MQSeries®

Using Java™

SC34-5456-02

MQSeries®

Using Java™

SC34-5456-02

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix D, “Notices” on page 287.

Third edition (November 1999)

This edition applies to the following products:

- MQSeries classes for Java Version 5.1 on the following platforms:
  - MQSeries for AIX® Version 5 Release 1
  - MQSeries for HP-UX Version 5 Release 1
  - MQSeries for Sun Solaris Version 5 Release 1
  - MQSeries for Windows NT® Version 5 Release 1
  - MQSeries for OS/2® Warp Version 5 Release 1
  - MQSeries for AS/400® Version 4 Release 2 Modification 1
  - MQSeries for AT&T GIS UNIX® Version 2 Release 2
  - MQSeries for SINIX and DC/OSx Version 2 Release 2
  - MQSeries for MVS/ESA™ Version 1 Release 2
  - MQSeries classes for Java for OS/390® Version 1 Release 2 (SupportPac MAG13)
- MQSeries Product Extension MA88 on the following platforms:
  - MQSeries for AIX® Version 5 Release 1
  - MQSeries for HP-UX Version 5 Release 1
  - MQSeries for Sun Solaris Version 5 Release 1
  - MQSeries for Windows NT® Version 5 Release 1

and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this book ix
Who this book is for ix
What you need to know to understand this book ix
How to use this book x
Abbreviations used in this book x

Summary of changes xi
Changes for this edition (SC34-5456-02) xi
Changes for the previous edition xi
Changes for the first edition xi

Part 1. Guidance for users 1

Chapter 1. Getting started 3
What is MQSeries classes for Java? 3
What is MQSeries classes for Java Message Service(JMS)? 3
Who should use MQ Java? 3
Connection options 4
Prerequisites 6

Chapter 2. Installation procedures 7
Installing MQ base Java Version 5.1 7
Installing Product Extension MA88 9
Web server configuration 10

Chapter 3. Using MQ base Java 11
Using the sample applet to verify the TCP/IP client 11
Verifying with the sample application 13
Running your own programs 15
Solving MQ base Java problems 15

Chapter 4. Using MQ JMS 17
Running the point-to-point IVT 18
The Publish/Subscribe Installation Verification Test 21
Running your own programs 24
Solving problems 24

Chapter 5. Using the MQ JMS administration tool 27
Invoking the Administration tool 27
Configuration 28
Administration commands 29
Manipulating subcontexts 30
Administering JMS objects 31

Part 2. Programming with MQ base Java 39

Chapter 6. Introduction for programmers 41
Why should I use the Java interface? 41
The MQSeries classes for Java interface 42

Java Developer's Kit . . . 42
MQSeries classes for Java class library . . . 43

Chapter 7. Writing MQ base Java programs . . . 45
Should I write applets or applications? . . . 45
Connection differences . . . 45
Example code fragments . . . 46
Operations on queue managers . . . 52
Accessing queues and processes . . . 53
Handling messages . . . 54
Handling errors . . . 55
Getting and setting attribute values . . . 55
Multithreaded programs . . . 56
Writing user exits . . . 57
Compiling and testing MQSeries classes for Java programs . . . 58

Chapter 8. Environment-dependent behavior . . . 61
Core details . . . 61
Restrictions and variations for core classes . . . 62
Version 5 extensions operating in other environments . . . 63

Chapter 9. The MQSeries classes for Java classes and interfaces . . . 67
MQChannelDefinition . . . 68
MQChannelExit . . . 70
MQDistributionList . . . 73
MQDistributionListItem . . . 75
MQEnvironment . . . 77
MQException . . . 81
MQGetMessageOptions . . . 83
MQManagedObject . . . 87
MQMessage . . . 90
MQMessageTracker . . . 108
MQProcess . . . 110
MQPutMessageOptions . . . 112
MQQueue . . . 115
MQQueueManager . . . 123
MQC . . . 131
MQReceiveExit . . . 132
MQSecurityExit . . . 134
MQSendExit . . . 136

Part 3. Programming with MQ JMS . . . 139

Chapter 10. Writing MQ JMS programs . . . 143
The JMS model . . . 143
Building a Connection . . . 144
Obtaining a Session . . . 147
Sending a message . . . 147
Receiving a message . . . 150
Closing down . . . 152
Handling errors . . . 152

Chapter 11. Programming Publish/Subscribe applications . . . 153

Writing a simple Pub/Sub application . . . 153
Using Topics . . . 155
Subscriber options . . . 158
Solving Pub/Sub problems . . . 159

Chapter 12. JMS messages . . . 161
Message selectors . . . 161
Mapping JMS messages onto MQSeries messages . . . 165

Chapter 13. JMS interfaces and classes . . . 179
Sun Java Message Service classes and interfaces . . . 179
MQSeries JMS classes . . . 182
BytesMessage . . . 184
Connection . . . 192
ConnectionFactory . . . 195
ConnectionMetaData . . . 198
DeliveryMode . . . 200
Destination . . . 201
ExceptionListener . . . 203
MapMessage . . . 204
Message . . . 212
MessageConsumer . . . 224
MessageListener . . . 226
MessageProducer . . . 227
MQQueueEnumeration * . . . 230
ObjectMessage . . . 231
Queue . . . 232
QueueBrowser . . . 234
QueueConnection . . . 236
QueueConnectionFactory . . . 238
QueueReceiver . . . 240
QueueRequestor . . . 241
QueueSender . . . 243
QueueSession . . . 246
Session . . . 249
StreamMessage . . . 253
TemporaryQueue . . . 260
TemporaryTopic . . . 261
TextMessage . . . 262
Topic . . . 263
TopicConnection . . . 265
TopicConnectionFactory . . . 267
TopicPublisher . . . 270
TopicRequestor . . . 273
TopicSession . . . 275
TopicSubscriber . . . 278

Part 4. Appendices . . . 279

Appendix A. Mapping between Administration tool properties and programmable properties . . . 281

Appendix B. Scripts provided with MQSeries classes for Java Message Service(JMS) 283

Appendix C. LDAP server configuration for Java objects 285
Checking your LDAP server configuration 285
Configuration procedures 285

Appendix D. Notices 287
Trademarks 288

Glossary of terms and abbreviations 289

Bibliography 291
MQSeries cross-platform publications 291
MQSeries platform-specific publications 292
Softcopy books 293
MQSeries information available on the Internet 294

Index 295

Figures

1. MQSeries classes for Java example applet 47
2. MQSeries classes for Java example application 50
3. Topic name hierarchy 156
4. JMS to MQSeries mapping model 165
5. JMS to MQSeries mapping model 175

Tables

1. Platforms and connection modes 5
2. MQ base Java V 5.1 installation directories 7
3. Sample CLASSPATH statements for the client and server 8
4. Environment variables for client and server 9
5. Product Extension installation directories 9
6. Sample CLASSPATH statements for the Product Extension 10
7. Environment variables for Product Extension 10
8. Classes required by JMS 17
9. Administration verbs 30
10. Syntax and description of commands used to manipulate subcontexts . 30
11. The JMS object types that are handled by the administration tool 31
12. Syntax and description of commands used to manipulate administered objects 31
13. Property names and valid values 33
14. The valid combinations of property and object type 34
15. Core classes restrictions and variations 62
16. Character set identifiers 93

17. Set methods on MQQueueConnectionFactory 146
18. Property names for queue URIs 148
19. Symbolic values for queue properties 149
20. MQRFH2 folders and properties used by JMS 167
21. Property datatypes and values 168
22. JMS properties mapping to MQMD fields 169
23. Outgoing message field mapping 170
24. Incoming message field mapping 174
25. Interface Summary 179
26. Class Summary 181
27. Package 'com.ibm.mq.jms' class Summary 182
28. Package 'com.ibm.jms' class summary 183
29. Comparison of representations of properties within the administration tool, and the programmable equivalents. 281
30. Utilities suppled with MQSeries classes for Java Message Service(JMS) 283

Tables

About this book

This book describes MQSeries classes for Java™ (MQ base Java) which can be used to access MQSeries systems, and MQSeries classes for Java Message Service(JMS) (MQ JMS) which can be used to access both Java Message Service and MQSeries applications.

Part 1 describes the use of MQ Java and MQ JMS, Part 2 provides assistance for programmers wanting to use MQ base Java, and Part 3 provides assistance for programmers wanting to use MQ JMS.

Note: This documentation is available in softcopy format only (PDF and HTML). It is available as part of the Product Extension MA88, and from the MQSeries family website at:

`http://www.ibm.com/software/ts/mqseries/`

This documentation **cannot** be ordered as a printed book.

Who this book is for

This information is written for programmers who are familiar with the procedural MQSeries application programming interface as described in the *Application Programming Guide*, and shows how to transfer this knowledge to become productive with the 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 Chapter 6, "Introducing the Message Queue Interface" in the *MQSeries Application Programming Guide* and in Chapter 3, "Call descriptions" in the *MQSeries Application Programming Reference* book
- Experience of MQSeries programs in general, or familiarity with the content of the other MQSeries publications

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

- CICS concepts
- Using the CICS Java 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).

How to use this book

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

There is a glossary and bibliography at the back of this book.

Abbreviations used in this book

The following abbreviations are used throughout this book:

MQ Java MQSeries classes for Java and MQSeries classes for Java Message Service(JMS) combined

MQ base Java MQSeries classes for Java

MQ JMS MQSeries classes for Java Message Service(JMS)

Summary of changes

This section describes changes to this edition of *MQSeries 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 (SC34-5456-02)

Major changes for this edition include:

Enhanced support for AIX, HP-UX, Windows, and Solaris
MQ base Java version 5.1.1 is now available as a Product Extension for these platforms. The Product Extension includes updated MQSeries client and bindings files together with JMS support.

Changes for the previous edition

Major changes in this edition include:

Enhanced support for OS/390
MQ base Java now supports CICS Transaction Server for OS/390 (Version 1.3) as well as the Java Virtual Machine (JVM) available under UNIX System Services.

Changes for the first edition

This version of MQ base Java is a consolidation of the MQSeries Client for Java and MQSeries Bindings for Java products and contains the following additions and enhancements:

Programmable transport options
MQSeries client and bindings code have been combined into a single Java package. The transport choice is now a programmable option making it possible to connect to the MQSeries server either as an MQSeries client, or through the Java Native Interface (JNI). Applications previously written specifically for MQSeries Client for Java or MQSeries Bindings for Java can still be run with this version of MQ base Java. The package `com.ibm.mqbind` can still be used on some platforms but it is deprecated and you are recommended not to use it in any new applications.

Repackaging into Java .jar files
The client, bindings, and common files have been repackaged into **.jar** files for easier installation and downloading to clients.

Support for connection using VisiBroker for Java
As an option, MQ base Java running on supported Windows® platforms can connect to the MQSeries server using an IIOP protocol. This support is provided using VisiBroker for Java in conjunction with Netscape Navigator, and requires Inprise VisiBroker for Java to be installed on the MQSeries server machine.

Part 1. Guidance for users

Chapter 1. Getting started . . . 3
What is MQSeries classes for Java? . . . 3
What is MQSeries classes for Java Message Service(JMS)? . . . 3
Who should use MQ Java? . . . 3
Connection options . . . 4
Client connection . . . 5
Using VisiBroker for Java . . . 5
Bindings connection . . . 6
Prerequisites . . . 6

Chapter 2. Installation procedures . . . 7
Installing MQ base Java Version 5.1 . . . 7
Installing Product Extension MA88 . . . 9
Web server configuration . . . 10

Chapter 3. Using MQ base Java . . . 11
Using the sample applet to verify the TCP/IP client . . . 11
Configuring your queue manager to accept client connections . . . 11
Running from appletviewer . . . 12
Running from a Web browser . . . 12
Customizing the verification applet . . . 13
Verifying with the sample application . . . 13
Using VisiBroker connectivity . . . 14
Using CICS Transaction Server for OS/390 . . . 14
Running your own programs . . . 15
Solving MQ base Java problems . . . 15
Tracing the sample applet . . . 15
Tracing the sample application . . . 15
Error messages . . . 16

Chapter 4. Using MQ JMS . . . 17
Post installation setup . . . 17
Additional setup for Pub/Sub mode . . . 18
Running the point-to-point IVT . . . 18
Point-to-point verification without JNDI . . . 19
Point-to-point verification with JNDI . . . 19
IVT error recovery . . . 21
The Publish/Subscribe Installation Verification Test . . . 21
Pub/Sub verification without JNDI . . . 22
Pub/Sub verification with JNDI . . . 23
PSIVT error recovery . . . 23
Running your own programs . . . 24
Solving problems . . . 24
Tracing programs . . . 24
Logging . . . 25

Chapter 5. Using the MQ JMS administration tool . . . 27
Invoking the Administration tool . . . 27
Configuration . . . 28
Security . . . 29

Administration commands 29
Manipulating subcontexts 30
Administering JMS objects 31
Object types 31
Verbs used with JMS objects 31
Creating objects 32
Properties 33
Property dependencies 36
The ENCODING property 36
Sample error conditions 37

Chapter 1. Getting started

This chapter gives an overview of MQSeries classes for Java and MQSeries classes for Java Message Service(JMS), and their uses.

What is MQSeries classes for Java?

MQSeries classes for Java (MQ base Java) allows a program written in the Java programming language to connect to MQSeries as an MQSeries client, or directly to an MQSeries server. It enables Java applets, applications, and servlets to issue calls and queries to MQSeries giving access to mainframe and legacy applications, typically over the Internet, without necessarily having any other MQSeries code on the client machine. With 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 is MQSeries classes for Java Message Service(JMS)?

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

There are a number of benefits that arise from using MQ JMS as the API for writing MQSeries applications. Some advantages derive from JMS being an open standard with multiple implementations, others result from additional features that are present in MQ JMS but not in 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

More information about the benefits of the JMS API can be found on Sun's web site at `http://java.sun.com`.

The extra function provided over MQ base Java includes:

- Asynchronous message delivery
- Message selectors
- Support for pub/sub messaging
- Structured message classes

Who should use MQ Java?

If your enterprise fits any of the following scenarios, you can gain significant advantage by using MQSeries classes for Java and MQSeries classes for Java Message Service(JMS):

- 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 MQSeries 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 MQSeries 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 MQSeries 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 and add the value of high integrity provided by MQSeries connectivity. An Internet Service provider that exploits MQSeries 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.

MQSeries and MQSeries classes for Java Message Service(JMS) 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 and 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 timeliness of the response is not impacted by network loading. In addition, the transactional nature of MQSeries 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.

MQSeries 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 run-time environment. These factors combine to reduce significantly the development time for multi-platform MQSeries applications, and future enhancements to applets are automatically picked up by end users as the applet code is downloaded.

Connection options

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

- As an MQSeries client using TCP/IP
- In bindings mode, connecting directly to MQSeries

MQ base Java on Windows NT can also connect using VisiBroker for JavaTable 1 on page 5 shows the connection modes that can be used for each platform.

Table 1. Platforms and connection modes

Server platform	Connection mode		
	Client		Bindings
	stand.	VisiB.	
Windows NT	yes	yes	yes
AIX	yes	no	yes
Solaris	yes	no	yes
OS/2	yes	no	yes
AS/400®	yes	no	no
HP-UX	yes	no	yes
AT&T GIS UNIX	yes	no	no
Sun OS	yes	no	no
SINIX and DC/OSx	yes	no	no
OS/390	no	no	yes

Note: HP-UX Java bindings support is only available for HP-UXv11 systems running the POSIX draft10 pthreaded version of MQSeries. You also require the HP-UX Developer's Kit for Java 1.1.7 (JDK), Release C.01.17.01 or above.

These options are described in more detail below.

Client connection

If you are using MQ Java as an MQSeries client, it can be installed either on the MQSeries server machine, which may also contain a Web server, or on a separate machine. Installation on the same machine as a Web server has the advantage of allowing you to download and run MQSeries client applications on machines that do not have MQ Java installed locally.

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

From within any Java-enabled Web browser
When running in this mode, the locations of the MQSeries queue managers that can be accessed may be constrained by the security restrictions of the browser being used.

Using an applet viewer
To use this method you must have the Java Developer's Kit (JDK) or Java Runtime Environment (JRE) installed on the client machine.

As a stand-alone Java program or in a Web application server
To use this method you must have the Java Developer's Kit (JDK) or Java Runtime Environment (JRE) installed on the client machine.

Using VisiBroker for Java

Connection through Visibroker is provided on the Wndows platform, as an alternative to using the standard MQSeries client protocols. This support is provided by VisiBroker for Java in conjunction with Netscape Navigator, and requires VisiBroker for Java and an MQSeries object server on the MQSeries server machine. A suitable object server is provided with MQ base Java.

Bindings connection

When used in bindings mode, MQ Java uses the JNI to call directly into the existing queue manager API rather than communicating through a network. This provides better performance for MQSeries applications than using network connections. Unlike the client mode, applications written using the bindings mode cannot be downloaded as applets.

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

Prerequisites

The following software is required to run MQ base Java:

- MQSeries for the server platform you wish to use.
- Java Developers Kit (JDK) for the server platform.
- Java Developers Kit, or Java Runtime Environment (JRE), or Java-enabled Web browser for client platforms. (See “Client connection” on page 5.)

  Note: To run MQ base Java applets (for example the installation verification program) inside a Web browser, you need a browser that can run Java 1.1.6 applets. Sun System's HotJava, Netscape Navigator 4, and Microsoft® Internet Explorer 4 are examples of browsers that meet this requirement.
- VisiBroker for Java. (Only if running on Windows with a VisiBroker connection.)
- For OS/390, OS/390 Version 2 Release 5 with UNIX System Services.

MQ base Java must be installed on your machine before you install and run MQ JMS. The following additional software is required if you want to use the Administration Tool (see Chapter 5, “Using the MQ JMS administration tool” on page 27):

- At least one of the following service provider packages:
  - Lightweight Directory Access Protocol (LDAP) - `ldap.jar`, `providerutil.jar`.
  - File system - `fscontext.jar`, `providerutil.jar`.
- A JNDI service provider. This is the resource in which physical representations of the administered objects is stored. It is expected that users of MQ JMS will use an LDAP server for this purpose, but the tools 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. Information to assist with this configuration is provided in Appendix C, “LDAP server configuration for Java objects” on page 285.

Chapter 2. Installation procedures

This section describes the installation of MQ base Java Version 5.1 and of the Product Extension MA88, which comprises MQ base Java Version 5.1.1 and MQ JMS. Go to the relevant section to find the information you require.

Installing MQ base Java Version 5.1

The MQ base Java Version 5.1, for all platforms except OS/390 and AS/400 can be installed from either the MQSeries Version 5.1 Software Server CD or the MQSeries Version 5.1 Software Client CD. Follow the installation instructions provided with the CD. If you choose the typical installation, MQ base Java is included in the installation. If you choose to customize your installation, make sure that MQSeries classes for Java is checked.

Notes:

1. If you want to use the native connection (bindings) mode, you must install from the server CD
2. On an OS/2 system, MQ base Java must be installed in an HPFS partition.

For OS/390 MQ base Java is supplied as a downloadable MQSeries SupportPac, available from `http://www.ibm.com/software/ts/mqseries/`.

For AS/400, MQ base Java can only be downloaded as a client.

The MQ base Java Version 5.1 files, documentation, and samples are installed in the directories shown in Table 2 and Table 5 on page 9.

Table 2. MQ base Java V 5.1 installation directories

Platform	Files	Directory
AIX	code Documentation Samples	usr/mqm/java/lib usr/mqm/html/mqjava usr/mqm/samp/javaclnt/*langdir*
HP-UX Solaris	code Documentation Samples	opt/mqm/java/lib opt/mqm/html/mqjava opt/mqm/samp/javaclnt/*langdir*
OS/2 Windows NT	code Documentation Samples	*install_dir*\java\lib *install_dir*\html\mqjava\ *install_dir*\tools\javaclnt\samples*langdir*
OS/390	code Samples	*install_dir*/java/lib *install_dir*/samp/mqjava

Note:
install_dir is the directory in which you chose to install MQ base Java
langdir is the language directory for your installation

MQSeries Java is contained in the following Java `.jar` files:

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

com.ibm.mq.iiop.jar This code supports only the Visibroker connection.

com.ibm.mqbind.jar This code supports only the bindings connection and is not supplied or supported on all platforms You are recommended not to use it in any new applications.

Notes:

1. **com.ibm.mq.iiop.jar** is supplied only on the Windows platform.
2. On OS/390, only the **com.ibm.mq.jar** file is supplied. This file supports the bindings connection to MQSeries from both UNIX System Services and CICS Transaction Server for OS/390.

After installation, you must update your CLASSPATH environment variable to include the MQSeries Java code and samples directories.Table 3 and Table 6 on page 10 show typical CLASSPATH settings for the various platforms

Table 3. Sample CLASSPATH statements for the client and server

Platform	Sample CLASSPATH
AIX	CLASSPATH=*jdk_dir*/lib/classes.zip: /usr/mqm/java/lib/com.ibm.mq.jar: /usr/mqm/java/lib/com.ibm.mqbind.jar: /usr/mqm/java/lib: /usr/mqm/samp/javaclnt/*lang_dir*:
HP-UX Solaris	CLASSPATH=*jdk_dir*/lib/classes.zip: /opt/mqm/java/lib/com.ibm.mq.jar: /opt/mqm/java/lib/com.ibm.mqbind.jar: /opt/mqm/java/lib /opt/mqm/samp/javaclnt/*lang_dir*:
OS/2	CLASSPATH=*jdk_dir*\lib\classes.zip; *install_dir*\java\lib\com.ibm.mq.jar; *install_dir*\java\lib\com.ibm.mqbind.jar; *install_dir*\java\lib; *install_dir*\tools\javaclnt\samples*lang_dir*;
Windows NT	CLASSPATH=C:*jdk_dir*\lib\classes.zip; *install_dir*\java\lib\com.ibm.mq.jar; *install_dir*\java\lib\com.ibm.mqiiop.jar; *install_dir*\java\lib\com.ibm.mqbind.jar; *install_dir*\java\lib\; *install_dir*\tools\javaclnt\samples*lang_dir*;
OS/390	CLASSPATH=/usr/lpp/J1.1/lib/classes.zip: *install_dir*/java/lib: *install_dir*/java/lib/com.ibm.mq.jar: *install_dir*/samp/mqjava

Note:
jdk_dir is the directory in which the JDK is installed.
install_dir is the directory in which you chose to install MQSeries.
lang_dir is the language directory for your installation

Additional environment variables need to be updated on some platforms as shown in Table 4 on page 9 and Table 7 on page 10.

Table 4. Environment variables for client and server	
Platform	**Environment variable**
AIX	LD_LIBRARY_PATH=/usr/mqm/lib
HP-UX	SHLIB_PATH=/opt/mqm/lib
Solaris	LD_LIBRARY_PATH=/opt/mqm/lib
OS/390	LIBPATH=*install_dir*/java/lib STEPLIB=*hlq*.SCSQAUTH:*hlq*.SQCSQANLE
Note: *hlq* is the high level qualifier for the MQSeries installation.	

Installing Product Extension MA88

The MQSeries Product Extension MA88 is available for AIX, HP_UX, Windows, and Solaris, platforms. The Product Extension contains:

- MQ base Java Version 5.1.1
- MQ JMS

You can install MQ base Java Version 5.1.1 alone if you only want the latest versions of the MQ base Java classes, or both packages if you want to use MQ JMS applications.

Note: If you install the product extension and subsequently install or reinstall base MQSeries, make sure that you do not install the base MQ Java version 5.1 as this will cause your MQSeries Java support to be back level.

Table 5. Product Extension installation directories		
Platform	**Files**	**Directory**
AIX	code Documentation Samples	usr/mqm/java/lib usr/mqm/java/doc usr/mqm/java/samples
HP-UX and Solaris	code Documentation Samples	opt/mqm/java/lib opt/mqm/java/doc opt/mqm/java/samples
Windows 95,98, and NT	code Documentation Samples	*install_dir*\lib *install_dir*\doc *install_dir*\samples
Note: *install_dir* is the directory in which you chose to install the Product Extension.		

Table 6. Sample CLASSPATH statements for the Product Extension	
Platform	**Sample CLASSPATH**
AIX	CLASSPATH=*jdk_dir*/lib/classes.zip: /usr/mqm/java/lib/com.ibm.mq.jar: /usr/mqm/java/lib: /usr/mqm/java/samples:
HP-UX and Solaris	CLASSPATH=*jdk_dir*/lib/classes.zip: /opt/mqm/java/lib/com.ibm.mq.jar: /opt/mqm/java/lib: /opt/mqm/java/samples:
Windows 95,98, and NT	CLASSPATH=C:*jdk_dir*\lib\classes.zip; *install_dir*\com.ibm.mq.jar; *install_dir*\com.ibm.mqiiop.jar; *install_dir*\lib\; *install_dir*\samples\;
Note: *jdk_dir* is the directory in which the JDK is installed. *install_dir* is the directory in which you chose to install the Product Extension.	

If you have existing applications that have a dependency on the deprecated bindings package, `com.ibm.mqbind`, you also need to add the `com.ibm.mqbind.jar` file to your classpath.

Table 7. Environment variables for Product Extension	
Platform	**Environment variable**
AIX	LD_LIBRARY_PATH=/usr/mqm/java/lib
HP_UX	LD_LIBRARY_PATH=/opt/mqm/java/lib
Solaris	SHLIB_PATH=/opt/mqm/java/lib
Windows95, 98, and NT	PATH=*install_dir*\lib
Note: *install_dir* is the installation directory for the Product Extension	

Web server configuration

If you install MQSeries Java on a Web server, you can download and run MQSeries Java applications on machines that do not have MQSeries Java installed locally. To make the MQSeries 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 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 OS/390 and served to clients.

Chapter 3. Using MQ base Java

This chapter describes how to configure your system to run the sample applet and application programs to verify your MQ base Java installation and how to modify the procedures to run your own programs.

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

Using the sample applet to verify the TCP/IP client

An installation verification applet, `mqjavac.html`, is provided as part of MQ base Java. The applet can be used to verify the TCP/IP connected client mode of MQ base Java. (See also “Verifying with the sample application” on page 13.)

The applet connects to a given queue manager, exercises all the MQSeries calls, and produces diagnostic messages in the event of any failures.

The applet can be run from the applet viewer supplied with your JDK (v1.1.6 or later), or any Java 1.1.6 enabled browser. When using the applet viewer you will be able to access a queue manager on any host. When using a Web browser, you will be able to access a queue manager only on the host from which the applet was loaded. This is your local machine if you have MQ base Java installed, or the machine on which your Web server is running if you download the applet from a Web server.

Note: When loading applets from a local installation, some Web browsers allow you to specify only the literal string "localhost" as the name of the host to connect to. Consult your Web browser documentation for further information.

In all cases, if the applet does not complete successfully, follow the advice given in the diagnostic messages and try to run the applet again.

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 procedure:
   a. Start your queue manager using the strmqm command
   b. Type

      ```
      runmqsc
      ```

      to start the runmqsc program
   c. Define a sample channel called JAVA.CHANNEL by typing:

      ```
      DEF CHL('JAVA.CHANNEL') CHLTYPE(SVRCONN) TRPTYPE(TCP) MCAUSER(' ') +
      DESCR('Sample channel for MQSeries Client for Java')
      ```

2. Start a listener program with the following commands:

   For OS/2 and NT operating systems:
   Issue the command:

   ```
   runmqlsr -t tcp [-m QMNAME] -p 1414
   ```

   Note: If you use the default queue manager, the -m flag is not required.

   Using VisiBroker for Java on the Windows NT operating system:
   Start the IIOP server with the following command:

   ```
   java com.ibm.mq.iiop.Server
   ```

   Note: To stop the IIOP server, issue the following command:

   ```
   java com.ibm.mq.iiop.samples.AdministrationApplet shutdown
   ```

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

Running from appletviewer

To use this method you must have the Java Developer's Kit (JDK) installed on your machine.

Local installation procedure

1. Change to your samples directory for your language
2. Type:

   ```
   appletviewer mqjavac.html
   ```

Web server installation procedure:
Enter the command:

```
appletviewer http://Web.server.host/MQJavaclient/mqjavac.html
```

Notes:

1. On some platforms the command is 'applet', and not 'appletviewer'.
2. On some platforms, you may need to select 'Properties' from the 'Applet' menu at the top left of your screen, and then set 'Network Access' to 'Unrestricted'.

Using this technique you should be able to connect to any queue manager running on any host to which you have TCP/IP access.

Running from a Web browser

To run the applet from a Web browser, first copy the contents of the samples directory (for your chosen language) into the directory that contains the MQ base Java code. This is necessary because browsers may not make use of your local CLASSPATH setting and so require all the files used by the applet to be in a single directory tree.

Local installation procedure

Note: This method requires a Java 1.1 capable browser.

Open your copied version of the file `mqjavac.html` in your Web browser. The file open procedure varies from between browsers, but the function is usually found on the 'File' menu and is likely to be called 'open', or 'open page'.

Web server installation procedure

1. Configure your Web server so that it can serve files located in this directory. (Consult your Web server documentation for details on how to do this.)
2. Open the URL:

   `http://Web.server.hostname/MQJavaclient/mqjavac.html`

Note: Because of security restrictions imposed by your browser, you will be able to connect only to a queue manager running on the same host as the Web server.

Customizing the verification applet

Optional parameters are included in the `mqjavac.html` file. These parameters allow you to modify the applet to suit your requirements. Each parameter is defined in a line of HTML which looks like the following:

```
<!PARAM name="xxx" value="yyy">
```

To specify a parameter value, remove the initial exclamation mark, and edit the value as desired. The following parameters can be specified:

hostname
Prefills the hostname edit box with the supplied value.

port
Prefills the port number edit box with the supplied value.

channel
Prefills the channel edit box with the supplied value.

queueManager
Prefills the queue manager edit box with the supplied value.

userID:
Uses the specified user ID when connecting to the queue manager.

password
Uses the specified password when connecting to the queue manager.

trace
Causes MQ base Java to write a trace log. Use this option only at the direction of IBM service.

Verifying with the sample application

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

1. **If you want to test a client connection:**
   Configure your queue manager as described in “Configuring your queue manager to accept client connections” on page 11.

   Note: Carry out the rest of the procedure on the client machine if you are testing a client connection. For a bindings connection it should be carried out on the MQSeries server machine.
2. Change to your samples directory.

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

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

   ```
   (1)Please enter the type of connection (MQSeries)               : (MQSeries)
   (2)Please enter the IP address of the MQSeries server          : myhost
   (3)Please enter the port to connect to                         : (1414)
   (3)Please enter the server connection channel name             : JAVA.CHANNEL
     Please enter the queue manager name                          :
     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 transport is functioning correctly.
     Press Enter to continue...
   ```

4. At prompt (1):

   Leave the default 'MQSeries'.

5. At prompt (2):

   - If you want to use TCP/IP connection, enter an MQSeries server hostname.
   - If you want to use native connection (bindings mode), leave the field blank. (Do not enter a name.)

Notes:

1. If you choose server connection, you do not see the prompts marked (3).
2. On OS/390 you do not see prompts (1), (2), or (3)

Using VisiBroker connectivity

If you are running with VisiBroker, the procedures described in “Configuring your queue manager to accept client connections” on page 11 are not required.

To test an installation using VisBroker, use the procedures described in “Verifying with the sample application” on page 13, but at prompt (1), type `VisiBroker` using the exact case.

Using CICS Transaction Server for OS/390

1. Define the sample application program 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. Execute 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.

Running your own programs

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

For information on writing MQ base Java applications and applets, see Part 2, “Programming with MQ base Java” on page 39.

Solving MQ base Java problems

If a program does not complete successfully, try running the installation verification applet or installation verification program, both of which are described in Chapter 3, “Using MQ base Java” on page 11, and follow the advice given in the diagnostic messages.

If you continue to have problems and need to contact the IBM service team, you may be asked to turn on the trace facility. The method of doing this depends on whether you are running the client or the bindings. Choose the appropriate section below to find the procedures for your system.

Tracing the sample applet

To run trace with the sample applet, edit the `mqjavac.html` file as follows:

In the line

```
<!PARAM name="trace" value="1">
```

remove the exclamation mark, and change the value from 1 to a number from 1 to 5 depending on the level of detail required. (The higher the number, the more information will be gathered.)

The line should then read:

```
<PARAM name="trace" value="n">
```

where 'n' is a number between 1 and 5.

The trace output appears in the Java console or in your Web browser's Java log file.

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 higher the number, the more information is gathered.)

For more information on using trace, and how to find and use the output on your platform, see *MQSeries System Administration*.

Tracing with CICS Transaction Server for OS/390

When using CICS Transaction Server for OS/390 it is not possible to supply command line arguments directly to the program. It is necessary to write a small wrapper program which invokes MQIVP.main() with the appropriate arguments.

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.

Recommended Action: Connect the server to the network and retry.

Unable to load file gatekeeper.ior

This failure can occur on a web server deploying VisiBroker applets, when the `gatekeeper.ior` file is not located in the correct place.

Recommended Action: Restart the VisiBroker Gatekeeper from the directory in which the applet is deployed. The `gatekeeper` file will be written to this directory.

Failure: Missing software, may be MQSeries, or VBROKER_ADM variable

This failure occurs in the MQIVP sample program if your Java software environment is incomplete.

Recommended Action: On the client, ensure that the VBROKER_ADM environment variable is set to address the VisiBroker for Java administration (`adm`) directory, and retry.

On the server, ensure that MQ base Java from MQSeries Version 5.1 is installed and retry.

NO_IMPLEMENT

There is a communications problem involving VisiBroker Smart Agents.

Recommended Action: Consult your VisiBroker documentation.

COMM_FAILURE

There is a communications problem involving VisiBroker Smart Agents.

Recommended Action: Use the same port number for all VisiBroker Smart Agents and retry. Consult your VisiBroker documentation.

MQRC_ADAPTER_NOT_AVAILABLE

If you get this error when you are trying to use Visibroker, it is likely that the JAVA class org.omg.CORBA.ORB cannot be found in the CLASSPATH.

Recommended action: Ensure that your CLASSPATH statement includes the path to the Visibroker `vbjorb.jar` and `vbjapp.jar` files.

MQRC_ADAPTER_CONN_LOAD_ERROR

If you see this error while running on OS/390, ensure that the MQSeries SCSQANLE, and SCSQAUTH datasets are in your STEPLIB statement.

Chapter 4. Using MQ JMS

This chapter describes the following tasks:

- Setting up your system to use the Test and sample programs
- Running the point-to-point Installation Verification Test (IVT) program to verify your MQSeries classes for Java Message Service(JMS) installation
- Running the sample Pub/Sub Installation Verification Test (PSIVT) program to verify your Pub/Sub installation
- Running your own programs

Post installation setup

To make all the necessary resources available to MQ JMS programs, the following system variables need to be updated:

Classpath

Successful operation of JMS programs requires a number of Java packages to be available to the JVM. These need to be specified on the classpath after the necessary packages have been obtained and installed.

Table 8 lists the classes that are required, and the package that they come from:

Table 8. Classes required by JMS

Class	Jar file	Notes
MQSeries JMS classes	com.ibm.mqjms.jar	provided with MQ JMS in the java/lib directory
com.ibm.mq.MQMessage	com.ibm.mq.jar	provided with MQSeries 5.1, in the java/lib subdirectory.
javax.jms.Message	jms.jar	provided with MQ JMS in the java/lib directory
javax.naming.InitialContext	jndi.jar	provided with MQ JMS in the java/lib directory
com/sun/jndi/toolkit/ComponentDirContext	providerutil.jar	provided with MQ JMS in the java/lib directory
com.sun.jndi.ldap.LdapCtxFactory	ldap.jar	provided with MQ JMS in the java/lib directory (This is the default service used by the IVT. Other services may be specified using the `-icf` parameter.)

Environment variables

There are a number of scripts provided in the `bin` subdirectory of the MQ JMS installation. These are intended as convenient shortcuts for a number of common actions. Many of these scripts assume that there is an environment variable named `MQ_JAVA_INSTALL_PATH` defined which points to the directory in which MQ JMS is installed. Setting this variable isn't mandatory, but if it is not set, the scripts in the `bin` directory must be edited accordingly.

On Windows NT the classpath and new environment variable can be set using the **Environment** tab of the **System Properties**. On Unix they 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.

Additional setup for Pub/Sub mode

Before the MQ JMS implementation of JMS Pub/Sub can be used, some additional setup is required:

Ensure the Broker is Running

To verify that the MQSeries Message Broker has been 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, then a message similar to the following is displayed:

```
MQSeries message broker for queue manager MY.QUEUE.MANAGER running.
```

If the operating system reports that it cannot execute the `dspmqbrk` command, ensure that the MQSeries Message Broker has been installed properly.

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

```
strmqbrk -m MY.QUEUE.MANAGER
```

Create the MQ JMS System Queues

For the MQ JMS Pub/Sub implementation to work correctly, a number of system queues must be created. A script has been suppled in the `bin` subdirectory of the MQ JMS installation to assist with this task. To use the script, enter the following command:

```
runmqsc MY.QUEUE.MANAGER < MQJMS_PSQ.mqsc
```

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

Running the point-to-point IVT

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

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

The program can be run in one of two possible modes.

Using JNDI lookup of administered objects

JNDI mode forces the program to obtain its administered objects (see “Administering JMS objects” on page 31 for a description of administered objects) from a JNDI namespace, which is the expected operation of JMS client applications. This invocation method has the same prerequisites as the administration tool (see Chapter 5, “Using the MQ JMS administration tool” on page 27).

With no JNDI lookup of administered objects

If the user does not wish to use JNDI, the administered objects can be created at run-time by running the IVT in no-JNDI mode. Since a JNDI based repository is relatively complex to set up, it is recommended 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 -nojndi
```

If the test completes successfully, output similar to the following should be seen:

```
5648-C60 (c) Copyright IBM Corp. 1999. All Rights Reserved.
MQSeries Classes for Java(tm) Message Service - 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: Message Class:   jms_text          JMSType:          null
JMSDeliveryMode: 2                              JMSExpiration:    0
JMSPriority:     4                              JMSMessageID:     ID:414d5120716
d312020202020202020203000c43713400000
JMSTimestamp:    935592657000                   JMSCorrelationID: null
JMSDestination:  queue:///SYSTEM.DEFAULT.LOCAL.QUEUE
JMSReplyTo:      null
JMSRedelivered:  false
JMS_IBM_Format:MQSTR            JMS_IBM_PutApplType:11
JMSXGroupSeq:1                  JMSXDeliveryCount:0
JMS_IBM_MsgType:8               JMSXUserID:kingdon
JMSXApplID:D:\jdk1.1.8\bin\java.exe
A simple text message from the MQJMSIVT program
Reply string equals original string
Closing QueueReceiver
Closing QueueSender
Closing Session
Closing Connection
IVT completed OK
IVT finished
```

Point-to-point verification with JNDI

In order for the IVT to run correctly using JNDI, 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 automate the task of creating these objects. Enter the command:

```
IVTSetup
```

The script invokes the MQ JMS Administration tool (see Chapter 5, “Using the MQ JMS administration tool” on page 27) 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)
CCSID(819)
TEMPMODEL(SYSTEM.DEFAULT.MODEL.QUEUE)
QMANAGER()
```

The MQQueue is bound under the name `ivtQ` (`cn=ivtQ`). The `QUEUE` property is given the value `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)
CCSID(1208)
PRIORITY(APP)
QMANAGER()
```

When the administered objects have been 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:

-t = turn tracing on (default: tracing off)

providerURL = 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 being used, (see `initCtxFact` below) , the URL is of the form:

`file://directorySpec`

Note: The *providerURL* string should be enclosed in quotation marks (").

initCtxFact = classname of 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 being used, this parameter should be set to:

`com.sun.jndi.fscontext.RefFSContextFactory`

If the test completes successfully, output similar to that from the non-JNDI output should be seen, except that the 'create' `QueueConnectionFactory` and `Queue` lines should indicate retrieval of the object from JNDI instead, as shown in the following code fragment:

```
5648-C60 (c) Copyright IBM Corp. 1999. All Rights Reserved.
MQSeries Classes for Java(tm) Message Service - 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
 ...
 ...
```

Although not strictly necessary, it is good practice to remove objects created by the IVTSetup script from the JNDI namespace the . 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 classpath, check that your classpath is set correctly as described in “Post installation setup” on page 17.
- If the IVT fails with a message 'failed to create MQQueueManager' with an additional message including the number 2059, this is an indication that MQSeries failed to connect to the default local queue manager on the machine on which the IVT was run. Check that the queue manager is running and that it is marked as the default queue manager.
- A message of 'failed to open MQ queue' indicates that a connection to the default queue manager was obtained, but that the 'SYSTEM.DEFAULT.LOCAL.QUEUE' could not be opened. This may indicate that the queue does not exist on your default queue manager, or that it is not enabled for PUT and GET. Add or enable the queue for the duration of the test.

The Publish/Subscribe Installation Verification Test

The Publish/Subscribe Installation Verification Test (PSIVT) is supplied in compiled form only, and can be found in the `com.ibm.mq.jms` package.

The PSIVT attempts to perform the following tasks:

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 the PSIVT is run, the message is published by the publisher and is received and displayed by the subscriber. The publisher publishes to the broker's default stream. The subscriber is non-durable, doesn't perform message selection, accepts messages from local connections, and performs a synchronous receive, waiting a maximum of 5 seconds for a message to arrive.

The PSIVT, like the IVT, may be run 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.

Pub/Sub verification without JNDI

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

To run the test without JNDI issue the following command:

```
PSIVTRun -nojndi [-m <qmgr>] [-t]
```

where:

-t turn tracing on (default: tracing off)

-nojndi JNDI lookup of the administered objects

-m <qmgr> specify queue manager to connect to.

If the test completes successfully, output similar to the following should be seen:

```
5648-C60 (c) Copyright IBM Corp. 1999. All Rights Reserved.
MQSeries Classes for Java(tm) Message Service
Publish/Subscribe Installation Verification Test
Creating a TopicConnectionFactory
Creating a Topic
Creating a Connection
Creating a Session
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...

Got message:

JMS Message class: jms_text
  JMSType:         null
  JMSDeliveryMode: 2
  JMSExpiration:   0
  JMSPriority:     4
  JMSMessageID:    ID:414d5120514d2e504f4c415249532e4254b7dc3753700000
  JMSTimestamp:    937232048000
  JMSCorrelationID:ID:414d5158000000000000000000000000000000000000000000
  JMSDestination:  topic
  ://MQJMS/PSIVT/Information
  JMSReplyTo:      null
  JMSRedelivered:  false
  JMS_IBM_Format:MQSTR
  UNIQUE_CONNECTION_ID:937232047753
  JMS_IBM_PutApplType:26
  JMSXGroupSeq:1
  JMSXDeliveryCount:0
  JMS_IBM_MsgType:8
  JMSXUserID:hollingl
  JMSXApplID:QM.POLARIS.BROKER
A simple text message from the MQJMSPSIVT program

Reply string equals original string
Closing TopicSubscriber
Closing TopicPublisher
Closing Session
Closing Connection
PSIVT completed OK
PSIVT finished
```

Pub/Sub 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`

These objects may be defined using the MQ JMS Administration Tool (see Chapter 5, “Using the MQ JMS administration tool” on page 27) using the following commands:

```
DEFINE TCF(ivtTCF)
```

to define the TopicConnectionFactory, and:

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

to define the Topic.

These definitions assume that a default queue manager is available, on which the broker is running. For details on configuring these objects to use a non-default queue manager, see “Administering JMS objects” on page 31. These objects should reside in a context pointed to by the `-url` command-line parameter described below.

To run the test using the JNDI, enter the following command:

```
PSIVTRun -url <purl> [-icf <initcf>] [-t]
```

where:

-t = turn tracing on (default: tracing off)

-url <purl> = specify the URL of the JNDI location in which the administered objects reside

-icf <initcf> = specify the initialContextFactory for JNDI [`com.sun.jndi.ldap.LdapCtxFactory`]

If the test completes successfully, output similar to that from the non-JNDI output should be seen, except that the 'create' `QueueConnectionFactory` and `Queue` lines should indicate retrieval of the object from JNDI instead.

PSIVT error recovery

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

- If you see the message:

  ```
  *** The broker is not running! Please start it using 'strmqbrk' ***
  ```

  this indicates that the broker is installed on the target queue manager, but its control queue contains outstanding messages. This indicates that the broker is not running, and it should be started using the `strmqbrk` command. (See “Additional setup for Pub/Sub mode” on page 18.)

- If the following message is displayed:

  ```
  Unable to connect to queue manager: <default>
  ```

  ensure that your MQSeries system has configured a default queue manager.

- If the following message is displayed:

  ```
  Unable to connect to queue manager: ...
  ```

ensure that the administered TopicConnectionFactory the PSIVT uses is configured with a valid queue manager name (or, if you are using the `-nojndi` option, you have supplied a valid queue manager using 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 the PSIVT uses is configured with the name of the queue manager on which the broker is installed (or, if you are using the `-nojndi` option, you have supplied a queue manager name using the `-m` option).

Running your own programs

For information on writing your own MQ JMS programs, see Chapter 10, “Writing MQ JMS programs” on page 143.

MQ JMS includes a utility file, `runjms` (`runjms.bat` on Windows NT) to help you to run programs that have been provided for you, or that you have written.

The utility provides default locations for the trace and log files and enables you to add any application runtime parameters that your application needs. As supplied, the script assumes that the environment variable MQ_JAVA_INSTALL_PATH has been set to the directory in which the MQ JMS is installed, and that the subdirectories `trace` and `log` within that directory are to be used for trace and log output. These are only suggested locations and the script can be edited to use any directory you choose.

Use the following command to run your application:

```
runjms <classname of application> [application specific arguments]
```

For information on writing MQ JMS applications and applets, see Part 3, “Programming with MQ JMS” on page 139.

Solving problems

If a program does not complete successfully, try running the installation verification program, which is described in Chapter 4, “Using MQ JMS” on page 17, and follow the advice given in the diagnostic messages.

Tracing programs

The MQJMS trace facility is provided to assist IBM staff diagnose customer problems.

Trace is disabled by default since the output rapidly becomes large, and is unlikely to be of use to the customer under normal circumstances. If you are asked to provide trace output it can be enabled by setting the Java property MQJMS_TRACE_LEVEL to one of the following values:

off no tracing

on traces MQ JMS calls only

base traces both MQ JMS calls and the underlying MQ base Java calls

For example:

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

By default, trace is output to a file named `mqjms.trc` in the current working directory. It can be redirected to a different directory using the Java property `MQJMS_TRACE_DIR`. For example:

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

The `runjms` utility script sets these properties using environment variables `MQJMS_TRACE_LEVEL` and `MQ_JAVA_INSTALL_PATH`: as follows:

```
java -DMQJMS_LOG_DIR=%MQ_JAVA_INSTALL_PATH%\log
-DMQJMS-TRACE_DIR=%MQ_JAVA_INSTALL_PATH%\trace
-DMQJMS_TRACE_LEVEL=%MQJMS_TRACE_LEVEL% %1 %2 %3 %4 %5 %6 %7 %8 %9
```

This is only a suggestion and can be modified as required.

Logging

The 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.

The output may be redirected to a file using a Java property which specifies the new location, for example:

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

The utility script `runjms` in the bin directory of the MQJMS installation sets this property to

```
<MQ_JAVA_INSTALL_PATH>/log
```

(Where `MQ_JAVA_INSTALL_PATH` is the path to your MQ JMS installation.) This is a suggestion, and can be modified as required.

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

```
formatLog <inputfile> <outputfile>
```

Chapter 5. Using the MQ JMS administration tool

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

The JMS objects that can be administered by the tool are:

- MQQueueConnectionFactory
- MQTopicConnectionFactory
- MQQueue
- MQTopic

These objects are described in more detail in “Administering JMS objects” on page 31.

The tool also allows administrators to manipulate directory namespace subcontexts within the JNDI. See “Manipulating subcontexts” on page 30.

Invoking the Administration tool

The administration tool has a command line interface which can be used interactively or to start a batch process. The interactive mode provides a command prompt where administration commands can be entered. 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 28)

A command prompt is displayed indicating 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 28). As the user traverses the directory namespace, the prompt changes to reflect this, so that the current context is always displayed in the prompt.

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

```
JMSAdmin <test.scp
```

where *test.scp* is a script file containing administration commands (see “Administration commands” on page 29). The last command in the file must be the `END` command.

Configuration

The administration tool needs to be configured with values for the following three parameters:

INITIAL_CONTEXT_FACTORY This indicates the service provider being used by the tool. There are currently two supported values for this property:

- com.sun.jndi.ldap.LdapCtxFactory (for LDAP)
- com.sun.jndi.fscontext.RefFSContextFactory (for file system context)

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

- ldap://hostname/contextname (for LDAP)
- file:[drive:]/pathname (for file system context)

SECURITY_AUTHENTICATION This indicates whether security credentials are passed over to your service provider by JNDI. This parameter is used only if an LDAP service provider is being 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, then the property defaults to none. See “Security” on page 29 for more details about security with the administration tool.

These parameters are set in a configuration file, the name of which may be supplied with the `-cfg` command-line parameter, as described in “Invoking the Administration tool” on page 27. If no configuration filename is specified the tool attempts to load the default configuration file (`JMSAdmin.config`), looking 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 MQ JMS installation.)

The configuration file is a plain-text file consisting of a set of key-value pairs, separated by an '=' as 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 will not be 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. This file should be edited to suit the setup of your system.

Security

Administrators need to be aware of the effect of the SECURITY_AUTHENTICATION property described in “Configuration” on page 28.

- If this parameter is set to *none*, then no security credentials are passed to the service provider by JNDI and "anonymous authentication" is performed.
- If the parameter is set to either simple or CRAM-MD5, then security credentials in the form of a user distinguished name (User DN) and password are passed through JNDI to the underlying service provider.

If security credentials are required, then the user will be prompted for these when the tool initializes.

Note: The text typed is echoed to the screen, and this includes the password. Care should therefore be taken to ensure that passwords are not disclosed to unauthorized users.

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, then the tool displays an appropriate error message and terminates.

Administration commands

More detailed information about security and JNDI can be found in the documentation at Sun's Java website (http://java.sun.com). 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 9 on page 30. 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 anything between 1 and 20 parameters. Details of the verbs which take at least one parameter are discussed in later sections of this chapter. Verb names are not case sensitive.

Commands are usually terminated using the carriage return key, but this can be over-ridden by keying the '+' symbol directly before the carriage return. This allows the entering of 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 should be ignored.

Table 9. Administration verbs

Verb		Description
Standard	**Shortform**	
ALTER	ALT	Change at least one of the properties of a given administered object
DEFINE	DEF	Create and store an administered object, or create a new subcontext
DISPLAY	DIS	Display the properties of one or more stored administered objects, or the contents of the current context
DELETE	DEL	Remove one or more administered objects from the namespace, or remove an empty subcontext
CHANGE	CHG	Alter the current context, allowing the user to traverse the directory namespace anywhere below the initial context (pending security clearance)
COPY	CP	Make a copy of a stored administered object, storing it under an alternative name
MOVE	MV	Alter the name under which an administered object is stored
END		Close the administration tool

Manipulating subcontexts

The verbs `CHANGE`, `DEFINE`, `DISPLAY` and `DELETE` allow the user to manipulate directory namespace subcontexts and their use is described in Table 10.

Table 10. Syntax and description of commands used to manipulate subcontexts

Command syntax	Description
DEFINE CTX(ctxName)	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 invalid.
DISPLAY CTX	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.
DELETE CTX(ctxName)	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.
CHANGE CTX(ctxName)	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.

Administering JMS objects

This section describes the four types of objects the administration tool can handle. It includes details about each of their configurable properties and the verbs that can be used to manipulate them.

Object types

The four type of administered objects are shown in Table 11.

Table 11. The JMS object types that are handled by the administration tool

Object Type		Description
Java	**Keyword**	
MQQueueConnectionFactory	QCF	The MQSeries implementation of the JMS QueueConnectionFactory interface. This represents a factory object for creating connections in the point-to-point domain of JMS.
MQTopicConnectionFactory	TCF	The MQSeries implementation of the JMS TopicConnectionFactory interface. This represents a factory object for creating connections in the publish/subscribe domain of JMS.
MQQueue	Q	The MQSeries implementation of the JMS Queue interface. This represents a destination for messages in the point-to-point domain of JMS.
MQTopic	T	The MQSeries implementation of the JMS Topic interface. This represents a destination for messages in the publish/subscribe domain of JMS.

The 'keyword' column indicates the strings that can be substituted for `TYPE` in the commands shown in Table 12.

Verbs used with JMS objects

The verbs `ALTER`, `DEFINE`, `DISPLAY`, `DELETE`, `COPY`, and `MOVE` allow the user to manipulate administered objects in the directory namespace and their use is summarized in Table 12.

Table 12 (Page 1 of 2). Syntax and description of commands used to manipulate administered objects

Command syntax	Description
ALTER TYPE(name) [property]*	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 invalid.

Table 12 (Page 2 of 2). Syntax and description of commands used to manipulate administered objects

Command syntax	Description
DEFINE TYPE(name) [property]*	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 invalid or already exists, or if the properties supplied are invalid.
DISPLAY TYPE(name)	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.
DELETE TYPE(name)	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.
COPY TYPE(nameA) TYPE(nameB)	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.
MOVE TYPE(nameA) TYPE(nameB)	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.

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 33).

LDAP naming considerations

If you want 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 the user to refer to object and context names without a prefix. If a prefix is not supplied, then the tool automatically adds a default prefix (currently `cn=`) to the name supplied by the user, as 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`) is without 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 server configuration for Java objects” on page 285.

Properties

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

`PROPERTY_NAME(property_value)`

Property names are case-insensitive, and are restricted to the set of recognized names shown in Table 13. Valid property values depend on the property and are also indicated in the table.

Table 13 (Page 1 of 2). Property names and valid values

Property		**Valid values (defaults in bold)**
Standard	**Shortform**	
DESCRIPTION	DESC	Any string
TRANSPORT	TRAN	• **BIND** - Connections use MQSeries bindings. • CLIENT - Client connection is used
CLIENTID	CID	Any string
QMANAGER	QMGR	Any string
HOSTNAME	HOST	Any string
PORT		Any positive integer
CHANNEL	CHAN	Any string
CCSID	CCS	Any positive integer
RECEXIT	RCX	Any string
RECEXITINIT	RCXI	Any string
SECEXIT	SCX	Any string
SECEXITINIT	SCXI	Any string
SENDEXIT	SDX	Any string
SENDXITINIT	SDXI	Any string
TEMPMODEL	TM	Any string

Table 13 (Page 2 of 2). Property names and valid values

Property		Valid values (defaults in bold)
Standard	**Shortform**	
BROKERVER	BVER	**V1** - The broker contained in SupportPac `ma0c` is used.
BROKERPUBQ	BPUB	Any string
BROKERQMGR	BQM	Any string
BROKERCONQ	BCON	Any string
EXPIRY	EXP	• **APP** - Expiry may be defined by the JMS application. • UNLIM - No expiry occurs. • Any positive integer representing expiry in milliseconds.
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.
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.
TARGCLIENT	TC	• **JMS** - Client is a JMS application. • MQ - Client is a non-JMS, traditional MQSeries application.
ENCODING	ENC	See “The ENCODING property” on page 36
QUEUE	QU	Any string
TOPIC	TOP	Any string

Many of the properties are only relevant to a specific subset of the object types. Table 14 shows which property-object type combinations are valid, and brief descriptions of each property.

Table 14 (Page 1 of 2). The valid combinations of property and object type

Property	Valid object types				Description
	QCF	**TCF**	**Q**	**T**	
DESCRIPTION	Y	Y	Y	Y	A description of the stored object
TRANSPORT	Y	Y			Whether connections will use the MQ Bindings, or a client connection
CLIENTID	Y	Y			A string identifier for the client
QMANAGER	Y	Y	Y		The name of the queue manager to connect to
PORT	Y	Y			The port to connect to
HOSTNAME	Y	Y			The name of the host on which the queue manager resides

Table 14 (Page 2 of 2). The valid combinations of property and object type

Property	Valid object types				Description
	QCF	TCF	Q	T	
CHANNEL	Y	Y			The name of the channel being used
CCSID	Y	Y	Y	Y	The coded-character-set-ID in use on connections
RECEXIT	Y	Y			Fully-qualified class name of the receive exit being used
RECEXITINIT	Y	Y			Receive exit initialization string
SECEXIT	Y	Y			Fully-qualified class name of the security exit being used
SECEXITINIT	Y	Y			Security exit initialization string
SENDEXIT	Y	Y			Fully-qualified class name of the send exit being used
SENDEXITINIT	Y	Y			Send exit initialization string
TEMPMODEL	Y				Name of the model queue from which temporary queues are created
BROKERVER		Y			The version of the broker being used
BROKERPUBQ		Y			The name of the queue onto which published messages are put
BROKERQMGR		Y			The queue manager on which the broker is running
BROKERCONQ		Y			Broker's control queue name
EXPIRY			Y	Y	The period after which messages at a destination expire
PRIORITY			Y	Y	The priority for messages sent to a destination
PERSISTENCE			Y	Y	The persistence of messages sent to a destination
TARGCLIENT			Y	Y	The type of the client being used
ENCODING			Y	Y	The encoding scheme used for this destination
QUEUE			Y		The underlying name of the queue representing this destination
TOPIC				Y	The underlying name of the topic representing this destination

Note: Appendix A, “Mapping between Administration tool properties and programmable properties” on page 281 shows the relationship between properties set by the tool and programmable properties.

Property dependencies

Some properties have dependencies on each other. This may mean, that supplying a property is meaningless unless another property has been set to a particular value. The two specific property groups where this can occur are Client properties and Exit initialization strings.

Client properties

If the `TRANSPORT(CLIENT)` property has not been explicitly set on a connection factory, then the transport used on connections provided by the factory is 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
- SENDEXIT
- SENDEXITINIT

Attempting to set any of these properties without setting the `TRANSPORT` property to `CLIENT` causes an error.

Exit initialization strings

Setting of any of the exit initialization strings is invalid 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 which is either normal or reversed

decimal encoding which is either normal or reversed

floating-point encoding which is either IEEE normal, IEEE reversed or System/390®.

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

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

where the first character represents *integer encoding*, the second *decimal encoding*, and the third *floating-point encoding*. This gives us a set of 12 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)
```

Sample error conditions

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

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
```

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 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
```

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
```

Part 2. Programming with MQ base Java

Chapter 6. Introduction for programmers 41
Why should I use the Java interface? 41
The MQSeries classes for Java interface 42
Java Developer's Kit 42
MQSeries classes for Java class library 43

Chapter 7. Writing MQ base Java programs 45
Should I write applets or applications? 45
Connection differences 45
Client connections 45
Bindings mode 46
Defining which connection to use 46
Example code fragments 46
Example applet code 46
Example application code 50
Operations on queue managers 52
Setting up the MQSeries environment 52
Connecting to a queue manager 52
Accessing queues and processes 53
Handling messages 54
Handling errors 55
Getting and setting attribute values 55
Multithreaded programs 56
Writing user exits 57
Compiling and testing MQSeries classes for Java programs 58
Running MQSeries classes for Java applets 58
Running MQSeries classes for Java applications 58
Running MQSeries classes for Java applications under CICS Transaction Server for OS/390 59
Tracing MQSeries Java programs 59

Chapter 8. Environment-dependent behavior 61
Core details 61
Restrictions and variations for core classes 62
Version 5 extensions operating in other environments 63

Chapter 9. The MQSeries classes for Java classes and interfaces 67
MQChannelDefinition 68
Variables 68
Constructors 69
MQChannelExit 70
Variables 70
Constructors 72
MQDistributionList 73
Constructors 73
Methods 73
MQDistributionListItem 75
Variables 75
Constructors 76
MQEnvironment 77

Variables . . . 77
Constructors . . . 80
Methods . . . 80
MQException . . . 81
Variables . . . 81
Constructors . . . 81
MQGetMessageOptions . . . 83
Variables . . . 83
Constructors . . . 86
MQManagedObject . . . 87
Variables . . . 87
Constructors . . . 88
Methods . . . 88
MQMessage . . . 90
Variables . . . 90
Constructors . . . 98
Methods . . . 98
MQMessageTracker . . . 108
Variables . . . 108
MQProcess . . . 110
Constructors . . . 110
Methods . . . 110
MQPutMessageOptions . . . 112
Variables . . . 112
Constructors . . . 114
MQQueue . . . 115
Constructors . . . 115
Methods . . . 115
MQQueueManager . . . 123
Variables . . . 123
Constructors . . . 123
Methods . . . 124
MQC . . . 131
MQReceiveExit . . . 132
Methods . . . 132
MQSecurityExit . . . 134
Methods . . . 134
MQSendExit . . . 136
Methods . . . 136

Chapter 6. Introduction for programmers

This chapter contains general information for programmers. For more detailed information about writing programs see Chapter 7, “Writing MQ base Java programs” on page 45.

Why should I use the Java interface?

The MQSeries classes for Java programming interface makes the many benefits of Java available to you as a developer of MQSeries 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/distributed environments, and a lot of emphasis has been placed on security. Java programs cannot overrun their run-time stack, cannot corrupt memory outside of their process space, and when downloaded from the Internet 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 run-time system is present.

If you write your application using MQSeries classes for Java, users can download the Java byte codes for your program (called *applets*) from the Internet and run them 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 and users automatically receive the latest version the next time they access the applet. This can significantly reduce the costs involved in installing and updating traditional client applications where a large number of desktops are involved. 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 MQSeries 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 MQSeries classes for Java interface

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

```
MQBACK, MQBEGIN, MQCLOSE, MQCMIT, MQCONN, MQCONNX,
MQDISC, MQGET, MQINQ, MQOPEN, MQPUT, MQPUT1, MQSET
```

These verbs all take, as a parameter, a handle to the MQSeries 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 MQSeries objects, which you act upon by calling methods on those objects, as in the following example.

Using the procedural interface, you disconnect from a queue manager 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 and you disconnect from it by calling the disconnect() method on that class.

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

Java Developer's Kit

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

MQSeries classes for Java requires JDK 1.1.6 or higher.

If you do not have the right JDK, go to the IBM Software Download Catalog which is available on the World Wide Web at location:

```
http: //www.ibm.com/software/download
```

You can also develop applications using the JDK included with the integrated development environment of IBM Visual Age for Java.

MQSeries classes for Java class library

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

The following classes are provided:

- MQChannelDefinition
- MQChannelExit
- MQDistributionList
- MQDistributionListItem
- MQEnvironment
- MQException
- MQGetMessageOptions
- MQManagedObject
- MQMessage
- MQMessageTracker
- MQPutMessageOptions
- MQProcess
- MQQueue
- MQQueueManager

and the following Java interfaces:

- MQC
- MQReceiveExit
- MQSecurityExit
- MQSendExit

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

```
import com.ibm.mq.*;
```

Chapter 7. Writing MQ base Java programs

To access MQSeries queues using MQSeries classes for Java, you write Java programs containing calls that put messages onto and get messages from MQSeries queues. The programs can take the form of Java *applets*, *servlets*, or Java *applications*.

This chapter provides information to assist with writing Java applets, servlets, and applications to interact with MQSeries systems. For details of individual classes, see Chapter 9, "The MQSeries classes for Java classes and interfaces" on page 67.

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 stand-alone.
- Applets can be downloaded from a Web server to a Web browser machine, but applications and servlets are not.

The following general rules apply:

- If you want to run your programs from machines that do not have MQSeries classes for Java installed locally, you should write applets.
- The native bindings mode of MQSeries 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 MQSeries classes for Java has some dependencies on the connection modes you want to use.

Client connections

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

- It supports only TCP/IP.
- It does not support connection tables.
- It does not read any MQSeries environment variables at startup.
- Information that would be stored in a channel definition and in environment variables is stored in a class called MQEnvironment, or 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 MQSeries classes for Java clients do not support the MQBEGIN verb or fast bindings.

For general information on MQSeries clients see the *MQSeries Clients* book.

Note: When you use the VisiBroker connection, the userid and password settings in MQEnvironment are not forwarded to the MQSeries server. The effective userid is that which applies to the IIOP server.

Bindings mode

The bindings mode of MQSeries 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 MQSeries 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
  ```

- For VisiBroker connections:

  ```
  MQC.TRANSPORT_PROPERTY, MQC.TRANSPORT_VISIBROKER
  MQC.ORB_PROPERTY, orb
  ```

MQEnvironment.hostname
Set the value of this variable follows:

- For client connections, set this to the hostname of the MQSeries server to which you want to connect
- For bindings mode, set this to null

Example code fragments

Two example code fragments are included in this section; Figure 1 on page 47 and Figure 2 on page 50. Each is written to use a particular connection with 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

```
// ============================================================================
//
// Licensed Materials - Property of IBM
//
// 5639-C34
//
// (c) Copyright IBM Corp. 1995,1999
//
// ============================================================================
// MQSeries Client for Java sample applet
//
// This sample runs as an applet using the appletviewer and HTML file,
// using the command :-
//            appletviewer MQSample.html
// Output is to the command line, NOT the applet viewer window.
//
// Note. If you receive MQSeries error 2 reason 2059 and you are sure your
// MQSeries and TCP/IP setup is correct,
// you should click on the "Applet" selection in the Applet viewer window
// select properties, and change "Network access" to unrestricted.

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

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 MQSeries 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 MQSeries 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 (Part 1 of 3). MQSeries classes for Java example applet

```
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 MQSeries 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,
                             null,           // default q manager
                             null,           // no dynamic q name
                             null);          // no alternate user id


    // Define a simple MQSeries 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 MQSeries 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);
```

Figure 1 (Part 2 of 3). MQSeries classes for Java example applet

```
      // 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 an MQSeries error?

    catch (MQException ex)
    {
      System.out.println("An MQSeries error occurred : Completion code " +
                         ex.completionCode +
                         " Reason code " + ex.reasonCode);
    }
    // Was it a Java buffer space error?
    catch (java.io.IOException ex)
    {
      System.out.println("An error occurred whilst writing to the
      message buffer: " + ex);
    }

  } // end of start

} // end of sample
```

Figure 1 (Part 3 of 3). MQSeries classes for Java example applet

Changing the connection to use VisiBroker for Java

Modify the line

```
MQEnvironment.properties.put (MQC.TRANSPORT_PROPERTY,
                              MQC.TRANSPORT_MQSERIES);
```

to

```
MQEnvironment.properties.put (MQC.TRANSPORT_PROPERTY,
                              MQC.TRANSPORT_VISIBROKER);
```

and add the following lines to initialize the ORB:

```
ORB orb=ORB.init(this,null);
MQEnvironment.properties.put(MQC.ORB_PROPERTY,orb);
```

You also need to add the following import statement to the beginning of the file:

```
import org.omg.CORBA.ORB;
```

You do not need to specify port number or channel if you are using VisiBroker.

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

```
// ============================================================================
// Licensed Materials - Property of IBM
// 5639-C34
// (c) Copyright IBM Corp. 1995, 1999
// ============================================================================
// MQSeries classes for Java sample application
//
// This sample runs as a Java application using the command :- java MQSample

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

import java.util.Hashtable;     // Required for properties

public class MQSample
{
  private String qManager = "your_Q_manager";  // define name of queue
                                               // manager to connect to.
  private MQQueueManager qMgr;                 // define a queue manager
                                               // object
  public static void main(String args[]) {
     new MQSample();
  }

  public MQSample() {
   try {

      java.util.Hashtable properties;

      // Create a connection to the queue manager
      qMgr = new MQQueueManager(qManager);

      // Set up the options on the queue we wish to open...
      // Note. All MQSeries Options are prefixed with MQC in Java.

      int openOptions = MQC.MQOO_INPUT_AS_Q_DEF |
                        MQC.MQOO_OUTPUT ;
```

Figure 2 (Part 1 of 2). MQSeries classes for Java example application

```
      // 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,
                               null,           // default q manager
                               null,           // no dynamic q name
                               null);          // no alternate user id

      // Define a simple MQSeries 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
      // put the message on the queue

      system_default_local_queue.put(hello_world,pmo);

      // get the message back again...
      // First define a MQSeries 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 an MQSeries error?
    catch (MQException ex)
    {
      System.out.println("An MQSeries error occurred : Completion code " +
                         ex.completionCode + " Reason code " + ex.reasonCode);
    }
    // Was it a Java buffer space error?
    catch (java.io.IOException ex)
    {
      System.out.println("An error occurred whilst writing to the message buffer: " + ex);
    }
  }
} // end of sample
```

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

Operations on queue managers

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

Setting up the MQSeries environment

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

The "C" based MQSeries 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, which allows you to specify the following details that are to be used during the connection attempt:

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

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

```
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 an MQSeries listener at port 1414. To specify a different port, use the code:

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

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

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

Note: If you are setting up a connection using VisiBroker for Java, see "Changing the connection to use VisiBroker for Java" on page 49.

Connecting to a queue manager

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

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

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

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

Calling the disconnect method causes all open queues and processes that you have accessed through that queue manager to be closed. It is good programming

practice, however, to close these resources yourself when you have finished using them. You do this with the close() method.

The commit() and backout() methods on a queue manager replace the MQCMIT and MQBACK calls of the procedural interface.

Accessing queues and processes

Queues and process are accessed using the MQQueueManager class. The MQOD (object descriptor structure) has been collapsed into the parameters of these methods. For example, to open a queue on a queue manager "queueManager", use the following code:

```
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, close it using the close() method, as in the following example:

```
queue.close();
```

With MQSeries classes for Java you can also create a queue 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 since this does not apply to processes.

The accessProcess method returns a new object of class MQProcess.

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

```
process.close();
```

With MQSeries classes for Java you can also create a process 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, and 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 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 *MQSeries 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 shows the creation of a message and putting it onto a queue:

```
// Build a new message containing my age followed by my name
MQMessage myMessage = new MQMessage();
myMessage.writeInt(25);

String name = "Wendy Ling";
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.

There is no need to specify a maximum message size because 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:

```
// 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);
```

The number format used by the read and write methods can be altered by setting the *encoding* member variable.

The character set to use for reading and writing strings can be altered by setting the *characterSet* member variable.

See “MQMessage” on page 90 for more details.

Note: Using the writeUTF() method of MQMessage automatically encodes the length of the string as well as the Unicode bytes it contains. When your message is to 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 an MQSeries 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 MQSeries. You can decide at which point in your program you want to deal with the possibility of failure by surrounding your code with 'try' and 'catch' blocks, as in the following example:

```
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);
}
```

Getting and setting attribute values

For many of the common attributes, the classes MQManagedObject, MQQueue, MQProcess, and MQQueueManager contain getXXX() and setXXX() methods which 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 using the *new* operator, it is automatically opened for 'inquiry'. When you access a process object using the accessProcess() method, it is automatically opened for 'inquiry'. When you access a queue object using the accessQueue() method, it is *not* automatically opened for either 'inquire' or 'set' operations, because automatically adding these options can cause problems with some types of remote queues. To use the inquire, set, and getXXX/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

There is no need for the SelectorCount, IntAttrCount and CharAttrLength parameters 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;
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 and, when the button is pressed, it fetches a message from the queue.

Because the Java runtime environment is inherently multithreaded, your application initialization will take place in one thread, and the code that is executed in response to the button press executes in a separate thread (the user interface thread).

With the "C" based MQSeries client this would cause a problem, since handles cannot be shared across multiple threads. MQSeries 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 MQSeries classes for Java ensures that, for a given connection (queue manager object instance), all access to the target MQSeries queue manager is synchronized. This means that a thread wishing to issue a call to a queue manager is blocked until all other calls in progress for that connection have completed. If you require simultaneous access to the same queue manager from within your program, create a new queue manager object for each thread requiring concurrent access. (This is equivalent to issuing a separate MQCONN call for each thread.)

Note: In the CICS Transaction Server for OS/390 environment, only the main (first) thread is allowed to issue CICS or MQSeries 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.

Writing user exits

MQSeries 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. There are three exit interfaces defined in the MQSeries package:

- MQSendExit
- MQReceiveExit
- MQSecurityExit

The following sample defines a class that implements all three:

```
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.

The *agentBuffer* parameter contains the data that is about to be sent (in the case of the send exit), or has just been received (in the case of the receive and security exits). There is no need for a length parameter, because the expression agentBuffer.length tells you the length of the array.

For the Send and Security exits, your exit code should return the byte array that you wish to be sent to the server. For a Receive exit, your code should return the modified data that you wish to be interpreted by the MQSeries classes for Java.

The simplest possible exit body is:

```
{
  return agentBuffer;
}
```

If your program is to run as a downloaded Java applet, note that under the security restrictions placed on it you will not be able to 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.

Compiling and testing MQSeries classes for Java programs

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

To compile a class "MyClass.java", use the command:

```
javac MyClass.java
```

Running MQSeries classes for Java applets

If you are writing 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>
<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 MQSeries classes for Java applications

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

```
java MyClass
```

Note: The '.class' extension is omitted from the class name.

Running MQSeries classes for 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 using the supplied CEDA transaction.
2. Ensure that the MQSeries CICS adapter is installed in your CICS system. (See *MQSeries System Management Guide* for details.)
3. Ensure that the JVM environment specified in the DHFJVM parameter of your CICS startup JCL includes appropriate CLASSPATH and LIBPATH entries.
4. Initiate the transaction using any of your normal processes.

For more information on running CICS Java transactions, refer to your CICS system documentation.

Tracing MQSeries Java programs

MQSeries classes for Java includes a trace facility, which can be used to produce diagnostic messages if you suspect 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:

```
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 you are running it from your local disk using the appletviewer command, you also have the option of redirecting the trace output to a file of your choice. The following code fragment shows an example of how to make the redirection to a file called `myapp.trc`:

```
import java.io.*;

try {
  FileOutputStream
  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 5 different levels of tracing:

1 Provides entry, exit and exception tracing
2 Provides parameter information in addition to 1
3 Provides transmitted and received MQSeries 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, issue the command `java_g` in place of `java` to run an application, or `appletviewer_g` instead of `appletviewer` to run an applet.

Notes:

1. `java_g` is not supported for High Performance Java (HPJ) applications on OS/390.

Chapter 8. Environment-dependent behavior

This chapter describes the behavior of the Java classes in the various environments in which they can be used. The MQSeries classes for Java classes allow you create applications that can be used in the following environments:

1. MQSeries Client for Java connected to an MQSeries V2.x server
2. MQSeries Client for Java connected to an MQSeries V5 server
3. MQSeries Bindings for Java executing on an MQSeries V5 server
4. MQSeries Bindings for Java executing on an MQSeries for MVS/ESA V1.2 server
5. MQSeries Bindings for Java executing on an MQSeries for MVS/ESA V1.2 server with CICS Transaction Server for OS/390 Version 1.3

In all cases the MQSeries classes for Java code makes use of services provided by the underlying MQSeries server. There are differences in the level of function (for example MQSeries V5 provides a superset of the function of V2), and in terms of behavior of some of the API calls and options. The differences in behavior are mainly minor, and mostly occur between the OS/390 (MQSeries for MVS/ESA) servers and the servers on other platforms.

MQSeries classes for Java provides a 'core' of classes, which provide consistent function and behavior in all the environments, and 'V5 extensions', which are designed for use only in environments 2 and 3. The core and extensions are documented below.

Core details

MQSeries classes for Java contains the following core of classes, which can be used in all environments with only the minor variations listed in “Restrictions and variations for core classes” on page 62.

- MQEnvironment
- MQException
- MQGetMessageOptions
  Excluding:
  - MatchOptions
  - GroupStatus
  - SegmentStatus
  - Segmentation
- MQManagedObject
  Excluding:
  - inquire()
  - set()
- MQMessage
  Excluding:
  - groupId
  - messageFlags
  - messageSequenceNumber
  - offset
  - originalLength

- MQPutMessageOptions
  Excluding:
  - knownDestCount
  - unknownDestCount
  - invalidDestCount
  - recordFields
- MQProcess
- MQQueue
- MQQueueManager
  Excluding:
  - begin()
  - accessDistributionList()
- 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

Although the core classes generally behave consistently across all environments, there are some minor restrictions and variations which are documented in Table 15.

Apart from these documented variations, the core classes give consistent behavior across all environments, even if the equivalent MQSeries classes normally have environment differences. In general, the behavior will be that expected in environments 2 and 3.

Table 15 (Page 1 of 2). Core classes restrictions and variations

Class or element	Restrictions and variations
MQGMO_LOCK MQGMO_UNLOCK MQGMO_BROWSE_MSG_UNDER_CURSOR	Cause MQRC_OPTIONS_ERROR when used in environments 4 or 5.
MQPMO_NEW_MSG_ID MQPMO_NEW_CORREL_ID MQPMO_LOGICAL_ORDER	Give errors except in environments 2 and 3. (See V5 extensions.)
MQGMO_SYNCPOINT_IF_PERSISTENT MQGMO_LOGICAL_ORDER MQGMO_COMPLETE_MESSAGE MQGMO_ALL_MSGS_AVAILABLE MQGMO_ALL_SEGMENTS_AVAILABLE	Give errors except in environments 2 and 3. (See V5 extensions.)
MQGMO_MARK_SKIP_BACKOUT	Causes MQRC_OPTIONS_ERROR except in environment 4 and 5.
MQCNO_FASTPATH_BINDING	Supported only in environment 3. (See V5 extensions.)
MQPMRF_* fields	Supported only in environments 2 and 3.

Table 15 (Page 2 of 2). Core classes restrictions and variations

Class or element	Restrictions and variations
Putting a message with MQQueue.priority > MaxPriority	Rejected with MQCC_FAILED and MQRC_PRIORITY_ERROR in environments 4 and 5. Other environments accept it with the warnings MQCC_WARNING and MQRC_PRIORITY_EXCEEDS_MAXIMUM and treat the message as if it were put with MaxPriority.
BackoutCount	Environments 4 and 5 return a maximum backout count of 255, even if the message has been backed out more than 255 times.
Default dynamic queue name	CSQ.* for environments 4 and 5. AMQ.* for other systems.
MQMessage.report options: MQRO_EXCEPTION_WITH_FULL_DATA MQRO_EXPIRATION_WITH_FULL_DATA MQRO_COA_WITH_FULL_DATA MQRO_COD_WITH_FULL_DATA MQRO_DISCARD_MSG	Not supported if a report message is generated by an OS/390 queue manager, although they may be set in all environments. This issue affects all Java environments, because the OS/390 queue manager could be distant from the Java application. Avoid relying on any of these options if there is a chance that an OS/390 queue manager could be involved.
MQQueueManager.commit() and MQQueueManager.backout()	In environment 5 these methods return MQRC_ENVIRONMENT_ERROR. In this environment applications should use the JCICS task synchronization methods: com.ibm.cics.server.Task.commit(), and com.ibm.cics.server.Task.rollback().
MQQueueManager constructor	In environments 4 and 5, if the options present in MQEnvironment (and the optional properties argument) imply a client connection, the constructor fails with MQRC_ENVIRONMENT_ERROR. In environments 4 and 5, the constructor may also return MQRC_CHAR_CONVERSION_ERROR. Ensure that the National Language Resources component of the OS/390 Language Environment is installed. In particular, ensure that conversions are available between the IBM-1047 and ISO8859-1 code pages. In environments 4 and 5, the constructor may also return MQRC_UCS2_CONVERSION_ERROR. The MQSeries classes for Java attempt to convert from Unicode to the queue manager code page, and default to IBM-500 if a specific code page is unavailable. Ensure that you have appropriate conversion tables for Unicode, which should be installed as part of the OS/390 C/C++ optional feature, and ensure that the Language Environment can locate the tables. See the *OS/390 C/C++ Programming Guide*, SC09-2362, for more information about enabling UCS-2 conversions.

Version 5 extensions operating in other environments

MQSeries classes for Java contains the following functions specifically designed to use the API extensions introduced in MQSeries V5. These functions operate as designed only in environments 2 and 3. This topic describes how they can be expected to behave in other environments.

MQQueueManager constructor option

An optional integer argument is included in the MQQueueManager constructor. 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 used in the MQSeries V5 bindings (environment 3). If it is used in any other environment, it is ignored.

MQQueueManager.begin() method

This can be used only in environment 3. In any other environment it fails with MQRC_ENVIRONMENT_ERROR.

MQPutMessageOptions options.

The following flags may be set into the MQPutMessageOptions options fields in any environment, but if used with a subsequent MQQueue.put() in any environment other than 2 or 3, the put() fails with MQRC_OPTIONS_ERROR:

- MQPMO_NEW_MSG_ID
- MQPMO_NEW_CORREL_ID
- MQPMO_LOGICAL_ORDER

MQGetMessageOptions options.

The following flags may be set into the MQGetMessageOptions options fields in any environment, but if used with a subsequent MQQueue.get() in any environment other than 2 or 3, the get() fails with MQRC_OPTIONS_ERROR:

- MQGMO_SYNCPOINT_IF_PERSISTENT
- MQGMO_LOGICAL_ORDER
- MQGMO_COMPLETE_MESSAGE
- MQGMO_ALL_MSGS_AVAILABLE
- MQGMO_ALL_SEGMENTS_AVAILABLE

MQGetMessageOptions fields

Values may be set into the following fields, regardless of the environment, but if the MQGetMessageOptions used on a subsequent MQQueue.get() is found to contain non-default values when running in any environment other than 2 or 3, the get() fails with MQRC_GMO_ERROR. This means that in environments other than 2 or 3, these fields will always be set to their initial values after every successful get().

- MatchOptions
- GroupStatus
- SegmentStatus
- Segmentation

Distribution Lists

The following classes are used to create Distribution Lists:

- MQDistributionList
- MQDistributionListItem
- MQMessageTracker

You can create and populate MQDistributionList and MQDistributionListItems in any environment, but you can only create and open MQDistributionList successfully in environments 2 and 3. An attempt to create and open one in any other environment is rejected with MQRC_OD_ERROR.

MQPutMessageOptions fields

Four fields in MQPMO are rendered as the following member variables in the MQPutMessageOptions class:

- knownDestCount
- unknownDestCount
- invalidDestCount
- recordFields

Although primarily intended for use with distribution lists, the MQSeries V5 server also fills in the DestCount fields after an MQPUT to a single queue. For example, if the queue resolves to a local queue, then knownDestCount is set to 1 and the other two fields to 0. In environments 2 and 3, the values set by the V5 server are returned in the MQPutMessageOptions class. In the other environments return 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 to 0.

recordFields is used with distribution lists. A value may be written into recordFields at any time, regardless of the environment, but is ignored if the MQPutMessage options are used on a subsequent MQQueue.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 non-default values, and then does a put() to or get() in an environment other than 2 or 3, the put() or get() raises an exception (MQRC_MD_ERROR). A successful put() or get() in an environment other than 2 or 3, always leaves the new MQMD fields set to their default values. A grouped or segmented message should not normally be sent to a Java application running against a queue manager that is not MQSeries Version 5 or higher. If such an application does issue a get, and the physical message to be retrieved happens to be part of a group or segmented message (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 containing the values for the new fields.

Chapter 9. The MQSeries classes for Java classes and interfaces

This chapter describes all the MQSeries 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
- MQPutMessageOptions
- MQProcess
- MQQueue
- MQQueueManager

and the following interfaces:

- MQC
- MQReceiveExit
- MQSecurityExit
- MQSendExit

MQChannelDefinition

```
java.lang.Object
  |
  └─ com.ibm.mq.MQChannelDefinition
```

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 MQSeries in bindings mode.

Variables

channelName
`public String channelName`

The name of the channel through which the connection is established.

queueManagerName
`public String queueManagerName`

The name of the queue manager to which the connection is made.

maxMessageLength
`public int maxMessageLength`

The maximum length of message that can be sent to the queue manager.

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.

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.

connectionName
`public String connectionName`

The TCP/IP hostname of the machine on which the queue manager resides.

remoteUserId

```
public String remoteUserId
```

The user id used to establish the connection.

remotePassword

```
public String remotePassword
```

The password used to establish the connection.

Constructors

MQChannelDefinition

```
public MQChannelDefinition()
```

MQChannelExit

```
java.lang.Object
   |
   └─ com.ibm.mq.MQChannelExit
```

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 MQSeries client for Java should take next.

Note: This class does not apply when connecting directly to MQSeries in bindings mode.

Variables

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
```

MQXR_INIT
```
public final static int MQXR_INIT
```

MQXR_TERM
```
public final static int MQXR_TERM
```

MQXR_XMIT
```
public final static int MQXR_XMIT
```

MQXR_SEC_MSG
```
public final static int MQXR_SEC_MSG
```

MQXR_INIT_SEC
```
public final static int MQXR_INIT_SEC
```

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
```

MQXCC_CLOSE_CHANNEL
```
public final static int MQXCC_CLOSE_CHANNEL
```

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_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.

MQXR_SEC_MSG
Indicates to the security exit that a security message has been received from the queue manager.

MQXR_INIT_SEC
Indicates that the exit is to initiate the security dialog with the queue manager.

exitResponse

```
public int exitResponse
```

Set by the exit to indicate the action that MQSeries classes for Java should take next. Valid values are:

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 MQSeries client for Java.

MQXCC_SUPPRESS_FUNCTION
Set by the security exit to indicate that communications with the queue manager should be shut down.

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.

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_CLOSE_CHANNEL
Set by any exit to indicate that the connection to the queue manager should be closed.

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.

exitUserArea
```
public byte exitUserArea[]
```
A storage area available for the exit to use. Any data placed in the exitUserArea is preserved by the MQSeries Client for Java across exit invocations with the same exitID. (That is to say, the send, receive, and security exits each have their own, independent, user areas.)

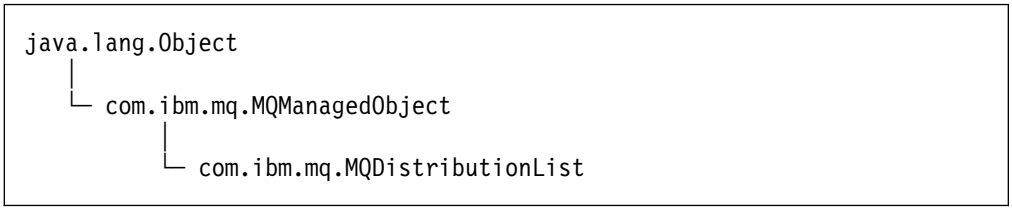
capabilityFlags
```
public static final int capabilityFlags
```
Indicates the capability of the queue manager. Only the MQC.MQCF_DIST_LISTS flag is supported.

fapLevel
```
public static final int fapLevel
```
The negotiated Format and Protocol (FAP) level.

Constructors

MQChannelExit
```
public MQChannelExit()
```

MQDistributionList



public class **MQDistributionList**
extends **MQManagedObject** (See page 87.)

Note: You can use this class only when connected to an MQSeries Version 5 (or higher) queue manager.

An MQDistributionList is created with MQDistributionList constructor or with 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 *MQSeries 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 129 for details of the remaining parameters.

Methods

put

```
public synchronized void put(MQMessage message,
             MQPutMessageOptions putMessageOptions ) Throws
```
MQException.

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 112 for details.)

Throws MQException if the put fails.

getFirstDistributionListItem

```
public MQDistributionListItem getFirstDistributionListItem()
```

Returns the first item in the distribution list, or *null* if the list is empty.

getValidDestinationCount

```
public int getValidDestinationCount()
```

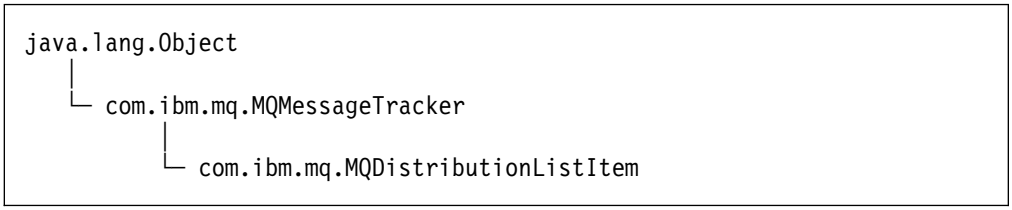
Returns the number of items in the distribution list that were opened successfully.

getInvalidDestinationCount

```
public int getInvalidDestinationCount()
```

Returns the number of items in the distribution list that failed to open successfully.

MQDistributionListItem



public class **MQDistributionListItem**
extends **MQMessageTracker** (See page 108.)

Note: You can use this class only when connected to an MQSeries 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".

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 "".

queueManagerName
`public String queueManagerName`

The name of the queue manager on which the queue is defined.

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

```
java.lang.Object
   |
   └─ com.ibm.mq.MQEnvironment
```

public class **MQEnvironment**
extends **Object**

Note: All the methods and attributes of this class apply to the MQSeries classes for Java client connections, but only enableTracing, disableTracing, properties, and version_notice apply to bindings connections.

MQEnvironment contains static member variables which control the environment in which an MQQueueManager object (and its corresponding connection to MQSeries) 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 constructing an MQQueueManager instance.

Variables

Note: Variables marked with * do not apply when connecting directly to MQSeries in bindings mode.

version_notice

```
public final static String version_notice
```

The current version of MQSeries classes for Java.

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 134.

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 136.

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 132.

hostname*

```
public static String hostname
```

The TCP/IP hostname of the machine on which the MQSeries 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.

port*

```
public static int port
```

The port to connect to. This is the port on which the MQSeries server is listening for incoming connection requests. The default value is 1414.

channel*

```
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.

userID*

```
public static String userID
```

Equivalent to the MQSeries 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 MQSeries client.

The default value is "".

password*

```
public static String password
```

Equivalent to the MQSeries 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 MQSeries client.

The default value is "".

properties

```
public static java.util.Hashtable properties
```

A set of key/value pairs defining the MQSeries 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:

Key	Value
MQC.CCSID_PROPERTY	Integer (Overrides MQEnvironment.CCSID.)
MQC.CHANNEL_PROPERTY	String (Overrides MQEnvironment.channel.)
MQC.CONNECT_OPTIONS_PROPERTY	Integer, defaults to MQC.MQCNO_NONE.
MQC.HOST_NAME_PROPERTY	String (Overrides MQEnvironment.hostname.)
MQC.ORB_PROPERTY	org.omg.CORBA.ORB (optional)
MQC.PASSWORD_PROPERTY	String (Overrides MQEnvironment.password.)
MQC.PORT_PROPERTY	Integer (Overrides MQEnvironment.port.)
MQC.RECEIVE_EXIT_PROPERTY	MQReceiveExit (Overrides MQEnvironment.receiveExit.)
MQC.SECURITY_EXIT_PROPERTY	MQSecurityExit (Overrides MQEnvironment.securityExit.)
MQC.SEND_EXIT_PROPERTY	MQSendExit (Overrides MQEnvironment.sendExit.)
MQC.TRANSPORT_PROPERTY	MQC.TRANSPORT_MQSERIES_BINDINGS or MQC.TRANSPORT_MQSERIES_CLIENT or MQC.TRANSPORT_VISIBROKER or MQC.TRANSPORT_MQSERIES (The default, which selects bindings or client based on the value of "hostname".)
MQC.USER_ID_PROPERTY	String (Overrides MQEnvironment.userID.)

CCSID*

```
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 MQSeries headers. All data in MQSeries 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 90.)

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.

Constructors

MQEnvironment

```
public MQEnvironment()
```

Methods

disableTracing

```
public static void disableTracing()
```

Turns off the MQSeries client for Java trace facility.

enableTracing

```
public static void enableTracing(int level)
```

Turns on the MQSeries 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)
```

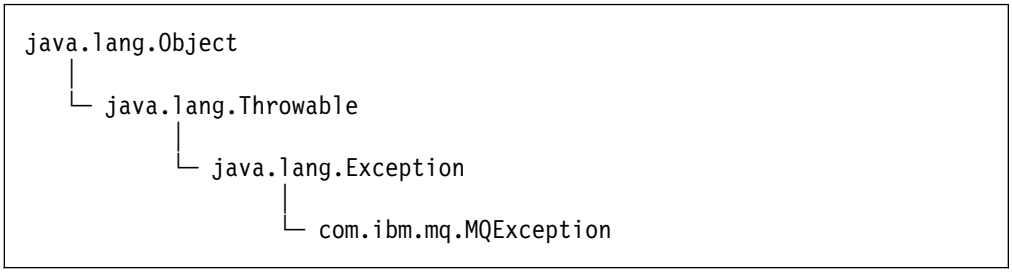
Turns on the MQSeries client for Java trace facility.

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

MQException



public class **MQException**
extends **Exception**

An MQException is thrown whenever an MQSeries 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 MQSeries completion codes, and constants beginning MQRC_ are MQSeries reason codes. The *MQSeries Application Programming Reference* contains a full description of these errors and their probable causes.

Variables

log

```
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.

completionCode

```
public int completionCode
```

MQSeries completion code giving rise to the error. The possible values are:

- MQException.MQCC_WARNING
- MQException.MQCC_FAILED

reasonCode

```
public int reasonCode
```

MQSeries reason code describing the error. For a full explanation of the reason codes refer to the *MQSeries Application Programming Reference*.

exceptionSource

```
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.

Constructors

MQException

```
public MQException(int completionCode,
                   int reasonCode,
                   Object source)
```

Construct a new MQException object.

Parameters

completionCode
The MQSeries completion code

reasonCode
The MQSeries reason code

source
The object in which the error occurred

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 61 for details.

Variables

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_NONE

MQC.MQGMO_WAIT
Wait for a message to arrive.

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_NO_SYNCPOINT
Get message without syncpoint control.

MQC.MQGMO_BROWSE_FIRST
Browse from start of queue.

MQC.MQGMO_BROWSE_NEXT
Browse from the current position in the queue.

MQC.MQGMO_BROWSE_MSG_UNDER_CURSOR*
Browse message under browse cursor.

MQC.MQGMO_MSG_UNDER_CURSOR
Get message under browse cursor.

MQC.MQGMO_LOCK*
Lock the message that is browsed.

MQC.MQGMO_UNLOCK*
Unlock a previously locked message.

MQC.MQGMO_ACCEPT_TRUNCATED_MSG
Allow truncation of message data.

MQC.MQGMO_FAIL_IF_QUIESCING
Fail if the queue manager is quiescing.

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.

MQC.MQGMO_SYNCPOINT_IF_PERSISTENT*
Get message with syncpoint control if message is persistent.

MQC.MQGMO_MARK_SKIP_BACKOUT*
Allow a unit of work to be backed out without reinstating the message on the queue.

Segmenting and grouping
MQSeries 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 as they are individual physical messages. Logical messages which 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_LOGICAL_ORDER*
Return messages in groups, and segments of logical messages, in logical order

MQC.MQGMO_COMPLETE_MSG*
Retrieve only complete logical 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

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.

resolvedQueueName

```
public String resolvedQueueName
```

This is an output field set by the queue manager 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.

matchOptions*

```
public int matchOptions
```

Selection criteria that determine which message is retrieved. The following match options can be set:

MQC.MQMO_MATCH_MSG_ID
Message id to be matched

MQC.MQMO_MATCH_CORREL_ID
Correlation id to be matched

MQC.MQMO_MATCH_GROUP_ID
Group id to be matched

MQC.MQMO_MATCH_MSG_SEQ_NUMBER
Match message sequence number

MQC.MQMO_NONE
No matching required

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_NOT_IN_GROUP
Message is not in a group.

MQC.MQGS_MSG_IN_GROUP
Message is in a group, but is not the last in the group.

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.

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_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.

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.

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

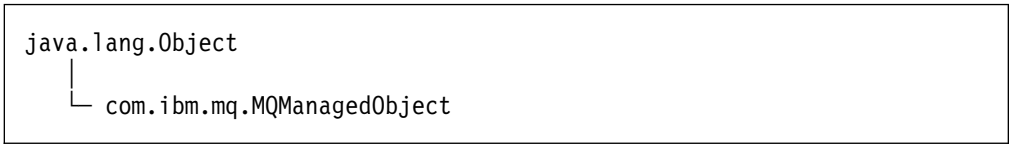
Constructors

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



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 specified (if any) when this resource was opened. Setting this attribute 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.

isOpen
`public boolean isOpen`

Indicates whether this resource is currently open. This attribute is *deprecated* and setting it has no effect.

connectionReference
`public  MQQueueManager connectionReference`

The queue manager to which this resource belongs. 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

Constructors

MQManagedObject

```
protected MQManagedObject()
```

Constructor method.

Methods

getDescription

```
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 *MQSeries 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.

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 *MQSeries 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.

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 MQSeries call 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 an MQSeries message. There is group of readXXX methods for reading data from a message, and a group of writeXXX data 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

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_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******
- 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**

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
```

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

messageType

```
public int messageType
```

Indicates the type of the message. The following values are currently defined by the system:

- MQC.MQMT_DATAGRAM
- MQC.MQMT_REQUEST
- MQC.MQMT_REPLY
- MQC.MQMT_REPORT

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.

expiry

```
public int 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

```
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_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. xreftext="encoding".

encoding

```
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 floating point data. 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 System/390

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
System/390 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.

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, which specifies that character data in the application message data is in the queue manager's character set. The additional character set values shown in Table 16 are supported.

Table 16 (Page 1 of 2). Character set identifiers

characterSet	Description
819	iso-8859-1 / latin1 / ibm819
912	iso-8859-2 / latin2 / ibm912
913	iso-8859-3 / latin3 / ibm913
914	iso-8859-4 / latin4 / ibm914
915	iso-8859-5 / cyrillic / ibm915
1089	iso-8859-6 / arabic / ibm1089
813	iso-8859-7 / greek / ibm813
916	iso-8859-8 / hebrew / ibm916
920	iso-8859-9 / latin5 / ibm920
37	ibm037
273	ibm273
277	ibm277
278	ibm278
280	ibm280
284	ibm284
285	ibm285
297	ibm297
420	ibm420
424	ibm424
437	ibm437 / PC Original
500	ibm500
737	ibm737 / PC Greek
775	ibm775 / PC Baltic
838	ibm838
850	ibm850 / PC Latin 1
852	ibm852 / PC Latin 2
855	ibm855 / PC Cyrillic
856	ibm856
857	ibm857 / PC Turkish
860	ibm860 / PC Portuguese
861	ibm861 / PC Icelandic

Table 16 (Page 2 of 2). Character set identifiers

characterSet	Description
862	ibm862 / PC Hebrew
863	ibm863 / PC Canadian French
864	ibm864 / PC Arabic
865	ibm865 / PC Nordic
866	ibm866 / PC Russian
868	ibm868
869	ibm869 / PC Modern Greek
870	ibm870
871	ibm871
874	ibm874
875	ibm875
918	ibm918
921	ibm921
922	ibm922
930	ibm930
933	ibm933
935	ibm935
937	ibm937
939	ibm939
942	ibm942
948	ibm948
949	ibm949
950	ibm950 / Big 5 Traditional Chinese
964	ibm964 / CNS 11643 Traditional Chinese
970	ibm970
1006	ibm1006
1025	ibm1025
1026	ibm1026
1097	ibm1097
1098	ibm1098
1112	ibm1112
1122	ibm1122
1123	ibm1123
1124	ibm1124
1381	ibm1381
1383	ibm1383
2022	JIS
932	PC Japanese
954	EUCJIS
1250	Windows Latin 2
1251	Windows Cyrillic
1252	Windows Latin 1
1253	Windows Greek
1254	Windows Turkish
1255	Windows Hebrew
1256	Windows Arabic
1257	Windows Baltic
1258	Windows Vietnamese
33722	ibm33722
5601	ksc-5601 Korean
1200	Unicode
1208	UTF-8

format

```
public String format
```

A format name used by the sender of the message to indicate to the receiver the nature of the data in the message. 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_NONE
No format name

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_DEAD_LETTER_HEADER
Dead letter header

MQC.MQFMT_EVENT
Event message

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.

priority

```
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.

persistence

```
public int persistence
```

Message persistence. The following values are defined:

- MQC.MQPER_PERSISTENT
- MQC.MQPER_NOT_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.

messageId

```
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 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.

correlationId

```
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.MQCI_NONE allows *any* correlation identifier to match.

For an MQQueue.put()() call, this specifies the correlation identifier to use.

The default value is MQC.MQCI_NONE.

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.

replyToQueueName

```
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 "".

replyToQueueManagerName

```
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.

userId

```
public String userId
```

Part of the identity context of the message; it identifies the user that originated this message.

The default value is "".

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 "".

putApplicationType

```
public int putApplicationType
```

The type of application that put the message. This may be a system 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.

putApplicationName

```
public String putApplicationName
```

The name of the application that put the message. The default value is "".

putDateTime

```
public GregorianCalendar putDateTime
```

The time and date that the message was put.

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 "".

groupId

```
public byte[] groupId
```

A byte string that identifies the message group to which the physical message belongs.

The default value is "MQC.MQGI_NONE".

messageSequenceNumber

```
public int messageSequenceNumber
```

The sequence number of a logical message within a group.

offset

```
public int offset
```

In a segmented message, the offset of data in a physical message from the start of a logical message.

messageFlags

```
public int messageFlags
```

Flags controlling the segmentation and status of a message.

originalLength

```
public int originalLength
```

The original length of a segmented message.

Constructors

MQMessage

```
public MQMessage()
```

Create a new message with default message descriptor information and an empty message buffer.

Methods

getTotalMessageLength

```
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 116.

getMessageLength

```
public int getMessageLength
```

Throws IOException.

The number of bytes of message data in this MQMessage object.

getDataLength

```
public int getDataLength()
```

Throws MQException.

The number of bytes of message data remaining to be read.

seek

```
public void seek(int pos)
```

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 MQSeries APIs.

getDataOffset

```
public int getDataOffset()
```

Throws IOException.

Return the current cursor position within the message data (the point at which read and write operations take effect).

clearMessage

```
public void clearMessage()
```

Throws IOException.

Discard any data in the message buffer and set the data offset back to zero.

getVersion

```
public int getVersion()
```

Returns the version of the structure in use.

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.

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.

readDouble

```
public double readDouble()
```

Throws IOException, EOFException.

Read a double from the current position in the message buffer. The behavior of this method is determined by the value of the encoding member variable.

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

```
public float readFloat()
```

Throws IOException, EOFException.

Read a float from the current position in the message buffer. The behavior of this method is determined by the value of the encoding member variable.

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

```
public void readFully(byte b[])
```

Throws Exception, EOFException.

Fill the byte array `b` with data from the message buffer.

readFully

```
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

```
public int readInt()
```

Throws IOException, EOFException.

Read an integer from the current position in the message buffer. The behavior of this method is determined by the value of the encoding member variable.

A value of MQC.MQENC_INTEGER_NORMAL reads a big-endian integer, a value of MQC.MQENC_INTEGER_REVERSED reads a little-endian integer.

readInt4

```
public int readInt4()
```

Throws IOException, EOFException.

Synonym for readInt(), provided for cross-language MQSeries API compatibility.

readLine

```
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

```
public long readLong()
```

Throws IOException, EOFException.

Read a long from the current position in the message buffer. The behavior of this method is determined by the value of the encoding member variable.

A value of MQC.MQENC_INTEGER_NORMAL reads a big-endian long, a value of MQC.MQENC_INTEGER_REVERSED reads a little-endian long.

readInt8

```
public long readInt8()
```

Throws IOException, EOFException.

Synonym for readLong(), provided for cross-language MQSeries API compatibility.

readObject

```
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

```
public short readShort()
```

Throws IOException, EOFException.

readInt2

```
public short readInt2()
```

Throws IOException, EOFException.

Synonym for readShort(), provided for cross-language MQSeries API compatibility.

readUTF

```
public String readUTF()
```

Throws IOException.

Read a UFT string, prefixed by a 2-byte length field, from the current position in the message buffer.

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 behavior of this method is determined by the value of the encoding member variable.

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.

readUInt2

```
public int readUInt2()
```

Throws IOException, EOFException.

Synonym for readUnsignedShort(), provided for cross-language MQSeries API compatibility.

readString

```
public String readString(int length)
```

Throws IOException, EOFException.

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).

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 (-9999999..9999999). 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 (-999999999999999 to 999999999999999). 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.

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.

write

```
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.

writeDouble

```
public void writeDouble(double v)
```

Throws IOException

Write a double into the message buffer at the current position. The behavior of this method is determined by the value of the encoding member variable.

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 behavior of this method is determined by the value of the encoding member variable.

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 behavior of this method is determined by the value of the encoding member variable.

A value of MQC.MQENC_INTEGER_NORMAL writes a big-endian integer, a value of MQC.MQENC_INTEGER_REVERSED writes a little-endian integer.

writeInt4

```
public void writeInt4(int v)
```

Throws IOException.

Synonym for writeInt(), provided for cross-language MQSeries API compatibility.

writeLong

```
public void writeLong(long v)
```

Throws IOException.

Write a long into the message buffer at the current position. The behavior of this method is determined by the value of the encoding member variable.

A value of MQC.MQENC_INTEGER_NORMAL writes a big-endian long, a value of MQC.MQENC_INTEGER_REVERSED writes a little-endian long.

writeInt8

```
public void writeInt8(long v)
```

Throws IOException.

Synonym for writeLong(), provided for cross-language MQSeries API compatibility.

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 behavior of this method is determined by the value of the encoding member variable.

A value of MQC.MQENC_INTEGER_NORMAL writes a big-endian short, a value of MQC.MQENC_INTEGER_REVERSED writes a little-endian short.

writeInt2

```
public void writeInt2(int v)
```

Throws IOException.

Synonym for writeShort(), provided for cross-language MQSeries API compatibility.

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 behavior of this method is determined by the value of the encoding member variable.

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

```
public void writeDecimal4(int v)
```

Throws IOException.

Write a 4-byte packed decimal format number into the message buffer at the current position. The behavior of this method is determined by the value of the encoding member variable.

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

```
public void writeDecimal8(long v)
```

Throws IOException.

Write an 8-byte packed decimal format number into the message buffer at the current position. The behavior of this method is determined by the value of the encoding member variable.

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.

writeUTF

```
public void writeUTF(String str)
```

Throws IOException.

Write a UFT string, prefixed by a 2-byte length field, into the message buffer at the current position.

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.

MQMessageTracker

```
java.lang.Object
   |
   └─ com.ibm.mq.MQMessageTracker
```

public abstract class **MQMessageTracker**
extends **Object**

Note: You can use this class only when connected to an MQSeries Version 5 (or higher) queue manager.

This class is inherited by MQDistributionListItem (on page 75) where it is used to tailor message parameters for a given destination in a distribution list.

Variables

feedback

```
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_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.

messageId

```
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.

correlationId

```
public byte correlationId[]
```

This specifies the correlation identifier to use when the message is put.

The default value is MQC.MQCI_NONE.

accountingToken

```
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".

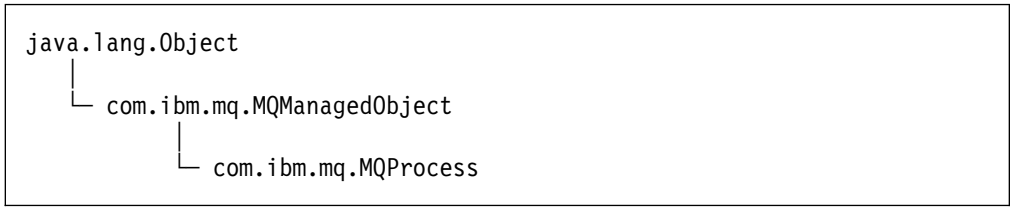
groupId

```
public byte[] groupId
```

A byte string that identifies the message group to which the physical message belongs.

The default value is "MQC.MQGI_NONE".

MQProcess



public class **MQProcess**
extends **MQManagedObject**. (on page 87.)

MQProcess provides inquire operations for MQSeries 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 123 for details of the remaining parameters.

Methods

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 81)

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
- 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.

close

```
public synchronized void close()
```

Throws MQException.

Override of "MQManagedObject.close" on page 89.

MQPutMessageOptions

```
java.lang.Object
   |
   └─ com.ibm.mq.MQPutMessageOptions
```

public class **MQPutMessageOptions**
extends **Object**

This class contains options that control the behavior of MQQueue.put().

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 “Version 5 extensions operating in other environments” on page 63 for details.

Variables

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_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.

MQC.MQPMO_NO_SYNCPOINT
Put a message without syncpoint control.

MQC.MQPMO_NO_CONTEXT
No context is to be associated with the message.

MQC.MQPMO_DEFAULT_CONTEXT
Associate default context with the message.

MQC.MQPMO_SET_IDENTITY_CONTEXT
Set identity context from the application.

MQC.MQPMO_SET_ALL_CONTEXT
Set all context from the application.

MQC.MQPMO_FAIL_IF_QUIESCING
Fail if the queue manager is quiescing.

MQC.MQPMO_NEW_MSG_ID*
Generate a new message id for each sent message.

MQC.MQPMO_NEW_CORREL_ID*
Generate a new correlation id for each sent message.

MQC.MQPMO_LOGICAL_ORDER*
Put logical messages and segments in message groups into their logical order.

MQC.MQPMO_NONE
No options specified. Do not use in conjunction with other options.

MQC.MQPMO_PASS_IDENTITY_CONTEXT
Pass identity context from an input queue handle.

MQC.MQPMO_PASS_ALL_CONTEXT
Pass all context from an input queue handle.

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_ALL_CONTEXT, set this field to refer to the MQQueue from which the context information should be taken.

The initial value of this field is null.

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_MSG_ID
Use the messageId attribute in the MQDistributionListItem.

MQC.MQPMRF_CORREL_ID
Use the correlationId attribute in the MQDistributionListItem.

MQC.MQPMRF_GROUP_ID
Use the groupId attribute in the MQDistributionListItem.

MQC.MQPMRF_FEEDBACK
Use the feedback attribute in the MQDistributionListItem.

MQC.MQPMRF_ACCOUNTING_TOKEN
Use the accountingToken attribute in the MQDistributionListItem.

The special value MQC.MQPMRF_NONE indicates that no fields are to be customized.

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.

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 may be different from the name of the queue manager from which the queue was accessed if the queue is a remote queue.

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.

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.

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.

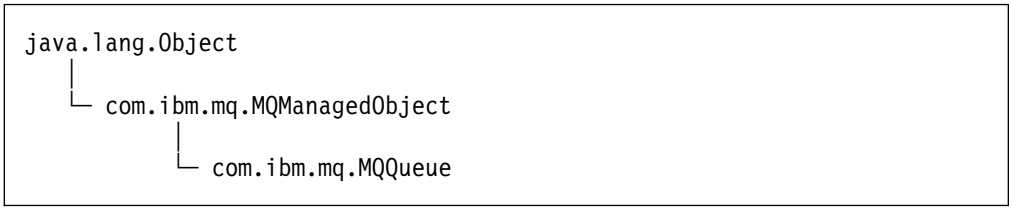
Constructors

MQPutMessageOptions

```
public MQPutMessageOptions()
```

Construct a new MQPutMessageOptions object with no options set, and a blank resolvedQueueName and resolvedQueueManagerName.

MQQueue



public class **MQQueue**
extends **MQManagedObject**. (See page 87.)

MQQueue provides inquire, set, put, and get operations for MQSeries queues. The inquire and set capabilities are inherited from MQ.MQManagedObject.

See also "MQQueueManager.accessQueue" on page 126.

Constructors

MQQueue:

```
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 126 for details of the remaining parameters.

Methods

get

```
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 212.)

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 MQSeries 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 MQSeries calls

until the get completes. If you need multiple threads to access MQSeries 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 83.)

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.

get

```
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 MQSeries 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 212.)

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 MQSeries 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 MQSeries calls until the get completes. If you need multiple threads to access MQSeries 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 83 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.

put

```
public synchronized void put(MQMessage message,
                             MQPutMessageOptions putMessageOptions)
```

Throws MQException.

Places a message onto the queue.

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 MQSeries queue.

Modifications to the MQMessage object after the put has completed do not affect the actual message on the MQSeries queue.

Performing a put updates the messageId and correlationId. This must be taken into consideration when making further calls to put/get using the same MQMessage object. Also, calling put does not clear the message data, so:

```
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 “MQGetMessageOptions” on page 83.)

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.

Note: All the following methods throw MQException if you call the method after you have closed the queue.

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

```
public int getDefinitionType()
```

Throws MQException.

Indicates how the queue was defined.

Returns
One of the following:

- MQC.MQQDT_PREDEFINED
- MQC.MQQDT_PERMANENT_DYNAMIC
- MQC.MQQDT_TEMPORARY_DYNAMIC

getMaximumDepth

```
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

```
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

```
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 MQSeries classes for Java (using accessQueue).

getOpenOutputCount

```
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 MQSeries classes for Java (using accessQueue).

getShareability

```
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

getInhibitPut

```
public int getInhibitPut()
```

Throws MQException.

Indicates whether or not put operations are allowed for this queue.

Returns
One of the following:

- MQC.MQQA_PUT_INHIBITED
- MQC.MQQA_PUT_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

getInhibitGet

```
public int getInhibitGet()
```

Throws MQException.

Indicates whether or not get operations are allowed for this queue.

Returns
The possible values are:

- MQC.MQQA_GET_INHIBITED
- MQC.MQQA_GET_ALLOWED

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

getTriggerControl

```
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

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

getTriggerData

```
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.

setTriggerData

```
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.

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.

setTriggerDepth

```
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.

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.

setTriggerMessagePriority

```
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.

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

setTriggerType

```
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

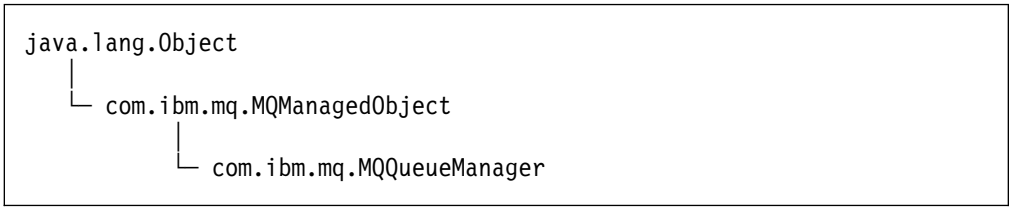
close

```
public synchronized void close()
```

Throws MQException.

Override of "MQManagedObject.close" on page 89.

MQQueueManager



public class **MQQueueManager**
extends **MQManagedObject**. (See page 87.)

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 61 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 MQSeries 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.

```
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 MQSeries 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 77.

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,
java.util.Hashtable properties)
```

The properties parameter takes a series of key/value pairs that describe the MQSeries 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 78.

Methods

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.

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.

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 MQSeries 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

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.

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.

getDistributionListCapable

```
public boolean getDistributionListCapable()
```

Indicates whether the queue manager supports distribution lists.

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.

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 "backout" that follows.

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 decription of "commit" above.

accessQueue

```
public synchronized MQQueue accessQueue
                          (
                          String queueName, int openOptions,
                          String queueManagerName,
                          String dynamicQueueName,
                          String alternateUserId
                          )
```

Throws MQException.

Establishes access to an MQSeries 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_BROWSE
Open to browse message

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

MQC.MQOO_INPUT_EXCLUSIVE
Open to get messages with exclusive access

MQC.MQOO_OUTPUT
Open to put messages

MQC.MQOO_INQUIRE
Open for inquiry - required if you wish to query properties

MQC.MQOO_SET
Open to set attributes

MQC.MQOO_SAVE_ALL_CONTEXT
Save context when message retrieved*****

MQC.MQOO_SET_IDENTITY_CONTEXT
Allows identity context to be set

MQC.MQOO_SET_ALL_CONTEXT
Allows all context to be set

MQC.MQOO_ALTERNATE_USER_AUTHORITY
Validate with the specified user identifier

MQC.MQOO_FAIL_IF_QUIESCING
Fail if the queue manager is quiescing

MQC.MQOO_BIND_AS_QDEF
Use default binding for queue

MQC.MQOO_BIND_ON_OPEN
Bind handle to destination when queue is opened

MQC.MQOO_BIND_NOT_FIXED
Do not bind to a specific destination

MQC.MQOO_PASS_ALL_CONTEXT
Allow all context to be passed

MQC.MQOO_PASS_IDENTITY_CONTEXT
Allow identity context to be passed

If more than one option is required the values can be added together or combined using the bitwise OR operator. See the MQSeries *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 to be 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.

See also "accessProcess" on page 128.

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 126 for details of the parameters.

queueManagerName, *dynamicQueueName*, and *alternateUserId* are set to "".

accessProcess

```
public synchronized MQProcess accessProcess
                           (
                         String processName,
                         int openOptions,
                         String queueManagerName,
                         String alternateUserId
                         )
```

Throws MQException.

Establishes access to an MQSeries 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 *MQSeries 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 to be 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
MQProcess that has been successfully opened.

Throws MQException if the open fails.

See also "MQQueueManager.accessQueue" on page 126.

accessProcess
This is a simplified version of the AccessProcess method previously described.

```
public synchronized MQProcess accessProcess
                         (
                        String processName,
                        int openOptions
                        )
```

This is a simplified version of the AccessQueue method previously described.

Parameters

processName
The name of the process to open.

openOptions
Options that control the opening of the process.

See "accessProcess" on page 128 for details of the options.

queueManagerName and *alternateUserId* are set to "".

accessDistributionList

```
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 to be 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 126.

accessDistributionList

This is a simplified version of the AccessDistributionList method previously described.

```
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" on page 129 for details of the parameters.

alternateUserId is set to "".

begin* (bindings connection only)

```
public synchronized void begin()
```

Throws MQException.

This method is supported only by the MQSeries classes for Java in bindings mode and it signals the queue manager that a new unit of work is starting.

isConnected

```
public boolean isConnected()
```

Returns the value of the isConnected variable.

MQC

public interface **MQC**
extends **Object**

The MQC interface defines all the constants used by the MQSeries Java programming interface. 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 can be found in Chapter 6, "MQSeries constants" in the *MQSeries Application Programming Reference* book.

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 MQSeries classes for Java.

Note: This interface does not apply when connecting directly to MQSeries 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:

```
// 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

```
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 MQSeries 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 MQSeries classes for Java what action to take next. See “MQChannelExit” on page 70 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 MQSeries 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 131
- “MQChannelDefinition” on page 68

MQSecurityExit

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 MQSeries 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:

```
// 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

```
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 MQSeries Client for Java what action to take next. See the “MQChannelExit” on page 70 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 131
- “MQChannelDefinition” on page 68

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 MQSeries Client for Java.

Note: This interface does not apply when connecting directly to MQSeries 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:

```
// 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...
  }
}
// in your main program...
MQEnvironment.sendExit = new MySendExit();
 ...    // other initialization
MQQueueManager qMgr        = new MQQueueManager("");
```

Methods

sendExit

```
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 MQSeries 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 MQSeries classes for Java what action to take next. See “MQChannelExit” on page 70 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 131
- “MQChannelDefinition” on page 68

Part 3. Programming with MQ JMS

Chapter 10. Writing MQ JMS programs 143
The JMS model 143
Building a Connection 144
Retrieving the factory from JNDI 144
Using the factory to create a connection 145
Creating factories at runtime 145
Choosing client or bindings transport 146
Obtaining a Session 147
Sending a message 147
Setting properties with the 'set' method 149
Message types 149
Receiving a message 150
Message Selectors 151
Asynchronous Delivery 151
Closing down 152
JVM hang at shutdown 152
Handling errors 152
Exception listener 152

Chapter 11. Programming Publish/Subscribe applications 153
Writing a simple Pub/Sub application 153
Import required packages 153
Obtain or create JMS objects 153
Publish messages 154
Receive subscriptions 155
Close down unwanted resources 155
Using Topics 155
Topic Names 155
Creating topics at runtime 157
Subscriber options 158
Creating non-durable subscribers 158
Creating durable subscribers 158
Using Message Selectors 158
Suppressing local publications 159
Combining the subscriber options 159
Solving Pub/Sub problems 159
Incomplete Pub/Sub close down 159
Handling broker reports 160

Chapter 12. JMS messages 161
Message selectors 161
Mapping JMS messages onto MQSeries messages 165
The MQRFH2 Header 166
JMS Fields and Properties with corresponding MQMD Fields 169
Mapping JMS fields onto MQSeries fields (Outgoing Messages) 170
Mapping MQSeries fields onto JMS Fields (Incoming Messages) 174
Mapping JMS to a Native MQSeries Application 175
Message Body 175

Chapter 13. JMS interfaces and classes 179

Sun Java Message Service classes and interfaces . . . 179
MQSeries JMS classes . . . 182
BytesMessage . . . 184
Methods . . . 184
Connection . . . 192
Methods . . . 192
ConnectionFactory . . . 195
MQSeries constructor . . . 195
Methods . . . 195
ConnectionMetaData . . . 198
MQSeries constructor . . . 198
Methods . . . 198
DeliveryMode . . . 200
Fields . . . 200
Destination . . . 201
MQSeries constructors . . . 201
Methods . . . 201
ExceptionListener . . . 203
Methods . . . 203
MapMessage . . . 204
Methods . . . 204
Message . . . 212
Fields . . . 212
Methods . . . 212
MessageConsumer . . . 224
Methods . . . 224
MessageListener . . . 226
Methods . . . 226
MessageProducer . . . 227
MQSeries constructors . . . 227
Methods . . . 227
MQQueueEnumeration * . . . 230
Methods . . . 230
ObjectMessage . . . 231
Methods . . . 231
Queue . . . 232
MQSeries constructors . . . 232
Methods . . . 232
QueueBrowser . . . 234
Methods . . . 234
QueueConnection . . . 236
Methods . . . 236
QueueConnectionFactory . . . 238
MQSeries constructor . . . 238
Methods . . . 238
QueueReceiver . . . 240
Methods . . . 240
QueueRequestor . . . 241
Constructors . . . 241
Methods . . . 241
QueueSender . . . 243
Methods . . . 243
QueueSession . . . 246
Methods . . . 246

Session 249
Fields 249
Methods 249
StreamMessage 253
Methods 253
TemporaryQueue 260
Methods 260
TemporaryTopic 261
MQSeries constructor 261
Methods 261
TextMessage 262
Methods 262
Topic 263
MQSeries constructor 263
Methods 263
TopicConnection 265
Methods 265
TopicConnectionFactory 267
MQSeries constructor 267
Methods 267
TopicPublisher 270
Methods 270
TopicRequestor 273
Constructors 273
Methods 273
TopicSession 275
MQSeries constructor 275
Methods 275
TopicSubscriber 278
Methods 278

Chapter 10. Writing MQ JMS programs

This chapter provides information to assist with writing MQ JMS applications. It provides 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 and it is important to understand this view, and how it maps onto the underlying MQSeries 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.

It is important to 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 MQSeries terms:

Connection
Provides a scope for temporary queues, as well as a place to hold the parameters that control how to connect to MQSeries (for example, the name of the queue manager, and the name of the remote host if using the MQSeries 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 MQSeries rules apply:

- Only a single operation can be in progress per HCONN at any given time, so the MessageProducers or MessageConsumers associated with a Session can not 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 'Pub/Sub' behavior.

The point to point versions are:

- QueueConnection
- QueueSession
- QueueSender
- QueueReceiver.

One of the key ideas in JMS is that it is possible, and strongly recommended, to write application programs using 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', which are so named because they 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, allowing the JMS application to be insulated from provider-specific information. Details of how to create and store factory objects are given in Chapter 5, “Using the MQ JMS administration tool” on page 27.

If you do not have a JNDI namespace available, see “Creating factories at runtime” on page 145.

Retrieving the factory from JNDI

To retrieve an object from a JNDI namespace an initial context must be setup as shown in this fragment taken from the `IVTRun` sample file:

```
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

(see Sun's JNDI documentation for more details of JNDI usage).

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`.

```
environment.put(Context.REFERRAL, "throw");
```

Having obtained an initial context, objects are retrieved from the namespace with the `lookup()` method. The following code retrieves a QueueConnectionFactory named `ivtQCF` from an LDAP-based namespace:

```
QueueConnectionFactory factory;
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:

```
QueueConnection connection;
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 as it requires references to MQSeries specific classes.

The following code creates a QueueConnectionFactory with all default settings:

```
factory = new com.ibm.mq.jms.MQQueueConnectionFactory();
```

(The `com.ibm.mq.jms.` prefix may be omitted if you choose to 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 17 on page 146 can be used to customize the factory with MQSeries specific information.

Starting the connection

The JMS specification defines that connections should be created in the 'stopped' state. Until the connection is started, no messages can be received by MessageConsumers that are associated with the connection. To start the connection, issue the following command:

```
connection.start();
```

Table 17. Set methods on MQQueueConnectionFactory	
Method	**Description**
setCCSID(int)	Used to set the `MQEnvironment.CCSID` property
setChannel(String)	The name of the channel for a client connection
setHostName(String)	The name of the host for a client connection
setPort(int)	The port for a client connection
setQueueManager(String)	The name of the queue manager
setTemporaryModel(String)	The name of a model queue used to generate a temporary destination as a result of a call to `QueueSession.createTemporaryQueue()`. It is recommended that this be the name of a temporary dynamic queue rather than a permanent dynamic queue.
setTransportType(int)	Specify how to connect to MQSeries. The options currently available are: • JMSC.MQJMS_TP_BINDINGS_MQ (the default) • JMSC.MQJMS_TP_CLIENT_MQ_TCPIP. `JMSC` is in the package `com.ibm.mq.jms`
setReceiveExit(String) setSecurityExit(String) setSendExit(String) setReceiveExitInit(String) setSecutityExitInit(String) setSendExitInit(String)	These methods exist to allow the use of the send, receive and security exits provided by the underlying MQSeries Classes for Java. The `set*Exit` methods take the name of a class which implements the relevant exit methods. (See the MQSeries 5.1 product documentation for details.) In addition the class must implement a constructor with a single `String` parameter. This string is used to provide any initialization data that may be required by the exit, and is set to the value provided in the corresponding `set*ExitInit` method.

Choosing client or bindings transport

MQJMS can communicate with MQSeries using either the client or bindings transports. Use of the Java bindings requires the JMS application and the MQSeries queue manager to be located on the same machine. The client permits the queue manager to be on a different machine to the application.

The transport to be used is determined by the contents of the connection factory object. Chapter 5, “Using the MQ JMS administration tool” on page 27 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:

```
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 has been 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:

```
QueueSession session;

boolean transacted = false;
session = connection.createQueueSession(transacted,
                                        Session.AUTO_ACKNOWLEDGE);
```

Note: A connection is thread safe, but sessions (and objects 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:

```
Queue ioQueue;
ioQueue = (Queue)ctx.lookup( qLookup );
```

MQ JMS provides an implementation of Queue in `com.ibm.mq.jms.MQQueue`. It contains properties for controlling the details of MQSeries specific behavior, but in many cases it is possible to use the default values. JMS defines a standard way of specifying the destination which minimizes the MQSeries 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.

MQ JMS accepts two forms for the `string` parameter of `createQueue()`.

- The first is the name of the MQSeries 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) and allows the specification of remote queues (queues on a queue manager other than the one to which you have connected), as well as the setting of 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, a further '/', the name of the queue, and optionally, a list of name-value pairs to 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 has been 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 18 lists the names that can be used in the name-value part of the URI. A disadvantage of this format is that it doesn't support symbolic names for the values, so where appropriate the 'special' values have been indicated below. It should be noted that these special values may be subject to change. (See “Setting properties with the 'set' method” on page 149 for an alternative method for setting properties.)

Table 18. Property names for queue URIs

Property	Description	Values
expiry	Lifetime of the message in milliseconds	0 for unlimited, positive integers for timeout (ms)
priority	Priority of the message	0 through 9, -1=QDEF, -2=APP
persistence	Whether the message should be 'hardened' to disk	1=non-persistent, 2=persistent, -1=QDEF, -2=APP
CCSID	Character set of the destination	integers - valid values listed in base MQSeries documentation.
targetClient	Whether the receiving application is JMS compliant or not	0=JMS, 1=MQ
encoding	How to represent numeric fields	An integer value as described in the base MQSeries documentation

where:

QDEF - a special value that means the property should be determined by the configuration of the MQSeries queue.

APP - a special value that means the JMS application can control this property.

After obtaining a Queue object (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 using the `send` method:

```
queueSender.send(outMessage);
```

Setting properties with the 'set' method

Queue properties can be set by creating an instance of `com.ibm.mq.jms.MQQueue` using the default constructor and then filling in the required values with public `set` methods. This method allows symbolic names to be used for the property values, but, 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.

```
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 19 shows the symbolic property values that are supplied with MQ JMS for use with the `set` methods.

Table 19. Symbolic values for queue properties

Property	Admin tool keyword	Values
expiry	UNLIM APP	JMSC.MQJMS_EXP_UNLIMITED JMSC.MQJMS_EXP_APP
priority	APP QDEF	JMSC.MQJMS_PRI_APP JMSC.MQJMS_PRI_QDEF
persistence	APP QDEF PERS NON	JMSC.MQJMS_PER_APP JMSC.MQJMS_PER_QDEF JMSC.MQJMS_PER_PER JMSC.MQJMS_PER_NON
targetClient	JMS MQ	JMSC.MQJMS_CLIENT_JMS-COMPLIANT JMSC.MQJMS_CLIENT_NONJMS_MQ
encoding	Integer(N) Integer(R) Decimal(N) Decimal(R) Float(N) Float(R) Native	JMSC.MQJMS_ENCODING_INTEGER_NORMAL JMSC.MQJMS_ENCODING_INTEGER_REVERSED JMSC.MQJMS_ENCODING_DECIMAL_NORMAL JMSC.MQJMS_ENCODING_DECIMAL_REVERSED JMSC.MQJMS_ENCODING_FLOAT_IEEE_NORMAL JMSC.MQJMS_ENCODING_FLOAT_IEEE_REVERSED JMSC.MQJMS_ENCODING_NATIVE

See “The ENCODING property” on page 36 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:

```
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 can be found in Chapter 13, “JMS interfaces and classes” on page 179.

Receiving a message

Messages are received with a QueueReceiver. This is created from a Session using the `createReceiver()` method. This method takes a Queue parameter to define where the messages are received from. See “Sending a message” on page 147 for a details of how to create a Queue object.

The sample program creates a receiver and reads back the test message with the following code:

```
QueueReceiver queueReceiver = session.createReceiver(ioQueue);
Message inMessage = queueReceiver.receive(1000);
```

The parameter in the receive call is a timeout in milliseconds which defines how long the method should wait if no message is immediately available. The parameter may be omitted, in which case the call blocks indefinitely. If you want no 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 then when the message is received the object 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, the 'instanceof' operator may be used 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 extracting the content from a TextMessage:

```
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 for selecting a subset of the messages on a queue to be returned by a receive call. When creating a QueueReceiver, a string can be provided which contains an SQL 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 can be found in Chapter 12, “JMS messages” on page 161.

The following example shows how to select for a user defined property named `myProp`:

```
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 has been created the selector is fixed for the lifetime of that receiver. This means that, if different selectors are required it is necessary to create new receivers.

Asynchronous Delivery

As an alternative to making calls to `QueueReceiver.receive()`, a method can be registered to be called automatically when a suitable message is available. The following fragment illustrates the mechanism:

```
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: Using 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 associated with a Session using asynchronous delivery.

Closing down

Garbage collection alone cannot release all MQSeries resources in a timely manner, particularly if the application requires the creation of many short lived JMS objects at the Session level or below. 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.

JVM hang at shutdown

If an MQ JMS application finishes without calling `Connection.close()` some JVMs will appear to hang. If this problem occurs, the application should be edited to include a call to `Connection.close()`, or the JVM may be terminated with `Ctrl-C`.

Handling errors

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 within it. For JMS this can be a valuable way of passing important detail from the underlying transport. In the case of MQ JMS, when MQSeries raises an MQException, this exception is usually included as the embedded exception within a JMSException.

The implementation of JMSException does not include the embedded exception in the output of its `toString()` method. It is therefore necessary to explicitly check for an embedded exception and print it out, as shown in the following fragment:

```
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

When using asynchronous message delivery, the application code is not able to catch exceptions raised by failures to receive messages, 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 can be found in Sun's JMS documentation.

Chapter 11. Programming Publish/Subscribe applications

This section provides the reader with an introduction to the programming model used when writing Publish/Subscribe applications using the MQSeries Classes for Java Message Service.

Writing a simple Pub/Sub application

This section walks through a simple MQSeries classes for Java Message Service(JMS) application.

Import required packages

An MQSeries classes for Java Message Service(JMS) application starts with a number of import statements which should at least include the following:

```
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 used for point-to-point as shown in the following:

Obtain a TopicConnectionFactory

The preferred method of doing this is to use JNDI lookup to, maintain portability of the application code. The following code initializes a JNDI context:

```
String CTX_FACTORY = "com.sun.jndi.ldap.LdapCtxFactory";
String INIT_URL    = "ldap://server.company.com/o=company_us,c=us";

Java.util.Hashtable env = new java.util.Hashtable();
env.put( Context.INITIAL_CONTEXT_FACTORY, CTX_FACTORY );
env.put( Context.PROVIDER_URL,            INIT_URL );
env.put( Context.REFERRAL,                "throw" );

Context ctx = null;
try {
   ctx = new InitialDirContext( env );
} catch( NamingException nx ) {
   // Add code to handle inability to connect to JNDI context
}
```

Note: The `CTX_FACTORY` and `INIT_URL` variables need customizing to suit your installation and your JNDI service provider .

The properties required by JNDI initialization are placed in a hashtable, that 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 using a lookup key which the administrator has defined:

```
TopicConnectionFactory factory;
factory = (TopicConnectionFactory)lookup("cn=sample.tcf");
```

Create a TopicConnection

This is created from the TopicConnectionFactory object. Connections are always initialized in a `stop` state and need to be started with the following code:

```
TopicConnection conn;
conn = factory.createTopicConnection();
conn.start();
```

Create a TopicSession

This is created with the TopicConnection. This method takes two parameters, one to signify whether the session is transacted, and one to specify the acknowledgement mode:

```
TopicSession session = conn.createTopicSession( false,
                                                Session.AUTO_ACKNOWLEDGE );
```

Obtain a Topic

This object is obtained from JNDI for use with TopicPublishers and TopicSubscribers that are created later. The following code retrieves a Topic:

```
Topic topic = null;
 try {
    topic = (Topic)ctx.lookup( "cn=sample.topic" );
 } catch( NamingException nx ) {
    // Add code to handle inability to retrieve Topic from JNDI
 }
```

Create consumers and producers of publications

Depending on the nature of the JMS client application being written, either a subscriber or a publisher (or both) needs to be created. Use the `createPublisher` and `createSubscriber` methods as follows:

```
// Create a publisher, publishing on the given topic
 TopicPublisher pub = session.createPublisher( topic );
// Create a subscriber, subscribing on the given topic
 TopicSubscriber sub = session.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 using the session, and then publishes the message:

```
// Create the TextMessage and place some data into it
TextMessage outMsg = session.createTextMessage();
outMsg.setText( "This is a short test string!" );

// Use the publisher to publish the message
pub.publish( outMsg );
```

Receive subscriptions

Subscribers need to be able to read the subscriptions that are delivered to them as in the following code:

```
// Retrieve the next waiting subscription
TextMessage inMsg = (TextMessage)sub.receive();

// Obtain the contents of the message
String payload = inMsg.getText();
```

This fragment of code performs a 'get-with-wait', which means that the receive call will block until a message becomes available. Alternative versions of the receive call are available (such as 'receiveNoWait') see “TopicSubscriber” on page 278.

Close down unwanted resources

It is important to free up all the resources used by the Pub/Sub application when it terminates. Use the `close()` method of objects that can be closed (publishers, subscribers, sessions and connections):

```
// Close publishers and subscribers
pub.close();
sub.close();

// Close sessions and connections
session.close();
conn.close();
```

Using Topics

This section discusses the use of JMS Topic objects in MQSeries classes for Java Message Service(JMS) applications

Topic Names

This section describes the use of topic names within MQSeries classes for Java Message Service(JMS).

Note: The JMS specification does not specify the exact details of the use and maintenance of topic hierarchies, and so this area may well vary from one provider to the next.

Topic names in MQJMS are arranged in a tree-like hierarchy, an example of which is shown in Figure 3 on page 156.

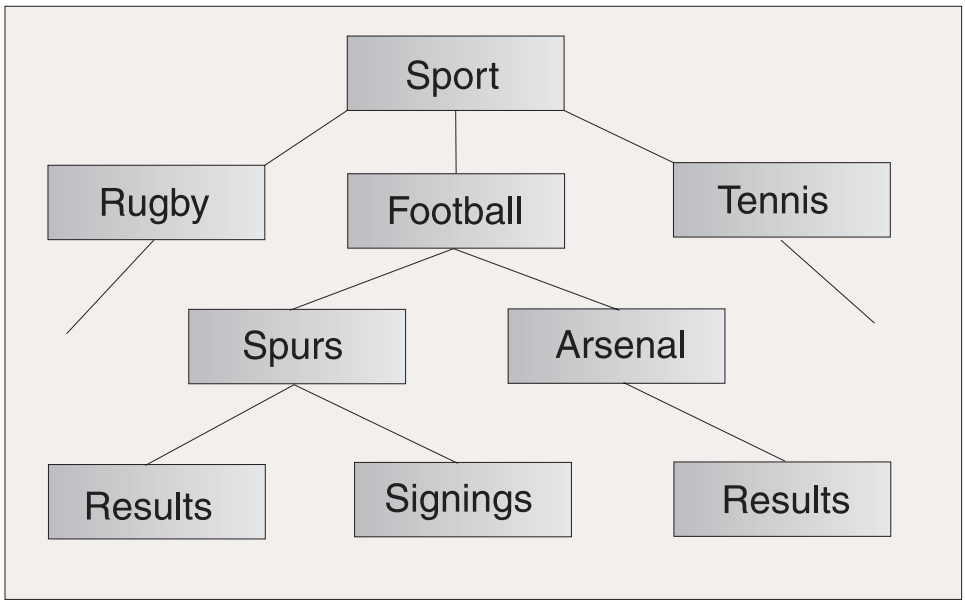


Figure 3. Topic name hierarchy

Levels in the tree are separated in a topic name by the '/' character, meaning that the 'Signings' node is represented as the topic name:

```
Sport/Football/Spurs/Signings
```

A powerful feature of the topic system in MQSeries classes for Java Message Service(JMS) is the use of wildcards. These allow subscribers to subscribe to more than one topic at a time. The '*' wildcard matches zero or more characters, whilst the '?' character matches a single character.

If a subscriber subscribes to the Topic representing the 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

There is no need explicitly to administer the topic hierarchies you use on the broker-side of your system. When the first publisher or subscriber on a given topic comes into existence, the state of the topics currently being published on and subscribed to is automatically created by the broker.

Note: A publisher cannot publish on a topic whose name contains wildcards.

Creating topics at runtime

There are four methods for creating Topic objects at run-time:

1. Construct a topic using the one-argument `MQTopic` constructor
2. Construct a topic 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 MQSeries 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 MQSeries classes for Java Message Service(JMS) 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 147.

The following code creates a topic for non-persistence, 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 creation of the object, the `baseTopicName` property is set 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. These should be set using the 'set' methods as described in “Setting properties with the 'set' method” on page 149. 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 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" );
```

Method 4: Using session.createTemporaryTopic()

A TemporaryTopic is a Topic which may only be consumed by subscribers 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 of using JMS subscribers. Some examples of their use are described in this section.

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.

Durable subscribers
These subscribers receive all the messages published on a topic, including those published while the subscriber is inactive

Creating non-durable subscribers

The subscriber created in Create consumers and producers of publications on page 154 is non-durable and is created with the following code:

```
// Create a subscriber, subscribing on the given topic
 TopicSubscriber sub = session.createSubscriber( topic );
```

Creating durable subscribers

Creating a durable subscriber is very similar to creating a non-durable subscriber, but a name which uniquely identifies the subscriber must also be provided:

```
// Create a durable subscriber, supplying a uniquely-identifying name
TopicSubscriber sub = session.createDurableSubscriber( topic, "D_SUB_000001" );
```

Unlike with non-durable subscribers, which automatically deregister themselves when their `close()` method is called (or when they fall out of scope), the system must be explicitly notified if the user wishes to terminate a durable subscription. Use the session's `unsubscribe()` method, passing in the unique name which created the subscriber:

```
// Unsubscribe the durable subscriber created above
session.unsubscribe( "D_SUB_000001" );
```

Using Message Selectors

Message selectors, which are discussed in detail in “Message Selectors” on page 151, can be used to filter out messages which do not satisfy given criteria. Message selectors are associated with a subscriber as follows:

```
// 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:

```
// Create a non-durable subscriber with the noLocal option set
TopicSubscriber sub = session.createSubscriber( topic, null, true );
```

Combining the subscriber options

The subscriber variations can be combined, allowing the user to create a durable subscriber which applies a selector and ignores local publications if they so wish. 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 );
```

Solving Pub/Sub problems

This section describes some of the problems you may experience when developing JMS client applications which use the publish/subscribe domain. Note that this section discusses problems specific to the Pub/Sub domain. Refer to “Handling errors” on page 152 and “Solving problems” on page 24 for more general trouble shooting guidance.

Incomplete Pub/Sub close down

It is important that JMS client applications surrender all external resources when they terminate. This is achieved by calling the `close()` method on all objects that can be closed once they are no longer required. For the pub/sub domain, these objects are:

- TopicConnection
- TopicSession
- TopicPublisher
- TopicSubscriber

The MQSeries classes for Java Message Service(JMS) 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 result in calls to 'close' on all TopicSubscribers and TopicPublishers the sessions created.

It is therefore important to call `connection.close()` for each of the connections an application has created, in order to ensure proper release of external resources.

There are some circumstances where this 'close' procedure may not complete, they include:

- Loss of an MQSeries client connection
- Unexpected application termination

In these circumstances, the `close()` is not called, and external resources remain open on the dead application's behalf. The main consequences of this are:

Broker state inconsistency
The MQSeries Message Broker may well contain registration information for subscribers and publishers which no longer exist. This means that the broker may continue forwarding messages to subscribers which will never receive them.

Subscriber queues remain
Part of the subscriber deregistration procedure is the removal of the underlying MQSeries queue that was used to receive subscriptions. If normal closure has not occurred, these queues remain and, in the case of broker state inconsistency, will continue to fill up with messages that will never be read.

Handling broker reports

The MQ JMS implementation uses report messages from the broker to confirm registration and deregistration commands. These reports are normally consumed by the MQSeries classes for Java Message Service(JMS) implementation, but under some error conditions they may be left 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 MQSeries classes for Java Message Service(JMS) 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. The application may also be used to clear the queue of messages when a problem has been diagnosed and/or fixed.

The compiled form of the tool is installed in the `<MQ_JAVA_INSTALL_PATH>/bin` directory, and should be invoked from this directory with the following command:

```
java PSReportDump [-m queueManager] [-clear]
```

where:

-m queueManager = specify the name of the queue manager to use

-clear = clear the queue of messages after dumping its contents

Ouput is sent to the screen, but can be redirected to file if required.

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 used by both clients and providers to identify and route messages.

Properties Each message contains a built-in facility for supporting application defined property values. Properties provide an efficient mechanism for supporting application defined message filtering.

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 for linking one message with another. It typically links a reply message with its requesting message. JMSCorrelationID can hold either a provider-specific message ID, an application-specific String or a provider-native byte[] value.

Message selectors

A Message contains a built-in facility for supporting application defined property values. In effect, this provides a mechanism for adding application specific header fields to a message. Properties allow an application, via message selectors, to have a JMS provider select/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 prior to sending a message. When a client receives a message, its properties are in read-only mode. If a client attempts to set properties at this point, a MessageNotWriteableException 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. Although JMS does not define a policy for what should or should not be made a

property, application developers should note that JMS providers are likely to 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 by message header the messages it's interested in. 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 of evaluation of a message selector is from left to right within precedence level. Parenthesis can be used to change this order. Predefined selector literals and operator names are written here in upper case; however, they are case insensitive.

A selector can contain:

- Literals
  - A string literal is enclosed in single quotes with single quote represented by doubled single quote such as '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, -57.9E2 or a numeric value with a decimal such as 7., -95.7, +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, and JMSType. 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 a property is referenced that does not exist in a message its value is NULL. If it does exist, its value is the corresponding property value.
- Whitespace is the same as that defined for Java: space, horizontal tab, form feed and line terminator.
- Expressions:
  - A selector is a conditional expression; a selector that evaluates to true matches; 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 () for ordering expression evaluation is supported.
- Logical operators in precedence order: NOT, AND, OR
- Comparison operators: =, >, >=, <, <=, <> (not equal)
  - Only like type values can be compared. One exception to this rule 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 the comparison of non-like types is attempted, 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'))
  - If 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; '%' stands for any sequence of characters (including the empty sequence); and 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 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 and 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 containing NULL values is defined by SQL 92 NULL semantics. A brief description of these semantics is provided here.

- 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 the reason for restricting exact numeric literals to those without a decimal (and the addition of numerics with a decimal as an alternate representation for an approximate numeric value).

SQL comments are not supported.

Mapping JMS messages onto MQSeries messages

This section describes how the JMS message structure that is described in the first part of this chapter is mapped onto an MQSeries message. It is of interest to programmers wishing to transmit messages between JMS and traditional MQSeries applications, or to people who wish to manipulate messages transmitted between two JMS applications - for example in a message broker implementation.

MQSeries messages are composed of three components:

- The MQSeries Message Descriptor (MQMD)
- An MQSeries 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 which can be set using the MQSeries JMS administration tool. As the MQRFH2 is used to carry JMS-specific information, it should always be included in the message when the sender is aware that the receiving destination is a JMS application. It would normally be omitted when sending a message directly to a non-JMS (MQSeries Native application) as such an application would not expect an MQRFH2 in its MQSeries message. Figure 4 shows the transformation of the structures:

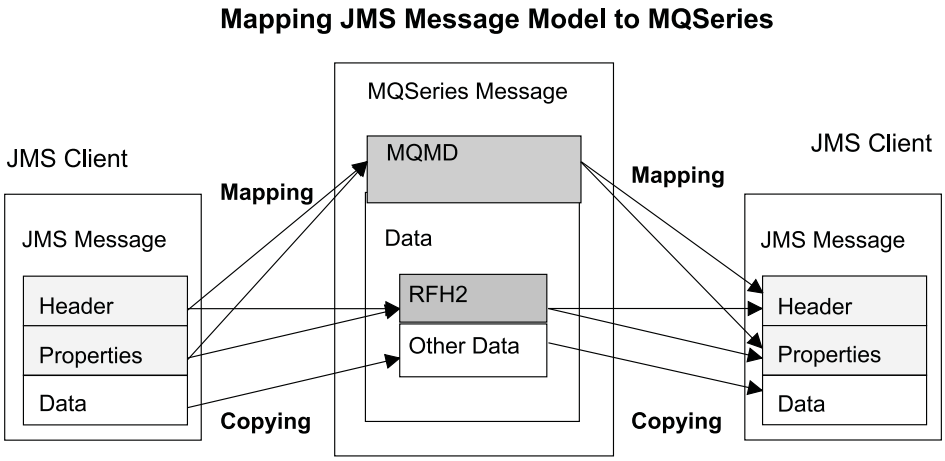


Figure 4. JMS to MQSeries mapping model

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, as 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 is used to carry JMS-specific data associated with the message content. The MQRFH2 Version 2 is an extensible header, and may be used to carry additional information, not directly associated with JMS, but this section covers only its use by JMS.

There are two parts of the header, a fixed and a variable portion.

Fixed portion

The fixed portion is modelled on the 'standard' MQSeries 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 valueset 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 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).

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 list:

1200	UCS2 open-ended
1208	UTF8
13488	UCS2 2.0 subset
17584	UCS2 2.1 subset (includes Euro symbol)

Variable Portion

The fixed portion is followed by the variable portion which contains a variable number of MQRFH2 Folders. Each folder contains a variable number of elements or properties. Folders are used to 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.

A full list of property names is shown in Table 20.

Table 20. MQRFH2 folders and properties used by JMS

JMS fields		**MQRFH2 fields**		
Name	**Java type**	**Folder name**	**Property name**	**Type/values**
JMSDestination	Destination	jms	Dst	string
JMSExpiration	long	jms	Exp	i8
JMSPriority	int	jms	Pri	i4
JMSDeliveryMode	int	jms	Dlv	i4
JMSCorrelationID	String	jms	Cid	string
JMSReplyTo	Destination	jms	Rto	string
JMSType	String	mcd	Type	string
JMSXGroupID	String	jms	Gid	string
JMSXGroupSeq	int	jms	Seq	i4
xxx (User Defined)	Any	usr.xxx	xxx	any
		mcd	Msd	jms_none jms_text jms_bytes jms_map jms_stream jms_object

The syntax used to express the properties in the variable portion is as follows:

NameValueLength (MQLONG)
Length in bytes of the NameValueData string immediately following 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:

```
<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:

```
<name dt="datatype">value</name>
```

The `dt="datatype"` element is optional and is omitted for many properties, as their datatype is predefined. If it is included, one or more space characters must be included before the 'dt='

`name` is the name of the property - see Table 20 on page 167

`datatype` must match, after folding, one of the literal values in Table 21

`value` is a string representation of the value to be conveyed, as shown in Table 21

A null value is encoded using the following syntax:

```
<name/>
```

Table 21. Property datatypes and values

Datatype	Value
string	Any sequence of characters excluding < and &
Boolean	The character 0 or 1 (1=="true")
bin.hex	Hexadecimal digits representing octets
i1	A number, expressed using digits 0..9, with optional sign (no fractions or exponent). Must lie in the range -128 to 127 inclusive
i2	A number, expressed using digits 0..9, with optional sign (no fractions or exponent). Must lie in the range -32768 to 32767 inclusive
i4	A number, expressed using digits 0..9, with optional sign (no fractions or exponent). Must lie in the range -2147483648 to 2147483647 inclusive
i8	A number, expressed using digits 0..9, with optional sign (no fractions or exponent). Must lie in the range -9223372036854775808 to 9223372036854775807 inclusive
int	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 doesn't wish to associate a particular precision with the property
r4	Floating point number, magnitude <= 3.40282347E+38, >= 1.175E-37 expressed using digits 0..9, optional sign, optional fractional digits, optional exponent
r8	Floating point number, magnitude <= 1.7976931348623E+308, >= 2.225E-307 expressed using digits 0..9, optional sign, optional fractional digits, optional exponent

A string value may contain spaces. The following escape sequences must be used in a string value:

`&` for the & character
`>` for the > character

The following escape sequences may be used, but are not required:

`<` for the < character
`&apos;` for the ' character
`&quot;` for the " character

JMS Fields and Properties with corresponding MQMD Fields

Table 22 lists the properties that are be mapped directly to MQMD fields.

Table 22. JMS properties mapping to MQMD fields

JMS field		**MQMD field**	
Header	**Java type**	**Field**	**C type**
JMSDeliveryMode	int	Persistence	MQLONG
JMSExpiration	long	Expiry	MQLONG
JMSPriority	int	Priority	MQLONG
JMSMessageID	String	MessageID	MQBYTE24
JMSTimestamp	long	PutDate PutTime	MQCHAR8 MQCHAR8
JMSCorrelationID	String	CorrelId	MQBYTE24
Properties			
JMSXUserID	String	UserIdentifier	MQCHAR12
JMSXAppID	String	PutApplName	MQCHAR28
JMSXDeliveryCount	int	BackoutCount	MQLONG
JMSXGroupID	String	GroupId	MQBYTE24
JMSXGroupSeq	int	MsgSeqNumber	MQLONG
Provider specific			
JMS_IBM_Report_Exception	int	Report	MQLONG
JMS_IBM_Report_Expiration	int	Report	MQLONG
JMS_IBM_Report_COA	int	Report	MQLONG
JMS_IBM_Report_COD	int	Report	MQLONG
JMS_IBM_Report_PAN	int	Report	MQLONG
JMS_IBM_Report_NAN	int	Report	MQLONG
JMS_IBM_Report_Pass_Msg_ID	int	Report	MQLONG
JMS_IBM_Report_Pass_Correl_ID	int	Report	MQLONG
JMS_IBM_MsgType	int	MsgType	MQLONG
JMS_IBM_Feedback	int	Feedback	MQLONG
JMS_IBM_Format	String	Format	MQCHAR8
JMS_IBM_PutApplType	int	PutApplType	MQLONG

Mapping JMS fields onto MQSeries fields (Outgoing Messages)

Table 23 shows how the header/property fields are mapped into MQMD/RFH2 fields at send() or publish() time.

For fields marked 'Set by Client', the value transmitted is the value held in the JMS message immediately prior to the send/publish() - the value in the JMS Message is left unchanged by the send/publish().

For fields marked 'set by Send Method', a value is assigned when the send/publish() is executed (any value held in the JMS Message is ignored) and 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 (Page 1 of 2). Outgoing message field mapping

JMS fields	**Xmitted in**		**Set by**
Name	**MQMD field**		
Header			
JMSDestination		MQRFH2	Send Method
JMSDeliveryMode	Persistence	MQRFH2	Send Method
JMSExpiration	Expiry	MQRFH2	Send Method
JMSPriority	Priority	MQRFH2	Send Method
JMSMessageID	MessageID		Send Method
JMSTimestamp	PutDate/PutTime		Send Method
JMSCorrelationID	CorrelId	MQRFH2	Client
JMSReplyTo	ReplyToQ/ReplyToQMgr	MQRFH2	Client
JMSType		MQRFH2	Client
JMSRedelivered			Receive-only
Properties			
JMSXUserID	UserIdentifier		Send Method
JMSXAppID	PutApplName		Send Method
JMSXDeliveryCount			Receive-only
JMSXGroupID	GroupId	MQRFH2	Client
JMSXGroupSeq	MsgSeqNumber	MQRFH2	Client
Provider specific			
JMS_IBM_Report_Exception	Report		Client
JMS_IBM_Report_Expiration	Report		Client

Table 23 (Page 2 of 2). Outgoing message field mapping			
JMS fields	**Xmitted in**		**Set by**
Name	**MQMD field**		
JMS_IBM_Report_COA/COD	Report		Client
JMS_IBM_Report_NAN/PAN	Report		Client
JMS_IBM_Report_Pass_Msg_ID	Report		Client
JMS_IBM_Report_Pass_Correl_ID	Report		Client
JMS_IBM_MsgType	MsgType		Client
JMS_IBM_Feedback	Feedback		Client
JMS_IBM_Format	Format		Client
JMS_IBM_PutApplType	PutApplType		Send Method

Mapping JMS header fields at send()/publish()

The following notes relate to the mapping of JMS fields at send()/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 148 for details of the URI notation).
- **JMSReplyTo to MQMD ReplyToQ, ReplyToQMgr, MQRFH2:** The Queue and QueueManager name is 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 148 for details of the URI notation).
- **JMSDeliveryMode to MQMD Persistence:** The JMSDeliveryMode value is set by the send/publish() Method or MessageProducer unless overridden by the Destination Object. 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 JMSDeliveryMode is set to a non-default value, 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 measured in milliseconds, but MQMD.expiry is in centiseconds.
  - If the send() method sets an unlimited time to live, then MQMD Expiry is set to MQEI_UNLIMITED, and no JMSExpiration is encoded in the MQRFH2.
  - If the send() method sets a limited time to live, and that time to live is less than 214748364.7 seconds (about 7 years), then 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, then MQMD.Expiry is set to MQEI_UNLIMITED, but 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 MQSeries. 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 MQSeries 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 set into the JMSMessageId field prior to a send() is overwritten.
- **JMSTimestamp from MQMD PutDate, PutTime:** After a send, the JMSTimestamp field is set equal to the date/time value given by the MQMD PutDate and PutTime fields. Any value set into the JMSMessageId field prior to a send() is overwritten.
- **JMSType to MQRFH2:** This string is set into the MQRFH2.
- **JMSCorrelationID to MQMD CorrelId, MQRFH2:** The JMSCorrelationID can hold one of the following:
  - **A provider specific message ID:** This should be a message identifier from a message previously sent or received, and so should be a string of 48 hexadecimal digits prefixed with 'ID:'. The prefix is removed and the remaining characters converted into binary and are then 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, is written into the MQMD CorrelID.

Mapping JMS Property Fields

These notes refer to the mapping of JMS property fields in MQSeries messages:

- **JMSXUserID from MQMD UserIdentifier:** JMSXUserID is set on return from send call.
- **JMSXAppID from MQMD PutApplName:** JSMXAppID 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 MQF_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 MQF_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 MQSeries 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 MQSeries 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.

JMS _IBM_Report_Exception	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

  Note: The Disposition report options cannot be set.

- **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 an MQSeries queue destination, MSGType is set to the value MQMT_REQUEST
  - If JMSReplyTo is not set, or is set to anything other than an MQSeries 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.

Mapping MQSeries fields onto JMS Fields (Incoming Messages)

Table 24 shows how the header/property fields are mapped into MQMD/MQRFH2 fields at send() or publish() time.

Table 24. Incoming message field mapping

JMS fields	**Retrieved from**	
Name	**MQMD field**	**MQRFH2**
JMS headers		
JMSDestination		jms.Dst
JMSDeliveryMode	Persistence	
JMSExpiration		jms.Exp
JMSPriority	Priority	
JMSMessageID	MessageID	
JMSTimestamp	PutDate PutTime	
JMSCorrelationID	CorrelId	jms.Cid
JMSReplyTo	ReplyToQ ReplyToQMgr	jms.Rto
JMSType		mcd.Type
JMSRedelivered	BackoutCount	
JMS properties		
JMSXUserID	UserIdentifier	
JMSXAppID	PutApplName	
JMSXDeliveryCount	BackoutCount	
JMSXGroupID	GroupId	jms.Gid
JMSXGroupSeq	MsgSeqNumber	jms.Seq
JMS provider specific		
JMS_IBM_Report_Exception	Report	
JMS_IBM_Report_Expiration	Report	
JMS_IBM_Report_COA	Report	
JMS_IBM_Report_COD	Report	
JMS_IBM_Report_PAN	Report	
JMS_IBM_Report_NAN	Report	
JMS_IBM_Report_ Pass_Msg_ID	Report	
JMS_IBM_Report_Pass_Correl_ID	Report	
JMS_IBM_MsgType	MsgType	
JMS_IBM_Feedback	Feedback	
JMS_IBM_Format	Format	
JMS_IBM_PutApplType	PutApplType	

Mapping JMS to a Native MQSeries Application

This section describes what happens if we send a message from a JMS Client application to a traditional MQSeries application which has no knowledge of MQRFH2 headers. Figure 5 is a diagram of the mapping.

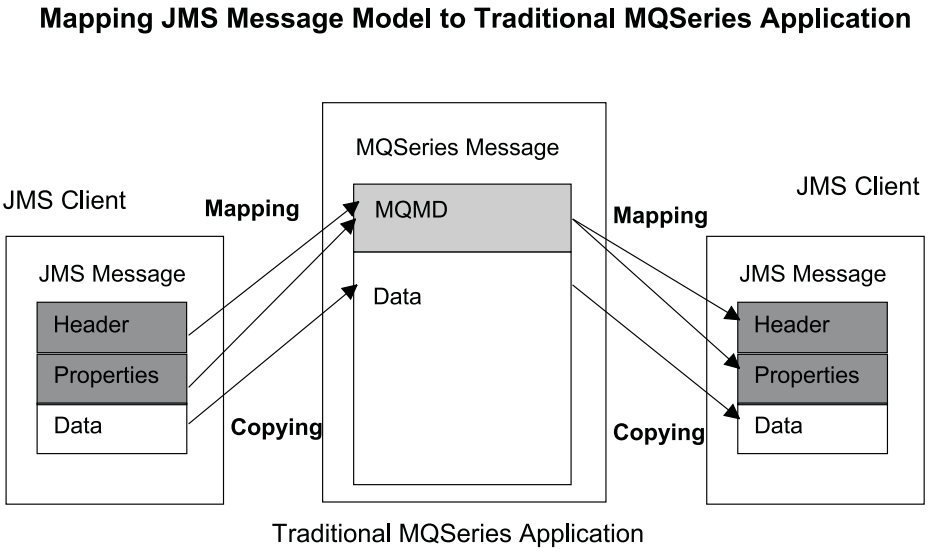


Figure 5. JMS to MQSeries mapping model

The administrator indicates that the JMS Client is communicating with such an application by setting the MQSeries Destination's TargetClient value to JMSC.MQJMS_CLIENT_NONJMS_MQ. This indicates no MQRFH2 field is to be produced.

The mapping from JMS to MQMD targeted at a Native MQSeries application is the same as for mapping JMS to MQMD targeted at a true JMS client. If an MQSeries message is received by JMS with the MQMD Format field set to other than MQFMT_RFH2 then we know we are receiving data from a nonJMS application. If the Format is MQFMT_STRING, the message is received as a JMS Text Message, otherwise it is received as a JMS Bytes message. As there is no MQRFH2, then only those JMS properties that are transmitted in the MQMD can be restored .

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 fashion

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 the MQ Java - such character sets are mainly used when sending a message to a non-JMS application.

If the character set is a double-byte set (including UTF16), then the ordering of the bytes is determined by the Destination object's integer encoding specification.

An incoming message is interpreted using the character set and encoding specified in the message itself. These specifications are carried in the rightmost MQSeries 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 1.02 specification, and associated JavaDoc.

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 be 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 carried in the rightmost MQSeries 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 resent without modification, then 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 explicitly sent in a BytesMessage is a UTF8 string. This is 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, and the character set value carried in an incoming MQSeries 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, so a JMS application that wishes to send a BytesMessage containing some textual data must itself 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="type">value</elementName1>
<elementName2 dt="type">value</elementName2>.....
</map>
```

where

`type` can take one of the values described in Table 20 on page 167. `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="type">value</elt>
<elt dt="type">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. JMS interfaces and classes

MQSeries classes for Java Message Service(JMS) consists of a number of java classes and interfaces based on the Sun javax.jms package of interfaces and classes. Clients should be written using the Sun interfaces and classes which are listed below and are described in detail in the following sections. The names of the MQSeries objects which implement the Sun interfaces and classes have a prefix of 'MQ' (unless stated otherwise in the object description). The descriptions detail any deviations of the MQSeries 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 not implemented in this release of MQSeries classes for Java Message Service(JMS).

Table 25 (Page 1 of 3). Interface Summary

Interface	Description
BytesMessage	A BytesMessage is used to send a message containing a stream of uninterpreted bytes.
Connection	A JMS Connection is a client's active connection to its JMS provider.
**ConnectionConsumer ** **	For application servers, Connections provide a special facility for creating a ConnectionConsumer.
ConnectionFactory	A ConnectionFactory encapsulates a set of connection configuration parameters that has been defined by an administrator.
ConnectionMetaData	ConnectionMetaData provides information describing the Connection.
DeliveryMode	Delivery modes supported by JMS.
Destination	The parent interface for Queue and Topic.
ExceptionListener	An exception listener is used to receive exceptions thrown by a Connections asynchronous delivery threads.
MapMessage	A MapMessage is used to send a set of name-value pairs where names are Strings and values are Java primitive types.
Message	The Message interface is the root interface of all JMS messages.
MessageConsumer	The parent interface for all message consumers.
MessageListener	A MessageListener is used to receive asynchronously delivered messages.
MessageProducer	A client uses a message producer to send messages to a Destination.
ObjectMessage	An ObjectMessage is used to send a message that contains a serializable Java object.

Table 25 (Page 2 of 3). Interface Summary

Interface	Description
Queue	A Queue object encapsulates a provider-specific queue name.
QueueBrowser	A client uses a QueueBrowser to look at messages on a queue without removing them.
QueueConnection	A QueueConnection is an active connection to a JMS point to point provider.
QueueConnectionFactory	A client uses a QueueConnectionFactory to create QueueConnections with a JMS PTP provider.
QueueReceiver	A client uses a QueueReceiver for receiving messages that have been delivered to a queue.
QueueSender	A client uses a QueueSender to send messages to a queue.
QueueSession	A QueueSession provides methods for creating QueueReceivers, QueueSenders, QueueBrowsers and TemporaryQueues.
ServerSession **	A ServerSession is an object implemented by an application server.
ServerSessionPool **	A ServerSessionPool is an object implemented by an application server to provide a pool of ServerSessions for processing the messages of a ConnectionConsumer.
Session	A JMS Session is a single threaded context for producing and consuming messages.
StreamMessage	A StreamMessage is used to send a stream of Java primitives.
TemporaryQueue	A TemporaryQueue is a unique Queue object created for the duration of a QueueConnection.
TemporaryTopic	A TemporaryTopic is a unique Topic object created for the duration of a TopicConnection.
TextMessage	A TextMessage is used to send a message containing a `java.lang.String`.
Topic	A Topic object encapsulates a provider-specific topic name.
TopicConnection	A TopicConnection is an active connection to a JMS Pub/Sub provider.
TopicConnectionFactory	A client uses a TopicConnectionFactory to create TopicConnections with a JMS Pub/Sub provider.
TopicPublisher	A client uses a TopicPublisher for publishing messages on a topic.
TopicSession	A TopicSession provides methods for creating TopicPublishers, TopicSubscribers and TemporaryTopics.
TopicSubscriber	A client uses a TopicSubscriber for receiving messages that have been published to a topic.
XAConnection **	XAConnection extends the capability of Connection by providing an XASession.

Table 25 (Page 3 of 3). Interface Summary	
Interface	**Description**
XAConnectionFactory **	Some application servers provide support for grouping JTS capable resource use into a distributed transaction.
XAQueueConnection **	XAQueueConnection provides the same create options as QueueConnection.
XAQueueConnectionFactory **	An XAQueueConnectionFactory provides the same create options as a QueueConnectionFactory.
XAQueueSession **	An XAQueueSession provides a regular QueueSession which can be used to create QueueReceivers, QueueSenders and QueueBrowsers.
XASession **	XASession extends the capability of Session by adding access to a JMS provider's support for JTA.
XATopicConnection **	An XATopicConnection provides the same create options as TopicConnection.
XATopicConnectionFactory **	An XATopicConnectionFactory provides the same create options as TopicConnectionFactory.
XATopicSession **	An XATopicSession provides a regular TopicSession which can be used to create TopicSubscribers and TopicPublishers.

Table 26. Class Summary	
Class	**Description**
QueueRequestor	JMS provides a QueueRequestor helper class to simplify making service requests.
TopicRequestor	JMS provides a TopicRequestor helper class to simplify making service requests.

MQSeries JMS classes

The following tables list the **com.ibm.mq.jms** and **com.ibm.jms** packages which contain the MQSeries classes that implement the sun interfaces.

Table 27. Package 'com.ibm.mq.jms' class Summary

Class	Implements
MQConnection	Connection
MQConnectionConsumer	ConnectionConsumer
MQConnectionFactory	ConnectionFactory
MQConnectionMetaData	ConnectionMetaData
MQDestination	Destination
MQMessageConsumer	MessageConsumer
MQMessageProducer	MessageProducer
MQQueue	Queue
MQQueueBrowser	QueueBrowser
MQQueueConnection	QueueConnection
MQQueueConnectionFactory	QueueConnectionFactory
MQQueueEnumeration	java.util.Enumeration from QueueBrowser
MQQueueReceiver	QueueReceiver
MQQueueSender	QueueSender
MQQueueSession	QueueSession
MQServerSession	ServerSession
MQServerSessionPool	ServerSessionPool
MQSession	Session
MQTemporaryQueue	TemporaryQueue
MQTemporaryTopic	TemporaryTopic
MQTopic	Topic
MQTopicConnection	TopicConnection
MQTopicConnectionFactory	TopicConnectionFactory
MQTopicPublisher	TopicPublisher
MQTopicSession	TopicSession
MQTopicSubscriber	TopicSubscriber

Table 28. Package 'com.ibm.jms' class summary	
Class	**Implements**
JMSBytesMessage	BytesMessage
JMSMapMessage	MapMessage
JMSMessage	Message
JMSObjectMessage	ObjectMessage
JMSStreamMessage	StreamMessage
JMSTextMessage	TextMessage

The following JMS interfaces are not implemented in this release of MQSeries classes for Java Message Service(JMS).

- ConnectionConsumer
- ServerSession
- ServerSessionPool
- XAConnection
- XAConnectionFactory
- XAQueueConnection
- XAQueueConnectionFactory
- XAQueueSession
- XASession
- XATopicConnection
- XATopicConnectionFactory
- XATopicSession

BytesMessage

public interface **BytesMessage**
extends **Message**

MQSeries class: **JMSBytesMessage**

```
java.lang.Object
   |
   +----com.ibm.jms.JMSMessage
            |
            +----com.ibm.jms.JMSBytesMessage
```

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

`public boolean readBoolean() throws JMSException`

Read a `boolean` from the bytes message.

Returns: the `boolean` value read.

Throws:

- MessageNotReadableException - if message in write-only mode.
- JMSException - if JMS fails to read message due to some internal JMS error.
- MessageEOFException - if end of message bytes

readByte

`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 message in write-only mode.
- MessageEOFException - if end of message bytes
- JMSException - if JMS fails to read message due to some internal JMS error.

readUnsignedByte

```
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 message in write-only mode.
- MessageEOFException - if end of message bytes
- JMSException - if JMS fails to read message due to some internal JMS error.

readShort

```
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 message in write-only mode.
- MessageEOFException - if end of message bytes
- JMSException - if JMS fails to read message due to some internal JMS error.

readUnsignedShort

```
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 message in write-only mode.
- MessageEOFException - if end of message bytes
- JMSException - if JMS fails to read message due to some 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 message in write-only mode.
- MessageEOFException - if end of message bytes

- JMSException - if JMS fails to read message due to some 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 message in write-only mode.
- MessageEOFException - if end of message bytes
- JMSException - if JMS fails to read message due to some 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 message in write-only mode.
- MessageEOFException - if end of message bytes
- JMSException - if JMS fails to read message due to some 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 message in write-only mode.
- MessageEOFException - if end of message bytes
- JMSException - if JMS fails to read message due to some internal JMS error.

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 message in write-only mode.
- MessageEOFException - if end of message bytes
- JMSException - if JMS fails to read message due to some internal JMS error.

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 message in write-only mode.
- MessageEOFException - if end of message bytes
- JMSException - if JMS fails to read message due to some internal JMS error.

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 message in write-only mode.
- JMSException - if JMS fails to read message due to some 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 message 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 message due to some internal JMS error.

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 message due to some 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 message due to some 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 message due to some 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 message due to some internal JMS error.

writeInt

```
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 message due to some internal JMS error.

writeLong

```
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 message due to some internal JMS error.

writeFloat

```
public void writeFloat(float value) throws JMSException
```

Convert the float argument to an `int` using `floatToIntBits` method in class `Float`, and then writes 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 message due to some 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 writes 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 message due to some 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 message due to some 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 message due to some 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 message due to some internal JMS error.

writeObject

```
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 (Integer, Double, Long etc.), Strings and byte arrays.

Parameters: `value` - the Java object to be written.

Throws:

- MessageNotWriteableException - if message in read-only mode.
- MessageFormatException - if object is invalid type.
- JMSException - if JMS fails to write message due to some 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 due to some internal JMS error.
- MessageFormatException - if message has an invalid format

Connection

public interface **Connection**
Subinterfaces: **QueueConnection** and **TopicConnection**

MQSeries class: **MQConnection**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQConnection
```

A JMS Connection is a client's active connection to its JMS provider.

See also: **QueueConnection** and **TopicConnection**

Methods

getClientID

```
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 JMS implementation fails to return the client ID for this Connection due to some internal error.

setClientID

```
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.

Parameters: clientID - the unique client identifier

Throws:

- JMSException - if JMS implementation fails to set the client ID for this Connection due to some internal error.
- InvalidClientIDException - if JMS client specifies an invalid 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.

getMetaData

```
public ConnectionMetaData getMetaData() throws JMSException
```

Get the metadata for this connection.

Returns: the connection metadata.

Throws: JMSException - general exception if JMS implementation fails to get the Connection metadata for this Connection.

See also: **ConnectionMetaData**

getExceptionListener

```
public ExceptionListener getExceptionListener()
                                                    throws JMSException
```

Get the ExceptionListener for this Connection.

Returns: The ExceptionListener for this Connection

Throws: JMSException - general exception if JMS implementation fails to get the Exception listener for this Connection.

setExceptionListener

```
public void setExceptionListener(ExceptionListener listener)
                                                    throws JMSException
```

Set an exception listener for this connection.

Parameters: handler - the exception listener.

Throws: JMSException - general exception if 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 JMS implementation fails to start the message delivery due to some internal error.

See also: stop

stop

```
public void stop() throws JMSException
```

Used to temporarily stop a Connection's delivery of incoming messages. It can be restarted using its `start` method. When stopped, delivery to all the Connection's message consumers is inhibited. synchronous receives block 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 JMS implementation fails to stop the message delivery due to some internal error.

See also: start

close

```
public void close() throws JMSException
```

Since a provider typically allocates significant resources outside the JVM on behalf of a Connection, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely 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 XASesssion, 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.

Throws: JMSException - if JMS implementation fails to close the connection due to internal error. For example, a failure to release resources or to close socket connection can lead to throwing of this exception.

ConnectionFactory

public interface **ConnectionFactory**
Subinterfaces: **QueueConnectionFactory** and **TopicConnectionFactory**

MQSeries 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.

See also: **QueueConnectionFactory** and **TopicConnectionFactory**

MQSeries constructor

MQConnectionFactory

```
public MQConnectionFactory()
```

Methods

setDescription *

```
public void setDescription(String x)
```

A short description of the object

getDescription *

```
public String getDescription()
Retrieve the object description
```

setTransportType *

```
public void setTransportType(int x) throws JMSException
```

Set the transport type to use. can be either
JMSC.MQJMS_TP_BINDINGS_MQ, or
JMSC.MQJMS_TP_CLIENT_MQ_TCPIP

getTransportType *

```
Retrieve the transport type
public int getTransportType()
```

setClientId *

```
public void setClientId(String x)
```

Sets the client Identifier to be used for all connections created using this Connection.

getClientId *

```
public String getClientId()
```

Gets the client Identifier that is used for all connections created using this ConnectionFactory.

setQueueManager *

```
public void setQueueManager(String x) throws JMSException
```

Set the name of the queue manager to connect to

getQueueManager *

```
public String getQueueManager()
```

Get the name of the queue manager

setHostName *

```
public void setHostName(String hostname)
```

For client only, the name of the host to connect to

getHostName *

```
public String getHostName()
```

Retrieve the name of the host

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 port is negative

getPort *

```
public int getPort()
```

For client connections only, get the port number

setChannel *

```
public void setChannel(String x) throws JMSException
```

For client only, set the channel to use

getChannel *

```
public String getChannel()
```

For client only, get the channel that was used

setCCSID *

```
public void setCCSID(int x) throws JMSException
```

Set the character set of the queue manager

getCCSID *

```
public int getCCSID()
```

Get the character set of the queue manager

setReceiveExit *

```
public void setReceiveExit(String receiveExit)
```

The name of a class that implements a receive exit

getReceiveExit *

```
public String getReceiveExit()
```

Get the name of the receive exit class

setReceiveExitInit *

```
public void setReceiveExitInit(String x)
```

Initialization string that is passed to the constructor of the receive exit class

getReceiveExitInit *

```
public String getReceiveExitInit()
```

Get the initialization string that was passed to the receive exit class

setSecurityExit *

```
public void setSecurityExit(String securityExit)
```

The name of a class that implements a security exit

getSecurityExit *

```
public String getSecurityExit()
```

Get the name of the security exit class

setSecurityExitInit *

```
public void setSecurityExitInit(String x)
```

Initialization string that is passed to the security exit constructor

getSecurityExitInit *

```
public String getSecurityExitInit()
```

Get the security exit inittialization string

setSendExit *

```
public void setSendExit(String sendExit)
```

The name of a class that implements a send exit

getSendExit *

```
public String getSendExit()
```

Get the name of the send exit class

setSendExitInit *

```
public void setSendExitInit(String x)
```

Initialization string that is passed to the constructor of send exit

getSendExitInit *

```
public String getSendExitInit()
```

get the send exit initialization string

ConnectionMetaData

public interface **ConnectionMetaData**

MQSeries class: **MQConnectionMetaData**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQConnectionMetaData
```

ConnectionMetaData provides information describing the Connection.

MQSeries constructor

MQConnectionMetaData

```
public MQConnectionMetaData()
```

Methods

getJMSVersion

```
public java.lang.String getJMSVersion() throws JMSException
```

Get the JMS version.

Returns: the JMS version.

Throws: JMSException - if some internal error occurs in JMS implementation during the metadata retrieval.

getJMSMajorVersion

```
public int getJMSMajorVersion() throws JMSException
```

Get the JMS major version number.

Returns: the JMS major version number.

Throws: JMSException - if some 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 some 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 some internal error occurs in JMS implementation ring during the property names retrieval

getJMSProviderName

```
public java.lang.String getJMSProviderName()
                                                    throws JMSException
```

Get the JMS provider name.

Returns: the JMS provider name.

Throws: JMSException - if some internal error occurs in JMS implementation during the meta-data retrieval.

getProviderVersion

```
public java.lang.String getProviderVersion()
                                                throws JMSException
```

Get the JMS provider version.

Returns: the JMS provider version.

Throws: JMSException - if some internal error occurs in JMS implementation during the metadata retrieval.

getProviderMajorVersion

```
public int getProviderMajorVersion() throws JMSException
```

Get the JMS provider major version number.

Returns: the JMS provider major version number.

Throws: JMSException - if some 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 some internal error occurs in JMS implementation during the metadata retrieval.

toString *

```
public String toString()
```

Overrides: toString in class Object

getJMSXPropertyNames

```
public java.util.Enumeration getJMSXPropertyNames()
                                                  throws JMSException
```

Get an enumeration of JMSX Property Names.

Returns:

an Enumeration of JMSX PropertyNames.

Throws:

JMSException - if some internal error occurs in JMS implementation during the property names retrieval.

DeliveryMode

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.

Destination

public interface **Destination**
Subinterfaces: **Queue** and **Topic**

MQSeries class: **MQDestination**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQDestination
```

The Destination object encapsulates provider-specific addresses.

See also: **Queue** and **Topic**

MQSeries constructors

MQDestination

```
public MQDestination()
```

Methods

setDescription *

```
public void setDescription(String x)
```

A short description of the object

getDescription *

```
public String getDescription()
```

Get the description of the object

setPriority *

```
public void setPriority(int priority) throws JMSException
```

Used to override the priority of all messages sent to this destination

getPriority *

```
public int getPriority()
```

Get the override priority value

setExpiry *

```
public void setExpiry(int expiry) throws JMSException
```

Used to override the expiry of all messages sent to this destination

getExpiry *

```
public int getExpiry()
```

Get the value of the expiry for this destination

setPersistence *

```
public void setPersistence(int persistence)
                                    throws JMSException
```

Used to override the persistence of all messages sent to this destination

getPersistence *

```
public int getPersistence()
```

Get the value of the persistence for this destination

setTargetClient *

```
public void setTargetClient(int targetClient)
                                    throws JMSException
```

Flag to indicate whether or not the remote application is JMS compliant

getTargetClient *

```
public int getTargetClient()
```

Get JMS compliance indicator flag

setCCSID *

```
public void setCCSID(int x) throws JMSException
```

Character set to be used to encode text strings in messages sent to this destination. See Table 16 on page 93 for a list of allowed values.

getCCSID *

```
public int getCCSID()
```

Get the name of the character set that is used by this destination

setEncoding *

```
public void setEncoding(int x) throws JMSException
```

Specifies the encoding to be used for numeric fields in messages sent to this destination. See 92 for a list of allowed values

getEncoding *

```
public int getEncoding()
```

Get the encoding that is used for this destination.

ExceptionListener

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 describing 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 and typically 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**

MQSeries 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:

JMSException - if JMS fails to read message due to some internal JMS error.

MessageFormatException - if this type conversion is invalid.

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:

JMSException - if JMS fails to read message due to some internal JMS error.

MessageFormatException - if this type conversion is invalid.

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 message due to some internal JMS error.

MessageFormatException - if this type conversion is invalid.

getChar

```
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 message due to some internal JMS error.

MessageFormatException - if this type conversion is invalid.

getInt

```
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 message due to some internal JMS error.

MessageFormatException - if this type conversion is invalid.

getLong

```
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 message due to some internal JMS error.

MessageFormatException - if this type conversion is invalid.

getFloat

```
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 message due to some internal JMS error.

MessageFormatException - if this type conversion is invalid.

getDouble

```
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 message due to some internal JMS error.

MessageFormatException - if this type conversion is invalid.

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:

JMSException - if JMS fails to read message due to some internal JMS error.

MessageFormatException - if this type conversion is invalid.

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 message due to some internal JMS error.

MessageFormatException - if this type conversion is invalid.

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 message due to some internal JMS error.

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 message due to some internal JMS error

setBoolean

```
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 message in read-only mode.

setByte

```
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 message in read-only mode.

setShort

```
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 message 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 message in read-only mode.

setInt

```
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 message in read-only mode.

setLong

```
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 message in read-only mode

setFloat

```
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 message in read-only mode.

setDouble

```
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 message in read-only mode

setString

```
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 message in read-only mode.

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 message 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 message in read-only mode.

setObject

```
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 invalid

MessageNotWriteableException - if message in read-only mode.

itemExists

```
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.

Message

public interface **Message**
Subinterfaces: **BytesMessage**, **MapMessage**, **ObjectMessage**, **StreamMessage**,and **TextMessage**

MQSeries 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

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 due to internal JMS error.

See also: setJMSMessageID()

setJMSMessageID

```
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 due to internal JMS error.

See also: getJMSMessageID()

getJMSTimestamp

```
public long getJMSTimestamp() throws JMSException
```

Get 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.

Returns: the message timestamp

Throws: JMSException - if JMS fails to get the Timestamp due to internal JMS error.

See also: setJMSTimestamp()

setJMSTimestamp

```
public void setJMSTimestamp(long timestamp)
                                    throws JMSException
```

Set the message timestamp.

Parameters: timestamp - the timestamp for this message

Throws: JMSException - if JMS fails to set the timestamp due to some internal JMS error.

See also: getJMSTimestamp()

getJMSCorrelationIDAsBytes

```
public byte[] getJMSCorrelationIDAsBytes()
                                    throws JMSException
```

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 correlationId due to some internal JMS error.

See also: setJMSCorrelationID(), getJMSCorrelationID(), setJMSCorrelationIDAsBytes()

setJMSCorrelationIDAsBytes

```
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 correlationId due to some internal JMS error.

See also: setJMSCorrelationID(), getJMSCorrelationID(), getJMSCorrelationIDAsBytes()

getJMSCorrelationID

```
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 correlationId due to some internal JMS error.

See also: setJMSCorrelationID(), getJMSCorrelationIDAsBytes(), setJMSCorrelationIDAsBytes()

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 due to some internal JMS error.

See also: setJMSReplyTo()

setJMSCorrelationID

```
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 typically 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: JMSException - if JMS fails to set correlationId due to some internal JMS error.

See also: getJMSCorrelationID(), getJMSCorrelationIDAsBytes(), setJMSCorrelationIDAsBytes()

setJMSReplyTo

```
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 due to some internal JMS error.

See also: getJMSReplyTo()

getJMSDestination

```
public Destination getJMSDestination() throws JMSException
```

Get 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.

Returns: the destination of this message.

Throws: JMSException - if JMS fails to get JMS Destination due to some internal JMS error.

See also: setJMSDestination()

setJMSDestination

```
public void setJMSDestination(Destination destination)
                                        throws JMSexception
```

Set the destination for this message.

Parameters: destination - the destination for this message.

Throws: JMSException - if JMS fails to set JMS Destination due to some internal JMS error.

See also: getJMSDestination()

getJMSDeliveryMode

```
public int getJMSDeliveryMode() throws JMSException
```

Get 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.

Returns: the delivery mode of this message.

Throws: JMSException - if JMS fails to get JMS DeliveryMode due to some internal JMS error.

See also: setJMSDeliveryMode(), DeliveryMode

setJMSDeliveryMode

```
public void setJMSDeliveryMode(int deliveryMode)
                                      throws JMSException
```

Set the delivery mode for this message.

Parameters: deliveryMode - the delivery mode for this message.

Throws: JMSException - if JMS fails to set JMS DeliveryMode due to some internal JMS error.

See also: getJMSDeliveryMode(), DeliveryMode

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.

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.

Returns: set to true if this message is being redelivered.

Throws: JMSException - if JMS fails to get JMS Redelivered flag due to some internal JMS error.

See also: setJMSRedelivered()

setJMSRedelivered

```
public void setJMSRedelivered(boolean redelivered)
                                        throws JMSException
```

Set to indicate whether this message is being redelivered.

Parameters: redelivered - an indication of whether this message is being redelivered.

Throws: JMSException - if JMS fails to set JMSRedelivered flag due to some internal JMS error.

See also: getJMSRedelivered()

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 due to some internal JMS error.

See also: setJMSType()

setJMSType

```
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 insures 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 due to some internal JMS error.

See also: getJMSType()

getJMSExpiration

```
public long getJMSExpiration() throws JMSException
```

Get 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.

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 due to some internal JMS error.

See also: setJMSExpiration()

setJMSExpiration

```
public void setJMSExpiration(long expiration)
                                          throws JMSException
```

Set the message's expiration value.

Parameters: expiration - the message's expiration time

Throws: JMSException - if JMS fails to set JMS message expiration due to some internal JMS error.

See also: getJMSExpiration()

getJMSPriority

```
public int getJMSPriority() throws JMSException
```

Get the message priority.

Returns: the message priority

Throws: JMSException - if JMS fails to get JMS message priority due to some internal JMS error.

See also: setJMSPriority() for priority levels

setJMSPriority

```
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 due to some internal JMS error.

See also: getJMSPriority()

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 due to some internal JMS error.

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 if property exists due to some 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 Property due to some internal JMS error.
- MessageFormatException - if this type conversion is invalid

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 Property due to some internal JMS error.
- MessageFormatException - if this type conversion is invalid.

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 Property due to some internal JMS error.
- MessageFormatException - if this type conversion is invalid.

getIntProperty

```
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 Property due to some internal JMS error.
- MessageFormatException - if this type conversion is invalid.

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 Property due to some internal JMS error.
- MessageFormatException - if this type conversion is invalid.

getFloatProperty

```
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 Property due to some internal JMS error.
- MessageFormatException - if this type conversion is invalid.

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 Property due to some internal JMS error.
- MessageFormatException - if this type conversion is invalid.

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 Property due to some internal JMS error.
- MessageFormatException - if this type conversion is invalid.

getObjectProperty

```
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, then a Integer is returned). If there is no property by this name, a null value is returned.

Throws: JMSException - if JMS fails to get Property due to some 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 Property names due to some 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 due to some internal JMS error.
- MessageNotWriteableException - if 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 due to some internal JMS error.
- MessageNotWriteableException - if properties are read-only

setShortProperty

```
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 due to some internal JMS error.
- MessageNotWriteableException - if properties are read-only

setIntProperty

```
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:

- JMSException - if JMS fails to set Property due to some internal JMS error.
- MessageNotWriteableException - if properties are read-only

setLongProperty

```
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 due to some internal JMS error.
- MessageNotWriteableException - if properties are read-only

setFloatProperty

```
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:

- JMSException - if JMS fails to set Property due to some internal JMS error.
- MessageNotWriteableException - if 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.

Throws:

- JMSException - if JMS fails to set Property due to some internal JMS error.
- MessageNotWriteableException - if properties are read-only

setStringProperty

```
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 Property due to some internal JMS error.
- MessageNotWriteableException - if properties are read-only

setObjectProperty

```
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 due to some internal JMS error.
- MessageFormatException - if object is invalid
- MessageNotWriteableException - if properties are read-only

acknowledge

```
public void acknowledge() throws JMSException
```

Acknowledge this and all previous messages received by the session.

Throws: JMSException - if JMS fails to acknowledge due to some 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 due to some internal JMS error.

MessageConsumer

public interface **MessageConsumer**
Subinterfaces: **QueueReceiver** and **TopicSubscriber**

MQSeries 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

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 message selector due to some JMS error

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 message listener due to some JMS error

See also: setMessageListener

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 due to some JMS error

See also: getMessageListener

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 due to some error.

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 due to some 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 due to some error.

close

```
public void close() throws JMSException
```

Since a provider may allocate some resources on behalf of a MessageConsumer outside the JVM, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough.

This call blocks until a receive or message listener in progress has completed.

Throws: JMSException - if JMS fails to close the consumer due to some error.

MessageListener

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**

MQSeries class: **MQMessageProducer**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQMessageProducer
```

A client uses a message producer to send messages to a Destination.

MQSeries constructors

MQMessageProducer

```
public MQMessageProducer()
```

Methods

setDisableMessageID

```
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 MQSeries classes for Java Message Service(JMS) implementation.

Parameters: value - indicates if message IDs are disabled.

Throws: JMSException - if JMS fails to set disabled message Id due to some internal error.

getDisableMessageID

```
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 disabled message Id due to some internal error.

setDisableMessageTimestamp

```
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 MQSeries classes for Java Message Service(JMS) implementation.

Parameters: value - indicates if message timestamps are disabled.

Throws: JMSException - if JMS fails to set disabled message timestamp due to some 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 disabled message timestamp due to some internal error.

setDeliveryMode

```
public void setDeliveryMode(int deliveryMode)
                                        throws JMSException
```

Set the producer's default delivery mode.

Delivery mode is set to PERSISTENT by default.

Parameters: deliveryMode - the message delivery mode for this message producer.

Throws: JMSException - if JMS fails to set delivery mode due to some internal error.

See also: getDeliveryMode

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 delivery mode due to some internal error.

See also: setDeliveryMode

setPriority

```
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 priority due to some internal error.

See also: getPriority

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 priority due to some internal error.

See also: setPriority

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.

Parameters: timeToLive - the message time to live in milliseconds; zero is unlimited

Throws: JMSException - if JMS fails to set Time to Live due to some internal error.

See also: getTimeToLive

getTimeToLive

```
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 Time to Live due to some internal error.

See also: setTimeToLive

close

```
public void close() throws JMSException
```

Since a provider may allocate some resources on behalf of a MessageProducer outside the JVM, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough.

Throws: JMSException - if JMS fails to close the producer due to some error.

MQQueueEnumeration *

public class **MQQueueEnumeration**
extends **Object**
implements **Enumeration**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQQueueEnumeration
```

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 MQ queue 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 of resetting an instance of this class - it acts as a 'one-shot' mechanism.

See also: **MQQueueBrowser**

Methods

hasMoreElements

`public boolean hasMoreElements()`

Indicate whether we can return another message

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.

ObjectMessage

public interface **ObjectMessage**
extends **Message**

MQSeries 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

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 object due to some internal JMS error
- MessageFormatException - if object serialization fails
- MessageNotWriteableException - if message in read-only mode

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 object due to some internal JMS error
- MessageFormatException - if object deserialization fails

Queue

public interface **Queue**
extends **Destination** Subinterfaces: **TemporaryQueue**

MQSeries 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.

MQSeries 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 148.

MQQueue *

```
public MQQueue(String queueManagerName,
               String queueName)
```

Methods

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: JMSException - if JMS implementation for Queue fails to return queue name due to some internal error.

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

getReference *

```
public Reference getReference() throws NamingException
```

Create a reference for this queue

Returns: a reference for this object

Throws: NamingException

setBaseQueueName *

```
public void setBaseQueueName(String x) throws JMSException
```

Set the value of the MQSeries queue name.

Note: This method should only be used by the administration tool. It makes no attempt to decode queue:qmgr:queue format strings.

getBaseQueueName *

```
public String getBaseQueueName()
```

Returns: the value of the MQSeries Queue name.

setBaseQueueManagerName *

```
public void setBaseQueueManagerName(String x) throws JMSException
```

Set the value of the MQSeries queue manager name.

Note: This method should only be used by the administration tool.

getBaseQueueManagerName *

```
public String getBaseQueueManagerName()
```

Returns: the value of the MQSeries Queue manager name

QueueBrowser

public interface **QueueBrowser**

MQSeries class: **MQQueueBrowser**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQQueueBrowser
```

A client uses a QueueBrowser to look at messages on a queue without removing them.

Note: The MQSeries class **MQQueueEnumeration** is used to hold the browse cursor.

See also: **QueueReceiver**

Methods

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 due to some JMS error.

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 due to some JMS error.

getEnumeration

```
public java.util.Enumeration getEnumeration()
                                              throws JMSException
```

Get an enumeration for browsing the current queue messages in the order they would be received.

Returns: an enumeration for browsing the messages

Throws: JMSException - if JMS fails to get the enumeration for this browser due to some JMS error.

Note: If the browser is created for a non-existent queue,this is not detected until the first call to getEnumeration.

close

```
public void close() throws JMSException
```

Since a provider may allocate some resources on behalf of a QueueBrowser outside the JVM, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough.

Throws: JMSException - if a JMS fails to close this Browser due to some JMS error.

QueueConnection

public interface **QueueConnection**
extends **Connection**
Subinterfaces: **XAQueueConnection**

MQSeries 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.

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.

Throws: JMSException - if JMS Connection fails to create a session due to some internal error or lack of support for specific transaction and acknowledgement mode.

createConnectionConsumer

```
public ConnectionConsumer createConnectionConsumer
                                          (Queue queue,
                                          java.lang.String messageSelector,
                                          ServerSessionPool sessionPool,
                                          int maxMessages)
                                          throws JMSException
```

Create a connection consumer for this connection.

Note: This method is not implemented in MQSeries classes for Java Message Service(JMS).

close *

```
public void close() throws JMSException
```

Overrides: close in class MQConnection

QueueConnectionFactory

public interface **QueueConnectionFactory**
extends **ConnectionFactory**
Subinterfaces: **XAQueueConnectionFactory**

MQSeries 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 PTP provider.

See also: **ConnectionFactory**

MQSeries 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 due to some internal error.
- JMSSecurityException - if client authentication fails due to invalid 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_TCPIP (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 due to some internal error.
- JMSSecurityException - if client authentication fails due to invalid user name or password.

setTemporaryModel *

```
public void setTemporaryModel(String x) throws JMSException
```

getTemporaryModel *

```
public String getTemporaryModel()
```

getReference *

```
public Reference getReference() throws NamingException
```

Create a reference for this queue connection factory

Returns: a reference for this object

Throws: NamingException

QueueReceiver

public interface **QueueReceiver**
extends **MessageConsumer**

MQSeries 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 classes 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 due to some internal error.

QueueRequestor

```
java.lang.Object
    |
    +----javax.jms.QueueRequestor
```

public class **QueueRequestor**
extends **java.lang.Object**

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 the session parameter to be 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

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.

Parameters: message - the message to send.

Returns: the reply message.

Throws: JMSException - if a JMS error occurs.

close

```
public void close() throws JMSException
```

Since a provider may allocate some resources on behalf of a QueueRequestor outside the JVM, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough.

Note: This method closes the Session object passed to the QueueRequestor constructor.

Throws: JMSException - if a JMS error occurs.

QueueSender

public interface **QueueSender**
extends **MessageProducer**

MQSeries 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, not associated with any given Queue.

See also: **MessageProducer**

Methods

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 due to some internal error.

send

```
public void send(Message message) throws JMSException
```

Send a message to the queue. Use the QueueSender's default delivery mode, timeToLive and priority.

Parameters: message - the message to be sent

Throws:

- JMSException - if JMS fails to send the message due to some internal error.
- MessageFormatException - if invalid message specified
- InvalidDestinationException - if a client uses this method with a Queue sender with an invalid queue.

send

```
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 due to some internal error.
- MessageFormatException - if invalid message specified
- InvalidDestinationException - if a client uses this method with a Queue sender with an invalid queue.

send

```
public void send(Queue queue,
                 Message message) throws JMSException
```

Send a message to the specified queue with the QueueSender's default delivery mode, timeToLive 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 due to some internal error.
- MessageFormatException - if invalid message specified
- InvalidDestinationException - if a client uses this method with an invalid queue.

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 due to some internal error.
- MessageFormatException - if invalid message specified
- InvalidDestinationException - if a client uses this method with an invalid queue.

close *

```
public void close() throws JMSException
```

Since a provider may allocate some resources on behalf of a MessageProducer outside the JVM, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough.

Throws: JMSException if JMS fails to close the producer due to some error.

Overrides: close in class MQMessageProducer

QueueSession

public interface QueueSession
extends Session

MQSeries class: **MQQueueSession**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQSession
            |
            +----com.ibm.mq.jms.MQQueueSession
```

A QueueSession provides methods for creating QueueReceivers, QueueSenders, QueueBrowsers and TemporaryQueues.

See also: **Session**

The following methods are inherited from **MQsession**:

- close
- commit
- rollback
- recover

Methods

createQueue

```
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 148.

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 due to some JMS error.

createReceiver

```
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 due to some JMS error.
- InvalidDestinationException - if invalid Queue specified.

createReceiver

```
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 matching the message selector expression are delivered

Throws:

- JMSException - if a session fails to create a receiver due to some JMS error.
- InvalidDestinationException - if invalid Queue specified.
- InvalidSelectorException - if the message selector is invalid.

createSender

```
public QueueSender createSender(Queue queue)
                                        throws JMSException
```

Create a QueueSender to send messages to the specified queue.

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 due to some JMS error.
- InvalidDestinationException - if invalid Queue specified.

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 due to some JMS error.
- InvalidDestinationException - if invalid Queue 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 matching the message selector expression are delivered

Throws:

- JMSException - if a session fails to create a browser due to some JMS error.
- InvalidDestinationException - if invalid Queue specified.
- InvalidSelectorException - if the message selector is invalid.

createTemporaryQueue

```
public TemporaryQueue createTemporaryQueue()
                                        throws JMSException
```

Create a temporary queue. It's 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 due to some JMS error.

Session

public interface **Session**
extends java.lang.**Runnable**
Subinterfaces: **QueueSession**, **TopicSession** and **XASession**

MQSeries 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** and **TopicSession**,

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

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 due to some 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 due to some 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 due to some 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 due to some 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 due to some 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 due to some 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 due to some 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 due to some internal error.

getTransacted

```
public boolean getTransacted() throws JMSException
```

Is the session in transacted mode?

Returns: true if in transacted mode

Throws: JMSException - if JMS fails to return the transaction mode due to internal error in JMS Provider.

commit

```
public void commit() throws JMSException
```

Commit all messages done in this transaction and releases any locks currently held.

Throws:

- JMSException - if JMS implementation fails to commit the transaction due to some internal error.
- TransactionRolledBackException - if the transaction gets rolled back due to some internal error during commit.

rollback

```
public void rollback() throws JMSException
```

Rollback any messages done in this transaction and releases any locks currently held.

Throws: JMSException - if JMS implementation fails to rollback the transaction due to some internal error.

close

```
public void close() throws JMSException
```

Since a provider may allocate some resources on behalf of a Session outside the JVM, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely 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 JMS implementation fails to close a Session due to some internal error.

recover

```
public void recover() throws JMSException
```

Stop message delivery in this session, and restart sending messages with the oldest unacknowledged message.

Throws: JMSException - if JMS implementation fails to stop message delivery and restart message send due to due to some internal error.

getMessageListener

```
public MessageListener getMessageListener()
                                    throws JMSException
```

Return the session's distinguished message listener.

Note: This method is not implemented.

Returns: the message listener associated with this session.

Throws: JMSException - if JMS fails to get the message listener due to an internal error in JMS Provider.

See also: setMessageListener

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 not used by regular JMS clients.

Note: This method is not implemented

Parameters: listener - the message listener to associate with this session.

Throws: JMSException - if JMS fails to set the message listener due to an internal error in JMS Provider.

See also: getMessageListener

run

```
public void run()
```

Reserved for use with the advanced server facilities.

Note: This method is not implemented

StreamMessage

public interface **StreamMessage**
extends **Message**

MQSeries 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 message due to some internal JMS error.
- MessageEOFException - if an end of message stream is received
- MessageFormatException - if this type conversion is invalid
- MessageNotReadableException - if message 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 message due to some internal JMS error.
- MessageEOFException - if an end of message stream is received
- MessageFormatException - if this type conversion is invalid
- MessageNotReadableException - if message in write-only mode.

readShort

```
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 message due to some internal JMS error.
- MessageEOFException - if an end of message stream is received
- MessageFormatException - if this type conversion is invalid
- MessageNotReadableException - if message in write-only mode.

readChar

```
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 message due to some internal JMS error.
- MessageEOFException - if an end of message stream is received
- MessageFormatException if this type conversion is invalid
- MessageNotReadableException if message 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 message due to some internal JMS error.
- MessageEOFException - if an end of message stream is received
- MessageFormatException if this type conversion is invalid
- MessageNotReadableException if message 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 message due to some internal JMS error.
- MessageEOFException - if an end of message stream
- MessageFormatException if this type conversion is invalid
- MessageNotReadableException if message 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 message due to some internal JMS error.
- MessageEOFException - if an end of message stream
- MessageFormatException if this type conversion is invalid
- MessageNotReadableException - if message in write-only mode.

readDouble

```
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 message due to some internal JMS error.
- MessageEOFException - if an end of message stream is received
- MessageFormatException - if this type conversion is invalid
- MessageNotReadableException - if message in write-only mode.

readString

```
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 message due to some internal JMS error.

- MessageEOFException - if an end of message stream is received
- MessageFormatException - if this type conversion is invalid
- MessageNotReadableException - if message in write-only mode

readBytes

```
public int readBytes(byte[] value)
             throws JMSExceptioneam message.
```

Read a byte array field from the stream message into the specified byte lbrk.rbrk. object (the read buffer). If the buffer size is less than or equal to the size of the data in the message field, then 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 message due to some internal JMS error.
- MessageEOFException - if an end of message stream is received
- MessageFormatException - if this type conversion is invalid
- MessageNotReadableException - if message in write-only mode.

readObject

```
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, then a Integer is returned)

Throws:

- JMSException - if JMS fails to read message due to some internal JMS error.
- MessageEOFException - if an end of message stream is received
- NotReadableException if message in write-only mode.

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 message due to some internal JMS error.
- MessageNotWriteableException - if message 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 message due to some internal JMS error.
- MessageNotWriteableException - if message in read-only mode.

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 message due to some internal JMS error.
- MessageNotWriteableException - if message in read-only mode.

writeChar

```
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 message due to some internal JMS error.
- MessageNotWriteableException - if message 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 message due to some internal JMS error.
- MessageNotWriteableException - if message 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 message due to some internal JMS error.
- MessageNotWriteableException - if message in read-only mode.

writeFloat

```
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 message due to some internal JMS error.
- MessageNotWriteableException - if message in read-only mode.

writeDouble

```
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 message due to some internal JMS error.
- MessageNotWriteableException - if message 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 message due to some internal JMS error.
- MessageNotWriteableException - if message 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 message due to some internal JMS error.
- MessageNotWriteableException - if message in read-only mode.

writeBytes

```
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 message due to some internal JMS error.
- MessageNotWriteableException - if message 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 message due to some internal JMS error.
- MessageNotWriteableException - if message in read-only mode.
- MessageFormatException - if the object is invalid

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 due to some internal JMS error.
- MessageFormatException - if message has an invalid format

TemporaryQueue

public interface **TemporaryQueue**
extends **Queue**

MQSeries 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 created for the duration of a QueueConnection.

Methods

delete

`public void` **`delete`**`() throws JMSException`

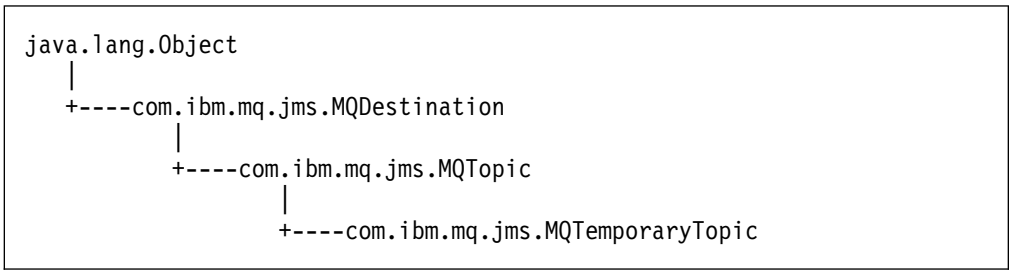
Delete this temporary queue. If there are still existing senders or receivers using it, then a JMSException will be thrown.

Throws: JMSException - if JMS implementation fails to delete a TemporaryQueue due to some internal error.

TemporaryTopic

public interface **TemporaryTopic**
extends **Topic**

MQSeries class: **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.

MQSeries 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, then a JMSException will be thrown.

Throws: JMSException - if JMS implementation fails to delete a TemporaryTopic due to some internal error.

TextMessage

public interface **TextMessage**
extends **Message**

MQSeries class: **JMSTextMessage**

```
java.lang.Object
   |
   +----com.ibm.jms.JMSMessage
           |
           +----com.ibm.jms.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

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 due to some internal JMS error.
- MessageNotWriteableException - if message in read-only mode.

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 text due to some internal JMS error.

Topic

public interface **Topic**
extends **Destination**
Subinterfaces: **TemporaryTopic**

MQSeries 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.

See also: **Destination**

MQSeries constructor

MQTopic

```
public MQTopic()
public MQTopic(string URItopic)
```

See TopicSession.createTopic.

Methods

getTopicName

`public java.lang.String` **`getTopicName`**`() throws JMSException`

Get the name of this topic in URI format. (URI format is described on 148.)

Note: Clients that depend upon the name, are not portable.

Returns: the topic name

Throws: JMSException - if JMS implementation for Topic fails to return topic name due to some internal error.

toString

```
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

getReference *

```
public Reference getReference()
```

Create a reference for this queue

Returns: a reference for this object

Throws: NamingException

setBaseTopicName *

```
public void setBaseTopicName(String x)
```

set method for the underlying MQSeries topic name.

getBaseTopicName

```
public String getBaseTopicName()
```

get method for the underlying MQSeries topic name.

TopicConnection

public interface **TopicConnection**
extends **Connection**

MQSeries class: **MQTopicConnection**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQConnection
           |
           +----com.ibm.mq.jms.MQTopicConnection
```

A TopicConnection is an active connection to a JMS Pub/Sub provider.

See also: **Connection**

Methods

createTopicSession

```
public TopicSession createTopicSession(boolean transacted,
                                       int acknowledgeMode)
                                       throws JMSException
```

Create a TopicSession

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 due to some internal error or lack of support for specific transaction and acknowledgement mode.

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 not used by regular JMS clients.

Note: This method is not implemented

Parameters:

- topic - the topic to access
- messageSelector - only messages with properties matching 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 JMS Connection fails to create a connection consumer due to some internal error or invalid arguments for sessionPool.
- InvalidSelectorException - if the message selector is invalid.

See also: ConnectionConsumer

createDurableConnectionConsumer

```
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 not used by regular JMS clients.

Note: This method is not implemented

Parameters:

- topic - the topic to access
- subscriptionName - durable subscription name
- messageSelector - only messages with properties matching the message selector expression are delivered
- sessionPool - the serversession 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 JMS Connection fails to create a connection consumer due to some internal error or invalid arguments for sessionPool and message selector.

See also: ConnectionConsumer

TopicConnectionFactory

public interface **TopicConnectionFactory**
extends **ConnectionFactory**

MQSeries 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.

See also: **ConnectionFactory**

MQSeries constructor

MQTopicConnectionFactory

```
public MQTopicConnectionFactory()
```

Methods

createTopicConnection

```
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 due to some internal error.

JMSSecurityException - if client authentication fails due to invalid 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.

Note: This method is valid only for transport type IBM_JMS_TP_CLIENT_MQ_TCPIP. See ConnectionFactory.

Parameters:

userName - the caller's user name

password - the caller's password

Returns: a newly created topic connection.

Throws:

JMSException - if JMS Provider fails to create a Topic Connection due to some internal error.

JMSSecurityException - if client authentication fails due to invalid user name or password.

setBrokerControlQueue *

```
public void setBrokerControlQueue(String x)
                           throws JMSException
```

Set method for brokerControlQueue attribute

Parameters: brokerControlQueue - the name of the broker control queue

getBrokerControlQueue *

```
public String getBrokerControlQueue()
```

Get method for brokerControlQueue attribute

Returns: the broker's control queue name

setBrokerQueueManager *

```
public void setBrokerQueueManager(String x)
                                  throws JMSException
```

Set method for brokerQueueManager attribute

Parameters: brokerQueueManager - the name of the broker's Queue Manager

getBrokerQueueManager *

```
public String getBrokerQueueManager()
```

Get method for brokerQueueManager attribute

Returns: the broker's queue manager name

setBrokerPubQueue *

```
public void setBrokerPubQueue(String x) throws JMSException
```

Set method for brokerPubQueue attribute

Parameters: brokerPubQueue - the name of the broker publish queue

getBrokerPubQueue *

```
public String getBrokerPubQueue()
```

Get method for brokerPubQueue attribute

Returns: the broker's publish queue name

getBrokerVersion *

```
public int getBrokerVersion()
```

setBrokerVersion *

```
public void setBrokerVersion(int x) throws JMSException
```

getReference *

```
public Reference getReference()
```

Return a reference for this topic connection factory

Returns: a reference for this topic connection factory

Throws: NamingException

TopicPublisher

public interface **TopicPublisher**
extends **MessageProducer**

MQSeries 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

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 topic for this topic publisher due to some internal error.

publish

```
public void publish(Message message) throws JMSException
```

Publish a Message to the topic Use the topic's default delivery mode, timeToLive and priority.

Parameters: message - the message to publish

Throws:

- JMSException - if JMS fails to publish the message due to some internal error.
- MessageFormatException - if invalid message specified
- InvalidDestinationException - if a client uses this method with a Topic Publisher with an invalid topic.

publish

```
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.

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 due to some internal error.
- MessageFormatException - if invalid message specified
- InvalidDestinationException - if a client uses this method with a Topic Publisher with an invalid topic.

publish

```
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, timeToLive 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 due to some internal error.
- MessageFormatException - if invalid message specified
- InvalidDestinationException - if a client uses this method with an invalid topic.

publish

```
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.

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 due to some internal error.
- MessageFormatException - if invalid message specified
- InvalidDestinationException - if a client uses this method with an invalid topic.

close *

```
public void close() throws JMSException
```

Since a provider may allocate some resources on behalf of a MessageProducer outside the JVM, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough.

Throws: JMSException if JMS fails to close the producer due to some error.

Overrides: close in class MQMessageProducer

TopicRequestor

```
java.lang.Object
    |
    +----javax.jms.TopicRequestor
```

public class **TopicRequestor**
extends java.lang.Object 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

TopicRequestor

```
public TopicRequestor(TopicSession session,
                      Topic topic) throws JMSException
```

Constructor for the TopicRequestor class. This implementation assumes the session parameter to be 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

request

```
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.

close

```
public void close() throws JMSException
```

Since a provider may allocate some resources on behalf of a TopicRequestor outside the JVM, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough.

Note: This method closes the Session object passed to the TopicRequestor constructor.

Throws: JMSException - if a JMS error occurs.

TopicSession

public interface **TopicSession**
extends **Session**

MQSeries class: **MQTopicSession**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQSession
           |
           +----com.ibm.mq.jms.MQTopicSession
```

A TopicSession provides methods for creating TopicPublishers, TopicSubscribers and TemporaryTopics.

See also: **Session**

MQSeries constructor

MQTopicSession

```
public MQTopicSession(boolean transacted,
                      int acknowledgeMode) throws JMSException
```

See TopicConnection.createTopicSession.

Methods

createTopic

```
public Topic createTopic(java.lang.String topicName)
                                          throws JMSException
```

Create a Topic given a URI format Topic name. (URI format is described on page 148.) 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 due to some JMS error.

createSubscriber

```
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 due to some JMS error.
- InvalidDestinationException - if invalid Topic specified.

createSubscriber

```
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 matching 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 due to some JMS error or invalid selector.
- InvalidDestinationException - if invalid Topic specified.
- InvalidSelectorException - if the message selector is invalid.

createDurableSubscriber

```
public TopicSubscriber createDurableSubscriber(Topic topic,
                                               java.lang.String name)
                                               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 and/or message selector.

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 due to some JMS error.
- InvalidDestinationException - if invalid Topic specified.

See TopicSession.unsubscribe.

createDurableSubscriber

```
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.

Parameters:

- topic - the topic to subscribe to
- name - the name used to identify this subscription.

- messageSelector - only messages with properties matching 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 due to some JMS error or invalid selector.
- InvalidDestinationException - if invalid Topic specified.
- InvalidSelectorException - if the message selector is invalid.

createPublisher

```
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 due to some JMS error.
- InvalidDestinationException - if invalid Topic specified.

createTemporaryTopic

```
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 due to some JMS error.

unsubscribe

```
public void unsubscribe(java.lang.String name)
                                     throws JMSException
```

Unsubscribe a durable subscription that has been created by a client.

Parameters: name - the name used to identify this subscription.

Throws:

- JMSException - if JMS fails to unsubscribe to durable subscription due to some JMS error.
- InvalidDestinationException - if invalid Topic specified

TopicSubscriber

public interface **TopicSubscriber**
extends **MessageConsumer**

MQSeries 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

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 due to some internal error.

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 due to some internal error.

Part 4. Appendices

Appendix A. Mapping between Administration tool properties and programmable properties 281

Appendix B. Scripts provided with MQSeries classes for Java Message Service(JMS) 283

Appendix C. LDAP server configuration for Java objects 285
Checking your LDAP server configuration 285
Configuration procedures 285

Appendix D. Notices 287
Trademarks 288

Appendix A. Mapping between Administration tool properties and programmable properties

MQSeries Classes for Java Message Service provides facilities for setting and querying the properties of administered objects using the Administration tool, or from within an application program. Table 29 shows the mapping between the property names used from within 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 29. Comparison of representations of properties within the administration tool, and the programmable equivalents.

Property	Member variable name	Property value mapping	
		Tool	Program
DESCRIPTION	description		
TRANSPORT	transportType	• BIND • CLIENT	JMSC.MQJMS_TP_BINDINGS_MQ JMSC.MQJMS_TP_CLIENT_MQ_TCPIP
CLIENTID	clientId		
QMANAGER	queueManager*		
HOSTNAME	hostName		
PORT	port		
CHANNEL	channel		
CCSID	CCSID		
RECEXIT	receiveExit		
RECEXITINIT	receiveExitinit		
SECEXIT	securityExit		
SECEXITINIT	securityExitInit		
SENDEXIT	sendExit		
SENDEXITINIT	sendExitInit		
TEMPMODEL	temporaryModel		
BROKERVER	brokerVersion	• V1 • V2	JMSC.MQJMS_BROKER_V1 JMSC.MQJMS_BROKER_V2
BROKERPUBQ	brokerPubQueue		
BROKERQMGR	brokerQueueManager		
BROKERCONQ	brokerControlQueue		
EXPIRY	expiry	• APP • UNLIM	JMSC.MQJMS_EXP_APP JMSC.MQJMS_EXP_UNLIMITED
PRIORITY	priority	• APP • QDEF	JMSC.MQJMS_PRI_APP JMSC.MQJMS_PRI_QDEF
PERSISTENCE	persistence	• APP • QDEF • PERS • NON	JMSC.MQJMS_PER_APP JMSC.MQJMS_PER_QDEF JMSC.MQJMS_PER_PER JMSC.MQJMS_PER_NON
TARGCLIENT	targetClient	• JMS • MQ	JMSC.MQJMS_CLIENT_JMS_COMPLIANT JMSC.MQJMS_CLIENT_NONJMS_MQ
ENCODING	encoding		
QUEUE	baseQueueName		
TOPIC	baseTopicName		

Note: * for an MQQueue object, the member variable name is baseQueueManagerName

Appendix B. Scripts provided with MQSeries classes for Java Message Service(JMS)

The following files are provided in the `bin` directory of your MQ JMS installation. These scripts are provided to assist with common tasks that need to be performed while installing or using MQ JMS. Table 30 lists the scripts and their uses.

Table 30. Utilities suppled with MQSeries classes for Java Message Service(JMS)

Utility	Use
IVTRun.bat IVTTidy.bat IVTSetup.bat	Used to run the point-to-point installation verification test program, described in “Running the point-to-point IVT” on page 18
PSIVTRun.bat	Used to run the Pub/Sub installation verification test program described in “The Publish/Subscribe Installation Verification Test” on page 21.
formatLog.bat	Used to convert binary log files to plain text,described in “Logging” on page 25
JMSAdmin.bat	Used to run the administration tool, described in Chapter 5, “Using the MQ JMS administration tool” on page 27
JMSAdmin.config	Configuration file for the administration tool, described in “Configuration” on page 28
runjms.bat	A utility script to assist with the running of JMS applications, described in “Running your own programs” on page 24
PSReportDump.class	Used to view broker report messages, described in “Handling broker reports” on page 160
Note: On UNIX systems the extension '.bat' is omitted from the filenames.	

Appendix C. LDAP server configuration for Java objects

If you are using JNDI to store MQJMS administered objects and you are using an LDAP server as your JNDI service provider, the server must be LDAP v3, such as the SecureWay eNetwork Directory v3.1., and it must be configured to store Java objects.

Checking your LDAP server configuration

To check whether the LDAP server is already configured to accept Java objects, run the MQ JMS Administration Tool in LDAP mode (see “Invoking the Administration tool” on page 27).

Attempt to create and display a test object using the following commands:

```
DEFINE QCF(ldapTest)
DISPLAY QCF(ldapTest)
```

If no exception occurs, then your server is properly configured configured and you can proceed to store JMS objects.

If a 'SchemaViolationException' is returned, your server is not configured for the storing of Java objects. The procedures below are provided to assist you with the configuration task.

Configuration procedures

Many LDAP servers provide tools that allow you to administer the server, refer to your server documentation for details on using these tools. The tools should allow you to view and update the schema, which contains 'attribute' and 'objectclass' definitions.

Ensure that the schema contains the following objectclass definitions, adding them if necessary:

```
( 1.3.6.1.4.1.42.2.27.4.2.1
        NAME 'javaContainer'
        SUP top
        MUST cn )

( 1.3.6.1.4.1.42.2.27.4.2.4
        NAME 'javaObject'
        SUP top
        ABSTRACT
        MUST javaClassName
        MAY javaCodebase )

( 1.3.6.1.4.1.42.2.27.4.2.5
        NAME 'javaSerializedObject'
        SUP javaObject
        AUXILIARY
        MUST javaSerializedData
        MAY javacodebase )
```

```
( 1.3.6.1.4.1.42.2.27.4.2.7
        NAME 'javaNamingReference'
        SUP javaObject
        AUXILIARY
        MUST javaclassname
        MAY ( javaFactory $ javaReferenceAddress ) )
```

In addition, ensure that the schema contains the following attribute definitions, updating the schema if necessary:

```
( 1.3.6.1.4.1.42.2.27.4.1.3
        NAME 'javaReferenceAddress'
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )

( 1.3.6.1.4.1.42.2.27.4.1.4
        NAME 'javaFactory'
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )

( 1.3.6.1.4.1.42.2.27.4.1.6
        NAME 'javaCodebase'
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )
```

When you have completed the updates, stop and restart the LDAP server, and repeat the configuration checking procedure described in “Checking your LDAP server configuration” on page 285.

Appendix D. 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.

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.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	AS/400	BookManager
IBM	IBMLink	MQSeries
MVS/ESA	OS/2	OS/390
OS/400	SupportPac	System/390
VSE/ESA		

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Windows and Windows NT are registered trademarks of Microsoft Corporation in the United States and/or other countries.

Other company, product, and service names may be trademarks or service marks of others.

Glossary of terms and abbreviations

This glossary describes terms used in this book and words used with other than their everyday meaning. In some cases, a definition may not be the only one applicable to a term, but it gives the particular sense in which the word is used in this book.

If you do not find the term you are looking for, see the index or the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

applet. A Java program which is designed to run only on a web page.

Application Programming Interface (API). An Application Programming Interface consists of the functions and variables that programmers are allowed to use in their applications.

channel. See MQI channel.

class. A class is an encapsulated collection of data and methods to operate on the data. A class may be instantiated to produce an object that is an instance of the class.

client. In MQSeries, a client is a run-time component that provides access to queuing services on a server for local user applications.

encapsulation. Encapsulation is an object-oriented programming technique that makes an object's data private or protected and allows programmers to access and manipulate the data only through method calls.

Hypertext Markup Language (HTML). A language used to define informa tion that is to be displayed on the World Wide Web.

Internet Inter-ORB Protocol (IIOP). A standard for TCP/IP communications between ORBs from different vendors.

instance. An instance is an object. When a class is instantiated to produce an object, we say that the object is an instance of the class.

interface. An interface is a class that contains only abstract methods and no instance variables. An interface provides a common set of methods that can be implemented by subclasses of a number of different classes.

Internet. The Internet is a cooperative public network of shared information. Physically, the Internet uses a subset of the total resources of all the currently existing public telecommunication networks. Technically, what distinguishes the Internet as a cooperative public network is its use of a set of protocols called TCP/IP (Transport Control Protocol/Internet Protocol).

Java Developers Kit (JDK). A package of software distributed by Sun Microsystems for Java developers. It includes the Java interpreter, Java classes and Java development tools: compiler, debugger, disassembler, appletviewer, stub file generator, and documentation generator.

Java Naming and Directory Service (JNDI). An API specified in the Java programming language. It provides naming and directory functions to applications written in the Java programming language.

Java Message Service (JMS). Sun microsystem's API for accessing enterprise messaging systems from Java programs.

Lightweight Directory Access Protocol (LDAP). LDAP is a client-server protocol for accessing a directory service.

message. In message queuing applications, a message is a communication sent between programs.

message queue. See queue

message queuing. A programming technique in which each program within an application communicates with the other programs by putting messages on queues.

method. Method is the object-oriented programming term for a function or procedure.

MQI channel. An MQI channel connects an MQSeries client to a queue manager on a server system and transfers MQI calls and responses in a bidirectional manner.

MQSeries. MQSeries is a family of IBM licensed programs that provide message queuing services.

object. (1) In Java, an object is an instance of a class. A class models a group of things; an object models a particular member of that group. (2) In MQSeries, an object is a queue manager, a queue, or a channel.

Object Request Broker (ORB). An application framework that provides interoperability between objects, built in different languages, running on different machines, in heterogeneous distributed environments.

package. A package in Java is a way of giving a piece of Java code access to a specific set of classes. Java code that is part of a particular package has access to

all the classes in the package and to all non-private methods and fields in the classes.

private. A private field is not visible outside its own class.

protected. A protected field is visible only within its own class, within a subclass, or within packages of which the class is a part

public. A public class or interface is visible everywhere. A public method or variable is visible everywhere that its class is visible

queue. A queue is an MQSeries object. Message queueing applications can put messages on, and get messages from, a queue

queue manager. a queue manager is a system program the provides message queuing services to applications.

server. (1) An MQSeries a server is a queue manager that provides message queuing services to client applications running on a remote workstation. (2) More generally, a server is a program that responds to requests for information in the particular two-program information flow model of client/server. (3) The computer on which a server program runs.

servlet. A Java program which is designed to run only on a web server.

subclass. A subclass is a class that extends another. The subclass inherits the public and protected methods and variables of its superclass.

superclass. A superclass is a class that is extended by some other class. The superclass's public and protected methods and variables are available to the subclass.

Transmission Control Protocol/Internet Protocol (TCP/IP). A set of communication protocols that support peer-to-peer connectivity functions for both local and wide area networks.

Visibroker for Java. An Object Request Broker (ORB) written in Java

Web. See World Wide Web.

Web browser. A program that formats and displays information that is distributed on the World Wide Web.

World Wide Web (Web). The World Wide Web is an Internet service, based on a common set of protocols, which allows a particularly configured server computer to distribute documents across the Internet in a standard way.

Bibliography

This section describes the documentation available for all current MQSeries products.

MQSeries cross-platform publications

Most of these publications, which are sometimes referred to as the MQSeries "family" books, apply to all MQSeries Level 2 products. The latest MQSeries Level 2 products are:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V4R2
- MQSeries for AT&T GIS UNIX V2.2
- MQSeries for Compaq (DIGITAL) OpenVMS V2.2.1.1
- MQSeries for DIGITAL UNIX (Compaq Tru64 UNIX) V2.2.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for OS/390 V2.1
- MQSeries for SINIX and DC/OSx V2.2
- MQSeries for Sun Solaris V5.1
- MQSeries for Tandem NonStop Kernel V2.2.0.1
- MQSeries for VSE/ESA V2.1
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1
- MQSeries for Windows NT V5.1

Any exceptions to this general rule are indicated.

MQSeries Brochure
The *MQSeries Brochure*, G511-1908, gives a brief introduction to the benefits of MQSeries. It is intended to support the purchasing decision, and describes some authentic customer use of MQSeries.

MQSeries: An Introduction to Messaging and Queuing
MQSeries: An Introduction to Messaging and Queuing, GC33-0805, describes briefly what MQSeries is, how it works, and how it can solve some classic interoperability problems. This book is intended for a more technical audience than the *MQSeries Brochure*.

MQSeries Planning Guide
The *MQSeries Planning Guide*, GC33-1349, describes some key MQSeries concepts, identifies items that need to be considered before MQSeries is installed, including storage requirements, backup and recovery, security, and migration from earlier releases, and specifies hardware and software requirements for every MQSeries platform.

MQSeries Intercommunication
The *MQSeries Intercommunication* book, SC33-1872, defines the concepts of distributed queuing and explains how to set up a distributed queuing network in a variety of MQSeries environments. In particular, it demonstrates how to (1) configure communications to and from a representative sample of MQSeries products, (2) create required MQSeries objects, and (3) create and configure MQSeries channels. The use of channel exits is also described.

MQSeries Queue Manager Clusters
MQSeries Queue Manager Clusters, SC34-5349, describes MQSeries clustering. It explains the concepts and terminology and shows how you can benefit by taking advantage of clustering. It details changes to the MQI, and summarizes the syntax of new and changed MQSeries commands. It shows a number of examples of tasks you can perform to set up and maintain clusters of queue managers.

This book applies to the following MQSeries products only:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V4R2
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for OS/390 V2.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries Clients
The *MQSeries Clients* book, GC33-1632, describes how to install, configure, use, and manage MQSeries client systems.

MQSeries System Administration
The *MQSeries System Administration* book, SC33-1873, supports day-to-day management of local and remote MQSeries objects. It includes topics such as security, recovery and restart, transactional support, problem determination, and the dead-letter queue handler. It also includes the syntax of the MQSeries control commands.

This book applies to the following MQSeries products only:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries Command Reference
The *MQSeries Command Reference*, SC33-1369, contains the syntax of the MQSC commands, which are used by MQSeries system operators and administrators to manage MQSeries objects.

MQSeries Programmable System Management
The *MQSeries Programmable System Management* book, SC33-1482, provides both reference and guidance information for users of MQSeries events, Programmable Command Format (PCF) messages, and installable services.

MQSeries Administration Interface Programming Guide and Reference
The *MQSeries Administration Interface Programming Guide and Reference*, SC34-5390, provides information for users of the MQAI. The MQAI is a programming interface that simplifies the way in which applications manipulate Programmable Command Format (PCF) messages and their associated data structures.

This book applies to the following MQSeries products only:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V4R2
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries Messages
The *MQSeries Messages* book, GC33-1876, which describes "AMQ" messages issued by MQSeries, applies to these MQSeries products only:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1

This book is available in softcopy only.

For other MQSeries platforms, the messages are supplied with the system. They do not appear in softcopy manual form.

MQSeries Application Programming Guide
The *MQSeries Application Programming Guide*, SC33-0807, provides guidance information for users of the message queue interface (MQI). It describes how to design, write, and build an MQSeries application. It also includes full descriptions of the sample programs supplied with MQSeries.

MQSeries Application Programming Reference
The *MQSeries Application Programming Reference*, SC33-1673, provides comprehensive reference information for users of the MQI. It includes: data-type descriptions; MQI call syntax; attributes of MQSeries objects; return codes; constants; and code-page conversion tables.

MQSeries Application Programming Reference Summary
The *MQSeries Application Programming Reference Summary*, SX33-6095, summarizes the information in the *MQSeries Application Programming Reference* manual.

MQSeries Using C++
MQSeries Using C++, SC33-1877, provides both guidance and reference information for users of the MQSeries C++ programming-language binding to the MQI. MQSeries C++ is supported by these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for AS/400 V4R2
- MQSeries for OS/390 V2.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries C++ is also supported by MQSeries clients supplied with these products and installed in the following environments:

- AIX
- HP-UX
- OS/2
- Sun Solaris
- Windows NT
- Windows 3.1
- Windows 95 and Windows 98

MQSeries Using Java
MQSeries Using Java, SC34-5456, provides both guidance and reference information for users of the MQSeries Bindings for Java and the MQSeries Client for Java. MQSeries classes for Java are supported by these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V4R2
- MQSeries for HP-UX V5.1
- MQSeries for MVS/ESA V1.2
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries platform-specific publications

Each MQSeries product is documented in at least one platform-specific publication, in addition to the MQSeries family books.

MQSeries for AIX

MQSeries for AIX Version 5 Release 1 Quick Beginnings, GC33-1867

MQSeries for AS/400

MQSeries for AS/400 Version 4 Release 2 Administration Guide, SC33-1956

MQSeries for AS/400 Version 4 Release 2 Application Programming Reference (ILE RPG), SC33-1957

MQSeries for AT&T GIS UNIX

MQSeries for AT&T GIS UNIX Version 2 Release 2 System Management Guide, SC33-1642

MQSeries for Compaq (DIGITAL) OpenVMS

MQSeries for Compaq (DIGITAL) OpenVMS Version 2 Release 2.1.1 System Management Guide, GC33-1791

MQSeries for Digital UNIX (Compaq Tru64 UNIX)

MQSeries for Digital UNIX Version 2 Release 2.1 System Management Guide, GC34-5483

MQSeries for HP-UX

MQSeries for HP-UX Version 5 Release 1 Quick Beginnings, GC33-1869

MQSeries for OS/2 Warp

MQSeries for OS/2 Warp Version 5 Release 1 Quick Beginnings, GC33-1868

MQSeries for OS/390

MQSeries for OS/390 Version 2 Release 1 Licensed Program Specifications, GC34-5377

MQSeries for OS/390 Version 2 Release 1 Program Directory

MQSeries for OS/390 Version 2 Release 1 System Management Guide, SC34-5374

MQSeries for OS/390 Version 2 Release 1 Messages and Codes, GC34-5375

MQSeries for OS/390 Version 2 Release 1 Problem Determination Guide, GC34-5376

MQSeries link for R/3

MQSeries link for R/3 Version 1 Release 2 User's Guide, GC33-1934

MQSeries for SINIX and DC/OSx

MQSeries for SINIX and DC/OSx Version 2 Release 2 System Management Guide, GC33-1768

MQSeries for Sun Solaris

MQSeries for Sun Solaris Version 5 Release 1 Quick Beginnings, GC33-1870

MQSeries for Tandem NonStop Kernel

MQSeries for Tandem NonStop Kernel Version 2 Release 2.0.1 System Management Guide, GC33-1893

MQSeries for VSE/ESA

MQSeries for VSE/ESA Version 2 Release 1 Licensed Program Specifications, GC34-5365

MQSeries for VSE/ESA Version 2 Release 1 System Management Guide, GC34-5364

MQSeries for Windows

MQSeries for Windows Version 2 Release 0 User's Guide, GC33-1822

MQSeries for Windows Version 2 Release 1 User's Guide, GC33-1965

MQSeries for Windows NT

MQSeries for Windows NT Version 5 Release 1 Quick Beginnings, GC34-5389

MQSeries for Windows NT Using the Component Object Model Interface, SC34-5387

MQSeries LotusScript Extension, SC34-5404

Softcopy books

Most of the MQSeries books are supplied in both hardcopy and softcopy formats.

BookManager format

The MQSeries library is supplied in IBM BookManager format on a variety of online library collection kits, including the *Transaction Processing and Data* collection kit, SK2T-0730. You can view the softcopy books in IBM BookManager format using the following IBM licensed programs:

BookManager READ/2
BookManager READ/6000
BookManager READ/DOS
BookManager READ/MVS
BookManager READ/VM
BookManager READ for Windows

HTML format

Relevant MQSeries documentation is provided in HTML format with these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V4R2
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1 (compiled HTML)
- MQSeries link for R/3 V1.2

The MQSeries books are also available in HTML format from the MQSeries product family Web site at:

`http://www.ibm.com/software/ts/mqseries/`

Portable Document Format (PDF)

PDF files can be viewed and printed using the Adobe Acrobat Reader.

If you need to obtain the Adobe Acrobat Reader, or would like up-to-date information about the platforms on which the Acrobat Reader is supported, visit the Adobe Systems Inc. Web site at:

```
http://www.adobe.com/
```

PDF versions of relevant MQSeries books are supplied with these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V4R2
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1
- MQSeries link for R/3 V1.2

PDF versions of all current MQSeries books are also available from the MQSeries product family Web site at:

```
http://www.ibm.com/software/ts/mqseries/
```

PostScript format

The MQSeries library is provided in PostScript (.PS) format with many MQSeries Version 2 products. Books in PostScript format can be printed on a PostScript printer or viewed with a suitable viewer.

Windows Help format

The *MQSeries for Windows User's Guide* is provided in Windows Help format with MQSeries for Windows Version 2.0 and MQSeries for Windows Version 2.1.

MQSeries information available on the Internet

The MQSeries product family Web site is at:

```
http://www.ibm.com/software/ts/mqseries/
```

By following links from this Web site you can:

- Obtain latest information about the MQSeries product family.
- Access the MQSeries books in HTML and PDF formats.
- Download MQSeries SupportPacs.

Index

A

accessing queues and processes 53
administered objects 31, 144
administering JMS objects 31
administration
  commands 29
  verbs 29
administration tool
  configuration file 28
  configuring 28
  overview 27
  starting 27
advantages of Java interface 41
applet example 46
applet viewer
  using 5
  with sample applet 12
applet viewer, using 11
applets versus applications 45
applets, running 58
application example 50
applications
  closing 152
  Pub/Sub, writing 153
  unexpected termination 159
applications versus applets 45
asynchronous message delivery 151

B

behavior in different environments 61
benefits of JMS 3
bibliography 291
bindings
  connection 6
  connection, programming 46
  verifying 13
bindings transport, choosing 146
body, message 161
BookManager 293
broker reports 160
BROKERCONQ object property 34
BROKERPUBQ object property 34
BROKERQMGR object property 34
BROKERVER object property 34
building a connection 144
bytes message 161
BytesMessage interface 184
BytesMessage type 150

C

CCSID object property 34
CHANGE (administration verb) 29
CHANNEL object property 34
choosing transport 146
CICS Transaction Server
  running applications 59
  using 14
CICS Transaction Server for OS/390 xi
class library 43
classes, core 61
classes, JMS 179
classes, MQSeries classes for Java 67
  MQC 131
  MQChannelDefinition 68
  MQChannelExit 70
  MQDistributionList 73
  MQDistributionListItem 75
  MQEnvironment 77
  MQException 81
  MQGetMessageOptions 83
  MQManagedObject 87
  MQMessage 90
  MQMessageTracker 108
  MQProcess 110
  MQPutMessageOptions 112
  MQQueue 115
  MQQueueManager 123
  MQReceiveExit 132
  MQSecurityExit 134
  MQSendExit 136
classpath, configuring 17
CLASSPATH, updating 8
client properties 36
client transport, choosing 146
CLIENTID object property 34
clients
  configuring queue manager 11
  connection 5
  programming 45
  verifying 13
closing
  applications 152
  JMS resources in Pub/Sub mode 155
  resources 152
code examples 46
com.ibm.jms, package 183
com.ibm.mq.iiop.jar 8
com.ibm.mq.jar 8
com.ibm.mq.jms, package 182

com.ibm.mqbind.jar 8
combinations, valid, of objects and properties 35
commands, administration 29
compiling MQSeries classes for Java programs 58
configuration file, for administration tool 28
configuring
  for Pub/Sub 18
  queue manager for clients 11
  the administration tool 28
  Web server 10
  your classpath 17
  your installation 17
connecting to a queue manager 52
connection
  building 144
  creating 145
  interface 143
  MQSeries, losing 159
  starting 145
Connection interface 192
connection type, defining 46
ConnectionFactory interface 195
ConnectionMetaData interface 198
connections xi
  binding 6
  client 5
  client, programming 45
  programming 45
converting the log file 25
COPY (administration verb) 29
core classes 61
  exceptions 62
  extensions for V5 63
createQueueSession method 147
createReceiver method 150
createSender method 147
creating
  a connection 145
  factories at runtime 145
  JMS objects 32
  Topics at runtime 157
customizing the sample applet 13

D

default trace and log output locations 24
DEFINE (administration verb) 29
defining connection type 46
defining transport 146
DELETE (administration verb) 29
DeliveryMode interface 200
dependencies, property 36
DESCRIPTION object property 34
Destination interface 201
differences between applets and applications 45
differences due to environment 61
directories, installation 7
disconnecting from a queue manager 52
DISPLAY (administration verb) 29
durable subscribers 158

E

ENCODING object property 36
END (administration verb) 29
environment differences 61
environment variables, configuring 17
  refid-jmscfi.environment variables 17
error
  conditions for object creation 37
  recovery, IVT 21
  recovery, PSIVT 23
error messages 16
errors
  logging 25
  runtime, handling 152
errors, handling 55
example code 46
exception listener 152
ExceptionListener interface 203
exceptions
  JMS 152
  MQSeries 152
exceptions to core classes 62
exit string properties 36
EXPIRY object property 34
extensions to core classes for V5 63
extra function provided over MQ Java 3

F

factories, creating at runtime 145
formatLog utility 25, 283
function, extra provided over MQ Java 3

G

getting started 3
glossary 289

H

handling
  errors 55
  messages 54
handling runtime errors 152
headers, message 161
HOSTNAME object property 34
HTML (Hypertext Markup Language) 293
Hypertext Markup Language (HTML) 293

I

IIOP
  connection, programming 45
IIOP support xi
  IIOP xi
  visibroker xi
import statements 153
INITIAL_CONTEXT_FACTORY parameter 28
inquire and set 56
installation
  Installation Verification Test program for Pub/Sub (PSIVT) 21
  IVT error recovery 21
  PSIVT error recovery 23
  setup 17
  verifying 17
installation directories 7
Installation Verification Test program (IVT) 18
installing MQ base Java 9
interface, programming 42
interfaces
  JMS 143, 179
  MQSeries 143
introduction 3
introduction for programmers 41
introduction to JMS 3
IVT (Installation Verification Test program) 18
IVTRun utility 19, 20, 283
IVTSetup utility 19, 283
IVTTidy utility 21, 283

J

jar files 8
Java classes 43, 67
Java Developers Kit 42
Java interface, advantages 41
javax.jms package 179
JDK 42
JMS
  administered objects 144
  benefits 3
  classes 179
  exception listener 152
  exceptions 152
  interfaces 143, 179
  introduction 3
  message types 149
  messages 161
  model 143
  objects for Pub/Sub 153
  objects, administering 31
  objects, creating 32
  objects, properties 33
  programs, writing 143
JMS *(continued)*
  resources, closing in Pub/Sub mode 155
JMSAdmin utility 283
JMSAdmin.config utility 283
JMSBytesMessage class 184
JMSCorrelationID header field 161
JMSMapMessage class 204
JMSMessage class 212
JMSStreamMessage class 253
JMSTextMessage class 262
JNDI
  retrieving 144
  security considerations 29

L

LDAP naming considerations 32
library, Java classes 43
listener, JMS exception 152
local publications, suppressing 159
log file
  converting 25
  default output location 24
logging errors 25

M

manipulating subcontexts 30
map message 161
MapMessage interface 204
MapMessage type 150
mapping properties between admin tool and programs 281
message
  body 161
  delivery, asynchronous 151
  headers 161
  properties 161
  selectors 151, 161
  selectors and SQL 162
  selectors in Pub/Sub mode 158
  sending 147
  types 149, 161
Message interface 212
MessageConsumer interface 143, 224
MessageListener interface 226
MessageProducer interface 143, 227
MessageProducer object 147
messages
  error 16
  handling 54
  JMS 161
  publishing 154
  receiving 150
  receiving in Pub/Sub mode 155
  selecting 151, 161

model, JMS 143
MOVE (administration verb) 29
MQC 131
MQChannelDefinition 68
MQChannelExit 70
MQConnection class 192
MQConnectionFactory class 195
MQConnectionMetaData class 198
MQDeliveryMode class 200
MQDestination class 201
MQDistributionList 73
MQDistributionListItem 75
MQEnvironment 46, 52, 77
MQException 81
MQGetMessageOptions 83
MQIVP
  listing 14
  sample application 13
  tracing 15
mqjavac
  tracing 15
  using to verify 11
MQManagedObject 87
MQMessage 90
MQMessageConsumer class 224
MQMessageProducer interface 227
MQMessageTracker 108
MQObjectMessage class 231
MQProcess 110
MQPutMessageOptions 112
MQQueue 115
  (JMS object) 31
  class 232
  for verification 19
MQQueueBrowser class 234
MQQueueConnection class 236
MQQueueConnectionFactory
  (JMS object) 31
  class 238
  for verification 19
  set methods 146
MQQueueEnumeration class 230
MQQueueManager 53, 123
MQQueueReceiver class 240
MQQueueSender interface 243
MQQueueSession class 246
MQReceiveExit 132
MQSecurityExit 134
MQSendExit 136
MQSeries
  connection, losing 159
  exceptions 152
  interfaces 143
MQSeries classes for Java classes 67
MQSeries publications 291
MQSeries software client CD 9
MQSeries software server CD 9
MQSeries supported verbs 42
MQSeriesV5 extensions 63
MQSession class 249
MQTemporaryQueue class 260
MQTemporaryTopic class 261
MQTopic
  (JMS object) 31
  class 263
MQTopicConnection class 265
MQTopicConnectionFactory
  (JMS object) 31
  class 267
MQTopicPublisher class 270
MQTopicSession class 275
MQTopicSubscriber class 278
multithreaded programs 56

N

names, of Topics 155
naming considerations, LDAP 32
Netscape Navigator, using 5
non-durable subscribers 158

O

object creation, error conditions 37
ObjectMessage interface 231
ObjectMessage type 150
objects
  administered 144
  JMS, administering 31
  JMS, creating 32
  JMS, properties 33
  message 161
  retrieving from JNDI 144
objects and properties, valid combinations 35
obtaining a session 147
Operations on queue managers 52
options
  connection 4
OS/390 support xi
overview 3

P

package
  com.ibm.jms 183
  com.mq.ibm.jms 182
  javax.jmss 179
PDF (Portable Document Format) 294
PERSISTENCE object property 34
platform differences 61

point-to-point installation verification 18
PORT object property 34
Portable Document Format (PDF) 294
PostScript format 294
prerequisite software 6
prerequisites 6
PRIORITY object property 34
problems, solving 15, 24
problems, solving in Pub/Sub mode 159
processes, accessing 53
Product Extensions 9
progams, running 58
programmers, introduction 41
programming
  bindings connection 46
  client connections 45
  compiling 58
  connections 45
  multithreaded 56
  tracing 59
  writing 45
programming interface 42
programs
  JMS, writing 143
  Pub/Sub, writing 153
  running 24
properties
  client 36
  dependencies 36
  mapping between admin tool and programs 281
  message 161
  of exit strings 36
  of JMS objects 33
  queue, setting 147
properties and objects, valid combinations 35
PROVIDER_URL parameter 28
PSIVT (Installation Verification Test program) 21
PSIVTRun utility 22, 283
Pub/Sub
publications
  MQSeries 291
publications (Pub/Sub), local suppressing 159
Publish/Subscribe Installation Verification Test program (PSIVT) 21
publishing messages 154

Q

QMANAGER object property 34
Queue
  interface 232
  object 144
queue manager
  connecting to 52
  disconnecting from 52
  operations on 52
queue manager, configuring for clients 11
QUEUE object property 34
queue properties
  setting 147
  setting with set methods 149
QueueBrowser interface 234
QueueConnection interface 236
QueueReceiver interface 240
QueueRequestor class 241
queues, accessing 53
QueueSender interface 243
QueueSession interface 246

R

reading strings 55
receiving
RECEXIT object property 34
RECEXITINIT object property 34
reports, broker 160
resources, closing 152
retrieving objects from JNDI 144
runjms utility 24, 283
running
  applets 58
  applications under CICS Transaction Server 59
  in a Web browser 5
  MQSeries classes for Java programs 58
  stand-alone 5
  with applet viewer 5
  your own programs 15
runtime

S

sample applet
  customizing 13
  tracing 15
  using to verify 11
  with applet viewer 12
  with Web browser 12
sample application
  tracing 15
  using to verify 13
sample application, Pub/Sub 153
scripts provided with MQSeries classes for Java Message Service(JMS) 283
SECEXIT object property 34
SECEXITINIT object property 34
security considerations, JNDI 29
SECURITY_AUTHENTICATION parameter 28
selecting a subset of messages 151, 161
selectors
  message 151, 161
  message in Pub/Sub mode 158
  message, and SQL 162

SENDEXIT object property 34
SENDEXITINIT object property 34
sending a message 147
session interface 143, 249
session, obtaining 147
set and inquire 56
set methods
  on MQQueueConnectionFactory 146
  using to set queue properties 149
setting
  queue properties 147
  queue properties with set methods 149
shutting down applications 152
softcopy books 293
software requirements 6
software, prerequisite 6
solving problems 15
  general 24
  in Pub/Sub mode 159
SQL for message selectors 162
stand-alone, running 5
starting a connection 145
starting the administration tool 27
stream message 161
StreamMessage interface 253
StreamMessage type 150
strings, reading and writing 55
subcontexts, manipulating 30
subscriber options 158
subscriptions, receiving 155
subset of messages, selecting 151, 161
Sun JMS interfaces and classes 179
Sun Web site 3
suppressing local publications 159

T

TARGCLIENT object property 34
TCP/IP
  client verifying 13
  connection, programming 45
TEMPMODEL object property 34
TemporaryQueue interface 260
TemporaryTopic interface 261
termination, unexpected 159
testing MQSeries classes for Java progams 59
text message 161
TextMessage interface 262
TextMessage type 150
Topic
  interface 153, 263
  names 155
  names, wildcards 156
  object 144
TOPIC object property 34
TopicConnection 153
TopicConnection interface 265
TopicConnectionFactory 153
  interface 267
  object 144
TopicPublisher 154
TopicPublisher interface 270
TopicRequestor class 273
TopicSession 153
TopicSession interface 275
TopicSubscriber 154
TopicSubscriber interface 278
trace, default output location 24
tracing
  programs 59
  sample applet 15
  the sample application 15
TRANSPORT object property 34
transport options xi
transport, choosing 146
types of JMS message 149
types of message 161

U

unexpected application termination 159
uniform resource identifier (URI) for queue properties 148
updating your CLASSPATH 8
URI for queue properties 148
user exits, writing 57
uses for MQSeries 3
using
  applet viewer 11
  CICS Transaction server 14
  MQ base Java 11
utilities provided with MQSeries classes for Java Message Service(JMS) 283

V

v5 extensions 63
valid combinations of objects and properties 35
verbs, MQSeries supported 42
verification
  with JNDI (PTP) 19
  with JNDI (Pub/Sub) 23
  without JNDI (PTP) 19
  without JNDI (Pub/Sub) 22
verifying
  client mode installation 11
  TCP/IP clients 13
  with the sample applet 11
  with the sample application 13
verifying your installation 17

versions of software required 6
Visibroker xi
  using 4, 5, 14

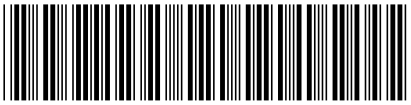
W

Web browser
  using 5
  with sample applet 12
Web server, configuring 10
widcards in topic names 156
Windows Help 294
writing
  programs 45
  strings 55
  user exits 57

X

XAQueueConnection interface 236
XAQueueConnectionFactory interface 238

SC34-5456-02



Spine information:

MQSeries® Using Java™