set a double property value with the given name into the Message.

Parameters:
    • name - the name of the double property.
    • value - the double property value to set in the Message.
setFloatProperty

```java
public void setFloatProperty(java.lang.String name, float value) throws JMSException
```

Set a float property value with the given name into the Message.

**Parameters:**
- name - the name of the float property.
- value - the float property value to set in the Message.

**Throws:**
- JMSExcetion - if JMS fails to set the property because of an internal JMS error.
- MessageNotWriteableException - if the properties are read-only.

setIntProperty

```java
public void setIntProperty(java.lang.String name, int value) throws JMSException
```

Set an integer property value with the given name into the Message.

**Parameters:**
- name - the name of the integer property.
- value - the integer property value to set in the Message.

**Throws:**
- JMSExcetion - if JMS fails to set the property because of an internal JMS error.
- MessageNotWriteableException - if the properties are read-only.

setJMSCorrelationID

```java
public void setJMSCorrelationID(java.lang.String correlationID) throws JMSException
```

Set the correlation ID for the message.

A client can use the JMSCorrelationID header field to link one message with another. A typical use is to link a response message with its request message.

**Note:** The use of a byte[] value for JMSCorrelationID is non-portable.

**Parameters:**
- correlationID - the message ID of a message being referred to.

**Throws:**
- JMSExcetion - if JMS fails to set the correlation ID because of an internal JMS error.

**See also:**
- getJMSCorrelationID(), getJMSCorrelationIDAsBytes(), setJMSCorrelationIDAsBytes()
setJMSCorrelationIDAsBytes

```java
public void setJMSCorrelationIDAsBytes(byte[] correlationID) throws JMSException
```

Set the correlation ID as an array of bytes for the message. A client can use this call to set the correlationID equal either to a messageID from a previous message, or to an application-specific string. Application-specific strings must not start with the characters ID:

**Parameters:**
- correlationID - the correlation ID as a string, or the message ID of a message being referred to.

**Throws:**
- JMSException - if JMS fails to set the correlation ID because of an internal JMS error.

**See also:**
- setJMSCorrelationID(), getJMSCorrelationID(), getJMSCorrelationIDAsBytes()

setJMSDeliveryMode

```java
public void setJMSDeliveryMode(int deliveryMode) throws JMSException
```

Set the delivery mode for this message.

Any value set using this method is ignored when the message is sent, but this method can be used to change the value in a received message.

To alter the delivery mode when a message is sent, use the setDeliveryMode method on the QueueSender or TopicPublisher (this method is inherited from MessageProducer).

**Parameters:**
- deliveryMode - the delivery mode for this message.

**Throws:**
- JMSException - if JMS fails to set JMS DeliveryMode because of an internal JMS error.

**See also:**
- getJMSDeliveryMode(), DeliveryMode

setJMSDestination

```java
public void setJMSDestination(Destination destination) throws JMSException
```

Set the destination for this message.

Any value set using this method is ignored when the message is sent, but this method can be used to change the value in a received message.

**Parameters:**
- destination - the destination for this message.

**Throws:**
- JMSException - if JMS fails to set JMS Destination because of an internal JMS error.
See also:
getJMSDestination()

setJMSExpiration

```java
public void setJMSExpiration(long expiration)
    throws JMSException
```

Set the message’s expiration value.

Any value set using this method is ignored when the message is sent, but this method can be used to change the value in a received message.

Parameters:
expiration - the message’s expiration time.

Throws:
JMSException - if JMS fails to set JMS message expiration because of an internal JMS error.

See also:
getJMSExpiration()

setJMSMessageID

```java
public void setJMSMessageID(java.lang.String id)
    throws JMSException
```

Set the message ID.

Any value set using this method is ignored when the message is sent, but this method can be used to change the value in a received message.

Parameters:
id - the ID of the message.

Throws:
JMSException - if JMS fails to set the message ID because of an internal JMS error.

See also:
getJMSMessageID()

setJMSPriority

```java
public void setJMSPriority(int priority)
    throws JMSException
```

Set the priority for this message.

JMS defines a ten level priority value, with 0 as the lowest priority, and 9 as the highest. In addition, clients should consider priorities 0-4 as gradations of normal priority, and priorities 5-9 as gradations of expedited priority.

Parameters:
priority - the priority of this message.

Throws:
JMSException - if JMS fails to set JMS message priority because of an internal JMS error.

See also:
getJMSPriority()
Message

**setJMSRedelivered**

```java
public void setJMSRedelivered(boolean redelivered)
throws JMSException
```

Set to indicate whether this message is being redelivered.

Any value set using this method is ignored when the message is sent, but this method can be used to change the value in a received message.

**Parameters:**
- `redelivered` - an indication of whether this message is being redelivered.

**Throws:**
- `JMSException` - if JMS fails to set JMSRedelivered flag because of an internal JMS error.

**See also:**
- `getJMSRedelivered()`

**setJMSReplyTo**

```java
public void setJMSReplyTo(Destination replyTo)
throws JMSException
```

Set where a reply to this message should be sent.

**Parameters:**
- `replyTo` - where to send a response to this message. A null value indicates that no reply is expected.

**Throws:**
- `JMSException` - if JMS fails to set ReplyTo Destination because of an internal JMS error.

**See also:**
- `getJMSReplyTo()`

**setJMSTimestamp**

```java
public void setJMSTimestamp(long timestamp)
throws JMSException
```

Set the message timestamp.

Any value set using this method is ignored when the message is sent, but this method can be used to change the value in a received message.

**Parameters:**
- `timestamp` - the timestamp for this message.

**Throws:**
- `JMSException` - if JMS fails to set the timestamp because of an internal JMS error.

**See also:**
- `getJMSTimestamp()`
setJMSType

```java
public void setJMSType(java.lang.String type)
    throws JMSException
```

Set the message type.

JMS clients should assign a value to type whether the application makes
use of it or not. This ensures that it is properly set for those providers that
require it.

**Parameters:**
- `type` - the class of message.

**Throws:**
- `JMSException` - if JMS fails to set JMS message type because of an
  internal JMS error.

**See also:**
- `getJMSType()`

setLongProperty

```java
public void setLongProperty(java.lang.String name, long value)
    throws JMSException
```

Set a long property value with the given name into the Message.

**Parameters:**
- `name` - the name of the long property.
- `value` - the long property value to set in the Message.

**Throws:**
- `JMSException` - if JMS fails to set Property because of an internal
  JMS error.
  - `MessageNotWriteableException` - if the properties are read-only.

setObjectProperty

```java
public void setObjectProperty(java.lang.String name, java.lang.Object value)
    throws JMSException
```

Set a property value with the given name into the Message.

**Parameters:**
- `name` - the name of the Java object property.
- `value` - the Java object property value to set in the Message.

**Throws:**
- `JMSException` - if JMS fails to set Property because of an internal
  JMS error.
  - `MessageFormatException` - if the object is not valid.
  - `MessageNotWriteableException` - if the properties are read-only.
Message

**setShortProperty**

```java
public void setShortProperty(java.lang.String name,
                             short value) throws JMSException
```

Set a short property value with the given name into the Message.

**Parameters:**

- name - the name of the short property.
- value - the short property value to set in the Message.

**Throws:**

- JMSException - if JMS fails to set Property because of an internal JMS error.
- MessageNotWriteableException - if the properties are read-only.

**setStringProperty**

```java
public void setStringProperty(java.lang.String name,
                               java.lang.String value) throws JMSException
```

Set a String property value with the given name into the Message.

**Parameters:**

- name - the name of the String property.
- value - the String property value to set in the Message.

**Throws:**

- JMSException - if JMS fails to set the property because of an internal JMS error.
- MessageNotWriteableException - if the properties are read-only.
MessageConsumer

public interface MessageConsumer
Subinterfaces: QueueReceiver and TopicSubscriber

WebSphere MQ class: MQMessageConsumer

java.lang.Object
| +----com.ibm.mq.jms.MQMessageConsumer

The parent interface for all message consumers. A client uses a message consumer to receive messages from a Destination.

Methods

getMessageListener

public MessageListener getMessageListener()
throws JMSException

Get the message consumer’s MessageListener.

Returns:
the listener for the message consumer, or null if a listener is not set.

Throws:
JMSException - if JMS fails to get the message listener because of a JMS error.

See also:
setMessageListener

getMessageSelector

public java.lang.String getMessageSelector()
throws JMSException

Get this message consumer’s message selector expression.

Returns:
this message consumer’s message selector.

Throws:
JMSException - if JMS fails to get the message selector because of a JMS error.

receive

public Message receive() throws JMSException

Receive the next message produced for this message consumer.

Returns:
the next message produced for this message consumer.

Throws:
JMSException - if JMS fails to receive the next message because of an error.
MessageConsumer

receive
public Message receive(long timeOut) throws JMSException

Receive the next message that arrives within the specified timeout interval. A timeout value of zero causes the call to wait indefinitely until a message arrives.

Parameters:
timeout - the timeout value (in milliseconds).

Returns:
the next message produced for this message consumer, or null if one is not available.

Throws:
JMSException - if JMS fails to receive the next message because of an error.

receiveNoWait
public Message receiveNoWait() throws JMSException

Receive the next message if one is immediately available.

Returns:
the next message produced for this message consumer, or null if one is not available.

Throws:
JMSException - if JMS fails to receive the next message because of an error.

setMessageListener
public void setMessageListener(MessageListener listener) throws JMSException

Set the message consumer's MessageListener.

Parameters:
messageListener - the messages are delivered to this listener.

Throws:
JMSException - if JMS fails to set message listener because of a JMS error.

See also:
getMessageListener

close
public void close() throws JMSException

Because a provider may allocate some resources outside of the JVM on behalf of a MessageConsumer, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

This call blocks until a receive or message listener in progress has completed.

Throws:
JMSException - if JMS fails to close the consumer because of an error.
public interface MessageListener

A MessageListener is used to receive asynchronously delivered messages.

Methods

onMessage

public void onMessage(Message message)

Pass a message to the Listener.

Parameters:
message - the message passed to the listener.

See also
Session.setMessageListener
MessageProducer

public interface MessageProducer
Subinterfaces: QueueSender and TopicPublisher

WebSphere MQ class: MQMessageProducer

java.lang.Object
   +----com.ibm.mq.jms.MQMessageProducer

A client uses a message producer to send messages to a Destination.

WebSphere MQ constructors

MQMessageProducer
   public MQMessageProducer()

Methods

close
   public void close() throws JMSException

Because a provider may allocate some resources outside of the JVM on behalf of a MessageProducer, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Throws:
   JMSException - if JMS fails to close the producer because of an error.

getDeliveryMode
   public int getDeliveryMode() throws JMSException

Get the producer’s default delivery mode.

Returns:
   the message delivery mode for this message producer.

Throws:
   JMSException - if JMS fails to get the delivery mode because of an internal error.

See also:
   setDeliveryMode

gGetDisableMessageID
   public boolean getDisableMessageID() throws JMSException

Get an indication of whether message IDs are disabled.

Returns:
   an indication of whether message IDs are disabled.

Throws:
   JMSException - if JMS fails to get the disabled message ID because of an internal error.
getDisableMessageTimestamp

public boolean getDisableMessageTimestamp() throws JMSException

Get an indication of whether message timestamps are disabled.

Returns:
an indication of whether message IDs are disabled.

Throws:
JMSException - if JMS fails to get the disabled message timestamp because of an internal error.

getPriority

public int getPriority() throws JMSException

Get the producer’s default priority.

Returns:
the message priority for this message producer.

Throws:
JMSException - if JMS fails to get the priority because of an internal error.

See also:
setPriority

getimeToLive

public long getTimeToLive() throws JMSException

Get the default length of time in milliseconds from its dispatch time that a produced message should be retained by the message system.

Returns:
the message time to live in milliseconds; zero is unlimited.

Throws:
JMSException - if JMS fails to get the Time to Live because of an internal error.

See also:
setTimeToLive

setDeliveryMode

public void setDeliveryMode(int deliveryMode) throws JMSException

Set the producer's default delivery mode.

Delivery mode is set to DeliveryMode.PERSISTENT by default.

Parameters:
deliveryMode - the message delivery mode for this message producer.

Throws:
JMSException - if JMS fails to set the delivery mode because of an internal error.

See also:
getDeliveryMode, DeliveryMode.NON_PERSISTENT, DeliveryMode.PERSISTENT
MessageProducer

**setDisableMessageID**

A method to set whether message IDs are disabled.

```java
public void setDisableMessageID(boolean value) throws JMSException
```

Set whether message IDs are disabled. Message IDs are enabled by default.

**Note:** This method is ignored in the WebSphere MQ classes for Java Message Service implementation.

**Parameters:**

- value - indicates whether message IDs are disabled.

**Throws:**

- JMSException - if JMS fails to set the disabled message ID because of an internal error.

**setDisableMessageTimestamp**

A method to set whether message timestamps are disabled.

```java
public void setDisableMessageTimestamp(boolean value) throws JMSException
```

Set whether message timestamps are disabled. Message timestamps are enabled by default.

**Note:** This method is ignored in the WebSphere MQ classes for Java Message Service implementation.

**Parameters:**

- value - indicates whether message timestamps are disabled.

**Throws:**

- JMSException - if JMS fails to set the disabled message timestamp because of an internal error.

**setPriority**

A method to set the producer's default priority.

```java
public void setPriority(int priority) throws JMSException
```

Set the producer’s default priority. Priority is set to 4, by default.

**Parameters:**

- priority - the message priority for this message producer.

**Throws:**

- JMSException - if JMS fails to set the priority because of an internal error.

**See also:**

- getPriority
setTimeToLive

public void setTimeToLive(long timeToLive)
    throws JMSException

Set the default length of time, in milliseconds from its dispatch time, that a produced message should be retained by the message system.

Time to live is set to zero by default.

---

**WebSphere MQ Event Broker note**

This method throws a JMSException if set to other than 0 when you make a direct connection to WebSphere MQ Event Broker.

---

**Parameters:**

timeToLive - the message time to live in milliseconds; zero is unlimited.

**Throws:**

JMSException - if JMS fails to set the Time to Live because of an internal error.

**See also:**

g getTimeToLive
public class MQQueueEnumeration extends Object implements Enumeration

Enumeration of messages on a queue. This class is not defined in the JMS specification, it is created by calling the getEnumeration method of MQQueueBrowser. The class contains a base MQQueue instance to hold the browse cursor. The queue is closed once the cursor has moved off the end of the queue.

There is no way to reset an instance of this class - it acts as a 'one-shot' mechanism.

See also: MQQueueBrowser

Methods

hasMoreElements

public boolean hasMoreElements()

Indicate whether another message can be returned.

nextElement

public Object nextElement() throws NoSuchElementException

Return the current message.

If hasMoreElements() returns 'true', nextElement() always returns a message. It is possible for the returned message to pass its expiry date between the hasMoreElements() and the nextElement calls.
public interface ObjectMessage
extends Message

WebSphere MQ class: JMSObjectMessage

java.lang.Object
|-- com.ibm.jms.JMSMessage
|   |-- com.ibm.jms.JMSObjectMessage

An ObjectMessage is used to send a message that contains a serializable Java object. It inherits from Message and adds a body containing a single Java reference. Only Serializable Java objects can be used.

See also: BytesMessage, MapMessage, Message, StreamMessage and TextMessage

Methods

getObject

public java.io.Serializable getObject()
   throws JMSException

Get the serializable object containing this message’s data. The default value is null.

Returns:
   the serializable object containing this message’s data.

Throws:
   • JMSException - if JMS fails to get the object because of an internal JMS error.
   • MessageFormatException - if object deserialization fails.

setObject

public void setObject(java.io.Serializable object)
   throws JMSException

Set the serializable object containing this message’s data. The ObjectMessage contains a snapshot of the object at the time setObject() is called. Subsequent modifications of the object have no effect on the ObjectMessage body.

Parameters:
   object - the message’s data.

Throws:
   • JMSException - if JMS fails to set the object because of an internal JMS error.
   • MessageFormatException - if object serialization fails.
   • MessageNotWriteableException - if the message is in read-only mode.
Queue

public interface Queue
extends Destination
Subinterfaces: TemporaryQueue

WebSphere MQ class: MQQueue

java.lang.Object
| +----com.ibm.mq.jms.MQDestination
| +----com.ibm.mq.jms.MQQueue

A Queue object encapsulates a provider-specific queue name. It is the way a client specifies the identity of a queue to JMS methods.

WebSphere MQ constructors

MQQueue *
public MQQueue()

Default constructor for use by the administration tool.

MQQueue *
public MQQueue(String URIqueue)

Create a new MQQueue instance. The string takes a URI format, as described on page [194].

MQQueue *
public MQQueue(String queueManagerName,
String queueName)

Methods

getBaseQueueManagerName *
public String getBaseQueueManagerName()

Returns:
the value of the WebSphere MQ Queue manager name.

getBaseQueueName *
public String getBaseQueueName()

Returns:
the value of the WebSphere MQ Queue name.

getQueueName
public java.lang.String getQueueName()
throws JMSException

Get the name of this queue.

Clients that depend upon the name are not portable.

Returns:
the queue name
Throws:
JMSEException - if JMS implementation for Queue fails to return the queue name because of an internal error.

generateReference *
public Reference getReference() throws NamingException

Create a reference for this queue.

Returns:
a reference for this object

Throws:
   NamingException

setBaseQueueManagerName *
public void setBaseQueueManagerName(String x) throws JMSException

Set the value of the WebSphere MQ queue manager name.

Note: This method should only be used by the administration tool.

setBaseQueueName *
public void setBaseQueueName(String x) throws JMSException

Set the value of the WebSphere MQ queue name.

Note: This method should only be used by the administration tool. It makes no attempt to decode queue:qmgr:queue format strings.

toString
public java.lang.String toString()

Return a pretty printed version of the queue name.

Returns:
the provider-specific identity values for this queue.

Overrides:
   toString in class java.lang.Object
public interface QueueBrowser

WebSphere MQ class: MQQueueBrowser

download lang.Object

A client uses a QueueBrowser to look at messages on a queue without removing them.

Note: The WebSphere MQ class MQQueueEnumeration is used to hold the browse cursor.

See also: QueueReceiver

Methods

close

public void close() throws JMSException

Because a provider may allocate some resources outside of the JVM on behalf of a QueueBrowser, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Throws:

JMSException - if a JMS fails to close this Browser because of a JMS error.

getEnumeration

public java.util.Enumeration getEnumeration() throws JMSException

Get an enumeration for browsing the current queue messages in the order that they would be received.

Returns:

an enumeration for browsing the messages.

Throws:

JMSException - if JMS fails to get the enumeration for this browser because of a JMS error.

Note: If the browser is created for a nonexistent queue, this is not detected until the first call to getEnumeration.

getMessageSelector

public java.lang.String getMessageSelector() throws JMSException

Get this queue browser’s message selector expression.

Returns:

this queue browser’s message selector.

Throws:

JMSException - if JMS fails to get the message selector for this browser because of a JMS error.
getQueue

public Queue getQueue() throws JMSException

Get the queue associated with this queue browser.

Returns:
   the queue.

Throws:
   JMSException - if JMS fails to get the queue associated with this Browser because of a JMS error.
QueueConnection

public interface QueueConnection
extends Connection
Subinterfaces: XAQueueConnection

WebSphere MQ class: MQQueueConnection

java.lang.Object
+-com.ibm.mq.jms.MQConnection
  +-com.ibm.mq.jms.MQQueueConnection

A QueueConnection is an active connection to a JMS point-to-point provider. A client uses a QueueConnection to create one or more QueueSessions for producing and consuming messages.

See also: Connection, QueueConnectionFactory, and XAQueueConnection

Methods

close *

   public void close() throws JMSException

   Overrides:
       close in class MQConnection.

createConnectionConsumer

   public ConnectionConsumer createConnectionConsumer
       (Queue queue,
        java.lang.String messageSelector,
        ServerSessionPool sessionPool,
        int maxMessages)
       throws JMSException

Create a connection consumer for this connection. This is an expert facility that is not used by regular JMS clients.

Parameters:

   • queue - the queue to access.
   • messageSelector - only messages with properties that match the message selector expression are delivered.
   • sessionPool - the server session pool to associate with this connection consumer.
   • maxMessages - the maximum number of messages that can be assigned to a server session at one time.
Returns:
the connection consumer.

Throws:
- JMSException - if the JMS Connection fails to create a connection consumer because of an internal error, or incorrect arguments for sessionPool and messageSelector.
- InvalidSelectorException - if the message selector is not valid.

See Also:
ConnectionConsumer

createQueueSession

public QueueSession createQueueSession(boolean transacted,
                                     int acknowledgeMode)
                                throws JMSException

Create a QueueSession.

Parameters:
- transacted - if true, the session is transacted.
- acknowledgeMode - indicates whether the consumer or the client will acknowledge any messages it receives. Possible values are:
  - Session.AUTO_ACKNOWLEDGE
  - Session.CLIENT_ACKNOWLEDGE
  - Session.DUPS_OK_ACKNOWLEDGE

This parameter is ignored if the session is transacted.

Returns:
a newly created queue session.

Throws:
JMSException - if JMS Connection fails to create a session because of an internal error, or lack of support for specific transaction and acknowledgement mode.
public interface QueueConnectionFactory
extends ConnectionFactory
Subinterfaces: XAQueueConnectionFactory

WebSphere MQ class: MQQueueConnectionFactory

java.lang.Object
 +----com.ibm.mq.jms.MQConnectionFactory
        +----com.ibm.mq.jms.MQQueueConnectionFactory

A client uses a QueueConnectionFactory to create QueueConnections with a JMS point-to-point provider.

See also: ConnectionFactory and XAQueueConnectionFactory

WebSphere MQ constructor

MQQueueConnectionFactory
  public MQQueueConnectionFactory()

Methods

createQueueConnection
  public QueueConnection createQueueConnection()
          throws JMSException

Create a queue connection with default user identity. The connection is created in stopped mode. No messages will be delivered until Connection.start method is explicitly called.

Returns:
  a newly created queue connection.

Throws:
  • JMSException - if JMS Provider fails to create Queue Connection because of an internal error.
  • JMSSecurityException - if client authentication fails because of a non valid user name or password.

createQueueConnection
  public QueueConnection createQueueConnection
          (java.lang.String userName,
           java.lang.String password)
          throws JMSException

Create a queue connection with specified user identity.

Note: This method can be used only with transport type JMSC.MQJMS_TP_CLIENT_MQ_TCPPIP (see ConnectionFactory). The connection is created in stopped mode. No messages will be delivered until Connection.start method is explicitly called.
Parameters:
- userName - the caller's user name.
- password - the caller's password.

Returns:
a newly created queue connection.

Throws:
- JMSException - if JMS Provider fails to create Queue Connection because of an internal error.
- JMSSecurityException - if client authentication fails because of a non valid user name or password.

getMessageRetention*
public int getMessageRetention()

Get method for messageRetention attribute.

Returns:
- JMSC.MQJMS_MRET_YES - unwanted messages remain on the input queue.
- JMSC.MQJMS_MRET_NO - unwanted messages are dealt with according to their disposition options.

getReference *
pubuc Reference getRefence() throws NamingException

Create a reference for this queue connection factory.

Returns:
a reference for this object.

Throws:
NamingException.

getTemporaryModel *
pubuc String getTemporaryModel()

setMessageRetention*
pubuc void setMessageRetention(int x) throws JMSException

Set method for messageRetention attribute.

Parameters:
Valid values are:
- JMSC.MQJMS_MRET_YES - unwanted messages remain on the input queue.
- JMSC.MQJMS_MRET_NO - unwanted messages are dealt with according to their disposition options.

setTemporaryModel *
pubuc void setTemporaryModel(String x) throws JMSException
QueueReceiver

public interface QueueReceiver
extends MessageConsumer

WebSphere MQ class: MQQueueReceiver

java.lang.Object
  +----com.ibm.mq.jms.MQMessageConsumer
  |    +----com.ibm.mq.jms.MQQueueReceiver

A client uses a QueueReceiver for receiving messages that have been delivered to a queue.

See also: MessageConsumer

This class inherits the following methods from MQMessageConsumer.

  • receive
  • receiveNoWait
  • close
  • getMessageListener
  • setMessageListener

Methods

getQueue

public Queue getQueue() throws JMSException

Get the queue associated with this queue receiver.

Returns:
  the queue.

Throws:
  JMSException - if JMS fails to get queue for this queue receiver because of an internal error.
public class QueueRequestor
extends java.lang.Object

java.lang.Object
    | ^----javax.jms.QueueRequestor

JMS provides this QueueRequestor helper class to simplify making service requests. The QueueRequestor constructor is given a non-transacted QueueSession and a destination Queue. It creates a TemporaryQueue for the responses, and provides a request() method that sends the request message and waits for its reply. Users are free to create more sophisticated versions.

See also: TopicRequestor

Constructors

QueueRequestor
public QueueRequestor(QueueSession session,
                        Queue queue)
                throws JMSException

This implementation assumes that the session parameter is non-transacted and either AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE.

Parameters:
• session - the queue session the queue belongs to.
• queue - the queue to perform the request/reply call on.

Throws:
    JMSException - if a JMS error occurs.

Methods

close
public void close() throws JMSException

Because a provider may allocate some resources outside of the JVM on behalf of a QueueRequestor, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Note: This method closes the Session object passed to the QueueRequestor constructor.

    Throws:
        JMSException - if a JMS error occurs.

request
public Message request(Message message)
                throws JMSException

Send a request and wait for a reply. The temporary queue is used for replyTo, and only one reply per request is expected.
QueueRequestor

Parameters:
message - the message to send.

Returns:
the reply message.

Throws:
JMSException - if a JMS error occurs.
QueueSender

public interface QueueSender
extends MessageProducer

WebSphere MQ class: MQQueueSender

java.lang.Object
    +----com.ibm.mq.jms.MQMessageProducer
        +----com.ibm.mq.jms.MQQueueSender

A client uses a QueueSender to send messages to a queue.

A QueueSender is normally associated with a particular Queue. However, it is possible to create an unidentified QueueSender that is not associated with any given Queue.

See also: MessageProducer

Methods

close *

public void close() throws JMSException

Because a provider may allocate some resources outside of the JVM on behalf of a QueueSender, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Throws:
JMSException if JMS fails to close the producer due to some error.

Overrides:
close in class MQMessageProducer.

getQueue

public Queue getQueue() throws JMSException

Get the queue associated with this queue sender.

Returns:
the queue.

Throws:
JMSException - if JMS fails to get the queue for this queue sender because of an internal error.
QueueSender

public void send(Message message) throws JMSException

Send a message to the queue. Use the QueueSender’s default delivery mode, time to live, and priority.

Parameters:
   message - the message to be sent.

Throws:
   • JMSException - if JMS fails to send the message because of an error.
   • MessageFormatException - if a non valid message is specified.
   • InvalidDestinationException - if a client uses this method with a Queue sender with a non valid queue.

public void send(Message message, int deliveryMode, int priority, long timeToLive) throws JMSException

Send a message specifying delivery mode, priority, and time to live to the queue.

Parameters:
   • message - the message to be sent.
   • deliveryMode - the delivery mode to use.
   • priority - the priority for this message.
   • timeToLive - the message’s lifetime (in milliseconds).

Throws:
   • JMSException - if JMS fails to send the message because of an internal error.
   • MessageFormatException - if a non valid message is specified.
   • InvalidDestinationException - if a client uses this method with a Queue sender with a non valid queue.

public void send(Queue queue, Message message) throws JMSException

Send a message to the specified queue with the QueueSender’s default delivery mode, time to live, and priority.

Note: This method can only be used with unidentified QueueSenders.

Parameters:
   • queue - the queue that this message should be sent to.
   • message - the message to be sent.

Throws:
   • JMSException - if JMS fails to send the message because of an internal error.
   • MessageFormatException - if a non valid message is specified.
send

public void send(Queue queue,
                 Message message,
                 int deliveryMode,
                 int priority,
                 long timeToLive) throws JMSException

Send a message to the specified queue with delivery mode, priority, and time to live.

**Note:** This method can only be used with unidentified QueueSenders.

**Parameters:**
- queue - the queue that this message should be sent to.
- message - the message to be sent.
- deliveryMode - the delivery mode to use.
- priority - the priority for this message.
- timeToLive - the message’s lifetime (in milliseconds).

**Throws:**
- JMSException - if JMS fails to send the message because of an internal error.
- MessageFormatException - if a non valid message is specified.
- InvalidDestinationException - if a client uses this method with a non valid queue.
QueueSession

public interface QueueSession
extends Session

WebSphere MQ class: MQQueueSession

java.lang.Object
   +----com.ibm.mq.jms.MQSession
   +----com.ibm.mq.jms.MQQueueSession

A QueueSession provides methods to create QueueReceivers, QueueSenders, QueueBrowsers and TemporaryQueues.

See also: Session

The following methods are inherited from MQSession:

- close
- commit
- rollback
- recover

Methods

createBrowser

public QueueBrowser createBrowser(Queue queue)
throws JMSException

Create a QueueBrowser to peek at the messages on the specified queue.

Parameters:

- queue - the queue to access.

Throws:

- JMSException - if a session fails to create a browser because of a JMS error.
- InvalidDestinationException - if a non valid Queue is specified.

createBrowser

public QueueBrowser createBrowser(Queue queue,
   java.lang.String messageSelector)
throws JMSException

Create a QueueBrowser to peek at the messages on the specified queue.

Parameters:

- queue - the queue to access.
- messageSelector - only messages with properties that match the message selector expression are delivered.

Throws:

- JMSException - if a session fails to create a browser because of a JMS error.
- InvalidDestinationException - if a non valid Queue is specified.
- InvalidSelectorException - if the message selector is not valid.
createQueue

```java
public Queue createQueue(java.lang.String queueName)
    throws JMSException
```

Create a Queue given a Queue name. This allows the creation of a queue with a provider specific name. The string takes a URI format, as described on page [194].

**Note:** Clients that depend on this ability are not portable.

**Parameters:**
- `queueName` - the name of this queue.

**Returns:**
a Queue with the given name.

**Throws:**
- JMSException - if a session fails to create a queue because of a JMS error.

createReceiver

```java
public QueueReceiver createReceiver(Queue queue)
    throws JMSException
```

Create a QueueReceiver to receive messages from the specified queue.

**Parameters:**
- `queue` - the queue to access.

**Throws:**
- JMSException - if a session fails to create a receiver because of a JMS error.
- InvalidDestinationException - if a non valid Queue is specified.

createReceiver

```java
public QueueReceiver createReceiver(Queue queue,
    java.lang.String messageSelector)
    throws JMSException
```

Create a QueueReceiver to receive messages from the specified queue.

**Parameters:**
- `queue` - the queue to access.
- `messageSelector` - only messages with properties that match the message selector expression are delivered.

**Throws:**
- JMSException - if a session fails to create a receiver because of a JMS error.
- InvalidDestinationException - if a non valid Queue is specified.
- InvalidSelectorException - if the message selector is not valid.

create Sender

```java
public QueueSender createSender(Queue queue)
    throws JMSException
```

Create a QueueSender to send messages to the specified queue.
QueueSession

Parameters:
queue - the queue to access, or null if this is to be an unidentified producer.

Throws:
- JMSException - if a session fails to create a sender because of a JMS error.
- InvalidDestinationException - if a non valid Queue is specified.

createTemporaryQueue

public TemporaryQueue createTemporaryQueue()

throws JMSException

Create a temporary queue. Its lifetime will be that of the QueueConnection unless deleted earlier.

Returns:
a temporary queue.

Throws:
JMSException - if a session fails to create a Temporary Queue because of a JMS error.
public interface Session
extends java.langRunnable
Subinterfaces: QueueSession, TopicSession, XAQueueSession, XASession, and XATopicSession

WebSphere MQ class: MQSession

java.lang.Object
 | +-----com.ibm.mq.jms.MQSession

A JMS Session is a single threaded context for producing and consuming messages.

See also: QueueSession, TopicSession, XAQueueSession, XASession, and XATopicSession

Fields

AUTO_ACKNOWLEDGE
public static final int AUTO_ACKNOWLEDGE

With this acknowledgement mode, the session automatically acknowledges a message when it has either successfully returned from a call to receive, or the message listener it has called to process the message successfully returns.

CLIENT_ACKNOWLEDGE
public static final int CLIENT_ACKNOWLEDGE

With this acknowledgement mode, the client acknowledges a message by calling a message’s acknowledge method.

DUPS_OK_ACKNOWLEDGE
public static final int DUPS_OK_ACKNOWLEDGE

This acknowledgement mode instructs the session to lazily acknowledge the delivery of messages.

Methods

close
public void close() throws JMSException

Because a provider may allocate some resources outside of the JVM on behalf of a Session, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Closing a transacted session rolls back any in-progress transaction. Closing a session automatically closes its message producers and consumer, so there is no need to close them individually.

Throws:
JMSException - if the JMS implementation fails to close a Session because of an internal error.
commit

```
public void commit() throws JMSException
```

Commit all messages done in this transaction and release any locks currently held.

WebSphere MQ Event Broker note

This always throws a JMSException when you have a direct connection to WebSphere MQ Event Broker.

Throws:

- JMSException - if JMS implementation fails to commit the transaction because of an internal error.
- TransactionRolledBackException - if the transaction gets rolled back because of an internal error during commit.

createBytesMessage

```
public BytesMessage createBytesMessage() throws JMSException
```

Create a BytesMessage. A BytesMessage is used to send a message containing a stream of uninterpreted bytes.

Throws:

- JMSException - if JMS fails to create this message because of an internal error.

createMapMessage

```
public MapMessage createMapMessage() throws JMSException
```

Create a MapMessage. A MapMessage is used to send a self-defining set of name-value pairs, where names are Strings, and values are Java primitive types.

Throws:

- JMSException - if JMS fails to create this message because of an internal error.

createMessage

```
public Message createMessage() throws JMSException
```

Create a Message. The Message interface is the root interface of all JMS messages. It holds all the standard message header information. It can be sent when a message containing only header information is sufficient.

Throws:

- JMSException - if JMS fails to create this message because of an internal error.
createObjectMessage

public ObjectMessage createObjectMessage()
    throws JMSException

Create an ObjectMessage. An ObjectMessage is used to send a message that contains a serializable Java object.

Throws:
    JMSException - if JMS fails to create this message because of an internal error.

createObjectMessage

public ObjectMessage createObjectMessage(java.io.Serializable object)
    throws JMSException

Create an initialized ObjectMessage. An ObjectMessage is used to send a message that contains a serializable Java object.

Parameters:
    object - the object to use to initialize this message.

Throws:
    JMSException - if JMS fails to create this message because of an internal error.

createStreamMessage

public StreamMessage createStreamMessage()
    throws JMSException

Create a StreamMessage. A StreamMessage is used to send a self-defining stream of Java primitives.

Throws:
    JMSException if JMS fails to create this message because of an internal error.

createTextMessage

public TextMessage createTextMessage()
    throws JMSException

Create a TextMessage. A TextMessage is used to send a message containing a String.

Throws:
    JMSException - if JMS fails to create this message because of an internal error.

createTextMessage

public TextMessage createTextMessage(java.lang.String string)
    throws JMSException

Create an initialized TextMessage. A TextMessage is used to send a message containing a String.

Parameters:
    string - the string used to initialize this message.

Throws:
    JMSException - if JMS fails to create this message because of an internal error.
**Session**

**getMessageListener**

```java
public MessageListener getMessageListener()
    throws JMSException
```

Return the session’s distinguished message listener.

**Returns:**
the message listener associated with this session.

**Throws:**
JMSException - if JMS fails to get the message listener because of an internal error in the JMS Provider.

**See also:**
setMessageListener

**getTransacted**

```java
public boolean getTransacted() throws JMSException
```

Is the session in transacted mode?

**WebSphere MQ Event Broker note**
This method always returns false when you have a direct connection to WebSphere MQ Event Broker.

**Returns:**
true if the session is in transacted mode.

**Throws:**
JMSException - if JMS fails to return the transaction mode because of an internal error in JMS Provider.

**recover**

```java
public void recover() throws JMSException
```

Stop message delivery in this session, and restart sending messages with the oldest unacknowledged message.

**WebSphere MQ Event Broker note**
This always throws a JMSException when you have a direct connection to WebSphere MQ Event Broker.

**Throws:**
JMSException - if the JMS implementation fails to stop message delivery and restart message send because of an internal error.
rollback

public void rollback() throws JMSException

Roll back any messages done in this transaction and release any locks currently held.

WebSphere MQ Event Broker note

This always throws a JMSException when you have a direct connection to WebSphere MQ Event Broker.

Throws:

JMSException - if the JMS implementation fails to roll back the transaction because of an internal error.

run

public void run()

This method is intended for use only by application servers.

WebSphere MQ Event Broker note

This always throws an IllegalStateException when you have a direct connection to WebSphere MQ Event Broker.

Specified by:

run in the interface java.lang.Runnable

See also:

ServerSession

setMessageListener

public void setMessageListener(MessageListener listener) throws JMSException

Set the session’s distinguished message listener. When it is set, no other form of message receipt in the session can be used. However, all forms of sending messages are still supported.

This is an expert facility that is not used by regular JMS clients.

Parameters:

listener - the message listener to associate with this session.

Throws:

JMSException - if JMS fails to set the message listener because of an internal error in the JMS Provider.

See also:

getMessageListener, ServerSessionPool, ServerSession
StreamMessage

public interface StreamMessage
extends Message

WebSphere MQ class: JMSStreamMessage

java.lang.Object
   +----com.ibm.jms.JMSMessage
         +----com.ibm.jms.JMSStreamMessage

A StreamMessage is used to send a stream of Java primitives.

See also: BytesMessage, MapMessage, Message, ObjectMessage and TextMessage

Methods

readBoolean

public boolean readBoolean() throws JMSException

Read a boolean from the stream message.

Returns:
   the boolean value read.

Throws:
   • JMSException - if JMS fails to read the message because of an internal JMS error.
   • MessageEOFException - if an end of message stream is received.
   • MessageFormatException - if this type conversion is not valid.
   • MessageNotReadableException - if the message is in write-only mode.

readByte

public byte readByte() throws JMSException

Read a byte value from the stream message.

Returns:
   the next byte from the stream message as an 8-bit byte.

Throws:
   • JMSException - if JMS fails to read the message because of an internal JMS error.
   • MessageEOFException - if an end of message stream is received.
   • MessageFormatException - if this type conversion is not valid.
   • MessageNotReadableException - if the message is in write-only mode.
readBytes

```java
class public int readBytes(byte[] value) throws JMSException

Read a byte array field from the stream message into the specified byte[] object (the read buffer). If the buffer size is less than, or equal to, the size of the data in the message field, an application must make further calls to this method to retrieve the remainder of the data. Once the first readBytes call on a byte[] field value has been done, the full value of the field must be read before it is valid to read the next field. An attempt to read the next field before that has been done will throw a MessageFormatException.

Parameters:
value - the buffer into which the data is read.

Returns:
the total number of bytes read into the buffer, or -1 if there is no more data because the end of the byte field has been reached.

Throws:
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if an end of message stream is received.
- MessageFormatException - if this type conversion is not valid.
- MessageNotReadableException - if the message is in write-only mode.
```

readChar

```java
public char readChar() throws JMSException

Read a Unicode character value from the stream message.

Returns:
a Unicode character from the stream message.

Throws:
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if an end of message stream is received.
- MessageFormatException if this type conversion is not valid.
- MessageNotReadableException if the message is in write-only mode.
```

readDouble

```java
public double readDouble() throws JMSException

Read a double from the stream message.

Returns:
a double value from the stream message.

Throws:
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if an end of message stream is received.
- MessageFormatException - if this type conversion is not valid.
StreamMessage

- MessageNotReadableException - if the message is in write-only mode.

readFloat

public float readFloat() throws JMSException

Read a float from the stream message.

Returns:
- a float value from the stream message.

Throws:
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if an end of message stream
- MessageFormatException if this type conversion is not valid.
- MessageNotReadableException - if the message is in write-only mode.

readInt

public int readInt() throws JMSException

Read a 32-bit integer from the stream message.

Returns:
- a 32-bit integer value from the stream message, interpreted as an int.

Throws:
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if an end of message stream is received.
- MessageFormatException if this type conversion is not valid.
- MessageNotReadableException if the message is in write-only mode.

readLong

public long readLong() throws JMSException

Read a 64-bit integer from the stream message.

Returns:
- a 64-bit integer value from the stream message, interpreted as a long.

Throws:
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if an end of message stream
- MessageFormatException if this type conversion is not valid.
- MessageNotReadableException if the message is in write-only mode.
readObject

```java
public java.lang.Object readObject() throws JMSException
```

Read a Java object from the stream message.

**Returns:**
- a Java object from the stream message in object format (for example, if it was set as an int, an `Integer` is returned).

**Throws:**
- `JMSException` - if JMS fails to read the message because of an internal JMS error.
- `MessageEOFException` - if an end of message stream is received.
- `MessageFormatException` - if this type conversion is not valid.
- `MessageNotReadableException` - if the message is in write-only mode.

readShort

```java
public short readShort() throws JMSException
```

Read a 16-bit number from the stream message.

**Returns:**
- a 16-bit number from the stream message.

**Throws:**
- `JMSException` - if JMS fails to read the message because of an internal JMS error.
- `MessageEOFException` - if an end of message stream is received.
- `MessageFormatException` - if this type conversion is not valid.
- `MessageNotReadableException` - if the message is in write-only mode.

readString

```java
public java.lang.String readString() throws JMSException
```

Read in a string from the stream message.

**Returns:**
- a Unicode string from the stream message.

**Throws:**
- `JMSException` - if JMS fails to read the message because of an internal JMS error.
- `MessageEOFException` - if an end of message stream is received.
- `MessageFormatException` - if this type conversion is not valid.
- `MessageNotReadableException` - if the message is in write-only mode.
StreamMessage

reset

public void reset() throws JMSException

Put the message in read-only mode, and reposition the stream to the beginning.

Throws:
- JMSException - if JMS fails to reset the message because of an internal JMS error.
- MessageFormatException - if the message has an not valid format.

writeBoolean

public void writeBoolean(boolean value) throws JMSException

Write a boolean to the stream message.

Parameters:
- value - the boolean value to be written.

Throws:
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

writeByte

public void writeByte(byte value) throws JMSException

Write out a byte to the stream message.

Parameters:
- value - the byte value to be written.

Throws:
- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

writeBytes

public void writeBytes(byte[] value) throws JMSException

Write a byte array to the stream message.

Parameters:
- value - the byte array to be written.

Throws:
- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.
**writeBytes**

```java
public void writeBytes(byte[] value, int offset, int length) throws JMSException
```

Write a portion of a byte array to the stream message.

**Parameters:**
- value - the byte array value to be written.
- offset - the initial offset within the byte array.
- length - the number of bytes to use.

**Throws:**
- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

**writeChar**

```java
public void writeChar(char value) throws JMSException
```

Write a char to the stream message.

**Parameters:**
- value - the char value to be written.

**Throws:**
- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

**writeDouble**

```java
public void writeDouble(double value) throws JMSException
```

Write a double to the stream message.

**Parameters:**
- value - the double value to be written.

**Throws:**
- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

**writeFloat**

```java
public void writeFloat(float value) throws JMSException
```

Write a float to the stream message.

**Parameters:**
- value - the float value to be written.

**Throws:**
- JMSException - if JMS fails to write the message because of an internal JMS error.
StreamMessage

- MessageNotWriteableException - if the message is in read-only mode.

writeInt

public void writeInt(int value) throws JMSException

Write an int to the stream message.

Parameters:
value - the int to be written.

Throws:
- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

writeLong

public void writeLong(long value) throws JMSException

Write a long to the stream message.

Parameters:
value - the long to be written.

Throws:
- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

writeObject

public void writeObject(java.lang.Object value) throws JMSException

Write a Java object to the stream message. This method only works for object primitive types (Integer, Double, Long, for example), Strings, and byte arrays.

Parameters:
value - the Java object to be written.

Throws:
- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.
- MessageFormatException - if the object is not valid.
writeShort

public void writeShort(short value) throws JMSException

Write a short to the stream message.

Parameters:
value - the short to be written.

Throws:
• JMSException - if JMS fails to write the message because of an internal JMS error.
• MessageNotWriteableException - if the message is in read-only mode.

writeString

public void writeString(java.lang.String value) throws JMSException

Write a string to the stream message.

Parameters:
value - the String value to be written.

Throws:
• JMSException - if JMS fails to write the message because of an internal JMS error.
• MessageNotWriteableException - if the message is in read-only mode.
TemporaryQueue

public interface TemporaryQueue
extends Queue

WebSphere MQ class: MQTemporaryQueue

java.lang.Object
  +----com.ibm.mq.jms.MQDestination
    +----com.ibm.mq.jms.MQQueue
      +----com.ibm.mq.jms.MQTemporaryQueue

A TemporaryQueue is a unique Queue object that is created for the duration of a QueueConnection.

Methods

delete
debut

public void delete() throws JMSException

Delete this temporary queue. If there are still existing senders or receivers using it, a JMSException will be thrown.

Throws:
  JMSException - if JMS implementation fails to delete a TemporaryQueue because of an internal error.
public interface TemporaryTopic
extends Topic

WebSphere MQ class: MQTemporaryTopic

java.lang.Object
    +----com.ibm.mq.jms.MQDestination
          +----com.ibm.mq.jms.MQTopic
                  +----com.ibm.mq.jms.MQTemporaryTopic

A TemporaryTopic is a unique Topic object created for the duration of a TopicConnection and can only be consumed by consumers of that connection.

WebSphere MQ constructor

MQTemporaryTopic
MQTemporaryTopic() throws JMSException

Methods

delete
    public void delete() throws JMSException

Delete this temporary topic. If there are still existing publishers or subscribers still using it, a JMSException will be thrown.

Throws:
    JMSException - if JMS implementation fails to delete a TemporaryTopic because of an internal error.
public interface TextMessage extends Message

WebSphere MQ class: JMSTextMessage

TextMessage is used to send a message containing a java.lang.String. It inherits from Message and adds a text message body.

See also: BytesMessage, MapMessage, Message, ObjectMessage and StreamMessage

Methods

getText

public java.lang.String getText() throws JMSException

Get the string containing this message’s data. The default value is null.

Returns:
the String containing the message’s data.

Throws:
JMSException - if JMS fails to get the text because of an internal JMS error.

setText

public void setText(java.lang.String string) throws JMSException

Set the string containing this message’s data.

Parameters:
string - the String containing the message’s data.

Throws:
JMSException - if JMS fails to set text because of an internal JMS error.
MessageNotWriteableException - if the message is in read-only mode.
public interface Topic
extends Destination
Subinterfaces: TemporaryTopic

WebSphere MQ class: MQTopic

java.lang.Object
   +----com.ibm.mq.jms.MQDestination
       +----com.ibm.mq.jms.MQTopic

A Topic object encapsulates a provider-specific topic name. It is the way a client specifies the identity of a topic to JMS methods.

---

WebSphere MQ Event Broker note

For direct connections to WebSphere MQ Event Broker, properties accessed by methods marked with a “§” are ignored.

See also: Destination

WebSphere MQ constructor

MQTopic
   public MQTopic()
   public MQTopic(string URItopic)

See TopicSession.createTopic.

Methods

getBaseTopicName *
   public String getBaseTopicName()
   get method for the underlying WebSphere MQ topic name.

getBrokerCCDurSubQueue * §
   public String getBrokerCCDurSubQueue()
   Get method for brokerCCDurSubQueue attribute.

   Returns:
   the name of the durable subscription queue (the brokerCCDurSubQueue) to use for a ConnectionConsumer.
getBrokerDurSubQueue

public String getBrokerDurSubQueue()

Get method for brokerDurSubQueue attribute.

Returns:
the name of the durable subscription queue (the brokerDurSubQueue) to use.

getAddressReference

public Reference getReference()

Create a reference for this topic.

Returns:
a reference for this object.

Throws:
  NamingException.

getTopicName

public java.lang.String getTopicName() throws JMSException

Get the name of this topic in URI format. (URI format is described in “Creating topics at runtime” on page 212)

Note: Clients that depend upon the name are not portable.

Returns:
the topic name.

Throws:
  JMSException - if JMS implementation for Topic fails to return the topic name because of an internal error.

setBaseTopicName

public void setBaseTopicName(String x)

set method for the underlying WebSphere MQ topic name.

setBrokerCCDurSubQueue

public void setBrokerCCDurSubQueue(String x) throws JMSException

Set method for brokerCCDurSubQueue attribute.

Parameters:
  brokerCCDurSubQueue - the name of the durable subscription queue to use for a ConnectionConsumer.

setBrokerDurSubQueue

public void setBrokerDurSubQueue(String x) throws JMSException

Set method for brokerDurSubQueue attribute.

Parameters:
  brokerDurSubQueue - the name of the durable subscription queue to use.
**toString**

```java
public String toString()
```

Return a pretty printed version of the Topic name.

**Returns:**
the provider specific identity values for this Topic.

**Overrides:**
toString in class Object.
public interface TopicConnection extends Connection
Subinterfaces: XATopicConnection

WebSphere MQ class: MQTopicConnection

A TopicConnection is an active connection to a JMS Publish/Subscribe provider.

See also: Connection, TopicConnectionFactory, and XATopicConnection

Methods

createConnectionConsumer

public ConnectionConsumer createConnectionConsumer
    (Topic topic,
    java.lang.String messageSelector,
    ServerSessionPool sessionPool,
    int maxMessages)
    throws JMSException

Create a connection consumer for this connection. This is an expert facility
that is not used by regular JMS clients.

WebSphere MQ Event Broker note

For a direct connection to WebSphere MQ Event Broker, if transact is
true then this method throws a JMSException.

Parameters:

• topic - the topic to access.
• messageSelector - only messages with properties that match the
  message selector expression are delivered.
• sessionPool - the server session pool to associate with this
  connection consumer.
• maxMessages - the maximum number of messages that can be
  assigned to a server session at one time.

Returns:

the connection consumer.

Throws:

• JMSException - if the JMS Connection fails to create a connection
  consumer because of an internal error, or because of incorrect
  arguments for sessionPool.
• InvalidSelectorException - if the message selector is not valid.

See also:

ConnectionConsumer
createDurableConnectionConsumer

```java
public ConnectionConsumer createDurableConnectionConsumer
    (Topic topic,
     java.lang.String subscriptionName,
     java.lang.String messageSelector,
     ServerSessionPool sessionPool,
     int maxMessages)
    throws JMSException
```

Create a durable connection consumer for this connection. This is an expert facility that is not used by regular JMS clients.

WebSphere MQ Event Broker note

For a direct connection to WebSphere MQ Event Broker, this method throws a JMSException.

Parameters:

- topic - the topic to access.
- subscriptionName - name of the durable subscription.
- messageSelector - only messages with properties that match the message selector expression are delivered.
- sessionPool - the server session pool to associate with this durable connection consumer.
- maxMessages - the maximum number of messages that can be assigned to a server session at one time.

Returns:

the durable connection consumer.

Throws:

- JMSException - if the JMS Connection fails to create a connection consumer because of an internal error, or because of incorrect arguments for sessionPool and messageSelector.
- InvalidSelectorException - if the message selector is not valid.

See also:

ConnectionConsumer

createTopicSession

```java
public TopicSession createTopicSession
    (boolean transacted,
     int acknowledgeMode)
    throws JMSException
```

Create a TopicSession.

WebSphere MQ Event Broker note

For a direct connection to WebSphere MQ Event Broker, if transact is true then this method throws a JMSException.

Parameters:

- transacted - if true, the session is transacted.
- acknowledgeMode - one of:
  - Session.AUTO_ACKNOWLEDGE
  - Session.CLIENT_ACKNOWLEDGE
Session.DUPS_OK_ACKNOWLEDGE

Indicates whether the consumer or the client will acknowledge any messages it receives. This parameter will be ignored if the session is transacted.

Returns:
a newly created topic session.

Throws:
JMSException - if JMS Connection fails to create a session because of an internal error, or a lack of support for the specific transaction and acknowledgement mode.
public interface TopicConnectionFactory
extends ConnectionFactory
Subinterfaces: XATopicConnectionFactory

WebSphere MQ class: MQTopicConnectionFactory

java.lang.Object
   +----com.ibm.mq.jms.MQConnectionFactory
   |    +----com.ibm.mq.jms.MQTopicConnectionFactory

A client uses a TopicConnectionFactory to create TopicConnections with a JMS Publish/Subscribe provider.

---

**WebSphere MQ Event Broker note**

For direct connections to WebSphere MQ Event Broker, properties accessed by methods marked with a “§” are ignored.

---

See also: ConnectionFactory and XATopicConnectionFactory

---

**WebSphere MQ constructor**

MQTopicConnectionFactory

class MQTopicConnectionFactory
public MQTopicConnectionFactory()

---

**Methods**

createTopicConnection

class TopicConnection
class TopicConnectionFactory
public TopicConnection createTopicConnection() throws JMSException

Create a topic connection with default user identity. The connection is created in stopped mode. No messages will be delivered until Connection.start method is explicitly called.

Returns:
a newly created topic connection.

Throws:

- JMSException - if JMS Provider fails to create a Topic Connection because of an internal error.
- JMSSecurityException - if client authentication fails because of a non valid user name or password.
createTopicConnection

public TopicConnection createTopicConnection
(java.lang.String userName,
java.lang.String password)
throws JMSException

Create a topic connection with specified user identity. The connection is
created in stopped mode. No messages will be delivered until
Connection.start method is explicitly called.

Parameters:
• userName - the caller’s user name.
• password - the caller’s password.

Returns:
a newly created topic connection.

Throws:
• JMSEception - if JMS Provider fails to create a Topic Connection
  because of an internal error.
• JMSSecurityException - if client authentication fails because of a
  non valid user name or password.

Note: This method is valid only for transport type
IBM_JMS_TP_CLIENT_MQ_TCPIP. See ConnectionFactory.

getBrokerCCSubQueue *

public String getBrokerCCSubQueue()

Get method for brokerCCSubQueue attribute.

Returns:
the name of the non-durable subscription queue to use for a
ConnectionConsumer.

getBrokerControlQueue *

public String getBrokerControlQueue()

Get method for brokerControlQueue attribute.

Returns:
the broker’s control queue name

getBrokerPubQueue *

public String getBrokerPubQueue()

Get method for brokerPubQueue attribute.

Returns:
the broker’s publish queue name.
getBrokerQueueManager * §
public String getBrokerQueueManager()

Get method for brokerQueueManager attribute.

Returns:
the broker’s queue manager name.

getBrokerSubQueue * §
public String getBrokerSubQueue()

Get method for brokerSubQueue attribute.

Returns:
the name of the non-durable subscription queue to use.

getBrokerVersion *
public int getBrokerVersion()

Get method for brokerVersion attribute.

Returns:
the broker’s version number

getCleanupInterval * §
public long getCleanupInterval()

Get method for cleanupInterval attribute.

Returns:
how often the cleanup utility will run, in milliseconds

getCleanupLevel * §
public int getCleanupLevel()

Get method for cleanupLevel attribute.

Returns:
the value of cleanupLevel

getPubAckInterval * §
public int getPubAckInterval()

Get method for pubAckInterval attribute.

Returns:
the interval, in number of messages, between publish requests that require acknowledgement from the broker.

ggetReference *
public Reference getReference()

Return a reference for this topic connection factory.

Returns:
a reference for this topic connection factory.

Throws:
NamingException.
TopicConnectionFactory

getStatusRefreshInterval * §
public int getStatusRefreshInterval()

Get method for statusRefreshInterval attribute.

Returns:
the number of milliseconds between transactions to refresh
publish/subscribe status.

getSubscriptionStore * §
public int getSubscriptionStore()

Get method for SUBSTORE property.

Returns:
an integer representing the current SUBSTORE property type.

setBrokerCCSubQueue * §
public void setBrokerCCSubQueue(String x) throws JMSException

Set method for brokerCCSubQueue attribute.

Parameters:
brokerSubQueue - the name of the non-durable subscription queue
to use for a ConnectionConsumer.

setBrokerControlQueue * §
public void setBrokerControlQueue(String x) throws JMSException

Set method for brokerControlQueue attribute.

Parameters:
brokerControlQueue - the name of the broker control queue.

setBrokerPubQueue * §
public void setBrokerPubQueue(String x) throws JMSException

Set method for brokerPubQueue attribute.

Parameters:
brokerPubQueue - the name of the broker publish queue.

setBrokerQueueManager * §
public void setBrokerQueueManager(String x) throws JMSException

Set method for brokerQueueManager attribute.

Parameters:
brokerQueueManager - the name of the broker’s Queue Manager.
TopicConnectionFactory

`setBrokerSubQueue` *

code:
```java
public void setBrokerSubQueue(String x) throws JMSException
```

Set method for brokerSubQueue attribute.

**Parameters:**
- `brokerSubQueue` - the name of the non-durable subscription queue to use.

`setBrokerVersion` *

code:
```java
public void setBrokerVersion(int x) throws JMSException
```

Set method for brokerVersion attribute.

**Parameters:**
- `brokerVersion` - the broker's version number.

`setCleanupInterval` *

code:
```java
public void setCleanupInterval(long x) throws JMSException
```

Set method for cleanupInterval attribute.

**Parameters:**
- `how often the cleanup utility will run, in milliseconds`

`setCleanupLevel` *

code:
```java
public void setCleanupLevel(int x) throws JMSException
```

Set method for cleanupLevel attribute.

**Parameters:**
- An integer representing one of the valid cleanupLevel values.
  
  These are represented by the constants:
  
  - `JMSC.MQJMS_CLEANUP_NONE`
  - `JMSC.MQJMS_CLEANUP_SAFE`
  - `JMSC.MQJMS_CLEANUP_STRONG`
  - `JMSC.MQJMS_CLEANUP_AS_PROPERTY`

`setPubAckInterval` *

code:
```java
public void setPubAckInterval(int x)
```

Set method for pubAckInterval attribute.

**Parameters:**
- `pubAckInterval` - the number of messages to use as interval.

`setStatusRefreshInterval` *

code:
```java
public void setStatusRefreshInterval(int x)
```

Set method for statusRefreshInterval attribute.

**Parameters:**
- `statusRefreshInterval` - the number of milliseconds between transactions to refresh publish/subscribe status.
**TopicConnectionFactory**

```java
setSubscriptionStore * §
```

```java
public void setSubscriptionStore(int x) throws JMSException
```

Set method for SUBSTORE property.

**Parameters:**

SubStore type - an integer representing one of the valid subStore values. These are represented by the constants:
- JMSC.MQJMS_SUBSTORE_QUEUE
- JMSC.MQJMS_SUBSTORE_BROKER
- JMSC.MQJMS_SUBSTORE_MIGRATE
TopicPublisher

public interface TopicPublisher extends MessageProducer

WebSphere MQ class: MQTopicPublisher

java.lang.Object
   +----com.ibm.mq.jms.MQMessageProducer
      +----com.ibm.mq.jms.MQTopicPublisher

A client uses a TopicPublisher for publishing messages on a topic. TopicPublisher is the Pub/Sub variant of a JMS message producer.

Methods

close *
public void close() throws JMSException

Because a provider may allocate some resources outside of the JVM on behalf of a TopicPublisher, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Throws:
JMSException if JMS fails to close the producer because of an error.

Overrides:
close in class MQMessageProducer.

getTopic
public Topic getTopic() throws JMSException

Get the topic associated with this publisher.

Returns:
this publisher’s topic

Throws:
JMSException - if JMS fails to get the topic for this topic publisher because of an internal error.

publish
public void publish(Message message) throws JMSException

Publish a Message to the topic Use the topic’s default delivery mode, time to live, and priority.

Parameters:
message - the message to publish

Throws:
• JMSException - if JMS fails to publish the message because of an internal error.
• MessageFormatException - if a non valid message is specified.
• InvalidDestinationException - if a client uses this method with a Topic Publisher with a non valid topic.
public void publish(Message message,
int deliveryMode,
int priority,
long timeToLive) throws JMSException

Publish a Message to the topic specifying delivery mode, priority, and time to live to the topic.

**WebSphere MQ Event Broker note**
If deliveryMode is PERSISTENT or timeToLive is greater than 0, this method throws a JMSException when you have a direct connection to WebSphere MQ Event Broker.

**Parameters:**
- message - the message to publish.
- deliveryMode - the delivery mode to use.
- priority - the priority for this message.
- timeToLive - the message’s lifetime (in milliseconds).

**Throws:**
- JMSException - if JMS fails to publish the message because of an internal error.
- MessageFormatException - if a non valid message is specified.
- InvalidDestinationException - if a client uses this method with a Topic Publisher with a non valid topic.

public void publish(Topic topic,
Message message) throws JMSException

Publish a Message to a topic for an unidentified message producer. Use the topic’s default delivery mode, time to live, and priority.

**Parameters:**
- topic - the topic to publish this message to.
- message - the message to send.

**Throws:**
- JMSException - if JMS fails to publish the message because of an internal error.
- MessageFormatException - if a non valid message is specified.
- InvalidDestinationException - if a client uses this method with a non valid topic.
public void publish(Topic topic,
Message message,
int deliveryMode,
int priority,
long timeToLive) throws JMSException

Publish a Message to a topic for an unidentified message producer, specifying delivery mode, priority, and time to live.

**WebSphere MQ Event Broker note**
If deliveryMode is PERSISTENT or timeToLive is greater than 0, this method throws a JMSException when you have a direct connection to WebSphere MQ Event Broker.

Parameters:
- topic - the topic to publish this message to.
- message - the message to send.
- deliveryMode - the delivery mode to use.
- priority - the priority for this message.
- timeToLive - the message’s lifetime (in milliseconds).

Throws:
- JMSException - if JMS fails to publish the message because of an internal error.
- MessageFormatException - if a non valid message is specified.
- InvalidDestinationException - if a client uses this method with a non valid topic.
public class TopicRequestor
extends java.lang.Object

java.lang.Object
   +----javax.jms.TopicRequestor

JMS provides this TopicRequestor class to assist with making service requests.

The TopicRequestor constructor is given a non-transacted TopicSession and a
destination Topic. It creates a TemporaryTopic for the responses, and provides a
request() method that sends the request message and waits for its reply. Users are
free to create more sophisticated versions.

Constructors

   public TopicRequestor(TopicSession session,
                          Topic topic) throws JMSException

Constructor for the TopicRequestor class. This implementation assumes
that the session parameter is non-transacted, and either
AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE.

Parameters:
   • session - the topic session the topic belongs to.
   • topic - the topic to perform the request/reply call on.

Throws:
   JMSException - if a JMS error occurs.

Methods

   public void close() throws JMSException

Because a provider may allocate some resources outside of the JVM on
behalf of a TopicRequestor, clients should close them when they are not
needed. You cannot rely on garbage collection to reclaim these resources
eventually, because this may not occur soon enough.

   Note: This method closes the Session object passed to the TopicRequestor
constructor.

   Throws:
       JMSException - if a JMS error occurs.

   public Message request(Message message) throws JMSException

Send a request and wait for a reply.

   Parameters:
       message - the message to send.
Returns:
    the reply message.

Throws:
    JMSException - if a JMS error occurs.
public interface TopicSession
extends Session

WebSphere MQ class: MQTopicSession

A TopicSession provides methods for creating TopicPublishers, TopicSubscribers
and TemporaryTopics.

See also: Session

WebSphere MQ constructor

MQTopicSession

public MQTopicSession(boolean transacted,
int acknowledgeMode) throws JMSException

See TopicConnection.createTopicSession.

Methods

createDurableSubscriber

public TopicSubscriber createDurableSubscriber
(Topic topic,
java.lang.String name) throws JMSException

Create a durable subscriber to the specified topic.

WebSphere MQ Event Broker note

This method throws a JMSException when you have a direct
connection to WebSphere MQ Event Broker.

Parameters:

• topic - the topic to subscribe to.
• name - the name used to identify this subscription.

Throws:

• JMSException - if a session fails to create a subscriber because of
a JMS error.
• InvalidDestinationException - if the Topic specified is not valid.

See TopicSession.unsubscribe
createDurableSubscriber

```java
public TopicSubscriber createDurableSubscriber(
    Topic topic,
    java.lang.String name,
    java.lang.String messageSelector,
    boolean noLocal)
    throws JMSException
```

Create a durable subscriber to the specified topic. A client can change an existing durable subscription by creating a durable subscriber with the same name and a new topic or message selector or both.

**WebSphere MQ Event Broker note**

This method throws a JMSException when you have a direct connection to WebSphere MQ Event Broker.

**Parameters:**

- `topic` - the topic to subscribe to.
- `name` - the name used to identify this subscription.
- `messageSelector` - only messages with properties that match the message selector expression are delivered. This value may be null.
- `noLocal` - if set, inhibits the delivery of messages published by its own connection.

**Throws:**

- `JMSException` - if a session fails to create a subscriber because of a JMS error or non valid selector.
- `InvalidDestinationException` - if the Topic specified is not valid.
- `InvalidSelectorException` - if the message selector is not valid.

createPublisher

```java
public TopicPublisher createPublisher(Topic topic)
    throws JMSException
```

Create a Publisher for the specified Topic.

**Parameters:**

- `topic` - the topic to publish to, or null if this is an unidentified producer.

**Throws:**

- `JMSException` - if a session fails to create a publisher because of a JMS error.
- `InvalidDestinationException` - if the Topic specified is not valid.
**createSubscriber**

```java
public TopicSubscriber createSubscriber(Topic topic) throws JMSException
```

Create a non-durable subscriber to the specified topic.

**Parameters:**
- topic - the topic to subscribe to

**Throws:**
- JMSException - if a session fails to create a subscriber because of a JMS error.
- InvalidDestinationException - if the Topic specified is not valid.

```java
public TopicSubscriber createSubscriber(Topic topic, java.lang.String messageSelector, boolean noLocal) throws JMSException
```

Create a non-durable subscriber to the specified topic.

**Parameters:**
- topic - the topic to subscribe to.
- messageSelector - only messages with properties that match the message selector expression are delivered. This value may be null.
- noLocal - if set, inhibits the delivery of messages published by its own connection.

**Throws:**
- JMSException - if a session fails to create a subscriber because of a JMS error or non valid selector.
- InvalidDestinationException - if the Topic specified is not valid.
- InvalidSelectorException - if the message selector is not valid.

**createTemporaryTopic**

```java
public TemporaryTopic createTemporaryTopic() throws JMSException
```

Create a temporary topic. Its lifetime will be that of the TopicConnection unless deleted earlier.

**Returns:**
- a temporary topic.

**Throws:**
- JMSException - if a session fails to create a temporary topic because of a JMS error.
createTopic

```java
public Topic createTopic(java.lang.String topicName)
    throws JMSException
```

Create a Topic given a URI format Topic name. (URI format is described in “Creating topics at runtime” on page 212) This allows the creation of a topic with a provider specific name.

**Note:** Clients that depend on this ability are not portable.

**Parameters:**
- topicName - the name of this topic.

**Returns:**
- a Topic with the given name.

**Throws:**
- JMSException - if a session fails to create a topic because of a JMS error.

unsubscribe

```java
public void unsubscribe(java.lang.String name)
    throws JMSException
```

Unsubscribe a durable subscription that has been created by a client.

**WebSphere MQ Event Broker note**

This method throws a JMSException when you have a direct connection to WebSphere MQ Event Broker.

**Note:** Do not use this method while an active subscription exists. You must close() your subscriber first.

**Parameters:**
- name - the name used to identify this subscription.

**Throws:**
- JMSException - if JMS fails to unsubscribe the durable subscription because of a JMS error.
- InvalidDestinationException - if the subscription name specified is not valid.
public interface TopicSubscriber extends MessageConsumer

WebSphere MQ class: MQTopicSubscriber

java.lang.Object
   +----com.ibm.mq.jms.MQMessageConsumer
   |   +----com.ibm.mq.jms.MQTopicSubscriber

A client uses a TopicSubscriber for receiving messages that have been published to a topic. TopicSubscriber is the Pub/Sub variant of a JMS message consumer.

See also: MessageConsumer and TopicSession.createSubscriber

MQTopicSubscriber inherits the following methods from MQMessageConsumer:
   close
   getMessageListener
   receive
   receiveNoWait
   setMessageListener

Methods

getNoLocal
   public boolean getNoLocal() throws JMSException

Get the NoLocal attribute for this TopicSubscriber. The default value for this attribute is false.

Returns:
   set to true if locally published messages are being inhibited.

Throws:
   JMSException - if JMS fails to get NoLocal attribute for this topic subscriber because of an internal error.

g getTopic
   public Topic getTopic() throws JMSException

Get the topic associated with this subscriber.

Returns:
   this subscriber’s topic.

Throws:
   JMSException - if JMS fails to get topic for this topic subscriber because of an internal error.
public interface XAConnection
Subinterfaces: XAQueueConnection and XATopicConnection

WebSphere MQ class: MQXACConnection

XAConnection extends the capability of Connection by providing an XASession. Refer to Appendix E, “JMS JTA/XA interface with WebSphere Application Server V4” on page 413 for details about how MQ JMS uses XA classes.

See also: XAQueueConnection and XATopicConnection
public interface XACreationFactory
Subinterfaces: XAQueueConnectionFactory and XATopicConnectionFactory

WebSphere MQ class: MQXAConnectionFactory

java.lang.Object

+----com.ibm.mq.jms.MQXAConnectionFactory

Some application servers provide support to group JTS-capable resource use into a distributed transaction. To include JMS transactions in a JTS transaction, an application server requires a JTS-aware JMS provider. A JMS provider exposes its JTS support by using a JMS XAConnectionFactory, which an application server uses to create XASessions. XAConnectionFactoryes are JMS-administered objects just like ConnectionFactories. It is expected that application servers use JNDI to find them.

Refer to [Appendix E, “JMS JTA/XA interface with WebSphere Application Server V4” on page 413] for details about how WebSphere MQ JMS uses XA classes.

See also: XAQueueConnectionFactory and XATopicConnectionFactory
public interface XAQueueConnection
extends QueueConnection and XAConnection

WebSphere MQ class: MQXAQueueConnection

XAQueueConnection provides the same create options as QueueConnection. The only difference is that, by definition, an XAConnection is transacted. Refer to Appendix E, “JMS JTA/XA interface with WebSphere Application Server V4” on page 413 for details about how WebSphere MQ JMS uses XA classes.

See also: XAConnection and QueueConnection

Methods

createQueueSession
    public QueueSession createQueueSession(boolean transacted,
                          int acknowledgeMode)
                        throws JMSException

Create a QueueSession.

Parameters:
    • transacted - if true, the session is transacted.
    • acknowledgeMode - indicates whether the consumer or the client will acknowledge any messages it receives. Possible values are:
        Session.AUTO_ACKNOWLEDGE
        Session.CLIENT_ACKNOWLEDGE
        Session.DUPS_OK_ACKNOWLEDGE

This parameter is ignored if the session is transacted.

Returns:
    a newly created queue session (note that this is not an XA queue session).

Throws:
    JMSException - if JMS Connection fails to create a queue session because of an internal error.

createXAQueueSession
    public XAQueueSession createXAQueueSession()

Create an XAQueueSession.

Throws:
    JMSException - if JMS Connection fails to create an XA queue session because of an internal error.
**XAQueueConnectionFactory**

public interface XAQueueConnectionFactory
extends QueueConnectionFactory and XAConnectionFactory

WebSphere MQ class: MQXAQueueConnectionFactory

java.lang.Object
   +----com.ibm.mq.jms.MQConnectionFactory
   |    +----com.ibm.mq.jms.MQQueueConnectionFactory
   |    |    +----com.ibm.mq.jms.MQXAQueueConnectionFactory

An XAQueueConnectionFactory provides the same create options as a QueueConnectionFactory. Refer to Appendix E, “JMS JTA/XA interface with WebSphere Application Server V4” on page 413 for details about how WebSphere MQ JMS uses XA classes.

See also: QueueConnectionFactory and XAConnectionFactory

**Methods**

createXAQueueConnection

public XAQueueConnection createXAQueueConnection()
throws JMSException

Create an XAQueueConnection using the default user identity. The connection is created in stopped mode. No messages are delivered until the Connection.start method is called explicitly.

Returns:
a newly created XA queue connection.

Throws:
- JMSException - if the JMS Provider fails to create an XA queue connection because of an internal error.
- JMSSecurityException - if client authentication fails because of a non valid user name or password.

createXAQueueConnection

public XAQueueConnection createXAQueueConnection
    (java.lang.String userName,
    java.lang.String password)
throws JMSException

Create an XA queue connection using a specific user identity. The connection is created in stopped mode. No messages are delivered until the Connection.start method is called explicitly.

Parameters:
- userName - the user name of the caller.
- password - the password for the caller.

Returns:
a newly created XA queue connection.
Throws:

- JMSException - if the JMS Provider fails to create an XA queue connection because of an internal error.
- JMSSecurityException - if client authentication fails because of a non valid user name or password.
public interface XAQueueSession
extends XASession

WebSphere MQ class: MQXAQueueSession

Java.lang.Object
  +----com.ibm.mq.jms.MQXASession
  |    +----com.ibm.mq.jms.MQXAQueueSession

An XAQueueSession provides a regular QueueSession that can be used to create QueueReceivers, QueueSenders and QueueBrowsers. Refer to Appendix E, “JMS JTA/XA interface with WebSphere Application Server V4” on page 413 for details about how WebSphere MQ JMS uses XA classes.

The XAResource that corresponds to the QueueSession can be obtained by calling the getXAResource method, which is inherited from XASession.

See also: XASession

Methods

getQueueSession
  public QueueSession getQueueSession() throws JMSException

Get the queue session associated with this XAQueueSession.

Returns:
the queue session object.

Throws:
JMSException - if a JMS error occurs.
XASession

public interface XASession
extends Session
Subinterfaces: XAQueueSession and XATopicSession

WebSphere MQ class: MQXASession

XASession extends the capability of Session by adding access to a JMS provider’s support for JTA. This support takes the form of a javax.transaction.xa.XAResource object. The functionality of this object closely resembles that defined by the standard X/Open XA Resource interface.

An application server controls the transactional assignment of an XASession by obtaining its XAResource. It uses the XAResource to assign the session to a transaction, prepare and commit work on the transaction, and so on.

An XAResource provides some fairly sophisticated facilities such as interleaving work on multiple transactions and recovering a list of transactions in progress.

A JTA-aware JMS provider must fully implement this functionality. To do this, a JMS provider could either use the services of a database that supports XA, or implement this functionality from scratch.

A client of the application server is given what appears to be a regular JMS Session. Behind the scenes, the application server controls the transaction management of the underlying XASession.

Refer to Appendix E, “JMS JTA/XA interface with WebSphere Application Server V4” on page 413 for details about how WebSphere MQ JMS uses XA classes.

See also: XAQueueSession and XATopicSession

Methods

commit

public void commit()
throws JMSException

This method should not be called for an XASession object. If it is called, it throws a TransactionInProgressException.

Specified by:
commit in the Session interface.

Throws:
TransactionInProgressException - if this method is called on an XASession.

getTransacted

public boolean getTransacted()
throws JMSException

Always returns true.
**XASession**

**Specified by:**
getTransacted in the Session interface.

**Returns:**
true - if the session is in transacted mode.

**Throws:**
JMSException - if JMS fails to return the transaction mode because of an internal error in the JMS Provider.

**getXAResource**

```java
public javax.transaction.xa.XAResource getXAResource()
```

Return an XA resource to the caller.

**Returns:**
an XA resource to the caller.

**rollback**

```java
public void rollback()
```

Throws JMSException

This method should not be called for an XASession object. If it is called, it throws a TransactionInProgressException.

**Specified by:**
rollback in the Session interface.

**Throws:**
TransactionInProgressException - if this method is called on an XASession.
public interface XATopicConnection
extends TopicConnection and XAConnection

WebSphere MQ class: MQXATopicConnection

An XATopicConnection provides the same create options as TopicConnection. The only difference is that, by definition, an XAConnection is transacted. Refer to Appendix E, “JMS JTA/XA interface with WebSphere Application Server V4” on page 413 for details about how WebSphere MQ JMS uses XA classes.

See also: TopicConnection and XAConnection

Methods

createTopicSession

    public TopicSession createTopicSession(boolean transacted,
                                            int acknowledgeMode)
    throws JMSException

Create a TopicSession.

Specified by:
    createTopicSession in interface TopicConnection.

Parameters:
    • transacted - if true, the session is transacted.
    • acknowledgeMode - one of:
        Session.AUTO_ACKNOWLEDGE
        Session.CLIENT_ACKNOWLEDGE
        Session.DUPS_OK_ACKNOWLEDGE

    Indicates whether the consumer or the client will acknowledge any messages it receives. This parameter will be ignored if the session is transacted.

Returns:
    a newly created topic session (note that this is not an XA topic session).

Throws:
    JMSException - if JMS Connection fails to create a topic session because of an internal error.

createXATopicSession

    public XATopicSession createXATopicSession()
    throws JMSException

Create an XATopicSession.
XATopicConnection

Throws:

JMSException - if the JMS Connection fails to create an XA topic session because of an internal error.
XATopicConnectionFactory

public interface XATopicConnectionFactory
extends TopicConnectionFactory and XAConnectionFactory

WebSphere MQ class: MQXATopicConnectionFactory

An XATopicConnectionFactory provides the same create options as
TopicConnectionFactory. Refer to Appendix E, "JMS JTA/XA interface with
WebSphere Application Server V4" on page 413 for details about how WebSphere
MQ JMS uses XA classes.

See also: TopicConnectionFactory and XAConnectionFactory

Methods

createXATopicConnection

public XATopicConnection createXATopicConnection()
throws JMSException

Create an XA topic connection using the default user identity. The
connection is created in stopped mode. No messages are delivered until
the Connection.start method is called explicitly.

Returns:
a newly created XA topic connection.

Throws:

• JMSException - if the JMS Provider fails to create an XA topic
  connection because of an internal error.
• JMSSecurityException - if client authentication fails because of a
  non valid user name or password.

createXATopicConnection

public XATopicConnection createXATopicConnection(java.lang.String userName,
java.lang.String password)
throws JMSException

Create an XA topic connection using the specified user identity. The
connection is created in stopped mode. No messages are delivered until
the Connection.start method is called explicitly.

Parameters:

• userName - the user name of the caller
• password - the password of the caller

Returns:
a newly created XA topic connection.

Throws:
XATopicConnectionFactory

- JMSException - if the JMS Provider fails to create an XA topic connection because of an internal error.
- JMSSecurityException - if client authentication fails because of a non valid user name or password.
XATopicSession

public interface XATopicSession
extends XASession

WebSphere MQ class: MQXATopicSession

java.lang.Object
    +----com.ibm.mq.jms.MQASession
        +----com.ibm.mq.jms.MQXASession
        +----com.ibm.mq.jms.MQXATopicSession

An XATopicSession provides a TopicSession, which can be used to create
TopicSubscribers and TopicPublishers. Refer to Appendix E, “JMS JTA/XA interface
with WebSphere Application Server V4” on page 413 for details about how
WebSphere MQ JMS uses XA classes.

The XAResource that corresponds to the TopicSession can be obtained by calling
the getXAResource method, which is inherited from XASession.

See also: TopicSession and XASession

Methods

getTopicSession

    public TopicSession getTopicSession()
    throws JMSException

    Get the topic session associated with this XATopicSession.

    Returns:
    the topic session object.

    Throws:
    • JMSException - if a JMS error occurs.
Part 4. Appendixes
Appendix A. Mapping between Administration tool properties and programmable properties

WebSphere MQ classes for Java Message Service provide facilities to set and query the properties of administered objects either using the WebSphere MQ JMS administration tool, or in an application program. Table 37 shows the mapping between each property name used with the administration tool and the corresponding member variable it refers to. It also shows the mapping between symbolic property values used in the tool and their programmable equivalents.

Table 37. Comparison of representations of property values within the administration tool and within programs.

<table>
<thead>
<tr>
<th>Property</th>
<th>Member variable name</th>
<th>Tool property values</th>
<th>Program property values</th>
</tr>
</thead>
<tbody>
<tr>
<td>BROKERCCDSUBQ</td>
<td>brokerCCDurSubQueue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BROKERCCSUBQ</td>
<td>brokerCCSubQueue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BROKERCONQ</td>
<td>brokerControlQueue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BROKERDURSUBQ</td>
<td>brokerDurSubQueue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BROKERMQMQGR</td>
<td>brokerQueueManager</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BROKERQUBQ</td>
<td>brokerSubQueue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BROKERQVER</td>
<td>brokerVersion</td>
<td>V1</td>
<td>JMSC.MQJMS_BROKER_V1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V2</td>
<td>JMSC.MQJMS_BROKER_V2</td>
</tr>
<tr>
<td>CCSID</td>
<td>CCSID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHANNEL</td>
<td>channel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLEANUP</td>
<td>cleanupLevel</td>
<td>NONE</td>
<td>JMSC.MQJMS_CLEANUP_NONE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAFE</td>
<td>JMSC.MQJMS_CLEANUP_SAFE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STRONG</td>
<td>JMSC.MQJMS_CLEANUP_STRONG</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ASPROP</td>
<td>JMSC.MQJMS_CLEANUP_AS_PROPERTY</td>
</tr>
<tr>
<td>CLEANUPINT</td>
<td>cleanupInterval</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLIENTID</td>
<td>clientId</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCODING</td>
<td>encoding</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPIRY</td>
<td>expiry</td>
<td>APP</td>
<td>JMSC.MQJMS_EXP_APP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UNLIM</td>
<td>JMSC.MQJMS_EXP_UNLIM</td>
</tr>
<tr>
<td>HOSTNAME</td>
<td>hostName</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSGBATCHSZ</td>
<td>msgBatchSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSGRETENTION</td>
<td>messageRetention</td>
<td>YES</td>
<td>JMSC.MQJMS_MRET_YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NO</td>
<td>JMSC.MQJMS_MRET_NO</td>
</tr>
<tr>
<td>PERSISTENCE</td>
<td>persistence</td>
<td>APP</td>
<td>JMSC.MQJMS_PER_APP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QDEF</td>
<td>JMSC.MQJMS_PER_QDEF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PERS</td>
<td>JMSC.MQJMS_PER_PER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NON</td>
<td>JMSC.MQJMS_PER_NON</td>
</tr>
<tr>
<td>POLLINGINT</td>
<td>pollingInterval</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PORT</td>
<td>port</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 37. Comparison of representations of property values within the administration tool and within programs. (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Member variable name</th>
<th>Tool property values</th>
<th>Program property values</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIORITY</td>
<td>priority</td>
<td>APP</td>
<td>JMSC.MQJMS_PRI_APP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QDEF</td>
<td>JMSC.MQJMS_PRI_QDEF</td>
</tr>
<tr>
<td>/pubAckInterval</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUEUE</td>
<td>baseQueueName</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QMANAGER</td>
<td>queueManager*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECEIXIT</td>
<td>receiveExit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECEIXINIT</td>
<td>receiveExitInit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECEXIT</td>
<td>securityExit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECEXITINIT</td>
<td>securityExitInit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SENDEXIT</td>
<td>sendExit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SENDEXEXIT</td>
<td>sendExitInit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSLCIPHERSUITE</td>
<td>sslCipherSuite</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSLCRL</td>
<td>sslCertStores</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSLPEERNAME</td>
<td>sslPeerName</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STATREFRESHINT</td>
<td>statusRefreshInterval</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYNCPOINTALLGETS</td>
<td>syncpointAllGets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TARGCLIENT</td>
<td>targetClient</td>
<td>JMS</td>
<td>JMSC.MQJMS_CLIENT_JMS_COMPLIANT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MQ</td>
<td>JMSC.MQJMS_CLIENT_NONJMS_MQ</td>
</tr>
<tr>
<td>TEMPMODEL</td>
<td>temporaryModel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOPIC</td>
<td>baseTopicName</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANSPORT</td>
<td>transportType</td>
<td>BIND</td>
<td>JMSC.MQJMS_TP_BINDINGS_MQ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CLIENT</td>
<td>JMSC.MQJMS_TP_CLIENT_MQ_TCPPIP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DIRECT</td>
<td>JMSC.MQJMS_TP_DIRECT_TCPPIP</td>
</tr>
<tr>
<td>USECONNPOOLING</td>
<td>useConnectionPooling</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: * for an MQQueue object, the member variable name is baseQueueManagerName
Appendix B. Scripts provided with WebSphere MQ classes for Java Message Service

The following files are provided in the bin directory of your WebSphere MQ JMS installation. These scripts are provided to assist with common tasks that need to be performed while installing or using WebSphere MQ JMS. Table 38 lists the scripts and their uses.

Table 38. Utilities supplied with WebSphere MQ classes for Java Message Service

<table>
<thead>
<tr>
<th>Utility</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cleanup.bat</td>
<td>Used to run the subscription cleanup utility, described in “Manual cleanup” on page 222</td>
</tr>
<tr>
<td>DefaultConfiguration</td>
<td>Used to run the default configuration application on non-Windows systems, as described in “JMS Postcard configuration” on page 20</td>
</tr>
<tr>
<td>formatLog.bat</td>
<td>Used to convert binary log files to plain text, described in “Logging” on page 36</td>
</tr>
<tr>
<td>IVTRun.bat</td>
<td>Used to run the point-to-point installation verification test program, described in “Running the point-to-point IVT” on page 28</td>
</tr>
<tr>
<td>IVTTidy.bat</td>
<td>Used to run the point-to-point installation verification test program, described in “Running the point-to-point IVT” on page 28</td>
</tr>
<tr>
<td>IVTSetup.bat</td>
<td>Used to run the point-to-point installation verification test program, described in “Running the point-to-point IVT” on page 28</td>
</tr>
<tr>
<td>JMSAdmin.bat</td>
<td>Used to run the administration tool, described in Chapter 5, “Using the WebSphere MQ JMS administration tool” on page 39</td>
</tr>
<tr>
<td>JMSAdmin.config</td>
<td>Configuration file for the administration tool, described in “Configuration” on page 40</td>
</tr>
<tr>
<td>postcard.bat</td>
<td>Used to start the JMS Postcard application, described in “JMS Postcard” on page 17</td>
</tr>
<tr>
<td>PSIVTRun.bat</td>
<td>Used to run the Pub/Sub installation verification test program described in “The publish/subscribe installation verification test” on page 32</td>
</tr>
<tr>
<td>PSReportDump.class</td>
<td>Used to view broker report messages, described in “Handling broker reports” on page 223</td>
</tr>
<tr>
<td>runjms.bat</td>
<td>A utility script to assist with the running of JMS applications, described in “Running your own WebSphere MQ JMS programs” on page 35</td>
</tr>
</tbody>
</table>

Note: On UNIX systems, the extension “.bat” is omitted from the filenames.
Scripts
Appendix C. LDAP schema definition for storing Java objects

This appendix gives details of the schema definitions (objectClass and attribute definitions) needed in an LDAP directory in order for it to store Java objects. Its intended audience is users wishing to use an LDAP server as their JNDI service provider in which to store WebSphere MQ JMS administered objects. You should ensure that your LDAP server schema contains the following definitions; the exact procedure to achieve this will vary from server to server. How to make the changes to some specific LDAP servers is covered later in this section.

Much of the data contained in this appendix has been taken from RFC 2713 Schema for Representing Java Objects in an LDAP Directory, which can be found at http://www.faqs.org/rfcs/rfc2713.html. LDAP server-specific information has been taken from Sun Microsystems’ JNDI 1.2.1 LDAP service provider, available at http://java.sun.com/products/jndi.

Checking your LDAP server configuration

To check whether the LDAP server is already configured to accept Java objects, run the WebSphere MQ JMS Administration Tool JMSAdmin against your LDAP server (see “Invoking the Administration tool” on page 39).

Attempt to create and display a test object using the following commands:

```
DEFINE QCF(ldapTest)
DISPLAY QCF(ldapTest)
```

If no errors occur, your server is properly configured to store Java objects and you can proceed to store JMS objects. However, if your LDAP server contains older schema definitions (for example, from an earlier draft of RFC 2713 such as the now-obsolete “draft-ryan-java-schema-00” and “draft-ryan-java-schema-01” specifications), you should update them with those presented here.

If a SchemaViolationException occurs, or if the message “Unable to bind to object” is returned, your server is not properly configured. Either your server is not configured to store Java objects, or permissions on the objects are not correct, or the provided suffix or context has not been set up. The following information should help you with the schema configuration part of your server setup.
Attribute definitions

Table 39. Attribute settings for javaCodebase

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID (Object Identifier)</td>
<td>1.3.6.1.4.1.42.2.27.4.1.7</td>
</tr>
<tr>
<td>Syntax</td>
<td>IA5 String (1.3.6.1.4.1.1466.115.121.1.26)</td>
</tr>
<tr>
<td>Maximum length</td>
<td>2048</td>
</tr>
<tr>
<td>Single/multi-valued</td>
<td>Multi-valued</td>
</tr>
<tr>
<td>User modifiable?</td>
<td>Yes</td>
</tr>
<tr>
<td>Matching rules</td>
<td>caseExactIA5Match</td>
</tr>
<tr>
<td>Access class</td>
<td>normal</td>
</tr>
<tr>
<td>Usage</td>
<td>userApplications</td>
</tr>
<tr>
<td>Description</td>
<td>URL(s) specifying the location of class definition</td>
</tr>
</tbody>
</table>

Table 40. Attribute settings for javaClassName

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID (Object Identifier)</td>
<td>1.3.6.1.4.1.42.2.27.4.1.6</td>
</tr>
<tr>
<td>Syntax</td>
<td>Directory String (1.3.6.1.4.1.1466.115.121.1.15)</td>
</tr>
<tr>
<td>Maximum length</td>
<td>2048</td>
</tr>
<tr>
<td>Single/multi-valued</td>
<td>Single-valued</td>
</tr>
<tr>
<td>User modifiable?</td>
<td>Yes</td>
</tr>
<tr>
<td>Matching rules</td>
<td>caseExactMatch</td>
</tr>
<tr>
<td>Access class</td>
<td>normal</td>
</tr>
<tr>
<td>Usage</td>
<td>userApplications</td>
</tr>
<tr>
<td>Description</td>
<td>Fully qualified name of distinguished Java class or interface</td>
</tr>
</tbody>
</table>

Table 41. Attribute settings for javaClassNames

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID (Object Identifier)</td>
<td>1.3.6.1.4.1.42.2.27.4.1.13</td>
</tr>
<tr>
<td>Syntax</td>
<td>Directory String (1.3.6.1.4.1.1466.115.121.1.15)</td>
</tr>
<tr>
<td>Maximum length</td>
<td>2048</td>
</tr>
<tr>
<td>Single/multi-valued</td>
<td>Multi-valued</td>
</tr>
<tr>
<td>User modifiable?</td>
<td>Yes</td>
</tr>
<tr>
<td>Matching rules</td>
<td>caseExactMatch</td>
</tr>
<tr>
<td>Access class</td>
<td>normal</td>
</tr>
<tr>
<td>Usage</td>
<td>userApplications</td>
</tr>
<tr>
<td>Description</td>
<td>Fully qualified Java class or interface name</td>
</tr>
</tbody>
</table>
Table 42. Attribute settings for javaFactory

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID (Object Identifier)</td>
<td>1.3.6.1.4.1.42.2.27.4.1.10</td>
</tr>
<tr>
<td>Syntax</td>
<td>Directory String (1.3.6.1.4.1.1466.115.121.1.15)</td>
</tr>
<tr>
<td>Maximum length</td>
<td>2048</td>
</tr>
<tr>
<td>Single/multi-valued</td>
<td>Single-valued</td>
</tr>
<tr>
<td>User modifiable?</td>
<td>Yes</td>
</tr>
<tr>
<td>Matching rules</td>
<td>caseExactMatch</td>
</tr>
<tr>
<td>Access class</td>
<td>normal</td>
</tr>
<tr>
<td>Usage</td>
<td>userApplications</td>
</tr>
<tr>
<td>Description</td>
<td>Fully qualified Java class name of a JNDI object factory</td>
</tr>
</tbody>
</table>

Table 43. Attribute settings for javaReferenceAddress

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID (Object Identifier)</td>
<td>1.3.6.1.4.1.42.2.27.4.1.11</td>
</tr>
<tr>
<td>Syntax</td>
<td>Directory String (1.3.6.1.4.1.1466.115.121.1.15)</td>
</tr>
<tr>
<td>Maximum length</td>
<td>2048</td>
</tr>
<tr>
<td>Single/multi-valued</td>
<td>Multi-valued</td>
</tr>
<tr>
<td>User modifiable?</td>
<td>Yes</td>
</tr>
<tr>
<td>Matching rules</td>
<td>caseExactMatch</td>
</tr>
<tr>
<td>Access class</td>
<td>normal</td>
</tr>
<tr>
<td>Usage</td>
<td>userApplications</td>
</tr>
<tr>
<td>Description</td>
<td>Addresses associated with a JNDI Reference</td>
</tr>
</tbody>
</table>

Table 44. Attribute settings for javaSerializedData

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID (Object Identifier)</td>
<td>1.3.6.1.4.1.42.2.27.4.1.8</td>
</tr>
<tr>
<td>Syntax</td>
<td>Octet String (1.3.6.1.4.1.1466.115.121.1.40)</td>
</tr>
<tr>
<td>Single/multi-valued</td>
<td>Single-valued</td>
</tr>
<tr>
<td>User modifiable?</td>
<td>Yes</td>
</tr>
<tr>
<td>Access class</td>
<td>normal</td>
</tr>
<tr>
<td>Usage</td>
<td>userApplications</td>
</tr>
<tr>
<td>Description</td>
<td>Serialized form of a Java object</td>
</tr>
</tbody>
</table>

objectClass definitions

Table 45. objectClass definition for javaSerializedObject

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID (Object Identifier)</td>
<td>1.3.6.1.4.1.42.2.27.4.2.5</td>
</tr>
<tr>
<td>Extends/superior</td>
<td>javaObject</td>
</tr>
<tr>
<td>Type</td>
<td>AUXILIARY</td>
</tr>
<tr>
<td>Required (must) attrs</td>
<td>javaSerializedData</td>
</tr>
</tbody>
</table>
objectClass definitions

Table 46. objectClass definition for javaObject

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID (Object Identifier)</td>
<td>1.3.6.1.4.1.42.2.27.4.2.4</td>
</tr>
<tr>
<td>Extends/superior</td>
<td>top</td>
</tr>
<tr>
<td>Type</td>
<td>ABSTRACT</td>
</tr>
<tr>
<td>Required (must) attrs</td>
<td>javaClassName</td>
</tr>
<tr>
<td>Optional (may) attrs</td>
<td>javaClassNames javaCodebase javaDoc description</td>
</tr>
</tbody>
</table>

Table 47. objectClass definition for javaContainer

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID (Object Identifier)</td>
<td>1.3.6.1.4.1.42.2.27.4.2.1</td>
</tr>
<tr>
<td>Extends/superior</td>
<td>top</td>
</tr>
<tr>
<td>Type</td>
<td>STRUCTURAL</td>
</tr>
<tr>
<td>Required (must) attrs</td>
<td>cn</td>
</tr>
</tbody>
</table>

Table 48. objectClass definition for javaNamingReference

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID (Object Identifier)</td>
<td>1.3.6.1.4.1.42.2.27.4.2.7</td>
</tr>
<tr>
<td>Extends/superior</td>
<td>javaObject</td>
</tr>
<tr>
<td>Type</td>
<td>AUXILIARY</td>
</tr>
<tr>
<td>Optional (may) attrs</td>
<td>javaReferenceAddress javaFactory</td>
</tr>
</tbody>
</table>

Server-specific configuration details

Netscape Directory (4.1 and earlier)

This level of Netscape Directory does not support the Octet String syntax, so Binary syntax (1.3.6.1.4.1.1466.115.121.1.5) should be used in its place. Netscape Directory 4.1 also has problems parsing an object class definition which contains a MUST clause without parentheses. The workaround is to add a superfluous value (objectClass) to each MUST clause.

Alternatively, you may wish to use the Sun-supplied schema modification applications described in “Sun Microsystems’ Schema Modification Applications” on page 407.

Microsoft® Active Directory

Within Active Directory, only the names of structural classes (not auxiliary classes) may appear in the object class attribute of an entry. Therefore, the abstract and auxiliary classes in the Java schema definition should be re-defined as structural. This has the following effects:

- the javaObject class should now inherit from javaContainer
- the javaNamingReference and javaSerializedObject classes should now inherit from javaObject

Instead of making these changes manually, you may wish to use the Sun-supplied schema modification applications described in “Sun Microsystems’ Schema Modification Applications” on page 407.
Sun Microsystems’ Schema Modification Applications

You can use your LDAP server’s administration tool (for example, the Directory Management Tool for IBM’s SecureWay® Directory) to verify or add the definitions described above. Alternatively, Sun Microsystems’ JNDI 1.2.1 LDAP service provider (available at http://java.sun.com/products/jndi) contains Java applications (CreateJavaSchema.java and UpdateJavaSchema.java) which will add or update the required schema definitions automatically. These applications contain workarounds for schema bugs and server-specific behavior in both Netscape Directory Server (pre-4.1 and 4.1) and Microsoft Windows 2000 Active Directory.

These applications are not packaged with WebSphere MQ classes for Java Message Service. Details on running them can be found in both the README and the application source contained in the Sun JNDI 1.2.1 LDAP service provider download.

iSeries OS/400 V4R5 Schema Modification

You can use your LDAP server’s administration tool (the Directory Management Tool for IBM’s SecureWay Directory) to verify or add the definitions described above.

OS/400 V4R5 LDAP Server is shipped with an out of date version of RFC 2713 schema for java objects. This schema must be updated to the schema as described above for correct operation with JMSAdmin. Modifying the schema requires the out of data definitions and any uses of those definitions to be deleted before the correct definitions can be added.

OS/400 V5R1 is shipped with the current version of RFC 2713 and does not require these changes.
Appendix D. Connecting with other products

This section covers:

- How to set up a connection from WebSphere MQ JMS to a publish/subscribe broker in “Setting up a publish/subscribe broker”
- How to use WebSphere MQ Integrator V2 to route or transform messages sent to or from a JMS client in “Transformation and routing with WebSphere MQ Integrator V2” on page 410

Setting up a publish/subscribe broker

You can use WebSphere MQ Integrator V2 or WebSphere MQ Event Broker as the publish/subscribe broker for WebSphere MQ JMS. You can link to either across a connection to base WebSphere MQ, or you can directly connect to WebSphere MQ Event Broker across a TCP/IP connection. Each method requires some setup activities:

Linking across WebSphere MQ

- Base WebSphere MQ
  
  First, you must create a broker publication queue. This is a WebSphere MQ queue on the broker queue manager; it is used to submit publications to the broker. You can choose your own name for this queue but it must match the queue name in your TopicConnectionFactory’s BROKERPUBQ property. By default, a TopicConnectionFactory’s BROKERPUBQ property is set to the value SYSTEM.BROKER.DEFAULT.STREAM so, unless you want to configure a different name in the TopicConnectionFactory, you should name the queue SYSTEM.BROKER.DEFAULT.STREAM.

- WebSphere MQ Integrator V2
  
  The next step is to set up a message flow within an execution group for the broker. The purpose of this message flow is to read messages from the broker publication queue. (If you want, you can set up multiple publication queues; each will need its own TopicConnectionFactory and message flow.)

  The basic message flow consists of an MQInput node (configured to read from the SYSTEM.BROKER.DEFAULT.STREAM queue) whose output is connected to the input of a Publication (or MQOutput) node.

  The message flow diagram would therefore look similar to the following:

  ![Figure 7. WebSphere MQ Integrator message flow](image)

  When this message flow is deployed and the broker is started, from the JMS application’s perspective the WebSphere MQ Integrator V2 broker...
behaves like a MQSeries Publish/Subscribe broker. The current subscription state can be viewed using the WebSphere MQ Integrator Control Center.

Notes:
1. No modifications are required to WebSphere MQ classes for Java Message Service.
2. MQSeries Publish/Subscribe and WebSphere MQ Integrator V2 brokers cannot coexist on the same queue manager.
3. Details of the WebSphere MQ Integrator V2 installation and setup procedure are described in the WebSphere MQ Integrator for Windows NT Version 2.0 Installation Guide.

Direct connection to WebSphere MQ Event Broker across TCP/IP

For this, you must set up a message flow within an execution group on WebSphere MQ Event Broker. This message flow is to read messages from the TCP/IP socket on which the broker is listening.

The basic message flow consists of a JMSIPOptimised flow set to listen on the port configured for direct connections. By default, this port is 1506.

Note: WebSphere MQ Event Broker can be configured to listen for both direct connections across TCP/IP from WebSphere MQ JMS and connections made across TCP/IP through WebSphere MQ. In this case, the two listeners must be configured on different ports. The default port for a WebSphere MQ connection is 1414.

Transformation and routing with WebSphere MQ Integrator V2

You can use WebSphere MQ Integrator V2 to route or transform messages that are created by a JMS client application, and to send or publish messages to a JMS client.

The WebSphere MQ JMS implementation uses the mcd folder of the MQRFH2 to carry information about the message, as described in “The MQRFH2 header” on page 230. By default, the Message Domain (Msd) property is used to identify whether the message is a text, bytes, stream, map, or object message. This value is set depending on the type of the JMS message.

If the application calls setJMSType it can set the mcd type field to a value of its choosing. This type field can be read by the WebSphere MQ Integrator message flow, and a receiving JMS application can use the getJMSType method to retrieve its value. This applies to all kinds of JMS message.

When a JMS application creates a text or bytes message, the application can set mcd folder fields explicitly by calling the setJMSType method and passing in a string argument in a special URI format as follows:

```
mcd://domain/[set]/[type][?format=fmt]
```

This URI form allows an application to set the mcd to a domain that is not one of the standard “jms_xxxx” values; for example, to domain “mrm”. It also allows the application to set any or all of the mcd set, type and format fields if desired.

The string argument to setJMSType is interpreted as follows:
1. If the string does not appear to be in the special URI format (i.e. it does not start with “mcd://”) then the string is added to the mcd folder as the type field.

2. If the string does start with “mcd://” and conforms to the URI format and the message is a Text or Bytes message, then the URI string is split into its constituent parts. The domain part overrides the jms_text or jms_bytes value that would otherwise have been generated, and the remaining parts (if present) are used to set the set, type and format fields in the mcd. Note that set, type and format are all optional.

3. If the string starts with “mcd://” and the message is a Map, Stream or Object message then the setJMSType call throws an exception. So you cannot override the domain, or provide a set or format for these classes of message, but you can provide a type if you wish.

When a WebSphere MQ message is received with an Msd domain other than one of the standard “jms_xxxx” values, it is instantiated as a JMS text or bytes message and a URI-style JMSType is assigned to it. The receiving application can read this using the getJMSType method.
Appendix E. JMS JTA/XA interface with WebSphere Application Server V4

WebSphere MQ classes for Java Message Service includes the JMS XA interfaces. These allow WebSphere MQ JMS to participate in a two-phase commit that is coordinated by a transaction manager that complies with the Java Transaction API (JTA).

This section describes how to use these features with the WebSphere Application Server, Advanced Edition, so that WebSphere Application Server can coordinate JMS send and receive operations, and database updates, in a global transaction.

Note: Before you use WebSphere MQ JMS and the XA classes with WebSphere Application Server, there might be additional installation or configuration steps. Refer to the Readme.txt file on the WebSphere MQ Using Java SupportPac Web page for the latest information (www.ibm.com/software/ts/mqseries/txppacs/ma88.html).

Using the JMS interface with WebSphere Application Server

This section provides guidance on using the JMS interface with the WebSphere Application Server, Advanced Edition.

You must already understand the basics of JMS programs, WebSphere MQ, and EJB beans. These details are in the JMS specification, the EJB V2 specification (both available from Sun), this manual, the samples provided with WebSphere MQ JMS, and other manuals for WebSphere MQ and WebSphere Application Server.

Administered objects

JMS uses administered objects to encapsulate vendor-specific information. This minimizes the impact of vendor-specific details on end-user applications. Administered objects are stored in a JNDI namespace, and can be retrieved and used in a portable manner without knowledge of the vendor-specific contents.

For standalone use, WebSphere MQ JMS provides the following classes:
- MQQueueConnectionFactory
- MQQueue
- MQTopicConnectionFactory
- MQTopic

WebSphere Application Server provides an additional pair of administered objects so that WebSphere MQ JMS can integrate with WebSphere Application Server:
- JMSWrapXAQueueConnectionFactory
- JMSWrapXATopicConnectionFactory

You use these objects in exactly the same way as the MQQueueConnectionFactory and MQTopicConnectionFactory. However, behind the scenes they use the XA versions of the JMS classes, and enlist the WebSphere MQ JMS XAResource in the WebSphere Application Server transaction.
Container-managed versus bean-managed transactions

Container-managed transactions are transactions in EJB beans that are demarcated automatically by the EJB container. Bean-managed transactions are transactions in EJB beans that are demarcated by the program (via the UserTransaction interface).

Two-phase commit versus one-phase optimization

The WebSphere Application Server coordinator only invokes a true two-phase commit if more than one XAResource is used in a particular transaction. Transactions that involve a single resource are committed using a one-phase optimization. This largely removes the need to use different ConnectionFactories for distributed and non-distributed transactions.

Defining administered objects

You can use the WebSphere MQ JMS administration tool to define the WebSphere Application Server-specific connection factories and store them in a JNDI namespace. The admin.config file in \texttt{MQ\_install\_dir/bin} should contain the following lines:

\begin{verbatim}
  INITIAL_CONTEXT_FACTORY=com.ibm.websphere.naming.WsnInitialContextFactory
  PROVIDER_URL=iiop://hostname/
\end{verbatim}

\texttt{MQ\_install\_dir} is the installation directory for WebSphere MQ JMS, and \texttt{hostname} is the name or IP address of the machine that is running WebSphere Application Server.

To access the \texttt{com.ibm.ejs.ns.jndi.CNInitialContextFactory}, you must add the file ejs.jar from the WebSphere Application Server lib directory to the CLASSPATH.

To create the new factories, use the define verb with the following two new types:

\begin{verbatim}
def WSCOF\(\text{name}\) \{properties\}
def WSTCF\(\text{name}\) \{properties\}
\end{verbatim}

These new types use the same properties as the equivalent QCF or TCF types, except that only the BIND transport type is allowed (and therefore, client properties cannot be configured). For details, see “Administering JMS objects” on page 43.

Retrieving administration objects

In an EJB bean, you retrieve the JMS-administered objects using the \texttt{InitialContext.lookup()} method, for example:

\begin{verbatim}
  InitialContext ic = new InitialContext();
  TopicConnectionFactory tcf = (TopicConnectionFactory) ic.lookup("jms/Samples/TCF1");
\end{verbatim}

The objects can be cast to, and used as, the generic JMS interfaces. Normally, there is no need to program to the WebSphere MQ specific classes in the application code.

Samples

There are three samples that illustrate the basics of using WebSphere MQ JMS with WebSphere Application Server Advanced Edition. These are in subdirectories of \texttt{MQ\_samples\_dir/ws}, where \texttt{MQ\_samples\_dir} is the samples directory for WebSphere MQ JMS. See Table 3 on page 8 to find where this is.
Sample1 demonstrates a simple put and get for a message in a queue by using container-managed transactions.

Sample2 demonstrates a simple put and get for a message in a queue by using bean-managed transactions.

Sample3 illustrates the use of the publish/subscribe API.

For details about how to build and deploy the EJB beans, please refer to the WebSphere Application Server documentation.

The readme.txt files in each sample directory include example output from each EJB bean. The scripts provided assume that a default queue manager is available on the local machine. If your installation varies from the default, you can edit these scripts as required.

Sample1

Sample1EJB.java, in the sample1 directory, defines two methods that use JMS:

- putMessage() sends a TextMessage to a queue, and returns the MessageID of the sent message
- getMessage() reads the message with the specified MessageID back from the queue

Before you run the sample, you must store two administered objects in the WebSphere Application Server JNDI namespace:

- **QCF1** a WebSphere Application Server-specific queue connection factory
- **Q1** a queue

Both objects must be bound in the jms/Samples sub-context.

To set up the administered objects, you can either use the WebSphere MQ JMS administration tool and set them up manually, or you can use the script provided.

The WebSphere MQ JMS administration tool must be configured to access the WebSphere Application Server namespace. For details about how to configure the administration tool, refer to "Configuring for WebSphere Application Server V3.5" on page 42.

To set up the administered objects with typical default settings, you can enter the following command to run the script admin.scp:

```
JMSAdmin < admin.scp
```

The bean must be deployed with the getMessage and putMessage methods marked as TX_REQUIRED. This ensures that the container starts a transaction before entering each method, and commits the transaction when the method completes. Within the methods, you do not need any application code that relates to the transactional state. However, remember that the message sent from putMessage occurs under syncpoint, and will not become available until the transaction is committed.

In the sample1 directory, there is a simple client program, Sample1Client.java, to call the EJB bean. There is also a script, runClient, to simplify running this program.
JMS JTA/XA interface with WebSphere Application Server

The client program (or script) takes a single parameter, which is used as the body of a TextMessage that will be sent by the EJB bean putMessage method. Then, the getMessage is called to read the message back off the queue and return the body to the client for display. The EJB bean sends progress messages to the standard output (stdout) of the application server, so you might wish to monitor that output during the run.

If the application server is on a machine that is remote from the client, you might need to edit Sample1Client.java. If you do not use the defaults, you may need to edit the runClient script to match the local installation path and name of the deployed jar file.

Sample2

Sample2EJB.java, in the sample2 directory, performs the same task as sample1, and requires the same administered objects. Unlike sample1, sample2 uses bean-managed transactions to control the transactional boundaries.

If you have not already run sample1, ensure that you set up the administered objects QCF1 and Q1, as described in “Sample1” on page 415.

The putMessage methods and getMessage methods start by obtaining an instance of UserTransaction. They use this instance to create a transaction via the UserTransaction.begin() method. After that, the main body of the code is the same as sample1 until the end of each method. At the end of each method, the transaction is completed by the UserTransaction.commit() call.

In the sample2 directory, there is a simple client program, Sample2Client.java, to call the EJB bean. There is also a script, runClient, to simplify running this program. You can use these in the same way as described for “Sample1” on page 415.

Sample3

Sample3EJB.java, in the sample3 directory, demonstrates the use of the publish/subscribe API with WebSphere Application Server. Publishing a message is very similar to the point to point case. However, there are differences when receiving messages via a TopicSubscriber.

Publish/subscribe programs commonly use nondurable subscribers. These nondurable subscribers exist only for the lifetime of their owning sessions (or less if the subscriber is closed explicitly). Also, they can receive messages from the broker only during that lifetime.

To convert sample1 to publish/subscribe, you might replace the QueueSender in putMessage with a TopicPublisher, and the QueueReceiver in getMessage with a nondurable TopicSubscriber. However, this would fail, because when the message is sent, the broker would not know of any subscribers to the topic. Therefore, the message would be discarded.

The solution is to create a durable subscriber before the message is published. Durable subscribers persist as a deliverable end-point beyond the lifetime of the session. Therefore, the message is available for retrieval during the call to getMessage().

The EJB bean includes two additional methods:
- createSubscription creates a durable subscription
JMS JTA/XA interface with WebSphere Application Server

- destroySubscription deletes a durable subscription

These methods (along with putMessage and getMessage) must be deployed with the TX_REQUIRED attribute.

Before you run sample3, you must store two administered objects in the WebSphere Application Server JNDI namespace:

TCF1
T1

Both objects must be bound in the jms/Samples sub-context.

To set up the administered objects, you can either use the WebSphere MQ JMS administration tool and set them up manually, or you can use a script. The script admin.scp is provided in the sample3 directory.

The WebSphere MQ JMS administration tool must be configured to access the WebSphere Application Server namespace. For details about how to configure the administration tool, refer to “Configuring for WebSphere Application Server V3.5” on page 42.

To set up the administered objects with typical default settings, you can enter the following command to run the script admin.scp:

JMSAdmin < admin.scp

If you have already run admin.scp to set up objects for sample1 or sample2, there will be error messages when you run admin.scp for sample3. (These occur when you attempt to create the jms and Samples sub-contexts.) You can safely ignore these error messages.

Also, before you run sample3, ensure that the WebSphere MQ publish/subscribe broker (SupportPac MA0C) is installed and running.

In the sample3 directory, there is a simple client program, Sample3Client.java, to call the EJB bean. There is also a script, runClient, to simplify running this program. You can use these in the same way as described for “Sample1” on page 415.
You may need to perform additional tasks to run an applet using WebSphere MQ Java classes in a Java virtual machine (JVM) at Java 1.2 level or greater. This is because the default security rules for applets with JVMs at these levels were changed to reduce the risk of damage by malevolent or misbehaving classes.

There are two different approaches that you can take:
1. Change the security settings on the browser and JVM to allow the use of WebSphere MQ Java packages.
2. Copy the WebSphere MQ Java classes to the same location as the applet you wish to run.

### Changing browser security settings

Different errors can result from trying to run the same applet in different environments; for example, in IBM VisualAge for Java, in appletviewer (supplied with most Development Kits for Java) or in a web browser such as Internet Explorer. The differences are to do with different security settings in each environment. You can change the behavior of the environments to allow an applet access to the classes it needs that are stored in package files.

In the following instructions, examples assume use of the Windows platform. On other platforms, the instructions will need slight modification.

#### For IBM VisualAge for Java:

You must change the java.policy file found in <vaj_install_dir>/ide/program/lib/security, where “<vaj_install_dir>” is the directory in which you installed IBM VisualAge for Java.

Refer to "Running WebSphere MQ Java applications under the Java 2 Security Manager" on page 10 for general instructions about changes to this file. For applets, you should also check for the following changes to the permissions:

1. Comment out the line
   ```java
   permission java.net.SocketPermission "localhost:1024-", "listen";
   ```
   and replace it with the following line:
   ```java
   permission java.net.SocketPermission ",", "accept, connect, listen, resolve";
   ```

2. Add the following lines:
   ```java
   permission java.util.PropertyPermission "MQJMS_LOG_DIR", "read";
   permission java.util.PropertyPermission "MQJMS_TRACE_DIR", "read";
   permission java.util.PropertyPermission "MQJMS_TRACE_LEVEL", "read";
   permission java.util.PropertyPermission "MQ JAVA INSTALL_PATH", "read";
   permission java.util.PropertyPermission "file.separator", "read";
   permission java.util.PropertyPermission "user.name", "read";
   permission java.util.PropertyPermission "com.ibm.mq.jms.cleanup", "read";
   permission java.lang.RuntimePermission "loadLibrary.*";
   ```

### Notes:

1. You may need to restart VisualAge for Java if you get the error message “Unknown Java Error” after repeated tests.
Changing browser security

2. Make sure that `<install_dir>/java/lib` is in the workspace classpath.

For appletviewer:

Find the policy file for your JDK and make the same changes as for IBM VisualAge for Java. For example, in the IBM Developer Kit for Windows, Java Technology Edition, Version 1.3, the java.policy file is found in the directory `<jdk_install_dir>/jre/lib/security`, where `<jdk_install_dir>` is the directory where the Developer Kit was installed.

For a web browser:

To achieve consistent behavior for applets within different web browsers use the Sun Java plug-in.
1. Install the Sun Java plug-in 1.3.01 or later.
   From this level, Netscape 6 is also supported.
2. Make the same changes to the java.policy file as listed above.
   The policy file is found in `<java_plugin_install_dir>/lib/security`.
3. Make sure that your HTML applet tags are changed to run with the plug-in.
   Download and run the Sun HTML Converter v1.3 to make the necessary changes.

Copying package class files

When a Java program is executed in the context of an applet (which is what is done when appletviewer is executed or a web browser is used), by default the Java program has significant security restrictions applied to it. One of these restrictions is that all environment variables in effect when the applet is launched are ignored. This includes CLASSPATH.

As a result, unless you make the changes described in "Changing browser security settings" on page 419 when an applet is executed, each and every class that it needs must also be available for download from the same location as the applet code itself.

To achieve this on a Windows system, perform the following steps (non-Windows users will need to perform similar tasks):
1. Download and install WINZIP (http://www.winzip.com) or equivalent file unzipping utility
2. Find the files containing the WebSphere MQ Java, or other package, classes that your applet needs.
   For example, WebSphere MQ base Java classes are in a file called `com.ibm.mq.jar` usually found in the `C:\Program Files\IBM\WebSphere MQ\Java\lib` folder.
3. Using the unzipping utility you installed in step 1, extract all the files in the .jar file into the folder that contains your applet.
   For the samples supplied with WebSphere MQ Java, the folder to use is `C:\Program Files\IBM\WebSphere MQ\Tools\Java\base`
   This will result in a sub-folder structure `com\ibm` being created.
4. Run your applet.
Appendix G. Information for SupportPac MA1G

This appendix contains information that is relevant to users of SupportPac MA1G “WebSphere MQ for MVS/ESA™ – WebSphere MQ classes for Java”. MA1G provides support for WebSphere MQ classes for Java from versions of OS/390 not supported by WebSphere MQ Java. It also provides support for CICS and High Performance Java (HPJ).

Users intending to use the WebSphere MQ base Java with CICS Transaction Server for OS/390 should be familiar with:
- Customer Information Control System (CICS) concepts
- Using the CICS Java Application Programming Interface (API)
- Running Java programs from within CICS

Users intending to use VisualAge for Java to develop OS/390 UNIX System Services High Performance Java (HPJ) applications should be familiar with the Enterprise Toolkit for OS/390 (supplied with VisualAge for Java Enterprise Edition for OS/390, Version 2).

Environments supported by SupportPac MA1G

SupportPac MA1G provides support for WebSphere MQ base Java from the following environments:
- OS/390 V2R6 or higher
- Java for OS/390, V1.1.8 or higher
- IBM MQSeries for MVS/ESA, Version 1.2 or higher
- High Performance Java (HPJ)

SupportPac MA1G also provides support for CICS TS1.3 or higher. Support for HPJ in this environment requires OS/390 V2R9 or higher.

SupportPac MA1G does not provide support for JMS.

Obtaining and installing SupportPac MA1G

SupportPac MA1G should be obtained from the WebSphere MQ web site http://www.ibm.com/software/mqseries. Follow links to “Download” and then “SupportPacs” to find the WebSphere MQ Java code.

The following procedure installs the WebSphere MQ classes for Java. The directory used for the installation needs at least 2MB of free storage. In the following, replace “/u/joe/mqm” with the path name of the directory you choose:

1. Remove any previous installation of this product using the following commands in the OpenEdition shell:
   ```
   cd /u/joe
   chmod -fr 700 mqm
   rm -rf mqm
   mkdir mqm
   ```

2. Using FTP binary mode, upload the file ma1g.tar.Z from your workstation to the HFS directory /u/joe/mqm.
Obtaining and installing

3. While in the OpenEdition shell, change to the installation directory /u/joe/mqm.
4. Uncompress and untar the file with the command
tar -xpozf ma1g.tar.Z
5. Set up your CLASSPATH and LIBPATH as described in “Environment variables” on page 8.

Verifying installation using the sample program

To verify installation of MA1G from Unix System Services (USS), follow the instructions in “Verifying with the sample application” on page 14.

To verify installation of MA1G from CICS Transaction Server:
1. Define the sample application program (MQIVP) to CICS.
2. Define a transaction to run the sample application.
3. Put the queue manager name into the file used for standard input.
4. Run the transaction.

The program output is placed in the files used for standard and error output.

Refer to CICS documentation for more information on running Java programs and setting the input and output files.

Features not provided by SupportPac MA1G

SupportPac MA1G provides a subset of features available to other WebSphere MQ base Java applications. In particular, it does not support the ConnectionPooling feature described in Chapter 7, “Writing WebSphere MQ base Java programs” on page 61. The following classes and methods are not supported:

- Classes and Interfaces
  - MQPoolServices
  - MQPoolServicesEvent
  - MQPoolToken
  - MQSimpleConnectionManager
  - MQPoolServicesEventListener
  - MQConnectionManager
  - ManagedConnection
  - ManagedConnectionFactory
  - ManagedConnectionMetaData

- Methods
  - MQEnvironment.getDefaultConnectionManager()
  - MQEnvironment.setDefaultConnectionManager()
  - MQEnvironment.addConnectionPoolToken()
  - MQEnvironment.removeConnectionPoolToken()
  - The six MQQueueManager constructors which allow a ConnectionManager or MQConnectionManager to be specified.

Attempting to use these classes, interfaces or methods will result in compile-time errors or run-time exceptions.
Running WebSphere MQ base Java applications under CICS
Transaction Server for OS/390

To run a Java application as a transaction under CICS, you must:

1. Define the application and transaction to CICS by using the supplied CEDA
   transaction.
2. Ensure that the WebSphere MQ CICS adapter is installed in your CICS system.
   (See WebSphere MQ for z/OS System Setup Guide for details.)
3. Ensure that the JVM environment specified in the DHFJVM parameter of your
   CICS startup JCL (Job Control Language) includes appropriate CLASSPATH
   and LIBPATH entries.
4. Initiate the transaction by using any of your normal processes.

For more information on running CICS Java transactions, refer to your CICS
system documentation.

Restrictions under CICS Transaction Server

In the CICS Transaction Server for OS/390 environment, only the main (first)
thread is allowed to issue CICS or WebSphere MQ calls. It is therefore not possible
to share MQQueueManager or MQQueue objects between threads in this
environment, or to create a new MQQueueManager on a child thread.

Chapter 8, “Environment-dependent behavior” on page 89 identifies some
restrictions and variations which apply to the WebSphere MQ classes for Java
when running against a z/OS & OS/390 queue manager. Additionally, when
running under CICS, the transaction control methods on MQQueueManager are
not supported. Instead of issuing MQQueueManager.commit() or
MQQueueManager.backout(), applications should use the JCICS task
synchronization methods, Task.commit() and Task.rollback(). The Task class is
supplied by JCICS in the com.ibm.cics.server package.
Appendix H. SSL CipherSuites supported by WebSphere MQ

The following table lists the CipherSpecs supported by WebSphere MQ, and their associated CipherSuite names. The CipherSpec name should be specified in the SSLCIPH property of the SVRCONN channel on the queue manager. The CipherSuite name should be specified in MQEnvironment.sslCipherSuite or MQC.SSL_CIPHER_SUITE_PROPERTY of WebSphere MQ base Java, or using the setSSLCipherSuite() method of MQConnectionFactory in JMS, or using the SSLCIPHERSUITE (SCPHS) property from JMSAdmin. The set of supported CipherSuites will vary between JSSE providers; those CipherSuites not supported by the IBM implementation of JSSE are marked with an asterisk.

Table 49. CipherSpecs and matching CipherSuites

<table>
<thead>
<tr>
<th>CipherSpec</th>
<th>CipherSuite</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES_SHA_EXPORT</td>
<td>SSL_RSA_WITH_DES_CBC_SHA</td>
</tr>
<tr>
<td>DES_SHA_EXPORT1024</td>
<td>SSL_RSA_EXPORT1024_WITH_DES_CBC_SHA *</td>
</tr>
<tr>
<td>NULL_MD5</td>
<td>SSL_RSA_WITH_NULL_MD5</td>
</tr>
<tr>
<td>NULL_SHA</td>
<td>SSL_RSA_WITH_NULL_SHA</td>
</tr>
<tr>
<td>RC2_MD5_EXPORT</td>
<td>SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5</td>
</tr>
<tr>
<td>RC4_56_SHA_EXPORT1024</td>
<td>SSL_RSA_EXPORT1024_WITH_RC4_56_SHA *</td>
</tr>
<tr>
<td>RC4_MD5_US</td>
<td>SSL_RSA_WITH_RC4_128_MD5</td>
</tr>
<tr>
<td>RC4_MD5_EXPORT</td>
<td>SSL_RSA_EXPORT_WITH_RC4_40_MD5</td>
</tr>
<tr>
<td>RC4_SHA_US</td>
<td>SSL_RSA_WITH_RC4_128_SHA</td>
</tr>
<tr>
<td>TRIPLE_DES_SHA_US</td>
<td>SSL_RSA_WITH_3DES_EDE_CBC_SHA</td>
</tr>
</tbody>
</table>
Appendix I. JMS exception messages

This section lists most common exceptions that can be generated by WebSphere MQ JMS. It does not include all messages that can be written to a trace file. If you receive an exception message not in this list (except in a trace file), or if the cause seems to be an error in WebSphere MQ JMS, please contact your IBM service representative.

Reading variables in a message
Some messages display text or numbers that vary according to the circumstances giving rise to the message; these are known as message variables. Message variables are indicated in this section by the use of numbers in braces; for example, {0}, {1}, and so on.

---

MQJMS0000 Method {0} has been invoked at an illegal or inappropriate time or if the provider is not in an appropriate state for the requested operation.
Explanation: The normal reason for this exception is that the SSL certificate stores have not been defined. {0} identifies the method that has caused the problem.
User Response: See "Using Secure Sockets Layer (SSL)" on page 199 for more information.

MQJMS0002 JMS Client attempted to set invalid clientId on a connection.
Explanation: An application attempted to set the clientId property of a valid connection to null, or attempted to set the clientId property of an invalid connection.
User Response: The clientId property on a connection can only be set once, only to a non-null value, and only before the connection is used. Ensure that the connection is valid and that the clientId value is not null.

MQJMS0003 Destination not understood or no longer valid.
Explanation: The queue or topic may have become unavailable, the application may be using an incorrect connection for the queue or topic, or the supplied destination is not of the correct type for this method.
User Response: Check that WebSphere MQ is still running and the queue manager is available. Check that the right connection is being used for your queue or topic.

MQJMS0004 JMS Client has given JMS Provider a message selector with invalid syntax.
Explanation: The message selector string is empty or contains an invalid value or syntax.
User Response: Check the linked WebSphere MQ exception reason and completion codes for more information.

MQJMS0005 Unexpected end of stream has been reached when a StreamMessage or BytesMessage is being read.
Explanation: The byte stream being read is shorter than the buffer supplied. This can also be caused by receiving a corrupt StreamMessage or BytesMessage.
User Response: Check the length of buffer supplied. Check system event logs for more information.

MQJMS0006 JMS Client attempts to use a data type not supported by a message or attempts to read data in the wrong type.
Explanation: Wrong data types used to read message property types.
User Response: Check that the message received and the properties to be read are of the type expected.

MQJMS0009 JMS Provider is unable to allocate the resources required for a method.
Explanation: Machine resources may be overloaded, the linked exception may give further information.
User Response: Check system resources and load.

MQJMS0010 Operation invalid because a transaction is in progress.
User Response: Wait for the current transaction to complete. See the linked WebSphere MQ exception for further information.

MQJMS0011 Call to Session.commit resulted in a rollback of the current transaction.
Explanation: The transaction failed resulting in a call to rollback to a safe state. See the linked exception for more information.
MQJMS1000 Failed to create JMS message.
Explanation: Invalid message type or properties were specified when creating a base message.
User Response: Check the linked WebSphere MQ exception Reason and Completion code for more information.

MQJMS1001 Unknown acknowledgement mode {0}.
Explanation: Invalid or no parameter {0} set for acknowledgement mode on the session.
User Response: See "Session" on page 345 for the possible values for acknowledgement mode.

MQJMS1004 Connection closed.
Explanation: An operation such as start() or stop() has been called on a connection that is already closed.
User Response: Ensure that the connection is open before performing any operation.

MQJMS1005 Unhandled state transition from {0} to {1}.
Explanation: The state transition is not valid, see log for more information.
User Response: Check the linked WebSphere MQ exception reason and completion code.

MQJMS1006 Invalid value for {0}: {1}
Explanation: Invalid value {1} for property {0}.
User Response: Check the linked WebSphere MQ exception reason and completion code. See Table 11 on page 46 for a list of valid values for this property.

MQJMS1008 Unknown value of transportType: {0}.
Explanation: The value given for transportType could not be used. {0} shows the invalid value.
User Response: See Table 11 on page 46 for a list of valid values for this property.

MQJMS1010 Not implemented.
Explanation: The function requested is not implemented. This can be thrown by message acknowledgement, if the session or acknowledgement parameters are invalid or incorrect.

MQJMS1011 Security credentials cannot be specified when using MQ bindings.
Explanation: The RRS queue does not support a client connection, and bindings connections do not support the specification of security credentials.
User Response: Ensure that you do not try to specify security credentials when using a bindings connection.

MQJMS1012 No message listener.
Explanation: The message listener has stopped or was never started.
User Response: Restart the message listener and retry.

MQJMS1013 Operation invalid while session is using asynchronous delivery.
Explanation: You cannot perform the requested operation while the session is actively using asynchronous delivery mode.

MQJMS1014 Operation invalid for identified producer.
Explanation: The QueueSender.send method has been performed on an identified QueueSender, which contradicts the JMS specification.

MQJMS1015 Unknown value of targetClient: {0}.
Explanation: The value for the targetClient property set by the application for this destination is not recognized by WebSphere MQ JMS.
User Response: See "Sending a message" on page 193 for valid values of the targetClient property.

MQJMS1017 Non-local MQ queue not valid for receiving or browsing.
Explanation: An attempt was made to perform an inappropriate operation on a non-local queue.
User Response: Check the queue properties.

MQJMS1018 No valid connection available.
Explanation: The queue is busy, there are network problems or a connection has not been defined for the object.
User Response: Create a valid connection for this operation.

MQJMS1019 Invalid operation for non-transacted session.
Explanation: Commit is not allowed on a session that is not transacted.
User Response: Check the linked IllegalStateException for more information. See "Session" on page 345 for further information.

MQJMS1020 Invalid operation for transacted session.
Explanation: Invalid acknowledgement mode for a transacted session. Acknowledge and Recover are not valid operations in transacted sessions.
User Response: See "Session" on page 345 for further information.
MQJMS1021 Recover failed: unacknowledged messages might not get redelivered.
Explanation: The system was unable to recover from a failure.
User Response: Contact your IBM representative.

MQJMS1022 Failed to redirect message.
Explanation: Unable to send message onto next destination.
User Response: Check trace settings and trace file existence. See "Tracing programs" on page 36 for more information on Trace.

MQJMS1023 Rollback failed.
Explanation: The system was unable to rollback to a safe state.
User Response: Restart the session, and check all required resources are available.

MQJMS1024 Session closed.
Explanation: The session timed out or was closed; or either the connection or the queue manager was closed, implicitly closing the session.
User Response: Check log settings are correct.

MQJMS1025 Failed to browse message.
Explanation: No message was available for browsing. There may be no message on the Queue.
User Response: Check the message type. Contact your IBM representative if there appears to be an internal error.

MQJMS1026 ExceptionListener threw exception: {0}.
User Response: Check linked exceptions for further information.

MQJMS1027 Failed to reconstitute destination from {0}.
Explanation: A message has been received which contains invalid destination information in the RFH2 header.
User Response: Ensure that the name strings exist. See "Tracing programs" on page 36 for more information on Trace.

MQJMS1028 Element name is null.
Explanation: A null name string was passed to one of the "get value by name" methods of MapMessage.
User Response: Ensure that all name strings being used to retrieve values are non-null.

MQJMS1029 Property name is null.
Explanation: The itemExists method of MapMessage was invoked with a null item name; or a null name string was used as an argument to a method which retrieves property values by name from a JMS message.
User Response: Ensure that the name strings indicated do not have null values.

MQJMS1031 An internal error has occurred. Please contact your system administrator.
Explanation: Internal Error.
User Response: Contact your IBM representative.

MQJMS1032 close() failed because of {0}.
Explanation: Internal Error. {0} indicates the reason for the error.
User Response: Contact your IBM representative.

MQJMS1033 start() failed because of {0}.
Explanation: {0} indicates why the session failed to start.
User Response: Contact your IBM representative.

MQJMS1034 MessageListener threw: {0}.
Explanation: A problem has occurred. {0} shows a symptom of the error.
User Response: Contact your IBM representative.

MQJMS1035 Cannot transmit non-MQ JMS messages.
Explanation: Wrong message type used. This is a possible internal problem.
User Response: Contact your IBM representative.

MQJMS1036 Failed to locate resource bundle.
Explanation: The resource bundle is either not present or not in the application’s classpath.
User Response: Check that the classpath includes the location of property files.

MQJMS1038 Failed to log error.
Explanation: Log settings may be incorrect, see the linked LogException.
User Response: Check log settings are correct.

MQJMS1039 Trace file does not exist
Explanation: Trace settings may be incorrect.
User Response: Check trace settings and trace file existence. See "Tracing programs" on page 36 for more information on Trace.
MQJMS1040 • MQJMS1055

MQJMS1040 Failed to connect to Trace stream.
Explanation: Trace settings may be incorrect.
User Response: See “Tracing programs” on page 36 for more information on Trace.

MQJMS1041 Failed to find system property {0}.
Explanation: The system property specified in {0} does not exist or was not found in the application’s classpath.
User Response: Check the classpath settings and the product installation.

MQJMS1042 Invalid delivery mode.
Explanation: Either an invalid value was specified for the delivery mode of a message producer, or an invalid delivery mode value was specified when publishing a message.
User Response: Check to ensure that the value specified is a valid enumeration for delivery mode.

MQJMS1043 JNDI failed due to {0}.
Explanation: {0} gives further information.
User Response: Check settings for LDAP, JNDI, and in the JMSAdmin.config file.

MQJMS1044 String is not a valid hexadecimal number - {0}.
Explanation: An attempt was made to specify a group ID or correlation ID which starts with the prefix “ID:” but is not followed by a well-formed hex value; or an attempt was made to receive a message which contains an RFH2 property of type bin.hex that does not have a well-formed hex value.
User Response: Ensure that a valid hex value always follows the “ID:” prefix when setting group ID or correlation ID values. Ensure that any RFH2 headers generated by non-JMS applications are well-formed.

MQJMS1045 Number outside of range for double precision S/390 Float {0}.
Explanation: This is a z/OS & OS/390 specific error.

MQJMS1046 The character set {0} is not supported.
Explanation: An attempt was made to send or receive a map message, stream message or text message whose body is encoded using a character set not supported by the JVM. In the case of text messages, this exception may be thrown when the body of the message is first queried, rather than at receive time.
User Response: Only set character encoding on a message to values known to be available to the receiving application.

MQJMS1047 The map message has an incorrect format.
Explanation: A map message was received, but its RFH2 header information is badly formatted.
User Response: Ensure any non-JMS applications are building well-formed RFH2 header information for inclusion in map messages.

MQJMS1048 The stream message has an incorrect format.
Explanation: A stream message was received, but its RFH2 header information is badly formatted.
User Response: Ensure any non-JMS applications are building well-formed RFH2 header information for inclusion in stream messages.

MQJMS1049 The JMS client attempted to convert a byte array to a String.
Explanation: Attempting to receive a byte array from a stream message using the readString method.
User Response: Either use the appropriate method to receive the data, or format the data placed into the stream message correctly.

MQJMS1050 The MQRFH2 header has an incorrect format.
Explanation: Receiving a message with a badly formed RFH2 header.
User Response: Ensure that any non-JMS applications building messages with RFH2 headers create well-formed RFH2 headers.

MQJMS1053 Invalid UTF-16 surrogate detected {0}.
Explanation: An invalid UTF-16 surrogate character has been encountered as part of a topic name or RFH2 property.
User Response: Ensure that, when specifying UTF-16, topic names or RFH2 properties are well-formed.

MQJMS1054 Invalid XML escape sequence detected {0}.
Explanation: An invalid XML escape sequence has been encountered in the RFH2 header of a received message.
User Response: Ensure that only valid XML escape sequences are placed into any RFH2 headers built by non-JMS applications.

MQJMS1055 The property or element in the message has incompatible datatype {0}.
Explanation: Attempting to retrieve a property from a JMS message using a accessor method which specifies an incompatible type. For example, attempting to retrieve an integer property using the getBooleanProperty method.
User Response: Use an accessor method defined by
the JMS specification as being able to retrieve property values of the required type.

MQJMS1056 Unsupported property or element datatype [0].
Explanation: This error is caused by one of the following:
1. Attempting to set a property of a JMS message using an object which is not one of the supported types.
2. Attempting to set or receive a message whose RFH2 contains a element representing a property which does not have a valid type associated with it.
User Response: Ensure that when setting message properties, an object type described as being valid in the JMS specification is used. If this exception occurs when receiving a message containing an RFH2 header sent by a non-JMS application, ensure that the RFH2 header is well-formed.

MQJMS1057 Message has no session associated with it.
Explanation: An attempt was made to acknowledge a message on a session which is not in an open state.
User Response: Ensure that the session associated with the message has been correctly opened. Check that the session has not been closed.

MQJMS1058 Invalid message property name: [0].
Explanation: Attempting to set a property that either does not have a valid property name, or is not a settable property.
User Response: Ensure that the property name used is a valid property name in accordance with the JMS specification. If the property name refers to a JMS or provider-specific extension property, ensure that this property is settable.

MQJMS1059 Fatal error - UTF8 not supported.
Explanation: The Java runtime environment you are using does not support the UTF-8 character encoding. JMS requires support for this encoding to perform some operations.
User Response: Consult the documentation and or provider of your Java runtime environment to determine how to obtain support for the UTF-8 character encoding.

MQJMS1060 Unable to serialize object.
Explanation: An attempt has been made to serialize an ObjectMessage which contains a non-serializable object.
User Response: Ensure that ObjectMessages only contain serializable objects. If the object placed inside an ObjectMessage references other objects, these must also be serializable.

MQJMS1061 Unable to deserialize object.
Explanation: De-serialization of an ObjectMessage failed.
User Response: Ensure that the ObjectMessage being received contains valid data. Ensure that the class files representing object data contained within the ObjectMessage are present on the machine deserializing the ObjectMessage. If the object contained within the ObjectMessage references other objects, ensure that these class files are also present.

MQJMS1066 Invalid message element name: [0].
Explanation: Attempting to set a message property using either an invalid property name, or the name of a property which cannot have its value set.
User Response: Ensure that the property name specified conforms to the JMS specification. If the property name supplied is that of a JMS property, or a vendor specific extension, ensure that this property name is settable.

MQJMS1067 Timeout invalid for MQ.
Explanation: An attempt was made to invoke the receive method on either a QueueReceiver or TopicSubscriber method, specifying a long timeout value which is not valid.
User Response: Ensure the timeout value specified is not negative and not greater than the value of Integer.MAX_VALUE.

MQJMS1068 Failed to obtain XAResource.
Explanation: JMS failed to create an XA Queue resource due to an error.
User Response: See the linked XAException for more information.

MQJMS1072 Could not inquire upon queue manager name.
Explanation: In createConnectionConsumer() or createDurableConnectionConsumer(), JMS could not determine the name of the queue manager.
User Response: Check your queue manager error logs for problems which may cause this. If there are no other error conditions, contact your IBM representative.

MQJMS1073 Specified MQ Queue is neither a QLOCAL nor a QALIAS.
Explanation: createConnectionConsumer() was called, but a queue of the wrong type was specified. Only QALIAS and QLOCALs can be used with the ConnectionConsumer feature.
User Response: Specify a queue of the correct type.
MQJMS1074 • MQJMS1087

MQJMS1074 Unable to process null message.
Explanation: Internal error in WebSphere MQ JMS.
User Response: Contact your IBM representative.

MQJMS1075 Error writing dead letter header.
Explanation: JMS attempted to requeue a message to the dead letter queue, but could not construct a dead letter header.
User Response: Use the linked exception to determine the cause of this error.

MQJMS1076 Error reading dead letter header.
Explanation: JMS attempted to interpret a message with a dead letter header, but encountered a problem.
User Response: Use the linked exception to determine the cause of this error.

MQJMS1077 Connection and Destination mismatch.
Explanation: An operation was requested, but the Destination class is incompatible with the Connection class. Topics cannot be used with QueueConnections and Queues cannot be used with TopicConnections.
User Response: Supply a suitable Destination. This may represent an internal error condition in JMS; in this case contact your IBM representative.

MQJMS1078 Invalid Session object.
Explanation: The JMS ConnectionConsumer feature attempted to deliver a batch of messages to a Session. However, the Session contained in the ServerSession object returned by the ServerSessionPool was not a WebSphere MQ JMS Session.
User Response: This is an error in the ServerSessionPool. If you have supplied a ServerSessionPool, check its behavior. In a J2EE application server, this may represent an error in the application server; in which case, refer to your application server’s documentation.

MQJMS1079 Unable to write message to dead letter queue.
Explanation: JMS attempted to requeue a message to the dead letter queue, but failed.
User Response: Use the linked exception to determine the cause of this error. If there is no linked exception, check that the queue manager has a defined dead letter queue. Once JMS has sent a message to the dead letter queue, the reason code stored in the message’s DLH can be used to determine why the message was dead-lettered.

MQJMS1080 No Backout-Requeue queue defined.
Explanation: JMS encountered a message which has been backed out more than the queue’s Backout Threshold, however the queue doesn’t have a Backout-Requeue queue defined.
User Response: Define a Backout-Requeue queue for the queue, or set the Backout Threshold to zero to disable poison message handling. Investigate the repeated backouts.

MQJMS1081 Message requeue failed.
Explanation: JMS found an error when requeuing a message which has been backed out more than the queue’s Backout Threshold.
User Response: Use the linked exception to determine the cause of this error. Investigate the repeated backouts.

MQJMS1082 Failure while discarding message.
Explanation: JMS encountered an error while discarding a message, or while generating an exception report for a message to be discarded.
User Response: Use the linked exception to determine the cause of this error.

MQJMS1083 Invalid message batch size (must be >0).
Explanation: An invalid batch size parameter was passed to createConnectionConsumer() or createDurableConnectionConsumer().
User Response: Set a batch size greater than zero. In a J2EE application server, this may represent an error in the application server. Refer to your application server’s documentation.

MQJMS1084 Null ServerSessionPool has been provided.
Explanation: The ServerSessionPool specified on createConnectionConsumer() or createDurableConnectionConsumer() was null.
User Response: Set an appropriate ServerSessionPool. In a J2EE application server, this may represent an error in the application server. Refer to your application server’s documentation.

MQJMS1085 Error writing RFH.
Explanation: JMS attempted to construct an RFH message header, but encountered an error.
User Response: Use the linked exception to determine the cause of this error.

MQJMS1086 Error reading RFH.
Explanation: JMS encountered an error while parsing an RFH message header.
User Response: Use the linked exception to determine the cause of this error.

MQJMS1087 Unrecognized or invalid RFH content.
Explanation: JMS expected to find an RFH message header, but found it to be missing, malformed or lacking required data.
User Response: Investigate the source of the message. This may represent an internal error condition in JMS; in this case, contact your IBM representative.
Appendix I. JMS exception messages 433

MQJMS1088 Mixed-domain consumers acting on the same input is forbidden.
Explanation: A point-to-point ConnectionConsumer is using the subscriber queue of a publish/subscribe ConnectionConsumer.
User Response: Do not attempt to access subscriber queues using the point-to-point ConnectionConsumer facilities of JMS. Check your TopicConnectionFactory and Topic objects to make sure they are not using a QLOCAL intended for use by point-to-point applications as a subscriber queue.

MQJMS1089 Exception occurred reading message body: {0}.
Explanation: JMS encountered an exception while reading data from a message. The message being read is likely to be a response message from the publish/subscribe broker.
User Response: Use the linked exception to determine the cause of this error.

MQJMS2000 Failed to close MQ queue.
Explanation: JMS attempted to close a WebSphere MQ queue, but encountered an error. The queue may already be closed, or another thread may be performing an MQGET while close() is called.
User Response: Use the linked exception to determine the cause of this error. You may be able to perform the close() later.

MQJMS2001 MQQueue reference is null.
Explanation: JMS attempted to perform some operation on a null MQQueue object.
User Response: Check your system setup, and that all required queue names have been specified. This may represent an internal error condition in JMS; in this case, contact your IBM representative.

MQJMS2002 Failed to get message from MQ queue.
Explanation: JMS attempted to perform an MQGET; however WebSphere MQ reported an error.
User Response: Use the linked exception to determine the cause of this error.

MQJMS2003 Failed to disconnect queue manager.
Explanation: JMS encountered an error while attempting to disconnect.
User Response: Use the linked exception to determine the cause of this error.

MQJMS2004 MQQueueManager reference is null.
Explanation: JMS attempted to perform an operation on a null MQQueueManager object.
User Response: Check that the relevant object has not been closed. This may represent an internal error condition in JMS; in this case, contact your IBM representative.

MQJMS2005 Failed to create MQQueueManager for {0}.
Explanation: JMS could not connect to a queue manager. {0} gives the name of the queue manager.
User Response: Use the linked exception to determine the cause of this error. Check the queue manager is running and, if using client attach, that the listener is running and the channel, port and hostname are set correctly. If no queue manager name has been specified, check that the default queue manager has been defined.

MQJMS2006 MQ problem: {0}.
Explanation: JMS encountered some problem with WebSphere MQ. {0} describes the problem.
User Response: Use the included text and linked exception to determine the cause of this error.

MQJMS2007 Failed to send message to MQ queue.
Explanation: JMS attempted to perform an MQPUT; however WebSphere MQ reported an error.
User Response: Use the linked exception to determine the cause of this error.

MQJMS2008 Failed to open MQ queue.
Explanation: JMS attempted to perform an MQOPEN; however WebSphere MQ reported an error.
User Response: Use the linked exception to determine the cause of this error. Check that the specified queue and queue manager are defined correctly.

MQJMS2009 MQQueueManager.commit() failed.
Explanation: JMS attempted to perform an MQCMIT; however WebSphere MQ reported an error.
User Response: Use the linked exception to determine the cause of this error.

MQJMS2010 Unknown value for MQ queue definitionType: {0}.
Explanation: Unable to delete the temporary queue as the definitionType is not valid.
User Response: Check the setting of definitionType.

MQJMS2011 Failed to inquire MQ queue depth.
Explanation: WebSphere MQ JMS is unable to tell how many messages are on the queue.
User Response: Check that the queue and queue manager are available.

MQJMS2012 XACLOSE failed.
Explanation: See linked XAException for more details.

MQJMS2013 Invalid security authentication supplied for MQQueueManager.
Explanation: Bad username or password or both. In bindings mode, a supplied user ID does not match the logged in user ID.
MQJMS3000  •  MQJMS3020

MQJMS3000 Failed to create a temporary queue from {0}.
Explanation: Creation of temporary queue failed.
User Response: See linked exception for more information. Check that the TemporaryModel parameter against the QueueConnectionFactory is set to a valid model queue.

MQJMS3001 Temporary queue already closed or deleted.
Explanation: Temporary queue no longer exists or is equal to null.
User Response: Check to see that the queue has been created, and that the session is still available.

MQJMS3002 Temporary queue in use.
Explanation: Another program is using the queue.
User Response: Wait for the temporary queue to become free or create another.

MQJMS3003 Cannot delete a static queue.
Explanation: Attempted to delete a queue of type static, where a temporary queue was expected.
User Response: Check the expected queue type for deletion.

MQJMS3004 Failed to delete temporary queue.
Explanation: The temporary queue may be persistent or busy.
User Response: See the linked WebSphere MQ exception for more details. Wait if the queue is busy, or delete the queue manually if it is persistent.

MQJMS3005 Publish/Subscribe failed due to {0}.
Explanation: General error. {0} shows the reason.
User Response: Check the linked WebSphere MQ exception reason and completion codes for more information. It is possible that the broker and queue manager versions are incompatible.

MQJMS3006 Topic reference is null.
Explanation: Topic supplied to a publisher is null.
User Response: Use non-null values.

MQJMS3008 Failed to build command {0}.
Explanation: Broker message command parameters incorrect.
User Response: Check linked exception for cause.

MQJMS3009 Failed to publish command to MQ queue.
Explanation: Invalid command, queue unavailable or broker errors.
User Response: Check linked WebSphere MQ exception reason and completion codes for more information.

MQJMS3010 Failed to build publish message.
Explanation: Unable to build the base message for the broker.
User Response: See the linked WebSphere MQ Exception for further details. Check settings and parameters are all correct. See Chapter 11, “Programming publish/subscribe applications” on page 203 for more information.

MQJMS3011 Failed to publish message to MQ queue.
Explanation: See linked Exception for more information.
User Response: Check settings and parameters are all correct. See Chapter 11, “Programming publish/subscribe applications” on page 203 for more information.

MQJMS3013 Failed to store admin. entry.
Explanation: An add to the admin or status queue failed due to duplication or some other error. See any linked exception for more information.
User Response: Check for duplicates and retry.

MQJMS3014 Failed to open subscriber queue {0}.
User Response: See linked exception for more information.

MQJMS3017 Failed to delete subscriber queue {0}.
Explanation: {0} gives the queue name. See linked exception for more information.

MQJMS3018 Unknown durable subscription {0}.
Explanation: Could not locate the given subscription. For example, during an unsubscribe request.

MQJMS3020 TemporaryTopic out of scope.
Explanation: The current connection ID does not match the connection that created the temporary topic.
MQJMS3021 Invalid subscriber queue prefix {0}.
Explanation: The name specified is not valid. It must begin with SYSTEM.JMS.D for durable subscriptions or SYSTEM.JMS.ND for non-durable subscriptions.

MQJMS3022 Durable re-subscribe must use same subscriber queue; specified: {0}, original: {1}.
Explanation: {0} and {1} show the differing queue names. Unable to get a subscription due to wrong queue manager or queue.
User Response: Check settings.

MQJMS3023 Subscription has an active TopicSubscriber.
Explanation: Can be caused by a queue open problem or if a subscription already exists on the JVM. If running in WebSphere Application Server there can be other causes. See linked exception, if set, for more information.
User Response: Check settings.

MQJMS3024 Illegal use of uninitialized clientId.
Explanation: The clientId in the connection has not been set.
User Response: Set the clientId before attempting to perform any operation.

MQJMS3025 TemporaryTopic in use.
Explanation: Something else is currently using the topic.
User Response: Wait until the topic is free or create another topic. Ensure subscribers de-register when finished.

MQJMS3026 QueueSender is closed.
User Response: Open or re-open the queue sender if required.

MQJMS3027 Local transactions not allowed with XA sessions.
Explanation: Unexpected error.
User Response: Contact your IBM representative.

MQJMS3028 TopicPublisher is closed.
User Response: Open or reopen the topic publisher if required.

MQJMS3029 Enlist failed (see linked Exception).
Explanation: JTSXA.enlist threw an exception that was caught by JMS.
User Response: Check the linked WebSphere MQ Exception reason and completion codes for more information. Contact your IBM representative.

MQJMS3031 clientId cannot be set after connection has been used.
Explanation: The clientId of a connection can be set only once and only before the connection is used.
User Response: Set the clientId before using the connection.

MQJMS3032 Resetting the clientId is not allowed.
Explanation: The clientId of a connection can be set only once and only before the connection is used.
User Response: Set the clientId before using the connection.

MQJMS3033 QueueReceiver is closed.
User Response: Open or reopen the receiver.

MQJMS3034 TopicSubscriber is closed.
User Response: Open or reopen the TopicSubscriber.

MQJMS3036 Broker side message selection valid only when using WebSphere MQ Integrator broker.
Explanation: Broker version and message selection are not consistent.
User Response: Ensure the broker version has been set in the ConnectionFactory. Use the method setBrokerVersion(JMSC.MQJMS_BROKER_V2) on the ConnectionFactory for WebSphere MQ Integrator or WebSphere MQ Event Broker.

MQJMS3037 Message Producer is closed.
Explanation: Either or both of the session and connection are closed.
User Response: Check to ensure that the session and connection are both available.

MQJMS3038 Message Consumer is closed.
Explanation: Either or both of the session and connection are closed.
User Response: Check to ensure that the session and connection are both available.

MQJMS3039 Illegal use of null name.
Explanation: Durable connection consumers must be named.
User Response: Check for null values.

MQJMS3040 Invalid broker control message content: {0}.
Explanation: {0} explains further.
User Response: Check the broker documentation for message content information.
MQJMS3042 Unrecognized message from Pub/Sub Broker.
Explanation: The message received from the broker was not of a recognized or supported format.
User Response: Check that the broker you are using is supported and refer to broker documentation for settings.

MQJMS3044 Cleanup level of NONE requested.
Explanation: Cleanup requested while cleanupLevel set to NONE.
User Response: Set cleanupLevel property to an appropriate value.

MQJMS3047 Subscription store type not supported by queue manager.
Explanation: Not an MQSPIQueue manager or deferred message not supported.
User Response: Possible incompatibility between queue manager version and broker. Specify a different type of subscription store or upgrade the queue manager. See “Subscription stores” on page 217 for more information.

MQJMS3048 Incorrect subscription store type.
Explanation: Subscription store changed within TopicConnection.
User Response: Contact your IBM representative.

MQJMS3049 Incorrect subscription type for this subscription store.
Explanation: TopicSubscriber was created with a different SUBSTORE setting than current TopicConnection.
User Response: Ensure TopicSubscribers are only used during the lifetime of their parent TopicConnection. See “Subscription stores” on page 217 for more information.

MQJMS4009 Context is not empty.
Explanation: Error deleting Context due to context not being empty.
User Response: Remove context contents before trying delete.

MQJMS4096 Binding non-administerable or not found.
Explanation: Administration object or service is equal to null.
User Response: Contact your IBM representative.

MQJMS4097 Context not found.
Explanation: Could not find a context to match the name given.
User Response: Ensure the correct context name is specified.

MQJMS4104 Object is inactive, so cannot perform directory operations.
Explanation: The JNDI service is inactive.
User Response: See Chapter 5, “Using the WebSphere MQ JMS administration tool” on page 39 for JMSAdmin and JNDI information.

MQJMS4106 Object is not a WebSphere MQ JMS administered object.

MQJMS4111 Unable to create context.
Explanation: Administration service failed.
User Response: Check LDAP and JNDI settings.

MQJMS4112 Unable to create a valid object, please check the parameters supplied.
Explanation: Consistency check failed.
User Response: Contact your IBM representative.

MQJMS4113 Unable to bind object.
Explanation: Administration service bind or copy or move operation failed.
User Response: Contact your IBM representative.

MQJMS4115 An invalid name was supplied.
Explanation: JMSAdmin error. An invalid name was supplied when trying to delete a context.
User Response: Refer to Chapter 5, “Using the WebSphere MQ JMS administration tool” on page 39 for more about using JMSAdmin.

MQJMS4121 Cannot open configuration file.
Explanation: Configuration file may not exist.
User Response: Check MQ_JAVA_INSTALL_PATH environment variable exists and points to the installation directory of the base Java classes.

MQJMS4127 Invalid property in this context.
Explanation: JMSAdmin object value is invalid in the current context.
User Response: See Chapter 5, “Using the WebSphere MQ JMS administration tool” on page 39 for more about JMSAdmin.

MQJMS4130 Context not found or unremovable.
Explanation: The specified child context could not be deleted.
User Response: Ensure the correct context name was specified.
MQJMS4131  Expected and actual object types do not match.
Explanation: Requested and retrieved objects are of different types.
User Response: Check that you have specified the correct object type.

MQJMS4132  Client-bindings attribute clash.
Explanation: Client properties specified for a bindings connection.
User Response: Ensure theConnectionFactory properties are correct.

MQJMS4133  ExitInit string supplied without Exit string.
Explanation: ExitInit string supplied but Exit is not set.
User Response: Set appropriate exit, or unset ExitInit string.

MQJMS4137  Unable to create a WebSphere MQ specific class. The WebSphere MQ classes may not have been installed or added to the classpath.
User Response: Check WebSphere Application Server installation and classpath variable.

MQJMS4139  Invalid authentication type supplied - using ‘none’.
Explanation: AdminService JNDI initialization parameters contain an invalid authorization scheme, so “none” is used as the value instead.
User Response: See Chapter 5, “Using the WebSphere MQ JMS administration tool” on page 39 for more information.

MQJMS5053  *** No broker response. Please ensure that the broker is running. ***
Explanation: Possible causes:
1. Broker is not running.
2. You are using BrokerVersion=V2 in your TopicConnectionFactory with the MQSeries Publish/Subscribe broker, which does not support this.
3. The Broker has rejected the Publication or Subscription and placed it on the SYSTEM.DEAD.LETTER.QUEUE
User Response: Ensure that your broker is running. Check the system event log for broker error messages. Check that the broker supports the BrokerVersion you are using. Check the SYSTEM.DEAD.LETTER.QUEUE for rejected messages.

MQJMS5054  The broker appears to be running, but the message did not arrive.
Explanation: Thrown by Installation Verification Test when the subscriber fails to receive the published message.
User Response: Check that you have set up the broker correctly. Check system event logs for broker error messages. Check the SYSTEM.DEAD.LETTER.QUEUE for messages rejected by the broker.

MQJMS5060  Unable to connect to queue manager.
Explanation: Thrown by Installation Verification Test.
User Response: Check that the queue manager is running and that its name is specified correctly in the IVTTest parameters.

MQJMS5061  Unable to access broker control queue on queue manager.
User Response: Check that the control queue exists. The default name is SYSTEM.BROKER.CONTROL.QUEUE.

MQJMS6040  Invalid socket family name: [0].
Explanation: An invalid socket family name was given to an instance of IMBSocketFactory. [0] shows the bad name.
User Response: Contact your IBM representative.

MQJMS6041  An exception occurred while attempting to load socket factory class [0], exception: <1/1>.
Explanation: Either a ClassNotFoundException, an InstantiationException or an IllegalAccessException occurred while trying to load a particular IMBSocketFactory. [0] gives the name of the class.
User Response: Contact your IBM representative.

MQJMS6059  An exception occurred while loading the minimal client security implementation.
User Response: Contact your IBM representative.

MQJMS6060  An unexpected exception in minimal client, exception = [0].
Explanation: An unusual or unexpected exception occurred at the minimal client. [0] gives more details.
User Response: Contact your IBM representative.

MQJMS6061  A specified topic was malformed, topic = [0].
Explanation: [0] gives the name of the malformed topic.
User Response: See “Using topics” on page 210 for more information.
<table>
<thead>
<tr>
<th>MQJMS6062</th>
<th>EOF was encountered while receiving data in the minimal client.</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Response:</td>
<td>Contact your IBM representative.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6063</th>
<th>The broker indicated an error on the minimal client connection.</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Response:</td>
<td>Refer to JMS or broker documentation. Contact your IBM representative.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6064</th>
<th>Connector.send was called with an illegal message value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explanation:</td>
<td>Connector.send was called with an illegal message value.</td>
</tr>
<tr>
<td>User Response:</td>
<td>See Chapter 7, “Writing WebSphere MQ base Java programs” on page 61 for more information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6065</th>
<th>An illegal value was encountered for a field, value = [0].</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explanation:</td>
<td>[0] shows the illegal value.</td>
</tr>
<tr>
<td>User Response:</td>
<td>See Table 37 on page 399 for a list of properties and their possible values.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6066</th>
<th>An unexpected internal error occurred in the minimal client.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explanation:</td>
<td>Internal problem.</td>
</tr>
<tr>
<td>User Response:</td>
<td>Contact your IBM representative.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6067</th>
<th>A bytes message operation was requested on something that is not a bytes message.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explanation:</td>
<td>The wrong message type was found.</td>
</tr>
<tr>
<td>User Response:</td>
<td>Check message type before performing type specific operations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6068</th>
<th>A text message operation was requested on something that is not a text message.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explanation:</td>
<td>The wrong message type was found.</td>
</tr>
<tr>
<td>User Response:</td>
<td>Check message type before performing type specific operations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6069</th>
<th>A stream message operation was requested on something that is not a stream message.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explanation:</td>
<td>The wrong message type was found.</td>
</tr>
<tr>
<td>User Response:</td>
<td>Check message type before performing type specific operations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6070</th>
<th>A map message operation was requested on something that is not a map message.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explanation:</td>
<td>The wrong message type was found.</td>
</tr>
<tr>
<td>User Response:</td>
<td>Check message type before performing type specific operations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6071</th>
<th>The broker sent an invalid message during authentication.</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Response:</td>
<td>See Chapter 11, “Programming publish/subscribe applications” on page 203 and the broker documentation for more information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6072</th>
<th>The broker requested an unavailable protocol during authentication.</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Response:</td>
<td>See Chapter 11, “Programming publish/subscribe applications” on page 203 and the broker documentation for more information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6073</th>
<th>Minimal client connection rejected because of authentication failure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Response:</td>
<td>See Chapter 11, “Programming publish/subscribe applications” on page 203 and the broker documentation for more information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6074</th>
<th>No QOP available in the minimal client.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explanation:</td>
<td>Indicates that QOP is not implemented in the current version of the minimal client.</td>
</tr>
<tr>
<td>User Response:</td>
<td>Contact your IBM representative.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6078</th>
<th>An attempt was made to write an invalid object type of class [0].</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explanation:</td>
<td>[0] identifies the invalid object’s class.</td>
</tr>
<tr>
<td>User Response:</td>
<td>See Chapter 11, “Programming publish/subscribe applications” on page 203 and the broker documentation for more information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6079</th>
<th>An exception occurred while attempting to load thread pooling support, exception = [0].</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explanation:</td>
<td>An exception was caught while attempting to load thread pooling support in the JMS client. Parameter [0] will give details of the exception.</td>
</tr>
<tr>
<td>User Response:</td>
<td>Contact your IBM representative.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6081</th>
<th>An attempt was made to read from a Stream message before a previous read has completed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explanation:</td>
<td>Internal error.</td>
</tr>
<tr>
<td>User Response:</td>
<td>Contact your IBM representative.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6083</th>
<th>An exception occurred while initializing a thread pool instance, exception = [0].</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explanation:</td>
<td>A SocketThreadPoolException was caught while initializing a thread pool instance in the JMS client. [0] gives details of the exception.</td>
</tr>
<tr>
<td>User Response:</td>
<td>Contact your IBM representative.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MQJMS6085</th>
<th>No ExceptionListener has been set.</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Response:</td>
<td>Create an ExceptionListener.</td>
</tr>
</tbody>
</table>
MQJMS6088 The client-side connection monitor is terminating.
User Response: Restart the connection.

MQJMS6090 Attempted to synchronously receive on a MessageConsumer for which a listener is active.
Explanation: MessageConsumer.receive() was called but a message listener is already active on the connection.
User Response: See Chapter 7, “Writing WebSphere MQ base Java programs” on page 61 for more information.

MQJMS6091 An InterruptedException occurred when starting or stopping delivery on the connection, exception = [0].
Explanation: Parameter [0] gives details of the exception.
User Response: Restart the connection.

MQJMS6096 A JMSPriority level of [0] is outside the range specified in JMS.
Explanation: Parameter [0] gives the value that is in error.
User Response: See Table 37 on page 399 for valid values.

MQJMS6097 The specified JMSMessageID, [0], is invalid.
Explanation: Incorrect syntax was used to specify a message ID in Message.setJMSMessageID. The correct syntax is ID:[0-9]+.
User Response: Check parameters. See Chapter 12, “JMS messages” on page 225 for more information on message IDs.

MQJMS6105 No more client parameter changes allowed.
Explanation: An attempt was made to set a SessionConfig parameter when no more changes are allowed. Internal error.
User Response: Contact your IBM representative.

MQJMS6106 An exception occurred when initializing parameter [0], exception [1].
Explanation: [0] identifies the failing parameter and [1] the caught exception.
User Response: Contact your IBM representative.

MQJMS6115 An exception occurred while creating the TopicConnection, exception [0].
Explanation: [0] gives details of the exception.
User Response: Contact your IBM representative.

MQJMS6116 This operation is not permitted on an entity that is closed.
Explanation: An operation was requested on a closed publisher, session, or connection.
User Response: Ensure that the publisher, session, or connection is open before trying this operation.

MQJMS6117 The [0] implementation of Topic is not supported.
Explanation: The Topic instance passed to a TopicPublisher or TopicSession method has an unsupported run-time implementation. [0] gives the class name of the unsupported implementation.
User Response: See “Using topics” on page 210 for more information on Topic implementations.

MQJMS6118 Topic [0] contains a wildcard, which is invalid for publishing.
Explanation: The Topic specified to a TopicPublisher method contained a wildcard. Wildcards are not allowed in Topics when publishing messages. The failing Topic is given by [0].
User Response: See “Using topics” on page 210 for more information.

MQJMS6119 An InterruptedException occurred while publishing, exception [0].
Explanation: An InterruptedException was caught while publishing a message. [0] gives details of the exception.

MQJMS6120 Attempted to use a temporary topic not created on the current connection.
Explanation: Invalid use of temporary topics and connections.
User Response: See Chapter 14, “JMS interfaces and classes” on page 261 for more information.

MQJMS6121 An InterruptedException occurred while subscribing, exception [0].
Explanation: An InterruptedException was caught while subscribing. [0] gives details of the exception.
MQJMS6122  An exception occurred when creating subscription to /{0}, /{1}.
Explanation: An invalid subject or query syntax was used in the creation of a subscriber, resulting in an exception. /{0} is the topic name, and /{1} is the caught exception.
User Response: See Chapter 11, “Programming publish/subscribe applications” on page 203 and the broker documentation for more information.

MQJMS6232  While creating a TopicSubscriber, attempting to add the subscription to the matching engine resulted in exception: /{0}.
Explanation: /{0} gives details of the exception.
User Response: See Chapter 11, “Programming publish/subscribe applications” on page 203 and the broker documentation for more information.

MQJMS6246  An incorrect use of a the Topic wildcard character /{0} was detected.
Explanation: The failing Topic is given by parameter /{0}.
User Response: See “Using topics” on page 210 for more information.

MQJMS6247  The Topic segment separator /{0} appears in an incorrect position.
Explanation: A subscription Topic separator was used incorrectly. /{0} shows the bad separator.
User Response: See “Using topics” on page 210 for more information.

MQJMS6249  The following exception occurred while parsing a subscription selector: /{0}.
Explanation: This may indicate a syntax error in your Selector.
User Response: See “Message selectors” on page 196 for more information.

MQJMS6250  The escape character was used to terminate the following pattern: /{0}.
Explanation: This may indicate a syntax error in your Selector.
User Response: See “Message selectors” on page 196 for more information.

MQJMS6251  The escape character /{0} passed to the pattern tool is longer than one character.
Explanation: This may indicate a syntax error in your Selector.
User Response: See “Message selectors” on page 196 for more information.

MQJMS6252  A message field was expected to contain a value of type /{0} but contained one of type /{1}.
Explanation: This may indicate a syntax error in your Selector.
User Response: See “Message selectors” on page 196 for more information.

MQJMS6312  Non-authorized subscription to topic /{0}.
Explanation: Attempting to create a subscription to a Topic that is not authorized for the client. /{0} gives the Topic string.
User Response: See Chapter 11, “Programming publish/subscribe applications” on page 203 and the broker documentation for more information.
Appendix J. Notices

This information was developed for products and services offered in the United States. IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.
Notices

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM’s application programming interfaces.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

- AIX
- CICS
- iSeries
- MVS/ESA
- OS/400
- System/390
- VSE/ESA
- zSeries
- AS/400
- IBM
- Language Environment
- OS/2
- SecureWay
- S/390
- WebSphere
- BookManager
- IBMLink
- MQSeries
- OS/390
- SupportPac
- VisualAge
- z/OS
Java, HotJava, JDK, and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.
confirm on arrival report options, message 126
confirm on delivery report options, message 126
connecting to a publish/subscribe broker 409
connecting to a queue manager 68
connecting to WebSphere MQ Event Broker 409
connecting to WebSphere MQ Integrator V2 409
connection
building 190
creating 191
interface 189
options 4
starting 192
WebSphere MQ, losing 220
Connection interface 279
connection pooling 73
example 74
connection type, defining 62
ConnectionConsumer class 243
ConnectionConsumer interface 282
ConnectionFactory interface 283
ConnectionMetaData interface 290
connector.jar 7
container-managed transactions 414
sample application 415
converting the log file 39
COPY (administration verb) 43
core classes 89
restrictions and variations 90, 423
CountingMessageListenerFactory.java 254
createQueueSession method 193
createReceiver method 196
createSender method 193
creating
a connection 191
factories at runtime 191
JMS objects 45
Topics at runtime 212

default connection pool 73
default trace and log output locations 35
DEFINE (administration verb) 43
defining connection type 62
defining transport 192
definition, LDAP schema 403
DELETE (administration verb) 43
DeliveryMode interface 292
dependencies, property 51
description interface 48, 401
Destination interface 293
differences between applets and applications 61
differences due to environment 423
directories, installation 8
disconnecting from a queue manager 68
DISPLAY (administration verb) 43
disposition options, message 126, 247
distribution lists
platform dependency 92
durable subscribers 214

E
ENCODING object property 52
END (administration verb) 43
environment dependencies 89
functions not with all platforms 92
distribution lists 92
MQGetMessageOptions fields 92
MQMD fields 93
MQPutMessageOptions fields 93
MQQueueManager begin() method 92
MQQueueManager constructor 92
restrictions and variations 90
MQCN0_FASTPATH_BINDING 90
MQGMO_* values 90
MQPMO_* values 90
MQPMRF_* values 90
MQRO_* values 91
z/OS & OS/390
environment differences 423
environment variables 8
configuring 23
error
conditions for object creation 54
handling 70
logging 36
recovery, IVT 31
recovery, PSIVT 35
runtime, handling 198
error messages 16
LDAP server 403
example code 62
exception listener 198
exception messages, JMS 427
exception report options, message 126, 247
ExceptionListener interface 295
exceptions
JMS 198
WebSphere MQ 198
exit string properties 52
expiration report options, message 126
EXPIRY object property 48, 401
extra function provided over WebSphere MQ Java 3

F
factories, creating at runtime 191
formatLog utility 39, 401
fscontext.jar 7
function, extra provided over WebSphere MQ Java 3
functions, Application Server Facilities 243

G
getting started 3

H
handling errors 70
JMS runtime errors 198
handling (continued)
messages 69
headers, message 225
HOSTNAME object property 48, 401

I
import statements 207
INITIAL_CONTEXT_FACTORY property 40, 41
inquire and set 71
installation
directories 8
Installation Verification Test program for publish/subscribe (PSIVT) 32
IVT error recovery 31
PSIVT error recovery 35
setup 23
verifying 17
Installation Verification Test program (IVT) 28
installing
WebSphere MQ classes for Java 7
WebSphere MQ classes for Java Message Service 7
interface, programming 58
interfaces
JMS 189, 261
WebSphere MQ 189
introduction
for programmers 57
WebSphere MQ classes for Java 3
WebSphere MQ classes for Java Message Service 3
IVT (Installation Verification Test program) 28
IVTrun utility 401
IVTrun utility 29, 30, 34
IVTSetup utility 30, 401
IVTTidy utility 31, 401

J
J2EE connector architecture 74
JAAS (Java Authentication and Authorization Service) 74, 173
jar files 7
Java 2 Platform Enterprise Edition (J2EE) 74
Java 2 Security Manager, running applications under 10
Java Authentication and Authorization Service (JAAS) 74, 173
Java classes 59
WebSphere MQ classes for Java
Java Development Kit (JDK) 58
Java interface, advantages 57
Java Transaction API (JTA) 389, 413
javaClassName
LDAP attribute setting 404
javaClassNames
LDAP attribute setting 404
javaCodebase
LDAP attribute setting 404
JavaContainer
LDAP objectClass definition 406
javaFactory
LDAP attribute setting 405
javaNamingReference
LDAP objectClass definition 406
javaObject
LDAP objectClass definition 406
javaReferenceAddress
LDAP attribute setting 405
javaSerializedData
LDAP attribute setting 405
javaSerializedObject
LDAP objectClass definition 405
javax.jms package 261
JDBC coordination 79
JDK (Java Development Kit) 58
JMS
administered objects 190
benefits 3
classes 261
exception listener 198
exceptions 198
reference 427
interfaces 189, 261
introduction 3
mapping of fields at send or publish 236
mapping with MQMD 233
message types 195
messages 225
model 189
objects for publish/subscribe 207
objects, administering 43
objects, creating 45
objects, properties 46
programs, writing 189
resources, closing in publish/subscribe mode 209
JMS exception messages 427
JMS JTA/XA Interface 413
JMS Postcard
accessibility 22
changing appearance 22
changing browser for help 22
default configuration 20
font and color settings 22
how it works 20
interoperability with other Postcard applications 21
receiving messages 20
sending a postcard 18
sending messages 21
signon 18
advanced options 18
starting 17
tidying up after use 21
using with one queue manager 18
using with two queue managers 19
Jms.jar 7
JMSAdmin configuration file 40, 41
JMSAdmin utility 39, 40
JMSAdmin.config file 39
JMSBytesMessage class 266
JMSCorrelationID header field 225
JMSMapMessage class 296
JMSMessage class 305
JMSStreamMessage class 350
JMSTextMessage class 360
JNDI
retrieving 190
security considerations 41
jndi.jar 7
JSSE
for SSL support 82, 199
JTA (Java Transaction API) 389, 413
JTA/JDBC coordination 79
installation
other platforms 79
Windows 79
known problems 81
limitations 81
switch file 79
usage 80
L
LDAP naming considerations 45
LDAP schema definition 403
LDAP server 30
attribute settings
javaClassName 404
javaClassNames 404
javaCodebase 404
javaFactory 405
javaObject 406
javaReferenceAddress 405
javaSerializedData 405
configuration 403
error messages 403
iSeries OS/400 V4R5 Schema Modification 407
Microsoft Active Directory 406
Netscape Directory 406
objectClass definitions
javaContainer 406
javaNamingReference 406
javaObject 406
javaSerializedObject 405
schema 403
Sun Microsystems' Schema Modification Applications 407
dap.jar 7
library, Java classes 59
listener, JMS exception 198
Load1.java 253
Load2.java 256
local publications, suppressing 214
log file
converting 39
default output location 35
logging errors 36
LoggingMessageListenerFactory.java 256
MA1G, SupportPac
special considerations for 421
ManagedConnection 180
ManagedConnectionFactory 183
ManagedConnectionMetaData 185
manipulating subcontexts 43
map message 225
MapMessage
interface 296
type 196
mapping properties between admin. tool and programs 399
mcd folder 410
message
body 225
delivery, asynchronous 197
error 16
handling 69
headers 225
message body 241
properties 225
selectors 197, 225
selectors and SQL 226
selectors in publish/subscribe mode 214
sending 193
types 195, 225
Message interface 305
MessageConsumer interface 189, 319
MessageListener interface 321
MessageListenerFactory.java 252
MessageProducer interface 189, 322
MessageProducer object 193
messages
JMS 225
mapping between JMS and WebSphere MQ 229
poison 246
publishing 209
receiving 196
receiving in publish/subscribe mode 209
selecting 197, 225
model, JMS 189
MOVE (administration verb) 43
MQC 171
MQChannelDefinition 96
MQChannelExit 98
MQCNO_FASTPATH_BINDING variations by environment 90
MQConnection class 279
MQConnectionConsumer class 243, 282
MQConnectionFactory class 283
MQConnectionManager 173
MQConnectionMetaData class 290
MQDeliveryMode class 292
MQDestination class 293
MQDistributionList 101
MQDistributionListItem 103
MQEnvironment 62, 67, 104
MQException 110
MQGetMessageOptions 112
MQGetMessageOptions fields variations by environment 92
MQGMO_* values variations by environment 90
MQIVP listing 15
sample application 14
tracing 15
mqjavac
using to verify 27
MQManagedObject 116
Index 447
Index  

QueueReceiver interface 336
QueueRequestor class 337
queues, accessing 68
QueueSender interface 339
QueueSession interface 342

R
reading strings 70
receiving
messages 196
messages in publish/subscribe mode 209
RECEXIT object property 48, 401
RECEXITINIT object property 48, 401
report options, message 126, 247
reports, broker 223
resources, closing 197
restrictions and variations
to core classes 423
retrieving objects from JNDI 190
runjms utility 35, 401
running
applets 86
applications under CICS Transaction Server 423
in a Web browser 5
programs 35
standalone program 5
the IVT 28
the PSIVT 32
WebSphere MQ classes for Java programs 86
with appletviewer 5
your own programs 15
runtime
creating factories 191
creating Topics 212
effects, handling 198

S
sample applet
using to verify 27
sample application
bean-managed transactions 416
bindings mode 65
corner-managed transactions 415
publish/subscribe 205
publish/subscribe with WebSphere Application Server 416
tracing 15
using Application Server Facilities 253
using to verify 14
WebSphere MQ JMS with WebSphere Application Server 414
dscheme, LDAP server 403
scripts provided with WebSphere MQ classes for Java Message Service 401
SECEXIT object property 48, 401
SECEXITINIT object property 48, 401
Secure Sockets Layer 72, 199
certificate revocation list (CRL) 83
CipherSpecs 82, 425
CipherSuites 82
CipherSuites supported 425
Distinguished names (DN) 83
enabling 82
handled by JSSE 82, 199
introduction 82, 199
properties 53
SSLCERTSTORES 200
SSLCIPHERSUITE 199
SSLPeerNAME 199
sslCertStores property 84
sslCipherSuite property 82
sslPeerName property 83
sslSocketFactory property 85
with user exits 72
security considerations, JNDI 41
Security policy definition file, editing 10
SECURITY_AUTHENTICATION property 40, 41
selecting a subset of messages 197, 225
selectors
message 197, 225
message in publish/subscribe mode 214
message, and SQL 226
SENDEXIT object property 48, 401
SENDEXITINIT object property 48, 401
sending a message 193
ServerSession sample code 249
ServerSessionPool sample code 249
Session class 243
Session interface 189, 345
session, obtaining 193
set and inquire 71
set methods
on MQQueueConnectionFactory 192
using to set properties 195
setJMSType method 410
setting
queue properties 193
queue properties with set methods 195
shutting down applications 198
software, prerequisites 6
solving problems 15
general 36
in publish/subscribe mode 219
SQL for message selectors 226
SSL
See Secure Sockets Layer
SSLCERTSTORES object property 200
sslCertStores property 84
SSLCIPHERSUITE object property 48, 53, 199
sslCipherSuite property 82
SSLCRL object property 48, 53
SSLPeerNAME object property 48, 53, 199
sslPeerName property 83
sslSocketFactory property 85
standalone program, running 5
starting a connection 192
starting the administration tool 39
STREAMREFRESHINT object property 48
stream message 225
StreamMessage interface 350
type 196
strings, reading and writing 70
subcontexts, manipulating 43
subscriber options 213
subscriptions, receiving 209
subset of messages, selecting 197, 225
SUBSTORE object property 48
Sun JMS interfaces and classes 261
Sun Web site 3
SupportPac MA1G
special considerations for 421
suppressing local publications 214
switch file for JTA/JDBC 79
SYNCPOINTALLGETS object property 48

T
TARGCLIENT object property 48, 401
TCP/IP
client verifying 14
connection, programming 61
TEMPPROFILE object property 48, 401
TemporaryQueue interface 358
TemporaryTopic interface 359
termination, unexpected 220
testing WebSphere MQ classes for Java programs 86
text message 225
textMessage interface 360
type 196
tokens, connection pooling 73
Topic
interface 207, 361
names 210
names, wildcards 211
object 190
TOPI C object property 48, 401
TopicConnection 207
interface 364
TopicConnectionFactory 207
interface 367
TopicLoad.java 257
TopicPublisher 209
interface 373
TopicRequestor class 376
TopicSession 207
interface 378
TopicSubscriber 209
interface 382
trace, default output location 35
tracing
programs 86
the sample application 15
WebSphere MQ for Java Message Service 36
transactions
bean-managed 414
transactions (continued)
  container-managed 414
  sample application 415, 416
TRANSPORT object property 48, 401
transport, choosing 192
two-phase commit, with WebSphere
  Application Server 414
types of JMS message 195, 225

U
unexpected application termination 220
uniform resource identifier (URI) for
  queue properties 194
  URI for queue properties 194
USE_INITIAL_DIR_CONTEXT
  property 41
USECONPOOILING object property 48
user exits
  with SSL 72
  writing 72
user exits, writing 198
uses for WebSphere MQ 3
using
  WebSphere MQ base Java 13
utilities provided with WebSphere MQ
  classes for Java Message Service 401

V
valid combinations of objects and
  properties 51
verbs, WebSphere MQ supported 58
verification
  with JNDI (point-to-point) 30
  with JNDI (publish/subscribe) 34
  without JNDI (point-to-point) 29
  without JNDI (publish/subscribe) 32
verifying
  client mode installation 27
  TCP/IP clients 14
  with the sample applet 27
  with the sample application 14
  your installation 17
versions of software required 6
VisiBroker
  using 4

W
Web browser
  using 5
Web server, configuring 10
WebSphere Application Server 249, 413
  configuration 42
  CosNaming namespace 40
  CosNaming repository 40, 42
  using with JMS 413
WebSphere MQ
  connection, losing 220
  exceptions 198
  interfaces 189
  messages 229
WebSphere MQ classes for Java
  classes 95
WebSphere MQ Event Broker
  connecting as publish/subscribe
    broker 409
WebSphere MQ Event Broker
  properties 52
WebSphere MQ Integrator V2
  connecting as publish/subscribe
    broker 409
transforming and routing
  messages 410
WebSphere MQ Message Descriptor
  (MQMD) 229
  mapping with JMS 233
WebSphere MQ Publish/Subscribe 24
WebSphere MQ supported verbs 58
wildcards in topic names 211
writing
  JMS programs 189
  programs 61
  publish/subscribe applications 204
  strings 70
  user exits 72, 198

X
XAConnection interface 383
XAConnectionFactory interface 384
XAQueueConnection interface 332, 385
XAQueueConnectionFactory
  interface 334, 386
XAQueueSession interface 388
XAResource 389
XASession interface 389
XAQueue interface 390
XAQueueSession interface 391
XATopicConnectionFactory interface 393
XATopicSession interface 395

Z
z/OS & OS/390
  differences with 91
Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

• By mail, to this address:
  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom

• By fax:
  – From outside the U.K., after your international access code use 44–1962–816151
  – From within the U.K., use 01962–816151

• Electronically, use the appropriate network ID:
  – IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  – IBMLink™: HURSLEY(IDRCF)
  – Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

• The publication title and order number
• The topic to which your comment applies
• Your name and address/telephone number/fax number/network ID.
Spine information:

IBM WebSphere MQ

WebSphere MQ Using Java