MQSeries®

Using Java™

SC34-5456-06

Note!

Before using this information and the product it supports, be sure to read the general information under "Appendix F. Notices" on page 363.

Seventh edition (January 2001)

This edition applies to IBM® MQSeries classes for Java Version 5.2.0 and MQSeries classes for Java Message Service Version 5.2, and to any subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures vii

Tables ix

About this book xi

Abbreviations used in this book. xi
Who this book is for xi
What you need to know to understand this book . . xi
How to use this book xii

Summary of changes xiii

| Changes to this edition (SC34-5456-06) xiii
Changes to the sixth edition (SC34-5456-05) . . . xiv
Changes to the fifth edition (SC34-5456-04) . . . xiv

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? 3
Who should use MQ Java? 3
Connection options 4
 Client connection 5
 Using VisiBroker for Java 6
 Bindings connection 6
Prerequisites 6

| **Chapter 2. Installation procedures . . . 7**

| Installing MQSeries classes for Java and MQSeries
| classes for Java Message Service 7
| Installing on UNIX 8
| Installing on AS/400. 9
| Installing on Linux 9
| Installing on Windows. 10
| Installation directories 10
| Environment variables. 10
| Web server configuration 12

Chapter 3. Using MQSeries classes for Java (MQ base Java). 13

Using the sample applet to verify the TCP/IP client 13
 Using the sample applet on AS/400 13
 Configuring your queue manager to accept client connections 13
 Running from appletviewer 14
 Customizing the verification applet 15
Verifying with the sample application 15
 Using VisiBroker connectivity 16
 Using CICS Transaction Server for OS/390 . . . 16
Running your own MQ base Java programs . . . 16
Solving MQ base Java problems 17
 Tracing the sample applet 17
 Tracing the sample application 17
 Error messages 18

Chapter 4. Using MQSeries classes for Java Message Service (MQ JMS) . . . 19

Post installation setup 19
 Additional setup for Publish/Subscribe mode . . 20
| Queues that require authorization for
| non-privileged users 20
Running the point-to-point IVT. 21
 Point-to-point verification without JNDI. . . . 21
 Point-to-point verification with JNDI 22
 IVT error recovery 24
The Publish/Subscribe Installation Verification Test 24
 Publish/Subscribe verification without JNDI . . 25
 Publish/Subscribe verification with JNDI . . . 26
 PSIVT error recovery 27
Running your own MQ JMS programs 27
Solving problems 28
 Tracing programs 28
 Logging 28

Chapter 5. Using the MQ JMS administration tool 31

Invoking the Administration tool 31
Configuration 32
 Configuring for WebSphere 33
 Security 33
Administration commands 34
Manipulating subcontexts 35
Administering JMS objects 35
 Object types 35
 Verbs used with JMS objects. 36
 Creating objects 37
 Properties 38
 Property dependencies 41
 The ENCODING property 42
 Sample error conditions 43

Part 2. Programming with MQ base Java 45

Chapter 6. Introduction for programmers 47

Why should I use the Java interface? 47
The MQSeries classes for Java interface 48
Java Development Kit 48
MQSeries classes for Java class library 49

Chapter 7. Writing MQ base Java programs 51

Should I write applets or applications? 51
Connection differences. 51
 Client connections 51

Bindings mode 52
Defining which connection to use 52
Example code fragments 52
Example applet code 52
Example application code 56
Operations on queue managers 58
Setting up the MQSeries environment 58
Connecting to a queue manager 58
Accessing queues and processes 59
Handling messages 60
Handling errors 61
Getting and setting attribute values 61
Multithreaded programs 62
Writing user exits 63
| Connection pooling 64
| Controlling the default connection pool 64
| The default connection pool and multiple
| components 66
| Supplying a different connection pool 67
| Supplying your own ConnectionManager . . . 68
Compiling and testing MQ base Java programs . . 69
Running MQ base Java applets 70
Running MQ base Java applications 70
Running MQ base Java applications under CICS
Transaction Server for OS/390 70
Tracing MQ base Java programs 70

Chapter 8. Environment-dependent behavior 73
Core details 73
Restrictions and variations for core classes 74
Version 5 extensions operating in other
environments 76

Chapter 9. The MQ base Java classes and interfaces 79
MQChannelDefinition 80
Variables 80
Constructors 81
MQChannelExit 82
Variables 82
Constructors 84
MQDistributionList 85
Constructors 85
Methods 85
MQDistributionListItem 87
Variables 87
Constructors 87
MQEnvironment. 88
Variables 88
Constructors 91
Methods 91
MQException 93
Variables 93
Constructors 93
MQGetMessageOptions 95
Variables 95
Constructors 98
MQManagedObject 99
Variables 99
Constructors. 100
Methods 100
MQMessage 102
Variables 102
Constructors. 110
Methods 110
MQMessageTracker 121
Variables 121
| MQPoolServices 123
| Constructors. 123
| Methods 123
| MQPoolServicesEvent 124
| Variables 124
| Constructors. 124
| Methods 125
| MQPoolToken 126
| Constructors. 126
MQProcess 127
Constructors. 127
Methods 127
MQPutMessageOptions 129
Variables 129
Constructors. 131
MQQueue 132
Constructors. 132
Methods 132
MQQueueManager 140
Variables 140
Constructors. 140
Methods 142
| MQSimpleConnectionManager 150
| Variables 150
| Constructors. 150
| Methods 150
MQC 152
| MQPoolServicesEventListener 153
| Methods 153
| MQConnectionManager 154
MQReceiveExit 155
Methods 155
MQSecurityExit 157
Methods 157
MQSendExit. 159
Methods 159
| ManagedConnection 161
| Methods 161
| ManagedConnectionFactory 164
| Methods 164
| ManagedConnectionMetaData 166
| Methods 166

Part 3. Programming with MQ JMS 167

Chapter 10. Writing MQ JMS programs 169
The JMS model 169
Building a connection 170
Retrieving the factory from JNDI 170
Using the factory to create a connection . . . 171
Creating factories at runtime 171
Choosing client or bindings transport 172

Obtaining a session 172
Sending a message 173
  Setting properties with the ‘set’ method . . . 174
  Message types 175
Receiving a message 175
  Message selectors 176
  Asynchronous delivery 177
Closing down 177
  Java Virtual Machine hangs at shutdown . . . 177
Handling errors 177
  Exception listener 178

Chapter 11. Programming Publish/Subscribe applications. 179
Writing a simple Publish/Subscribe application 179
  Import required packages 179
  Obtain or create JMS objects 179
  Publish messages 181
  Receive subscriptions. 181
  Close down unwanted resources 181
Using topics 181
  Topic names. 181
  Creating topics at runtime 182
Subscriber options. 183
  Creating non-durable subscribers. 184
  Creating durable subscribers 184
  Using message selectors 184
  Suppressing local publications. 184
  Combining the subscriber options 185
| Configuring the base subscriber queue 185
Solving Publish/Subscribe problems. 187
  Incomplete Publish/Subscribe close down. . . 187
  Handling broker reports. 188

Chapter 12. JMS messages 191
Message selectors 191
Mapping JMS messages onto MQSeries messages 195
  The MQRFH2 header. 196
  JMS fields and properties with corresponding MQMD fields 199
  Mapping JMS fields onto MQSeries fields (outgoing messages) 200
  Mapping MQSeries fields onto JMS fields (incoming messages) 204
  Mapping JMS to a native MQSeries application 205
  Message body 206

| **Chapter 13. MQ JMS Application Server Facilities 209**
| ASF classes and functions 209
| ConnectionConsumer. 209
| Planning an application 210
| Error handling 214
| Application server sample code 215
| MyServerSession.java. 217
| MyServerSessionPool.java 217
| MessageListenerFactory.java 218
| Examples of ASF use 219
| Load1.java 219
| CountingMessageListenerFactory.java 220
| ASFClient1.java. 220
| Load2.java 222
| LoggingMessageListenerFactory.java. 222
| ASFClient2.java. 222
| TopicLoad.java 223
| ASFClient3.java. 224
| ASFClient4.java. 225

Chapter 14. JMS interfaces and classes 227
Sun Java Message Service classes and interfaces 227
MQSeries JMS classes 230
BytesMessage 232
  Methods 232
Connection 240
  Methods 240
| ConnectionConsumer. 243
| Methods 243
ConnectionFactory. 244
  MQSeries constructor. 244
  Methods 244
ConnectionMetaData 248
  MQSeries constructor. 248
  Methods 248
DeliveryMode 250
  Fields 250
Destination 251
  MQSeries constructors 251
  Methods 251
ExceptionListener 253
  Methods 253
MapMessage 254
  Methods 254
Message 262
  Fields 262
  Methods 262
MessageConsumer 275
  Methods 275
MessageListener 277
  Methods 277
MessageProducer 278
  MQSeries constructors 278
  Methods 278
MQQueueEnumeration * 282
  Methods 282
ObjectMessage 283
  Methods 283
Queue. 284
  MQSeries constructors 284
  Methods 284
QueueBrowser 286
  Methods 286
QueueConnection 288
  Methods 288
QueueConnectionFactory 290
  MQSeries constructor. 290
  Methods 290
QueueReceiver 292
  Methods 292
QueueRequestor 293
  Constructors. 293

Methods 293
QueueSender 295
Methods 295
QueueSession 298
Methods 298
Session 301
Fields 301
Methods 301
StreamMessage 306
Methods 306
TemporaryQueue 314
Methods 314
TemporaryTopic 315
MQSeries constructor 315
Methods 315
TextMessage 316
Methods 316
Topic 317
MQSeries constructor 317
Methods 317
TopicConnection 319
Methods 319
TopicConnectionFactory 321
MQSeries constructor 321
Methods 321
TopicPublisher 324
Methods 324
TopicRequestor 327
Constructors 327
Methods 327
TopicSession 329
MQSeries constructor 329
Methods 329
TopicSubscriber 333
Methods 333
| XAConnection 334
| XAConnectionFactory 335
| XAQueueConnection 336
| Methods 336
| XAQueueConnectionFactory 337
| Methods 337
| XAQueueSession 339
| Methods 339
| XASession 340
| Methods 340
| XATopicConnection 342
| Methods 342
| XATopicConnectionFactory 344
| Methods 344
| XATopicSession 346
| Methods 346

Part 4. Appendixes 347

Appendix A. Mapping between Administration tool properties and programmable properties 349

Appendix B. Scripts provided with MQSeries classes for Java Message Service 351

Appendix C. LDAP server configuration for Java objects 353
Checking your LDAP server configuration . . . 353
Configuration procedures 353

Appendix D. Connecting to MQSeries Integrator V2. 355
Publish/subscribe 355
| Transformation and routing 356

| **Appendix E. JMS JTA/XA interface**
| **with WebSphere 357**
| Using the JMS interface with WebSphere 357
| Administered objects 357
| Container-managed versus bean-managed
| transactions 358
| Two-phase commit versus one-phase
| optimization 358
| Defining administered objects 358
| Retrieving administration objects 358
| Samples 358
| Sample1 359
| Sample2 360
| Sample3 360

Appendix F. Notices 363
Trademarks 364

Glossary of terms and abbreviations 365

Bibliography 369
MQSeries cross-platform publications 369
MQSeries platform-specific publications 369
Softcopy books 370
HTML format 370
Portable Document Format (PDF) 370
BookManager® format 371
PostScript format 371
Windows Help format 371
MQSeries information available on the Internet . . 371

Index 373

Sending your comments to IBM . . . 379

Figures

1. MQSeries classes for Java example applet 53
2. MQSeries classes for Java example application 56
3. Topic name hierarchy 182
4. JMS to MQSeries mapping model 195
5. JMS to MQSeries mapping model 206
6. ServerSessionPool and ServerSession functionality 216
7. MQSeries Integrator message flow 355

Tables

| 1. Platforms and connection modes 5
| 2. Product installation directories 10
| 3. Sample CLASSPATH statements for the product 10
| 4. Environment variables for the product . . . 11
| 5. Classes that are tested by IVT 24
6. Administration verbs 34
7. Syntax and description of commands used to manipulate subcontexts 35
8. The JMS object types that are handled by the administration tool 35
9. Syntax and description of commands used to manipulate administered objects 36
10. Property names and valid values 38
| 11. The valid combinations of property and object type 40
12. Core classes restrictions and variations . . . 74
13. Character set identifiers 105
14. Set methods on MQQueueConnectionFactory 171
15. Property names for queue URIs 174
16. Symbolic values for queue properties 175
17. MQRFH2 folders and properties used by JMS 197
18. Property datatypes and values 198
19. JMS properties mapping to MQMD fields 199
20. Outgoing message field mapping 200
21. Incoming message field mapping 204
| 22. Load1 parameters and defaults 220
| 23. ASFClient1 parameters and defaults 221
| 24. TopicLoad parameters and defaults 223
| 25. ASFClient3 parameters and defaults 224
26. Interface Summary. 227
27. Class Summary 229
28. Package 'com.ibm.mq.jms' class Summary 230
29. Package 'com.ibm.jms' class summary 231
30. Comparison of representations of properties within the administration tool, and the programmable equivalents. 349
31. Utilities supplied with MQSeries classes for Java Message Service 351

About this book

This book describes:

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

Note: This documentation is available in softcopy format only (PDF and HTML) as part of the product, and from the MQSeries family Web site at:

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

It **cannot** be ordered as a printed 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 combined
MQ base Java	MQSeries classes for Java
MQ JMS	MQSeries classes for Java Message Service

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 *MQSeries 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 the chapter about the Message Queue Interface in the *MQSeries Application Programming Guide* and the chapter about 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:

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

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

How to use this book

Part 1 of this book describes the use of MQ base 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.

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.

Summary of changes

This section describes changes in 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 to this edition (SC34-5456-06)

This edition includes updates for the new function introduced by MQ Java V5.2. This includes:

- Updates to the installation procedures. See “Chapter 2. Installation procedures” on page 7.
- Support for connection pooling, which can improve performance for applications and middleware that use multiple connections to MQSeries queue managers. See:
  - “Connection pooling” on page 64
  - “MQEnvironment” on page 88
  - “MQPoolServices” on page 123
  - “MQPoolServicesEvent” on page 124
  - “MQPoolToken” on page 126
  - “MQQueueManager” on page 140
  - “MQSimpleConnectionManager” on page 150
  - “MQConnectionManager” on page 154
  - “MQPoolServicesEventListener” on page 153
  - “ManagedConnection” on page 161
  - “ManagedConnectionFactory” on page 164
  - “ManagedConnectionMetaData” on page 166
- New subscriber queue configuration options to provide both a multiple queue and a shared queue approach for publish/subscribe applications. See:
  - “Properties” on page 38
  - “Configuring the base subscriber queue” on page 185
  - “Topic” on page 317
  - “TopicConnectionFactory” on page 321
- A new subscriber cleanup utility, to avoid any problems that result from the non-graceful closure of subscriber objects. See “Subscriber cleanup utility” on page 188.
- Support for Application Server Facilities, that is the concurrent processing of messages. See:
  - “Chapter 13. MQ JMS Application Server Facilities” on page 209
  - “ConnectionConsumer” on page 243
  - “QueueConnection” on page 288
  - “Session” on page 301
  - “TopicConnection” on page 319
- Updates to LDAP server configuration information. See “Appendix C. LDAP server configuration for Java objects” on page 353.

- Support for distributed transactions using the X/Open XA protocol. That is, MQ JMS includes XA classes so that MQ JMS can participate in a two-phase commit that is coordinated by an appropriate transaction manager. See:
  - “Appendix E. JMS JTA/XA interface with WebSphere” on page 357
  - “XAConnection” on page 334
  - “XAConnectionFactory” on page 335
  - “XAQueueConnection” on page 336
  - “XAQueueConnectionFactory” on page 337
  - “XAQueueSession” on page 339
  - “XASession” on page 340
  - “XATopicConnection” on page 342
  - “XATopicConnectionFactory” on page 344
  - “XATopicSession” on page 346

Changes to the sixth edition (SC34-5456-05)

Support for Linux included.

Changes to the fifth edition (SC34-5456-04)

Support for WebSphere™ and MQSeries Integrator V2

MQ base Java version 5.1.2 is now available as a Product Extension. It provides the ability to:

- Connect to MQSeries Integrator for Windows NT®, Version 2.0 to provide Publish/Subscribe support. See “Appendix D. Connecting to MQSeries Integrator V2” on page 355 for further details.
- Use WebSphere’s CosNaming JNDI service provider. See “Configuration” on page 32 for further details.

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? 3
Who should use MQ Java? 3
Connection options 4
Client connection 5
Using VisiBroker for Java 6
Bindings connection 6
Prerequisites 6

| **Chapter 2. Installation procedures** 7
| Installing MQSeries classes for Java and MQSeries
| classes for Java Message Service 7
| Installing on UNIX 8
| Installing on AS/400. 9
| Installing on Linux 9
| Installing on Windows. 10
| Installation directories 10
| Environment variables. 10
| Web server configuration 12

Chapter 3. Using MQSeries classes for Java (MQ base Java) 13
Using the sample applet to verify the TCP/IP client 13
Using the sample applet on AS/400 13
Configuring your queue manager to accept client connections 13
TCP/IP client. 13
Running from appletviewer 14
Customizing the verification applet 15
Verifying with the sample application 15
Using VisiBroker connectivity 16
Using CICS Transaction Server for OS/390 . . . 16
Running your own MQ base Java programs . . . 16
Solving MQ base Java problems 17
Tracing the sample applet 17
Tracing the sample application 17
Tracing with CICS Transaction Server for OS/390. 17
Error messages 18

Chapter 4. Using MQSeries classes for Java Message Service (MQ JMS) 19
Post installation setup 19
Additional setup for Publish/Subscribe mode . . 20
| Queues that require authorization for
| non-privileged users 20
Running the point-to-point IVT. 21
Point-to-point verification without JNDI 21
Point-to-point verification with JNDI 22
IVT error recovery 24
The Publish/Subscribe Installation Verification Test 24
Publish/Subscribe verification without JNDI . . 25
Publish/Subscribe verification with JNDI . . . 26
PSIVT error recovery 27
Running your own MQ JMS programs 27
Solving problems 28
Tracing programs 28
Logging 28

Chapter 5. Using the MQ JMS administration tool 31
Invoking the Administration tool 31
Configuration 32
Configuring for WebSphere 33
Security 33
Administration commands 34
Manipulating subcontexts 35
Administering JMS objects 35
Object types 35
Verbs used with JMS objects. 36
Creating objects 37
LDAP naming considerations 37
Properties 38
Property dependencies 41
The ENCODING property 42
Sample error conditions 43

Chapter 1. Getting started

This chapter gives an overview of MQSeries classes for Java and MQSeries classes for Java Message Service, 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
- Connect directly to an MQSeries server

It enables Java applets, applications, and servlets to issue calls and queries to MQSeries. This gives 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?

MQSeries classes for Java Message Service (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.

The use of MQ JMS as the API to write MQSeries applications has a number of benefits. Some advantages derive from JMS being an open standard with multiple implementations. Other advantages result from additional features that are present in 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 is 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 publish/subscribe 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:

- 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. They can also add value with the 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 provide an excellent infrastructure for access to enterprise applications and for development of complex Web applications. A service request from a Web browser can be queued then processed when possible, thus allowing a timely response to be sent to the end user, regardless of system loading. By placing this queue ’close’ to the user in network terms, the load on the network does not impact the timeliness of the response. Also, the transactional nature of 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 runtime environment. These factors combine to reduce the development time for multi-platform MQSeries applications significantly. Also, if there are enhancements to applets in the future, end users automatically pick these up as the applet code is downloaded.

Connection options

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

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

MQ base Java on Windows NT can also connect using VisiBroker for Java. Table 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
	Standard	VisiBroker	
Windows NT	yes	yes	yes
Windows® 2000	yes	no	yes
AIX®	yes	no	yes
Sun OS (v4.1.4 and earlier)	yes	no	no
Sun Solaris (v2.6, v2.8, V7, or SunOS v5.6, v5.7)	yes	no	yes
OS/2®	yes	no	yes
OS/400®	yes	no	yes
HP-UX	yes	no	yes
AT&T GIS UNIX	yes	no	no
SINIX and DC/OSx	yes	no	no
OS/390	no	no	yes
Linux	yes	no	no

Notes:

1. HP-UX Java bindings support is available only 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.
2. On HP-UXv10.20, Linux, Windows 95, and Windows 98, only TCP/IP client connectivity is supported.

The following sections describe these options in more detail.

Client connection

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

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

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

On the Windows platform, connection using VisiBroker is provided 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 Java Native Interface (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 that are 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

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

- 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.
- For OS/400, the AS/400® Developer Kit for Java, 5769-JV1, and the Qshell Interpreter, OS/400 (5769-SS1) Option 30.

To use the MQ JMS administration tool (see “Chapter 5. Using the MQ JMS administration tool” on page 31), you require the following additional software:

- 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 Java Naming and Directory Service (JNDI) service provider. This is the resource that stores physical representations of the administered objects. Users of MQ JMS will probably use an LDAP server for this purpose, but the tool also supports the use of the file system context service provider. If an LDAP server is used, it must be configured to store JMS objects. For information to assist with this configuration, refer to “Appendix C. LDAP server configuration for Java objects” on page 353.

| To use the XOpen/XA facilities of MQ JMS, you require MQSeries V5.2.

Chapter 2. Installation procedures

This chapter describes how to install the MQSeries classes for Java and MQSeries classes for Java Message Service product.

Installing MQSeries classes for Java and MQSeries classes for Java Message Service

This product is available for the AIX, AS/400, HP-UX, Linux, Sun Solaris, and Windows platforms. It contains:

- MQSeries classes for Java (MQ base Java) Version 5.2.0
- MQSeries classes for Java Message Service (MQ JMS) Version 5.2 (not AS/400)

For the connectivity available on each specific platform, refer to "Connection options" on page 4.

The product is supplied as compressed format files that are available from the MQSeries Web site, `http://www.ibm.com/software/mqseries/`.

Note: For OS/390, MQ base Java is supplied as an MQSeries SupportPac™ that can be downloaded from `http://www.ibm.com/software/mqseries/`.

For the latest versions of just the MQ base Java classes, you can install MQ base Java Version 5.2.0 alone. To use MQ JMS applications, you must install both MQ base Java and MQ JMS (together known as MQ Java).

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

com.ibm.mq.jar	This code includes support for all the connection options.
com.ibm.mq.iiop.jar	This code supports only the VisiBroker connection. It is supplied only on the Windows platform.
com.ibm.mqbind.jar	This code supports only the bindings connection and is not supplied or supported on all platforms. We recommend that you do not use it in any new applications.

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

com.ibm.mqjms.jar

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

connector.jar	Version 1.0 Public Draft
fscontext.jar	Early Access 4 Release
jms.jar	Version 1.0.2
jndi.jar	Version 1.1.2
ldap.jar	Version 1.0.3
providerutil.jar	Version 1.0

Installing MQ base Java and MQ JMS

Note: 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.

For installation instructions, see the section that is relevant to the platform you require:

AIX, HP-UX, and Sun Solaris	"Installing on UNIX"
AS/400	"Installing on AS/400" on page 9
Linux	"Installing on Linux" on page 9
Windows	"Installing on Windows" on page 10

When installation is complete, files and samples are installed in the locations shown in "Installation directories" on page 10.

After installation, you must update your environment variables, as shown in "Environment variables" on page 10.

Note: Take care if you install the product, then subsequently install or reinstall base MQSeries. Make sure that you do not install MQ base Java version 5.1, because your MQSeries Java support will revert back a level.

Installing on UNIX

This section describes how to install MQ Java on AIX, HP-UX, and Sun Solaris. For information about installing MQ base Java on Linux, see "Installing on Linux" on page 9.

Note: If this is a client-only installation (that is, an MQSeries server is NOT installed), you must set up the group and user ID mqm. For more information, please see the MQSeries Quick Beginnings manual relevant to your platform.

1. Log on as root.
2. Copy the file `ma88_`*`xxx`*`.tar.Z` in binary format, and store it in the directory /tmp, where *xxx* is the appropriate platform identifier:
   - `aix` AIX
   - `hp10` HP-UXv10
   - `hp11` HP-UXv11
   - `sol` Sun Solaris
3. Enter the following commands (where *xxx* is the appropriate platform identifier):
   ```
   uncompress -fv /tmp/ma88_xxx.tar.Z
   tar -xvf /tmp/ma88_xxx.tar
   rm /tmp/ma88_xxx.tar
   ```
   These commands create the necessary files and directories.
4. Use the appropriate installation tool for each platform:
   - For AIX, use `smitty` and:
     a. Uninstall all components that begin with mqm.java.
     b. Install components from the /tmp directory.
   - For HP-UX, use `sam` and install from the file `ma88_hp10` or `ma88_hp11`, as appropriate.

Note: Java does not support code page 1051 (which is the default for HP-UX). To run the Publish/Subscribe broker on HP-UX, you may need to change the CCSID of the broker's queue manager to an alternative value, for example 819.

- For Sun Solaris, enter the following command and select the options you require:

  ```
  pkgadd -d /tmp mqjava
  ```

  Then, enter the following command:

  ```
  rm -R /tmp/mqjava
  ```

Installing on AS/400

This section describes how to install MQ base Java on AS/400.

1. Copy the file `ma88_400.zip` to a directory on your PC.
2. Uncompress the file using InfoZip's Unzip facility.

   This creates the file `ma88_400.sav`.
3. Create a save file called MA88 in a suitable library on the AS/400, for example the QGPL library:

   ```
   CRTSAVF FILE(QGPL/MA88)
   ```
4. Transfer `ma88_400.sav` into this save file as a binary image. If you use FTP to do this, the put command should be similar to:

   ```
   PUT C:\TEMP\MA88_400.SAV QGPL/MA88
   ```
5. Install MQSeries classes for Java, product Id 5648C60, using RSTLICPGM:

   ```
   RSTLICPGM LICPGM(5648C60) DEV(*SAVF) SAVF(QGPL/MA88)
   ```
6. Delete the save file created in Step 3:

   ```
   DLTF FILE(QGPL/MA88)
   ```

Installing on Linux

This section describes how to install MQ Java on Linux.

For Linux, there are two installation files available, `ma88_linux.tgz` and MQSeriesJava-5.2.0-1.noarch.rpm. Each file provides an identical installation.

If you have root access to the target system, or use a Red Hat Package Manager (RPM) database to install packages, use MQSeriesJava-5.2.0-1.noarch.rpm.

If you do not have root access to the target system, or the target system does not have RPM installed, use `ma88_linux.tgz`.

To install using ma88_linux.tgz:

1. Select an installation directory for the product (for example, /opt).

   If this directory is not in your home directory, you may need to log in as root.
2. Copy the file `ma88_linux.tgz` into your home directory.
3. Change directory to your selected installation directory, for example:

   ```
   cd /opt
   ```
4. Enter the following command:

   ```
   tar -xpzf ~/ma88_linux.tgz
   ```

   This creates and populates a directory named mqm in the current directory (for example, /opt).

Installing on Linux

To install using MQSeriesJava-5.2.0-1.noarch.rpm:

1. Log in as root.
2. Copy MQSeriesJava-5.2.0-1.noarch.rpm into a working directory.
3. Enter the following command:

   ```
   rpm -i MQSeriesJava-5.2.0-1.noarch.rpm
   ```

   This installs the product to /opt/mqm/. It is possible to install to a different path (please refer to your RPM documentation for further details).

Installing on Windows

This section describes how to install MQ Java on Windows.

1. Create an empty directory called tmp and make it the current directory.
2. Copy the file `ma88_win.zip` to this directory.
3. Uncompress `ma88_win.zip` using InfoZip's Unzip facility.
4. Run `setup.exe` from this directory and follow the prompts on the resulting windows.

   Note: If you wish to install MQ base Java only, select the relevant options at this stage.

Installation directories

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

Table 2. Product installation directories

Platform	Directory
AIX	usr/mqm/java/
AS/400	/QIBM/ProdData/mqm/java/
HP-UX and Sun Solaris	opt/mqm/java/
Linux	*install_dir*/mqm/java/
Windows 95, 98, 2000, and NT	*install_dir*\
Note: *install_dir* is the directory in which you installed the product. On Linux, this is likely to be /opt.	

Environment variables

After installation, you must update your CLASSPATH environment variable to include the MQ base Java code and samples directories. Table 3 shows typical CLASSPATH settings for the various platforms.

Table 3. Sample CLASSPATH statements for the product

Platform	Sample CLASSPATH
AIX	CLASSPATH=*jdk_dir*/lib/classes.zip: /usr/mqm/java/lib/com.ibm.mq.jar: /usr/mqm/java/lib/connector.jar: /usr/mqm/java/lib: /usr/mqm/java/samples/base:
HP-UX and Sun Solaris	CLASSPATH=*jdk_dir*/lib/classes.zip: /opt/mqm/java/lib/com.ibm.mq.jar: /opt/mqm/java/lib/connector.jar: /opt/mqm/java/lib: /opt/mqm/java/samples/base:

Table 3. Sample CLASSPATH statements for the product (continued)

Platform	Sample CLASSPATH

Windows 95, 98, 2000, and NT	CLASSPATH=C:*jdk_dir*\lib\classes.zip;
	install_dir\lib\com.ibm.mq.jar;
	install_dir\lib\com.ibm.mq.iiop.jar;
	install_dir\lib\connector.jar;
	install_dir\lib\;
	install_dir\samples\base\;

AS/400	CLASSPATH=/QIBM/ProdData/mqm/java/lib/com.ibm.mq.jar: /QIBM/ProdData/mqm/java/lib/connector.jar: /QIBM/ProdData/mqm/java/lib: /QIBM/ProdData/mqm/java/samples/base:
Linux	CLASSPATH=*jdk_dir*/lib/classes.zip: *install_dir*/mqm/java/lib/com.ibm.mq.jar: *install_dir*/mqm/java/lib/connector.jar: *install_dir*/mqm/java/lib: *install_dir*/mqm/java/samples/base:
Notes: 1. *jdk_dir* is the directory in which the JDK is installed 2. *install_dir* is the directory in which you installed the product	

To use MQ JMS, you must include additional jar files in the classpath. These are listed in “Post installation setup” on page 19.

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

You must update additional environment variables on some platforms, as shown in Table 4.

Table 4. Environment variables for the product

Platform	Environment variable
AIX	LD_LIBRARY_PATH=/usr/mqm/java/lib
HP_UX	SHLIB_PATH=/opt/mqm/java/lib
Sun Solaris	LD_LIBRARY_PATH=/opt/mqm/java/lib
Windows 95, 98, 2000, and NT	PATH=*install_dir*\lib
Note: *install_dir* is the installation directory for the product	

Notes:

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

Web server configuration

If you install 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 MQSeries classes for Java (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
- 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

MQ base Java includes an installation verification applet, `mqjavac.html`. You can use the applet to verify the TCP/IP connected client mode of MQ base Java. (See also "Verifying with the sample application" on page 15.)

The applet connects to a given queue manager, exercises all the MQSeries calls, and produces diagnostic messages if there are any failures.

You can run the applet from the applet viewer supplied with your JDK. The applet viewer can access a queue manager on any host.

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.

Using the sample applet on AS/400

The OS/400 operating system does not have a native Graphical User Interface (GUI). To run the sample applet, you need to use the Remote Abstract Window Toolkit for Java (AWT), or the Class Broker for Java (CBJ), on graphics capable hardware. You can also verify the client from the command line (see "Verifying with the sample application" on page 15).

Configuring your queue manager to accept client connections

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

TCP/IP client

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

   For platforms other than AS/400:

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

Verifying client mode

For the AS/400 platform:

a. Start your queue manager by using the STRMQM command.

b. Define a sample channel called JAVA.CHANNEL by typing:

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

where `QMGRNAME` is the name of your queue manager, and `SOMEUSERID` is an AS/400 user id with appropriate authority to the MQSeries resources.

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, you can omit the -m option.

Using VisiBroker for Java on the Windows NT operating system:
Start the IIOP (Internet Inter-ORB Protocol) 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.

For the OS/400 operating system:
Issue the command:

```
STRMQMLSR MQMNAME(QMGRNAME)
```

where `QMGRNAME` is the name of your queue manager.

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’.

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

Customizing the verification applet

The `mqjavac.html` file includes some optional parameters. 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. You can specify the following parameters:

hostname	The value to display initially in the hostname edit box.
port	The value to display initially in the port number edit box.
channel	The value to display initially in the channel edit box.
queueManager	The value to display initially in the queue manager edit box.
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 other data to determine which connection mode you want to verify. Use the following procedure to verify your installation:

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

   To test a bindings connection, carry out the rest of this procedure on the 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.
4. At prompt (1), leave the default ’MQSeries’.
5. At prompt (2):
   - To use a TCP/IP connection, enter an MQSeries server hostname.
   - To use native connection (bindings mode), leave the field blank. (Do not enter a name.)

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

Installation verification program

```
Please enter the type of connection (MQSeries)              : (MQSeries)(1)
Please enter the IP address of the MQSeries server          : myhost(2)
Please enter the port to connect to                         : (1414)(3)
Please enter the server connection channel name             : JAVA.CHANNEL(3)
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...
```

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).
3. On OS/400, you can run the command `java MQIVP` only from the Qshell interactive interface (the Qshell is option 30 of OS/400, 5769-SS1). Alternatively, you can run the application by using the CL command `RUNJVA CLASS(MQIVP)`.
4. To use the MQSeries bindings for Java on OS/400, you must ensure that the library QMQMJAVA is in your library list.

Using VisiBroker connectivity

If you are using VisiBroker, the procedures described in "Configuring your queue manager to accept client connections" on page 13 are not required.

To test an installation that is using VisiBroker, use the procedures described in "Verifying with the sample application" on page 15, 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. Run the transaction.

The program output is placed in the files used for standard and error output.

Refer to CICS documentation for more information on running Java programs and setting the input and output files.

Running your own MQ base Java programs

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

For information on writing MQ base Java applications and applets, see "Part 2. Programming with MQ base Java" on page 45.

Solving MQ base Java problems

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

If the problems continue and you need to contact the IBM service team, you may be asked to turn on the trace facility. The method to do this depends on whether you are running in client mode or bindings mode. Refer to the following sections for the appropriate procedures for your system.

Tracing the sample applet

To run trace with the sample applet, edit the `mqjavac.html` file. Find the following 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 greater the number, the more information is 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 greater the number, the more information is gathered.)

| For more information about how to use trace, see “Tracing MQ base Java
| programs” on page 70.

Tracing with CICS Transaction Server for OS/390

When you use CICS Transaction Server for OS/390, it is not possible to supply command line arguments directly to the program. You need to write a small wrapper program that 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 the most recent version of MQ base Java 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 attempt to use VisiBroker, probably 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 MQSeries classes for Java Message Service (MQ JMS)

This chapter describes the following tasks:

- How to set up your system to use the Test and sample programs
- How to run the point-to-point Installation Verification Test (IVT) program to verify your MQSeries classes for Java Message Service installation
- How to run the sample Publish/Subscribe Installation Verification Test (PSIVT) program to verify your Publish/Subscribe installation
- How to run your own programs

Post installation setup

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

Classpath

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

Add the following .jar files to the classpath:

- com.ibm.mq.jar
- com.ibm.mqjms.jar
- connector.jar
- jms.jar
- jndi.jar
- jta.jar
- ldap.jar
- providerutil.jar

Environment variables

There are a number of scripts in the `bin` subdirectory of the MQ JMS installation. These are intended for use as convenient shortcuts for a number of common actions. Many of these scripts assume that the environment variable `MQ_JAVA_INSTALL_PATH` is defined, and that it points to the directory in which MQ JMS is installed. It is not mandatory to set this variable, but if you do not set it, you must edit the scripts in the `bin` directory accordingly.

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

Additional setup for Publish/Subscribe mode

Before you can use the MQ JMS implementation of JMS Publish/Subscribe, some additional setup is required:

Ensure that the Broker is Running

To verify that the MQSeries Publish/Subscribe broker is installed and is running, use the command:

```
dspmqbrk -m MY.QUEUE.MANAGER
```

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

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

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

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

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

Create the MQ JMS System Queues

For the MQ JMS Publish/Subscribe implementation to work correctly, a number of system queues must be created. A script is supplied, 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 typed the queue manager name correctly, and check that the queue manager is running.

Queues that require authorization for non-privileged users

Non-privileged users need authorization granted to access the queues used by JMS. For details about access control in MQSeries, see the chapter about protecting MQSeries objects in *MQSeries System Administration*.

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

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

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

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

SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE
SYSTEM.BROKER.CONTROL.QUEUE

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

SYSTEM.BROKER.DEFAULT.STREAM

Running the point-to-point IVT

This section describes 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.

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

With JNDI lookup of administered objects
JNDI mode forces the program to obtain its administered objects from a JNDI namespace, which is the expected operation of JMS client applications. (See "Administering JMS objects" on page 35 for a description of administered objects). This invocation method has the same prerequisites as the administration tool (see "Chapter 5. Using the MQ JMS administration tool" on page 31).

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

Point-to-point verification without JNDI

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

To run the test without JNDI, issue the following command:

```
IVTRun -nojndi
```

For client mode, to run the test without JNDI, issue the following command:

```
IVTRun -nojndi -client -m <qmgr> -host <hostname> [-port <port>]
             [-channel <channel>]
```

where:

qmgr	is the name of the queue manager to which you wish to connect
hostname	is the host on which the queue manager is running
port	is the TCP/IP port on which the queue manager's listener is running (default 1414)
channel	is the client connection channel (default SYSTEM.DEF.SVRCONN)

Point-to-point IVT

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

```
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
d3120202020202020202020203000c43713400000
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

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

```
Unable to bind to object
```

This message means that either the server is not storing Java objects, or the permissions on the objects or the suffix are not correct. See "Checking your LDAP server configuration" on page 353.

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

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

```
IVTSetup
```

The script invokes the MQ JMS Administration tool (see "Chapter 5. Using the MQ JMS administration tool" on page 31) 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 value of the `QUEUE` property becomes `QUEUE(SYSTEM.DEFAULT.LOCAL.QUEUE)`. All other properties have default values:

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

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

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

where:

-t
means turn tracing on (by default, tracing is off)

providerURL
is the JNDI location of the administered objects. If the default initial context factory is in use, this is an LDAP URL of the form:

```
ldap://hostname.company.com/contextName
```

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

```
file://directorySpec
```

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

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

```
com.sun.jndi.ldap.LdapCtxFactory
```

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

```
com.sun.jndi.fscontext.RefFSContextFactory
```

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

```
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
```

Point-to-point IVT

```
Retrieving a Queue from JNDI
Creating a QueueSender
 ...
 ...
```

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

IVT error recovery

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

- For assistance with any error messages involving classpath, check that your classpath is set correctly, as described in "Post installation setup" on page 19.
- The IVT might fail with a message 'failed to create MQQueueManager', with an additional message including the number 2059. This indicates that MQSeries failed to connect to the default local queue manager on the machine on which you ran the IVT. Check that the queue manager is running, and that it is marked as the default queue manager.
- A message of 'failed to open MQ queue' indicates that MQSeries connected to the default queue manager, but could not open the 'SYSTEM.DEFAULT.LOCAL.QUEUE'. This may indicate that either the queue does not exist on your default queue manager, or the queue is not enabled for PUT and GET. Add or enable the queue for the duration of the test.

Table 5 lists the classes that are tested by IVT, and the package that they come from:

Table 5. Classes that are tested by IVT

Class	Jar file
MQSeries JMS classes	com.ibm.mqjms.jar
com.ibm.mq.MQMessage	com.ibm.mq.jar
javax.jms.Message	jms.jar
javax.naming.InitialContext	jndi.jar
javax.resource.cci.Connection	connector.jar
javax.transaction.xa.XAException	jta.jar
com/sun/jndi/toolkit/ComponentDirContext	providerutil.jar
com.sun.jndi.ldap.LdapCtxFactory	ldap.jar

The Publish/Subscribe Installation Verification Test

The Publish/Subscribe Installation Verification Test (PSIVT) program is supplied only in compiled form. It is in the `com.ibm.mq.jms` package.

The PSIVT attempts to:

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

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

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

Publish/Subscribe verification without JNDI

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

To run the test without JNDI, issue the following command:

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

| For client mode, to run the test without JNDI, issue the following command:

```
| PSIVTRun -nojndi -client -m <qmgr> -host <hostname> [-port <port>]
|                [-channel <channel>] [-t]
```

| where:

-nojndi	means no JNDI lookup of the administered objects
qmgr	| is the name of the queue manager to which you wish to connect
hostname	| is the host on which the queue manager is running
port	| is the TCP/IP port on which the queue manager's listener is running (default 1414)
channel	| is the client connection channel (default SYSTEM.DEF.SVRCONN)
-t	means turn tracing on (default is off)

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

```
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:414d51580000000000000000000000000000000000000000
  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
```

Publish/Subscribe verification with JNDI

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

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

You can define these objects by using the MQ JMS Administration Tool (see "Chapter 5. Using the MQ JMS administration tool" on page 31) and using the following commands:

```
DEFINE TCF(ivtTCF)
```

This command defines the TopicConnectionFactory.

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

This command defines the Topic.

These definitions assume that a default queue manager, on which the broker is running, is available. For details on configuring these objects to use a non-default queue manager, see "Administering JMS objects" on page 35. These objects should reside in a context pointed to by the `-url` command-line parameter described below.

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

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

where:

-t	means turn tracing on (by default, tracing is off)
-url <purl>	is the URL of the JNDI location in which the administered objects reside
-icf <initcf>	is the initialContextFactory for JNDI `[com.sun.jndi.ldap.LdapCtxFactory]`

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

PSIVT error recovery

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

- If you see the following message:
  ```
  *** 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 some outstanding messages. This indicates that the broker is not running. To start it, use the `strmqbrk` command. (See "Additional setup for Publish/Subscribe mode" on page 20.)
- 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 that the PSIVT uses is configured with a valid queue manager name. Alternatively, if you used the `-nojndi` option, ensure that you supplied a valid queue manager (use the `-m` option).
- If the following message is displayed:
  ```
  Unable to access broker control queue on queue manager: ...
  Please ensure the broker is installed on this queue manager
  ```
  ensure that the administered TopicConnectionFactory that the PSIVT uses is configured with the name of the queue manager on which the broker is installed. If you used the `-nojndi` option, ensure that you supplied a queue manager name (use the `-m` option).

Running your own MQ JMS programs

For information on writing your own MQ JMS programs, see "Chapter 10. Writing MQ JMS programs" on page 169.

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

The utility provides default locations for the trace and log files, and enables you to add any application runtime parameters that your application needs. The supplied script assumes that the environment variable MQ_JAVA_INSTALL_PATH is set to the directory in which the MQ JMS is installed. The script also assumes that the subdirectories `trace` and `log` within that directory are used for trace and log output, respectively. These are only suggested locations, and you can edit the script 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 167.

Solving problems

If a program does not complete successfully, run the installation verification program, which is described in “Chapter 4. Using MQSeries classes for Java Message Service (MQ JMS)” on page 19, and follow the advice given in the diagnostic messages.

Tracing programs

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

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

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

on	traces 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
```

To disable trace, set MQJMS_TRACE_LEVEL to **off**.

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

For example:

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

The `runjms` utility script sets these properties by using the environment variables `MQJMS_TRACE_LEVEL` and `MQ_JAVA_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 you can modify it 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.

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

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

The utility script `runjms`, in the bin directory of the MQ JMSinstallation, 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, which you can modify as required.

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

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

Logging

Chapter 5. Using the MQ JMS administration tool

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

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

For details about these objects, refer to “Administering JMS objects” on page 35.

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

The tool also allows administrators to manipulate directory namespace subcontexts within the JNDI. See “Manipulating subcontexts” on page 35.

Invoking the Administration tool

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

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

```
JMSAdmin [-t] [-v] [-cfg config_filename]
```

where:

-t	Enables trace (default is trace off)
-v	Produces verbose output (default is terse output)
-cfg config_filename	The name of an alternative configuration file (see “Configuration” on page 32)

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

```
InitCtx>
```

indicating that the current context (that is, the JNDI context to which all naming and directory operations currently refer) is the initial context defined in the PROVIDER_URL configuration parameter (see “Configuration” on page 32).

Invoking the Administration tool

As you traverse the directory namespace, the prompt changes to reflect this, so that the prompt always displays the current context.

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

```
JMSAdmin <test.scp
```

where *test.scp* is a script file that contains administration commands (see “Administration commands” on page 34). The last command in the file must be the `END` command.

Configuration

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

INITIAL_CONTEXT_FACTORY
This indicates the service provider that the tool uses. There are currently three supported values for this property:
- com.sun.jndi.ldap.LdapCtxFactory (for LDAP)
- com.sun.jndi.fscontext.RefFSContextFactory (for file system context)
- com.ibm.ejs.ns.jndi.CNInitialContextFactory (to work with WebSphere’s CosNaming repository)

PROVIDER_URL
This indicates the URL of the session’s initial context, the root of all JNDI operations carried out by the tool. Three forms of this property are currently supported:
- ldap://hostname/contextname (for LDAP)
- file:[drive:]/pathname (for file system context)
- iiop://hostname[:port] /[?TargetContext=ctx] (to access “base” WebSphere CosNaming namespace)

SECURITY_AUTHENTICATION
This indicates whether JNDI passes over security credentials to your service provider. This parameter is used only when an LDAP service provider is used. This property can currently take one of three values:
- none (anonymous authentication)
- simple (simple authentication)
- CRAM-MD5 (CRAM-MD5 authentication mechanism)

If a valid value is not supplied, the property defaults to none. See “Security” on page 33 for more details about security with the administration tool.

These parameters are set in a configuration file. When you invoke the tool, you can specify this configuration by using the `-cfg` command-line parameter, as described in “Invoking the Administration tool” on page 31. If you do not specify a configuration file name, the tool attempts to load the default configuration file (`JMSAdmin.config`). It looks for this file first in the current directory, and then in the `<MQ_JAVA_INSTALL_PATH>/bin` directory. (Where <MQ_JAVA_INSTALL_PATH> is the path to your MQ JMS installation.)

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

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

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

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

Configuring for WebSphere

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

- CLASSPATH must include WebSphere's JNDI-related jar files:
  - For WebSphere V3.5:

    `<WSAppserver>\lib\ejs.jar`
- PATH for WebSphere V.3.5 must include:

  `<WSAppserver>\jdk\jre\bin`

  where <WSAppserver> is the install path for WebSphere.

Security

Administrators need to know about the effect of the `SECURITY_AUTHENTICATION` property described in "Configuration" on page 32.

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

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

Note: The text typed is echoed to the screen, and this includes the password. Therefore, take care 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, the tool displays an appropriate error message and terminates.

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

Administration commands

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

```
verb [param]*
```

where *verb* is one of the administration verbs listed in Table 6. 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 that take at least one parameter are discussed in later sections of this chapter.

Table 6. 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

Verb names are not case-sensitive.

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

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

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

Manipulating subcontexts

You can use the verbs CHANGE, DEFINE, DISPLAY and DELETE to manipulate directory namespace subcontexts. Their use is described in Table 7.

Table 7. 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 eight types of object that the administration tool can handle. It includes details about each of their configurable properties and the verbs that can manipulate them.

Object types

| Table 8 shows the eight types of administered objects. The Keyword column shows the strings that you can substitute for *TYPE* in the commands shown in Table 9 on page 36.

Table 8. 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.

Table 8. The JMS object types that are handled by the administration tool (continued)

Object Type		Description
Java	Keyword	
MQTopic	T	The MQSeries implementation of the JMS Topic interface. This represents a destination for messages in the publish/subscribe domain of JMS.
MQXAQueueConnectionFactory[1]	XAQCF	The MQSeries implementation of the JMS XAQueueConnectionFactory interface. This represents a factory object for creating connections in the point-to-point domain of JMS that use the XA versions of JMS classes.
MQXATopicConnectionFactory[1]	XATCF	The MQSeries implementation of the JMS XATopicConnectionFactory interface. This represents a factory object for creating connections in the publish/subscribe domain of JMS that use the XA versions of JMS classes.
JMSWrapXAQueueConnectionFactory[2]	WSQCF	The MQSeries implementation of the JMS QueueConnectionFactory interface. This represents a factory object for creating connections in the point-to-point domain of JMS that use the XA versions of JMS classes with WebSphere.
JMSWrapXATopicConnectionFactory[2]	WSTCF	The MQSeries implementation of the JMS TopicConnectionFactory interface. This represents a factory object for creating connections in the publish/subscribe domain of JMS that use the XA versions of JMS classes with WebSphere.

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

Verbs used with JMS objects

You can use the verbs `ALTER`, `DEFINE`, `DISPLAY`, `DELETE`, `COPY`, and `MOVE` to manipulate administered objects in the directory namespace. Table 9 summarizes their use. Substitute *TYPE* with the keyword that represents the required administered object, as listed in Table 8 on page 35.

Table 9. 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 9. Syntax and description of commands used to manipulate administered objects (continued)

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 38).

LDAP naming considerations

To store your objects in an LDAP environment, their names must comply with certain conventions. One of these is that object and subcontext names must include a prefix, such as `cn=` (common name), or `ou=` (organizational unit).

The administration tool simplifies the use of LDAP service providers by allowing you to refer to object and context names without a prefix. If you do not supply a prefix, the tool automatically adds a default prefix (currently `cn=`) to the name you supply.

This is shown in the following example.

```
InitCtx> DEFINE Q(testQueue)

InitCtx> DISPLAY CTX

    Contents of InitCtx

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

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

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

You may need to configure your LDAP server to store Java objects. Information to assist with this configuration is provided in “Appendix C. LDAP server configuration for Java objects” on page 353.

Properties

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

```
PROPERTY_NAME(property_value)
```

Property names are not case-sensitive, and are restricted to the set of recognized names shown in Table 10. This table also shows the valid property values for each property.

Table 10. 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 10. Property names and valid values (continued)

Property		Valid values (defaults in bold)
Standard	Shortform	
MSGRETENTION	MRET	• **Yes** - Unwanted messages remain on the input queue • No - Unwanted messages are dealt with according to their disposition options
BROKERVER	BVER	**V1** - The only value currently allowed.
BROKERPUBQ	BPUB	Any string (default is **SYSTEM.BROKER.DEFAULT.STREAM**)
BROKERSUBQ	BSUB	Any string (default is **SYSTEM.JMS.ND.SUBSCRIPTION.QUEUE**)
BROKERDURSUBQ	BDSUB	Any string (default is **SYSTEM.JMS.D.SUBSCRIPTION.QUEUE**)
BROKERCCSUBQ	CCSUB	Any string (default is **SYSTEM.JMS.ND.CC.SUBSCRIPTION.QUEUE**)
BROKERCCDSUBQ	CCDSUB	Any string (default is **SYSTEM.JMS.D.CC.SUBSCRIPTION.QUEUE**)
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** - The target of the message is a JMS application. • MQ - The target of the message is a non-JMS, traditional MQSeries application.
ENCODING	ENC	See “The ENCODING property” on page 42
QUEUE	QU	Any string
TOPIC	TOP	Any string

Many of the properties are relevant only to a specific subset of the object types. Table 11 on page 40 shows which property-object type combinations are valid, and gives a brief description of each property.

Table 11. The valid combinations of property and object type

Property	Valid object types						Description
	QCF	TCF	Q	T	WSQCF XAQCF	WSTCF XATCF	
DESCRIPTION	Y	Y	Y	Y	Y	Y	A description of the stored object
TRANSPORT	Y	Y			Y[1]	Y[1]	Whether connections will use the MQ Bindings, or a client connection
CLIENTID	Y	Y			Y	Y	A string identifier for the client
QMANAGER	Y	Y	Y		Y	Y	The name of the queue manager to connect to
PORT	Y	Y					The port on which the queue manager listens
HOSTNAME	Y	Y					The name of the host on which the queue manager resides
CHANNEL	Y	Y					The name of the client connection channel being used
CCSID	Y	Y	Y	Y			The coded-character-set-ID to be used 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				Y		Name of the model queue from which temporary queues are created
MSGRETENTION	Y				Y		Whether or not the connection consumer keeps unwanted messages on the input queue
BROKERVER		Y				Y	The version of the broker being used
BROKERPUBQ		Y				Y	The name of the broker input queue (stream queue)
BROKERSUBQ		Y				Y	The name of the queue from which non-durable subscription messages are retrieved
BROKERDURSUBQ				Y			The name of the queue from which durable subscription messages are retrieved
BROKERCCSUBQ		Y				Y	The name of the queue from which non-durable subscription messages are retrieved for a ConnectionConsumer
BROKERCCDSUBQ				Y			The name of the queue from which durable subscription messages are retrieved for a ConnectionConsumer
BROKERQMGR		Y				Y	The queue manager on which the broker is running

Table 11. The valid combinations of property and object type (continued)

Property	Valid object types						Description
	QCF	TCF	Q	T	WSQCF XAQCF	WSTCF XATCF	
BROKERCONQ		Y				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			Field indicates whether the MQSeries RFH2 format is used to exchange information with target applications
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

Notes:

1. For WSTCF, WSQCF, XATCF, and XAQCF objects, only the BIND transport type is allowed.
2. "Appendix A. Mapping between Administration tool properties and programmable properties" on page 349 shows the relationship between properties set by the tool and programmable properties.
3. The TARGCLIENT property indicates whether the MQSeries RFH2 format is used to exchange information with target applications.

   The MQJMS_CLIENT_JMS_COMPLIANT constant indicates that the RFH2 format is used to send information. Applications that use MQ JMS understand the RFH2 format. You should set the MQJMS_CLIENT_JMS_COMPLIANT constant when you exchange information with a target MQ JMS application.

   The MQJMS_CLIENT_NONJMS_MQ constant indicates that the RFH2 format is not used to send information. Typically, this value is used for an existing MQSeries application (that is, one that does not handle RFH2).

Property dependencies

Some properties have dependencies on each other. This may mean that it is meaningless to supply a property unless another property is set to a particular value. The 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, 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

If you attempt to set any of these properties without setting the `TRANSPORT` property to `CLIENT`, there will be an error.

Exit initialization strings
It is invalid to set any of the exit initialization strings unless the corresponding exit name has been supplied. The exit initialization properties are:
- RECEXITINIT
- SECEXITINIT
- SENDEXITINIT

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

The ENCODING property

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

integer encoding	this is either normal or reversed
decimal encoding	this is either normal or reversed
floating-point encoding	this is IEEE normal, IEEE reversed, or System/390®

The `ENCODING` is expressed as a three-character string with the following syntax:
`{N|R}{N|R}{N|R|3}`

In this string:
- `N` denotes normal
- `R` denotes reversed
- `3` denotes System/390
- the first character represents *integer encoding*
- the second character represents *decimal encoding*
- the third character represents *floating-point encoding*

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

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

The following examples show valid combinations for `ENCODING`:

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

Sample error conditions

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

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 . . . 47
Why should I use the Java interface? 47
The MQSeries classes for Java interface 48
Java Development Kit 48
MQSeries classes for Java class library 49

Chapter 7. Writing MQ base Java programs. . . 51
Should I write applets or applications? 51
Connection differences. 51
  Client connections 51
  Bindings mode 52
  Defining which connection to use 52
Example code fragments 52
  Example applet code 52
    Changing the connection to use VisiBroker for Java 55
  Example application code 56
Operations on queue managers. 58
  Setting up the MQSeries environment 58
  Connecting to a queue manager 58
Accessing queues and processes 59
Handling messages. 60
Handling errors 61
Getting and setting attribute values 61
Multithreaded programs 62
Writing user exits 63
| Connection pooling. 64
| Controlling the default connection pool 64
| The default connection pool and multiple components 66
| Supplying a different connection pool 67
| Supplying your own ConnectionManager . . . 68
Compiling and testing MQ base Java programs . . 69
  Running MQ base Java applets 70
  Running MQ base Java applications 70
  Running MQ base Java applications under CICS Transaction Server for OS/390 70
  Tracing MQ base Java programs 70

Chapter 8. Environment-dependent behavior . . 73
Core details 73
Restrictions and variations for core classes 74
Version 5 extensions operating in other environments. 76

Chapter 9. The MQ base Java classes and interfaces 79
MQChannelDefinition 80
  Variables 80
  Constructors 81
MQChannelExit 82
  Variables 82
  Constructors 84
MQDistributionList. 85
  Constructors 85
  Methods 85
MQDistributionListItem 87
  Variables 87
  Constructors 87
MQEnvironment. 88
  Variables 88
  Constructors 91
  Methods 91
MQException. 93
  Variables 93
  Constructors 93
MQGetMessageOptions 95
  Variables 95
  Constructors 98
MQManagedObject. 99
  Variables 99
  Constructors. 100
  Methods 100
MQMessage 102
  Variables 102
  Constructors. 110
  Methods 110
MQMessageTracker 121
  Variables 121
| MQPoolServices 123
| Constructors. 123
| Methods 123
| MQPoolServicesEvent 124
| Variables 124
| Constructors. 124
| Methods 125
| MQPoolToken 126
| Constructors. 126
MQProcess 127
  Constructors. 127
  Methods 127
MQPutMessageOptions 129
  Variables 129
  Constructors. 131
MQQueue 132
  Constructors. 132
  Methods 132
MQQueueManager 140
  Variables 140
  Constructors. 140
  Methods 142
| MQSimpleConnectionManager 150
| Variables 150
| Constructors. 150
| Methods 150
MQC 152
| MQPoolServicesEventListener 153
| Methods 153
| MQConnectionManager 154
MQReceiveExit 155
  Methods 155

MQSecurityExit 157
  Methods 157
MQSendExit. 159
  Methods 159
| ManagedConnection 161
| Methods 161
| ManagedConnectionFactory 164
| Methods 164
| ManagedConnectionMetaData. 166
| Methods 166

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 51.

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

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

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

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

Advantages of Java

If you place your applet on a Web server that is accessible outside the corporate firewall, anyone on the Internet can download and use your application. This means that you can get messages into your 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.

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

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

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

Java Development Kit

Before you can compile any applets or applications that you write, you must have access to the Java Development 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. It also contains the tools required to compile and run the applets and programs on each supported platform.

| You can download JDKs from 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 by using the JDK that is included with the integrated development environment of IBM Visual Age for Java.

To compile Java applications on the AS/400 platform, you must first install:
- The AS/400 Developer Kit for Java, 5769-JV1
- The Qshell Interpreter, OS/400 (5769-SS1) Option 30

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
- MQPoolServices
- MQPoolServicesEvent
- MQPoolToken
- MQPutMessageOptions
- MQProcess
- MQQueue
- MQQueueManager
- MQSimpleConnectionManager

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

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

In Java, a *package* is a mechanism for grouping sets of related classes together. The 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 use MQSeries classes for Java to access MQSeries queues, you write Java programs that contain calls that put messages onto, and get messages from, MQSeries queues. The programs can take the form of Java *applets*, Java *servlets*, or Java *applications*.

This chapter provides information to assist with writing Java applets, servlets, and applications to interact with MQSeries systems. For details of individual classes, see "Chapter 9. The MQ base Java classes and interfaces" on page 79.

Should I write applets or applications?

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

The main differences between applets and applications are:

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

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. Alternatively, this information can be passed as parameters when the connection is made.
- Error and exception conditions are written to a log specified in the MQException class. The default error destination is the Java console.

The 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.

Connection differences

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 the one that 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

Note: MQSeries for AS/400 does not support the use of MQBEGIN to initiate global units of work that are coordinated by the queue manager.

Defining which connection to use

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

MQEnvironment.properties
This can contain the following key/value pairs:
- For client and bindings connections:
```
MQC.TRANSPORT_PROPERTY, MQC.TRANSPORT_MQSERIES
```
- 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

This section includes two example code fragments; Figure 1 on page 53 and Figure 2 on page 56. Each one uses a particular connection and includes notes to describe the changes needed to use alternative connections.

Example applet code

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

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

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

Example code

```
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);

    // 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);

    // 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?
```

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

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

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 (object request broker):

```
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

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 {

      // 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);

      // 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
```

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

```
     // put the message on the queue

      system_default_local_queue.put(hello_world,pmo);

      // get the message back again...
      // First define a 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. MQSeries classes for Java example application (Part 2 of 2)

Operations on queue managers

This section describes how to connect to, and disconnect from, a queue manager using 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 you use the client connection to connect to a queue manager, 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. This class 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 55.

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();
```

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

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

Accessing queues and processes

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

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

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

The accessQueue method returns a new object of class MQQueue.

When you have finished using the queue, use the close() method to close it, as in the following example:

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

With MQSeries classes for Java, you can also create a queue by using the MQQueue constructor. The parameters are exactly the same as for the accessQueue method, with the addition of a queue manager parameter. For example:

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

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

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

The accessProcess method returns a new object of class MQProcess.

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

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

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

Handling messages

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

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

When the new message instance is created, all the MQMD parameters are automatically set to their default values, as defined in the *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 creates a message and puts 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.

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

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

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

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

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

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

See “MQMessage” on page 102 for more details.

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

Handling errors

Methods in the Java interface do not return a completion code and reason code. Instead, they throw an exception whenever the completion code and reason code resulting from 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 points in your program you want to deal with the possibility of failure. At these points, you can surround your code with ’try’ and ’catch’ blocks, as in the following example:

```
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. These methods allow you to get and set their attribute values. Note that for MQQueue, the methods will work only if you specify the appropriate ’inquire’ and ’set’ flags when you open the queue.

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

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

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

- charAttrs array

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

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

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

selectors[0] = MQIA_DEF_PRIORITY;
selectors[1] = MQCA_Q_DESC;

queue.inquire(selectors,intAttrs,charAttrs);

System.out.println("Default Priority = " + intAttrs[0]);
System.out.println("Description : " + new String(charAttrs,0));
```

Multithreaded programs

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

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

With the ″C″ based MQSeries client, this would cause a problem, because 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 (MQQueueManager object instance), all access to the target MQSeries queue manager is synchronized. Therefore, a thread wishing to issue a call to a queue manager is blocked until all other calls in progress for that connection are complete. If you require simultaneous access to the same queue manager from multiple threads within your program, create a new MQQueueManager object for each thread that requires concurrent access. (This is equivalent to issuing a separate MQCONN call for each thread.)

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. Three exit interfaces are 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.

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

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

The simplest possible exit body is:

```
{
  return agentBuffer;
}
```

Writing user exits

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

Connection pooling

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

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

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

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

Controlling the default connection pool

Consider the following example application, MQApp1:

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

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

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

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

```
import com.ibm.mq.*;
public class MQApp2
{
      public static void main(String[] args) throws MQException
      {
         MQPoolToken token=MQEnvironment.addConnectionPoolToken();

         for (int i=0; i<args.length; i++) {
            MQQueueManager qmgr=new MQQueueManager(args[i]);
            :
            : (do something with qmgr)
            :
            qmgr.disconnect();
         }

         MQEnvironment.removeConnectionPoolToken(token);

      }
}
```

The first bold line activates the default connection pool, by registering an MQPoolToken object with MQEnvironment.

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

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

The default connection pool stores a maximum of ten unused connections, and keeps unused connections active for a maximum of five minutes. The application can alter this (for details, see “Supplying a different connection pool” on page 67).

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

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

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

The default connection pool and multiple components

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

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

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

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

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

```
import com.ibm.mq.*;
public class MQApp3
{
      public static void main(String[] args)
      {
         for (int i=0; i<10; i++) {
            MQApp3_Thread thread=new MQApp3_Thread(i*60000);
            thread.start();
         }
      }
}

class MQApp3_Thread extends Thread
{
      long time;

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

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

         MQEnvironment.removeConnectionPoolToken(token);
      }
}
```

Supplying a different connection pool

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

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

The following code fragment demonstrates this behavior:

```
MQSimpleConnectionManager myConnMan=new MQSimpleConnectionManager();
myConnMan.setActive(MQSimpleConnectionManager.MODE_ACTIVE);
MQQueueManager qmgr=new MQQueueManager("my.qmgr.1", myConnMan);
:
: (do something with qmgr)
:
qmgr.disconnect();

MQQueueManager qmgr2=new MQQueueManager("my.qmgr.1", myConnMan);
:
: (do something with qmgr2)
:
qmgr2.disconnect();
myConnMan.setActive(MQSimpleConnectionManager.MODE_INACTIVE);
```

The connection that is forged during the first MQQueueManager constructor is stored in myConnMan after the qmgr.disconnect() call. The connection is then reused during the second call to the MQQueueManager constructor.

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

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

- MQSimpleConnectionManager.setTimeout()
- MQSimpleConnectionManager.setHighThreshold()

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

Connection pooling

The following application demonstrates this:

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

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

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

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

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

Supplying your own ConnectionManager

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

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

The ConnectionManager interface defines only one method:

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

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

The ConnectionManager searches its pool for a javax.resource.spi.ManagedConnection object that has been created with identical ManagedConnectionFactory and ConnectionRequestInfo objects. If the

ConnectionManager finds any suitable ManagedConnection objects, it creates a java.util.Set that contains the candidate ManagedConnections. Then, the ConnectionManager calls the following:

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

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

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

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

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

This connection handle can be returned from allocateConnection().

A ConnectionManager should register an interest in the ManagedConnection through:

```
mc.addConnectionEventListener()
```

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

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

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

```
MQPoolServices mqps=new MQPoolServices();
mqps.addMQPoolServicesEventListener(listener);
```

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

Compiling and testing MQ base Java programs

Before compiling MQ base 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 MQ base Java applets

If you write an applet (subclass of java.applet.Applet), you must create an HTML file referencing your class before you can run it. A sample HTML file might look as follows:

```
<html>
<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 MQ base Java applications

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

```
java MyClass
```

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

Running MQ base Java applications under CICS Transaction Server for OS/390

To run a Java application as a transaction under CICS, you must:

1. Define the application and transaction to CICS by using the supplied CEDA transaction.
2. Ensure that the MQSeries CICS adapter is installed in your CICS system. (See *MQSeries for OS/390 System Management Guide* for details.)
3. Ensure that the JVM environment specified in the DHFJVM parameter of your CICS startup JCL (Job Control Language) includes appropriate CLASSPATH and LIBPATH entries.
4. Initiate the transaction by using any of your normal processes.

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

Tracing MQ base Java programs

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

Tracing is controlled by the enableTracing and disableTracing methods of the MQEnvironment class. For example:

```
MQEnvironment.enableTracing(2);   // trace at level 2
 ...                              // these commands will be traced
MQEnvironment.disableTracing();   // turn tracing off again
```

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

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

```
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 five different levels of tracing:

1. Provides entry, exit, and exception tracing
2. Provides parameter information in addition to 1
3. Provides transmitted and received 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:

- For an application, run it by issuing the command `java_g` (instead of `java`)
- For an applet, run it by issuing the command `appletviewer_g` (instead of `appletviewer`)

Notes:

1. `java_g` is not supported for High Performance Java (HPJ) applications on OS/390.
2. `java_g` is not supported on OS/400, but similar function is provided by using `OPTION(*VERBOSE)` on the RUNJVA command.

Tracing MQ base Java programs

Chapter 8. Environment-dependent behavior

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

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

In all cases, the MQSeries classes for Java code uses services that are 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). There are also differences in the behavior of some API calls and options. Most behavior differences are minor, and most of them are 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. It also provides 'V5 extensions', which are designed for use only in environments 2 and 3. The following sections describe the core and extensions.

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 74.

- MQEnvironment
- MQException
- MQGetMessageOptions

  Excluding:
  - MatchOptions
  - GroupStatus
  - SegmentStatus
  - Segmentation
- MQManagedObject

  Excluding:
  - inquire()
  - set()
- MQMessage

  Excluding:
  - groupId
  - messageFlags
  - messageSequenceNumber
  - offset
  - originalLength

Core details

- MQPoolServices
- MQPoolServicesEvent
- MQPoolServicesEventListener
- MQPoolToken
- MQPutMessageOptions

  Excluding:
  - knownDestCount
  - unknownDestCount
  - invalidDestCount
  - recordFields
- MQProcess
- MQQueue
- MQQueueManager

  Excluding:
  - begin()
  - accessDistributionList()
- MQSimpleConnectionManager
- MQC

Notes:

1. Some constants are not included in the core (see "Restrictions and variations for core classes" for details), and you should not use them in completely portable programs.
2. Some platforms do not support all connection modes. On these platforms, you can use only the core classes and options that relate to the supported modes. (See Table 1 on page 5.)

Restrictions and variations for core classes

Although the core classes generally behave consistently across all environments, there are some minor restrictions and variations, which are documented in Table 12.

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

Table 12. 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_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_SYNCPOINT_IF_PERSISTENT	Gives errors in environment 1. (See V5 extensions.)
MQGMO_MARK_SKIP_BACKOUT	Causes MQRC_OPTIONS_ERROR except in environments 4 and 5.
MQCNO_FASTPATH_BINDING	Supported only in environment 3. (See V5 extensions.)

Table 12. Core classes restrictions and variations (continued)

Class or element	Restrictions and variations
MQPMRF_* fields	Supported only in environments 2 and 3.
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 that are 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 would behave in other environments.

MQQueueManager constructor option
The MQQueueManager constructor includes an optional integer argument. This maps onto the MQI's MQCNO.options field, and is used to switch between normal and fastpath connection. This extended form of the constructor is accepted in all environments, provided that the only options used are MQCNO_STANDARD_BINDING or MQCNO_FASTPATH_BINDING. Any other options cause the constructor to fail with MQRC_OPTIONS_ERROR. The fastpath option MQC.MQCNO_FASTPATH_BINDING is only honored when used in the MQSeries V5 bindings (environment 3). If this option 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. MQSeries for AS/400 does not support the use of the `begin()` method to initiate global units of work that are coordinated by the queue manager.

MQPutMessageOptions options
The following flags may be set into the MQPutMessageOptions options fields in any environment. However, if these flags are 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. However, if these flags are used with a subsequent MQQueue.get() in any environment other than 2 or 3, the get() fails with MQRC_OPTIONS_ERROR.
- MQGMO_LOGICAL_ORDER
- MQGMO_COMPLETE_MESSAGE
- MQGMO_ALL_MSGS_AVAILABLE
- MQGMO_ALL_SEGMENTS_AVAILABLE

| The following flag may be set into the MQGetMessageOptions options
| fields in any environment. However, if this flag is used with a subsequent
| MQQueue.get() in environment 1, the get() fails with
| MQRC_OPTIONS_ERROR.
| • MQGMO_SYNCPOINT_IF_PERSISTENT

MQGetMessageOptions fields

Values may be set into the following fields, regardless of the environment. However, if the MQGetMessageOptions used on a subsequent MQQueue.get() contains 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 are always 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, knownDestCount is set to 1 and the other two fields are set 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 are set to 0.

recordFields is used with distribution lists. A value may be written into recordFields at any time, regardless of the environment. However, it 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

V5 extensions

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 that runs 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 is 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 that contains the values for the new fields.

Chapter 9. The MQ base 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
- MQPoolServices
- MQPoolServicesEvent
- MQPoolToken
- MQPutMessageOptions
- MQProcess
- MQQueue
- MQQueueManager
- MQSimpleConnectionManager

The following interfaces are described:
- MQC
- MQPoolServicesEventListener
- MQConnectionManager
- MQReceiveExit
- MQSecurityExit
- MQSendExit
- ManagedConnection
- ManagedConnectionFactory
- ManagedConnectionMetaData

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, 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

```
java.lang.Object
   │
   └─ com.ibm.mq.MQManagedObject
           │
           └─ com.ibm.mq.MQDistributionList
```

public class **MQDistributionList**
extends **MQManagedObject** (See page 99.)

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

An MQDistributionList is created by using the MQDistributionList constructor, or by using the accessDistributionList method for MQQueueManager.

A distribution list represents a set of open queues to which messages can be sent using a single call to the put() method. (See "Distribution lists" in the *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 148 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 129 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

```
java.lang.Object
   │
   └─ com.ibm.mq.MQMessageTracker
            │
            └─ com.ibm.mq.MQDistributionListItem
```

public class **MQDistributionListItem**
extends **MQMessageTracker** (See page 121.)

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 that 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 you construct 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 157.

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 159.

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 155.

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 102.)

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.

| **setDefaultConnectionManager**

```
public static void setDefaultConnectionManager(MQConnectionManager cxManager)
```

| Sets the supplied MQConnectionManager to be the default
| ConnectionManager. The default ConnectionManager is used when there is
| no ConnectionManager specified on the MQQueueManager constructor.
| This method also empties the set of MQPoolTokens.

| **Parameters:**

| *cxManager*
| The MQConnectionManager to be the default ConnectionManager.

|

setDefaultConnectionManager

```
public static void setDefaultConnectionManager
                     (javax.resource.spi.ConnectionManager cxManager)
```

Sets the default ConnectionManager, and empties the set of MQPoolTokens. The default ConnectionManager is used when there is no ConnectionManager specified on the MQQueueManager constructor.

This method requires a JVM at Java 2 v1.3 or later, with JAAS 1.0 or later installed.

Parameters:

cxManager
The default ConnectionManager (which implements the javax.resource.spi.ConnectionManager interface).

getDefaultConnectionManager

```
public static javax.resource.spi.ConnectionManager
                                 getDefaultConnectionManager()
```

Returns the default ConnectionManager. If the default ConnectionManager is actually an MQConnectionManager, returns null.

addConnectionPoolToken

```
public static void addConnectionPoolToken(MQPoolToken token)
```

Adds the supplied MQPoolToken to the set of tokens. A default ConnectionManager can use this as a hint; typically, it are enabled only while there is at least one token in the set.

Parameters:

token The MQPoolToken to add to the set of tokens.

addConnectionPoolToken

```
public static MQPoolToken addConnectionPoolToken()
```

Constructs an MQPoolToken and adds it to the set of tokens. The MQPoolToken is returned to the application to be passed later into removeConnectionPoolToken().

removeConnectionPoolToken

```
public static void removeConnectionPoolToken(MQPoolToken token)
```

Removes the specified MQPoolToken from the set of tokens. If that MQPoolToken is not in the set, there is no action.

Parameters:

token The MQPoolToken to remove from the set of tokens.

MQException

```
java.lang.Object
   │
   └─ java.lang.Throwable
          │
          └─ java.lang.Exception
                 │
                 └─ com.ibm.mq.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 73 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, because they are individual physical messages.

Logical messages that form part of a group have the same `groupId` value, but each message in the group has a different `MsgSeqNumber` value. Messages in a group can also be segmented.

The following options can be used for dealing with segmented or grouped messages:

MQC.MQGMO_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 that the queue manager sets to the local name of the queue from which the message was retrieved. This will be different from the name used to open the queue if an alias queue or model queue was opened.

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

```
java.lang.Object
   │
   └─ com.ibm.mq.MQManagedObject
```

public class **MQManagedObject**
extends **Object**

MQManagedObject is a superclass for MQQueueManager, MQQueue and MQProcess. It provides the ability to inquire and set attributes of these resources.

Variables

alternateUserId
`public String alternateUserId`

The alternate user id (if any) specified when this resource was opened. Setting this attribute has no effect.

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 methods for writing data into a message. The format of numbers and strings used by these read and write methods can be controlled by the encoding and characterSet member variables. The remaining member variables contain control information that accompanies the application message data when a message travels between sending and receiving applications. The application can set values into the member variable before putting a message to a queue and can read values after retrieving a message from a queue.

Variables

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.

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. If the default
| value is used, CharacterSet 819 (iso-8859-1/latin/ibm819) is assumed. The
| character set values shown in Table 13 are supported.

Table 13. 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
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

Table 13. Character set identifiers (continued)

characterSet	Description
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 the nature of the data in the message to the receiver. You can use your own format names, but names beginning with the letters ″MQ″ have meanings that are defined by the queue manager. The queue manager built-in formats are:

MQC.MQFMT_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 that match those specified. The special value MQC.MQMI_NONE allows *any* message identifier to match.

For an MQQueue.put() call, this specifies the message identifier to use. If MQC.MQMI_NONE is specified, the queue manager generates a unique message identifier when the message is put. The value of this member variable is updated after the put to indicate the message identifier that was used.

The default value is MQC.MQMI_NONE.

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-defined or user-defined value. The following values are defined by the system:

- MQC.MQAT_AIX
- MQC.MQAT_CICS
- MQC.MQAT_DOS
- MQC.MQAT_IMS
- MQC.MQAT_MVS
- MQC.MQAT_OS2
- MQC.MQAT_OS400
- MQC.MQAT_QMGR
- MQC.MQAT_UNIX
- MQC.MQAT_WINDOWS
- MQC.MQAT_JAVA

The default value is the special value MQC.MQAT_NO_CONTEXT, which indicates that no context information is present in the message.

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 133.

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 value of the encoding member variable determines the behavior of this method.

Values of MQC.MQENC_FLOAT_IEEE_NORMAL and MQC.MQENC_FLOAT_IEEE_REVERSED read IEEE standard doubles in big-endian and little-endian formats respectively.

A value of MQC.MQENC_FLOAT_S390 reads a System/390 format floating point number.

readFloat

```
public float readFloat()
```

Throws IOException, EOFException.

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

Values of MQC.MQENC_FLOAT_IEEE_NORMAL and MQC.MQENC_FLOAT_IEEE_REVERSED read IEEE standard floats in big-endian and little-endian formats respectively.

A value of MQC.MQENC_FLOAT_S390 reads a System/390 format floating point number.

readFully

```
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 value of the encoding member variable determines the behavior of this method.

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

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 value of the encoding member variable determines the behavior of this method.

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

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 UTF 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 value of the encoding member variable determines the behavior of this method.

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

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 value of the encoding member variable determines the behavior of this method.

Values of MQC.MQENC_FLOAT_IEEE_NORMAL and MQC.MQENC_FLOAT_IEEE_REVERSED write IEEE standard floats in Big-endian and Little-endian formats respectively.

A value of MQC.MQENC_FLOAT_S390 writes a System/390 format floating point number. Note that the range of IEEE doubles is greater than the range of S/390® double precision floating point numbers, and so very large numbers cannot be converted.

writeFloat

```
public void writeFloat(float v)
```

Throws IOException.

Write a float into the message buffer at the current position. The value of the encoding member variable determines the behavior of this method.

Values of MQC.MQENC_FLOAT_IEEE_NORMAL and MQC.MQENC_FLOAT_IEEE_REVERSED write IEEE standard floats in big-endian and little-endian formats respectively.

A value of MQC.MQENC_FLOAT_S390 will write a System/390 format floating point number.

writeInt

```
public void writeInt(int v)
```

Throws IOException.

Write an integer into the message buffer at the current position. The value of the encoding member variable determines the behavior of this method.

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

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 value of the encoding member variable determines the behavior of this method.

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

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 value of the encoding member variable determines the behavior of this method.

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

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 value of the encoding member variable determines the behavior of this method.

A value of MQC.MQENC_DECIMAL_NORMAL writes a big-endian packed decimal, a value of MQC.MQENC_DECIMAL_REVERSED writes a little-endian packed decimal.

Parameters

v	can be in the range -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 value of the encoding member variable determines the behavior of this method.

A value of MQC.MQENC_DECIMAL_NORMAL writes a big-endian packed decimal, a value of MQC.MQENC_DECIMAL_REVERSED writes a little-endian packed decimal.

Parameters

v	can be in the range -9999999 to 9999999.

writeDecimal8

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

Throws IOException.

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

A value of MQC.MQENC_DECIMAL_NORMAL writes a big-endian packed decimal, a value of MQC.MQENC_DECIMAL_REVERSED writes a little-endian packed decimal.

Parameters:

v can be in the range -999999999999999 to 999999999999999.

writeUTF

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

Throws IOException.

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

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 87) 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.

MQPoolServices

```
java.lang.Object
   │
   └─ com.ibm.mq.MQPoolServices
```

public class **MQPoolServices**
extends **Object**

Note: Normally, applications do not use this class.

The MQPoolServices class can be used by implementations of ConnectionManager that are intended for use as the default ConnectionManager for MQSeries connections.

A ConnectionManager can construct an MQPoolServices object and, through it, register a listener. This listener receives events that relate to the set of MQPoolTokens that MQEnvironment manages. The ConnectionManager can use this information to perform any necessary startup or cleanup work.

See also “MQPoolServicesEvent” on page 124 and “MQPoolServicesEventListener” on page 153.

Constructors

MQPoolServices

```
public MQPoolServices()
```

Construct a new MQPoolServices object.

Methods

addMQPoolServicesEventListener

```
public void addMQPoolServicesEventListener
                               (MQPoolServicesEventListener listener)
```

Add an MQPoolServicesEventListener. The listener receives an event whenever a token is added or removed from the set of MQPoolTokens that MQEnvironment controls, or whenever the default ConnectionManager changes.

removeMQPoolServicesEventListener

```
public void removeMQPoolServicesEventListener
                                  (MQPoolServicesEventListener listener)
```

Remove an MQPoolServicesEventListener.

getTokenCount

```
public int getTokenCount()
```

Returns the number of MQPoolTokens that are currently registered with MQEnvironment.

MQPoolServicesEvent

```
java.lang.Object
   │
   └─ java.util.EventObject
           │
           └─ com.ibm.mq.MQPoolServicesEvent
```

Note: Normally, applications do not use this class.

An MQPoolServicesEvent is generated whenever an MQPoolToken is added to, or removed from, the set of tokens that MQEnvironment controls. An event is also generated when the default ConnectionManager is changed.

See also “MQPoolServices” on page 123 and “MQPoolServicesEventListener” on page 153.

Variables

TOKEN_ADDED
`public static final int TOKEN_ADDED`

The event ID used when an MQPoolToken is added to the set.

TOKEN_REMOVED
`public static final int TOKEN_REMOVED`

The event ID used when an MQPoolToken is removed from the set.

DEFAULT_POOL_CHANGED
`public static final int DEFAULT_POOL_CHANGED`

The event ID used when the default ConnectionManager changes.

ID `protected int ID`

The event ID. Valid values are:
TOKEN_ADDED
TOKEN_REMOVED
DEFAULT_POOL_CHANGED

token `protected MQPoolToken token`

The token. When the event ID is DEFAULT_POOL_CHANGED, this is null.

Constructors

MQPoolServicesEvent
`public MQPoolServicesEvent(Object source, int eid, MQPoolToken token)`

Constructs an MQPoolServicesEvent based on the event ID and the token.

MQPoolServicesEvent
`public MQPoolServicesEvent(Object source, int eid)`

Constructs an MQPoolServicesEvent based on the event ID.

Methods

getId
```
public int getId()
```

Gets the event ID.

Returns

The event ID, with one of the following values:
- TOKEN_ADDED
- TOKEN_REMOVED
- DEFAULT_POOL_CHANGED

getToken
```
public MQPoolToken getToken()
```

Returns the token that was added to, or removed from, the set. If the event ID is DEFAULT_POOL_CHANGED, this is null.

MQPoolToken

```
java.lang.Object
   │
   └─ com.ibm.mq.MQPoolToken
```

public class **MQPoolToken**
extends **Object**

An MQPoolToken can be used to enable the default connection pool. MQPoolTokens are registered with the MQEnvironment class before an application component connects to MQSeries. Later, they are deregistered when the component has finished using MQSeries. Typically, the default ConnectionManager is active while the set of registered MQPoolTokens is not empty.

MQPoolToken provides no methods or variables. ConnectionManager providers can choose to extend MQPoolToken so that hints can be passed to the ConnectionManager.

See "MQEnvironment.addConnectionPoolToken" on page 92 and "MQEnvironment.removeConnectionPoolToken" on page 92.

Constructors

MQPoolToken

```
public MQPoolToken()
```

Construct a new MQPoolToken object.

MQProcess

```
java.lang.Object
   │
   └─ com.ibm.mq.MQManagedObject
           │
           └─ com.ibm.mq.MQProcess
```

public class **MQProcess**
extends **MQManagedObject**. (See page 99.)

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 140 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 93).

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 101.

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 76 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. Note that, if the syncpoint control option is not specified, a default of 'no syncpoint' is assumed. This applies for all supported platforms, including OS/390.

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

```
java.lang.Object
   │
   └─ com.ibm.mq.MQManagedObject
              │
              └─ com.ibm.mq.MQQueue
```

public class **MQQueue**
extends **MQManagedObject**. (See page 99.)

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 145.

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 145 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 262.)

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 95.)

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 262.)

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 95 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.

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

```
msg.writeString("a");
q.put(msg,pmo);
msg.writeString("b");
q.put(msg,pmo);
```

puts two messages. The first contains "a" and the second "ab".

Parameters

message
: Message Buffer containing the Message Descriptor data and message to be sent.

putMessageOptions
: Options controlling the action of the put. (See "MQPutMessageOptions" on page 129)

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 101.

MQQueueManager

```
java.lang.Object
   │
   └─ com.ibm.mq.MQManagedObject
           │
           └─ com.ibm.mq.MQQueueManager
```

public class **MQQueueManager**
extends **MQManagedObject**. (See page 99.)

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 73 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 88.

MQQueueManager

```
public MQQueueManager(String queueManagerName,
                      MQConnectionManager cxManager)
```

Throws MQException.

This constructor connects to the specified Queue Manager, using the properties in MQEnvironment. The specified MQConnectionManager manages the connection.

MQQueueManager

```
public MQQueueManager(String queueManagerName,
                      ConnectionManager cxManager)
```

Throws MQException.

This constructor connects to the specified Queue Manager, using the properties in MQEnvironment. The specified ConnectionManager manages the connection.

This method requires a JVM at Java 2 v1.3 or later, with JAAS 1.0 or later installed.

MQQueueManager

```
public MQQueueManager(String queueManagerName,
                      int options)
```

Throws MQException.

This version of the constructor is intended for use only in bindings mode and it uses the extended connection API (MQCONNX) to connect to the queue manager. The *options* parameter allows you to choose fast or normal bindings. Possible values are:
- MQC.MQCNO_FASTPATH_BINDING for fast bindings *.
- MQC.MQCNO_STANDARD_BINDING for normal bindings.

MQQueueManager

```
public MQQueueManager(String queueManagerName,
                      int options,
                      MQConnectionManager cxManager)
```

Throws MQException.

This constructor performs an MQCONNX, passing the supplied options. The specified MQConnectionManager manages the connection.

MQQueueManager

```
public MQQueueManager(String queueManagerName,
                      int options,
                      ConnectionManager cxManager)
```

Throws MQException.

This constructor performs an MQCONNX, passing the supplied options. The specified ConnectionManager manages the connection.

This method requires a JVM at Java 2 v1.3 or later, with JAAS 1.0 or later installed.

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 89.

MQQueueManager

```
public MQQueueManager(String queueManagerName,
                      Hashtable properties,
                      MQConnectionManager cxManager)
```

Throws MQException.

This constructor connects to the named Queue Manager, using the supplied Hashtable of properties to override those in MQEnvironment. The specified MQConnectionManager manages the connection.

MQQueueManager

```
public MQQueueManager(String queueManagerName,
                      Hashtable properties,
                      ConnectionManager cxManager)
```

Throws MQException.

This constructor connects to the named Queue Manager, using the supplied Hashtable of properties to override those in MQEnvironment. The specified ConnectionManager manages the connection.

This method requires a JVM at Java 2 v1.3 or later, with JAAS 1.0 or later installed.

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.

Normally, any work performed as part of a unit of work is committed. However, if this connection is managed by a ConnectionManager, rather than an MQConnectionManager, the unit of work might be rolled back.

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 description 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 *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 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 147.

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 145 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 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 145.

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 147 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 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 145.

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 148 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 to the queue manager that a new unit of work is starting.

Do not use this method for applications that use local one-phase transactions.

isConnected

```
public boolean isConnected()
```

Returns the value of the isConnected variable.

MQSimpleConnectionManager

```
java.lang.Object      com.ibm.mq.MQConnectionManager
    │                          │
    └─ com.ibm.mq.MQSimpleConnectionManager
```

public class **MQSimpleConnectionManager**
implements **MQConnectionManager** (See page 154.)

An MQSimpleConnectionManager provides basic connection pooling functionality. You can use an MQSimpleConnectionManager either as the default Connection Manager, or as a parameter to an MQQueueManager constructor. When an MQQueueManager is constructed, the most-recently-used connection in the pool is used.

Connections are destroyed (by a separate thread) when they are unused for a specified period, or when there are more than a specified number of unused connections in the pool. You can specify the timeout period and the maximum number of unused connections.

Variables

MODE_AUTO
`public static final int MODE_AUTO`. See "setActive".

MODE_ACTIVE
`public static final int MODE_ACTIVE`. See "setActive".

MODE_INACTIVE
`public static final int MODE_INACTIVE`. See "setActive".

Constructors

MQSimpleConnectionManager
`public MQSimpleConnectionManager()`

Constructs an MQSimpleConnectionManager.

Methods

setActive
`public void setActive(int mode)`

Sets the active mode of the connection pool.

Parameters

mode The required active mode of the connection pool. Valid values are:

MODE_AUTO
The connection pool is active while the Connection Manager is the default Connection Manager and there is at least one token in the set of MQPoolTokens held by MQEnvironment. This is the default mode.

MODE_ACTIVE
The connection pool is always active. When MQQueueManager.disconnect() is called, the underlying connection is pooled, and potentially reused the next time that an MQQueueManager object is constructed.

Connections will be destroyed by a separate thread if they are unused for longer than the Timeout period, or if the size of the pool exceeds HighThreshold.

MODE_INACTIVE
The connection pool is always inactive. When this mode is entered, the pool of connections to MQSeries is cleared. When MQQueueManager.disconnect() is called, the connection that underlies any active MQQueueManager object is destroyed.

getActive

```
public int getActive()
```

Gets the mode of the connection pool.

Returns

The current active mode of the connection pool, with one of the following values (see “setActive” on page 150):

MODE_AUTO
MODE_ACTIVE
MODE_INACTIVE

setTimeout

```
public void setTimeout(long timeout)
```

Sets the Timeout value, where connections that remain unused for this length of time are destroyed by a separate thread.

Parameters

timeout
The value of the timeout in milliseconds.

getTimeout

```
public long getTimeout()
```

Returns the Timeout value.

setHighThreshold

```
public void setHighThreshold(int threshold)
```

Sets the HighThreshold. If the number of unused connections in the pool exceeds this value, the oldest unused connection in the pool is destroyed.

Parameters

threshold
The maximum number of unused connections in the pool.

getHighThreshold

```
public int getHighThreshold ()
```

Returns the HighThreshold value.

MQC

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

The MQC interface defines all the constants used by the MQ Java programming interface (except for completion code constants and error code constants). To refer to one of these constants from within your programs, prefix the constant name with "MQC.". For example, you can set the close options for a queue as follows:

```
MQQueue queue;
 ...
queue.closeOptions = MQC.MQCO_DELETE; // delete the
                                      // queue when
                                      // it is closed
 ...
```

A full description of these constants is in the *MQSeries Application Programming Reference*.

Completion code and error code constants are defined in the MQException class. See "MQException" on page 93.

MQPoolServicesEventListener

public interface **MQPoolServicesEventListener**
extends **Object**

Note: Normally, applications do not use this interface.

MQPoolServicesEventListener is for implementation by providers of default ConnectionManagers. When an MQPoolServicesEventListener is registered with an MQPoolServices object, the event listener receives an event whenever an MQPoolToken is added to, or removed from, the set of MQPoolTokens that MQEnvironment manages. It also receives an event whenever the default ConnectionManager changes.

See also "MQPoolServices" on page 123 and "MQPoolServicesEvent" on page 124.

Methods

tokenAdded

```
public void tokenAdded(MQPoolServicesEvent event)
```

Called when an MQPoolToken is added to the set.

tokenRemoved

```
public void tokenRemoved(MQPoolServicesEvent event)
```

Called when an MQPoolToken is removed from the set.

defaultConnectionManagerChanged

```
public void defaultConnectionManagerChanged(MQPoolServicesEvent event)
```

Called when the default ConnectionManager is set. The set of MQPoolTokens will have been cleared.

MQConnectionManager

This is a private interface that cannot be implemented by applications. MQSeries classes for Java supplies an implementation of this interface (MQSimpleConnectionManager), which you can specify on the MQQueueManager constructor, or through MQEnvironment.setDefaultConnectionManager.

See “MQSimpleConnectionManager” on page 150.

Applications or middleware that wish to provide their own ConnectionManager should implement javax.resource.spi.ConnectionManager. This requires Java 2 v1.3 with JAAS 1.0 installed.

MQReceiveExit

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 82 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 152
- “MQChannelDefinition” on page 80

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 82 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 152
- “MQChannelDefinition” on page 80

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 82 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 152
- “MQChannelDefinition” on page 80

ManagedConnection

public interface **javax.resource.spi.ManagedConnection**

Note: Normally, applications do not use this class; it is intended for use by implementations of ConnectionManager.

MQSeries classes for Java provides an implementation of ManagedConnection that is returned from ManagedConnectionFactory.createManagedConnection. This object represents a connection to an MQSeries Queue Manager.

Methods

getConnection

```
public Object getConnection(javax.security.auth.Subject subject,
                            ConnectionRequestInfo cxRequestInfo)
```

Throws ResourceException.

Creates a new connection handle for the physical connection represented by the ManagedConnection object. For MQSeries classes for Java, this returns an MQQueueManager object. The ConnectionManager normally returns this object from allocateConnection.

The subject parameter is ignored. If the cxRequestInfo parameter is not suitable, a ResourceException is thrown. Multiple connection handles can be used simultaneously for each single ManagedConnection.

destroy

```
public void destroy()
```

Throws ResourceException.

Destroys the physical connection to the MQSeries Queue Manager. Any pending local transaction is committed. For more details, see "getLocalTransaction" on page 162.

cleanup

```
public void cleanup()
```

Throws ResourceException.

Closes all open connection handles, and resets the physical connection to an initial state ready to be pooled. Any pending local transaction is rolled back. For more details, see "getLocalTransaction" on page 162.

associateConnection

```
public void associateConnection(Object connection)
```

Throws ResourceException.

MQSeries classes for Java does not currently support this method. A javax.resource.NotSupportedException is thrown.

ManagedConnection

addConnectionEventListener

```
public void addConnectionEventListener(ConnectionEventListener listener)
```

Adds a ConnectionEventListener to the ManagedConnection instance.

The listener is notified if a severe error occurs on the ManagedConnection, or when MQQueueManager.disconnect() is called on a connection handle that is associated with this ManagedConnection. The listener is not notified about local transaction events (see "getLocalTransaction").

removeConnectionEventListener

```
public void removeConnectionEventListener(ConnectionEventListener listener)
```

Removes a registered ConnectionEventListener.

getXAResource

```
public javax.transaction.xa.XAResource getXAResource()
```

Throws ResourceException.

MQSeries classes for Java does not currently support this method. A javax.resource.NotSupportedException is thrown.

getLocalTransaction

```
public LocalTransaction getLocalTransaction()
```

MQSeries classes for Java does not currently support this method. A javax.resource.NotSupportedException is thrown.

Currently, a ConnectionManager cannot manage the MQSeries local transaction, and registered ConnectionEventListeners are not informed about events relating to the local transaction. When cleanup() occurs, any ongoing unit of work is rolled back. When destroy() occurs, any ongoing unit of work is committed.

Existing API behavior is that an ongoing unit of work is committed at MQQueueManager.disconnect(). This existing behavior is preserved only when an MQConnectionManager (rather than a ConnectionManager) manages the connection.

getMetaData

```
public ManagedConnectionMetaData getMetaData()
```

Throws ResourceException.

Gets the meta data information for the underlying Queue Manager. See "ManagedConnectionMetaData" on page 166.

setLogWriter

```
public void setLogWriter(java.io.PrintWriter out)
```

Throws ResourceException.

Sets the log writer for this ManagedConnection. When a ManagedConnection is created, it inherits the log writer from its ManagedConnectionFactory.

MQSeries classes for Java does not currently use the log writer. See "MQException.log" on page 93 for more information about logging.

getLogWriter

```
public java.io.PrintWriter getLogWriter()
```

Throws ResourceException.

Returns the log writer for this ManagedConnection.

MQSeries classes for Java does not currently use the log writer. See "MQException.log" on page 93 for more information about logging.

ManagedConnectionFactory

public interface **javax.resource.spi.ManagedConnectionFactory**

Note: Normally, applications do not use this class.

MQSeries classes for Java provides an implementation of this interface to ConnectionManagers. A ManagedConnectionFactory is used to construct ManagedConnections, and to select suitable ManagedConnections from a set of candidates. For more details about this interface, see the J2EE Connector Architecture specification (refer to Sun's Web site at `http://java.sun.com`).

Methods

createConnectionFactory

```
public Object createConnectionFactory()
```

Throws ResourceException.

MQSeries classes for Java does not currently support the createConnectionFactory methods. This method throws a javax.resource.NotSupportedException.

createConnectionFactory

```
public Object createConnectionFactory(ConnectionManager cxManager)
```

Throws ResourceException.

MQSeries classes for Java does not currently support the createConnectionFactory methods. This method throws a javax.resource.NotSupportedException.

createManagedConnection

```
public ManagedConnection createManagedConnection
                              (javax.security.auth.Subject subject,
                               ConnectionRequestInfo cxRequestInfo)
```

Throws ResourceException.

Creates a new physical connection to an MQSeries Queue Manager, and returns a ManagedConnection object that represents this connection. MQSeries ignores the subject parameter.

matchManagedConnection

```
public ManagedConnection matchManagedConnection
                              (java.util.Set connectionSet,
                               javax.security.auth.Subject subject,
                               ConnectionRequestInfo cxRequestInfo)
```

Throws ResourceException.

Searches the supplied set of candidate ManagedConnections for an appropriate ManagedConnection. Returns either null, or a suitable ManagedConnection from the set that meets the criteria for connection.

setLogWriter

```
public void setLogWriter(java.io.PrintWriter out)
```

Throws ResourceException.

Sets the log writer for this ManagedConnectionFactory. When a ManagedConnection is created, it inherits the log writer from its ManagedConnectionFactory.

MQSeries classes for Java does not currently use the log writer. See “MQException.log” on page 93 for more information about logging.

getLogWriter

```
public java.io.PrintWriter getLogWriter()
```

Throws ResourceException.

Returns the log writer for this ManagedConnectionFactory.

MQSeries classes for Java does not currently use the log writer. See “MQException.log” on page 93 for more information about logging.

hashCode

```
public int hashCode()
```

Returns the hashCode for this ManagedConnectionFactory.

equals

```
public boolean equals(Object other)
```

Checks whether this ManagedConnectionFactory is equal to another ManagedConnectionFactory. Returns true if both ManagedConnectionFactories describe the same target Queue Manager.

ManagedConnectionMetaData

public interface **javax.resource.spi.ManagedConnectionMetaData**

Note: Normally, applications do not use this class; it is intended for use by implementations of ConnectionManager.

A ConnectionManager can use this class to retrieve meta data that is related to an underlying physical connection to a Queue Manager. An implementation of this class is returned from ManagedConnection.getMetaData().

Methods

getEISProductName

```
public String getEISProductName()
```

Throws ResourceException.

Returns “IBM MQSeries”.

getProductVersion

```
public String getProductVersion()
```

Throws ResourceException.

Returns a string that describes the command level of the MQSeries Queue Manager to which the ManagedConnection is connected.

getMaxConnections

```
public int getMaxConnections()
```

Throws ResourceException.

Returns 0.

getUserName

```
public String getUserName()
```

Throws ResourceException.

If the ManagedConnection represents a Client connection to a Queue Manager, this returns the user ID used for the connection. Otherwise, it returns an empty string.

Part 3. Programming with MQ JMS

Chapter 10. Writing MQ JMS programs 169
The JMS model 169
Building a connection 170
Retrieving the factory from JNDI 170
Using the factory to create a connection . . . 171
Creating factories at runtime 171
Starting the connection 171
Choosing client or bindings transport 172
Obtaining a session 172
Sending a message 173
Setting properties with the 'set' method . . . 174
Message types 175
Receiving a message 175
Message selectors 176
Asynchronous delivery 177
Closing down 177
Java Virtual Machine hangs at shutdown . . . 177
Handling errors 177
Exception listener 178

Chapter 11. Programming Publish/Subscribe applications 179
Writing a simple Publish/Subscribe application 179
Import required packages 179
Obtain or create JMS objects 179
Publish messages 181
Receive subscriptions. 181
Close down unwanted resources 181
Using topics 181
Topic names 181
Creating topics at runtime 182
Subscriber options 183
Creating non-durable subscribers 184
Creating durable subscribers 184
Using message selectors 184
Suppressing local publications. 184
Combining the subscriber options 185
Configuring the base subscriber queue 185
Default configuration. 185
Configuring non-durable subscribers . . . 186
Configuring durable subscribers 186
Re-creation and migration issues for durable subscribers 187
Solving Publish/Subscribe problems. 187
Incomplete Publish/Subscribe close down. . . 187
Subscriber cleanup utility 188
Handling broker reports. 188

Chapter 12. JMS messages 191
Message selectors 191
Mapping JMS messages onto MQSeries messages 195
The MQRFH2 header. 196
JMS fields and properties with corresponding MQMD fields 199
Mapping JMS fields onto MQSeries fields (outgoing messages) 200
Mapping JMS header fields at send()/publish() 201
Mapping JMS property fields 203
Mapping JMS provider-specific fields . . . 203
Mapping MQSeries fields onto JMS fields (incoming messages) 204
Mapping JMS to a native MQSeries application 205
Message body 206

Chapter 13. MQ JMS Application Server Facilities 209
ASF classes and functions 209
ConnectionConsumer. 209
Planning an application 210
General principles for point-to-point messaging 210
General principles for publish/subscribe messaging 211
Handling poison messages 212
Removing messages from the queue. . . . 213
Error handling 214
Recovering from error conditions. 214
Reason and feedback codes. 214
Application server sample code 215
MyServerSession.java. 217
MyServerSessionPool.java 217
MessageListenerFactory.java 218
Examples of ASF use 219
Load1.java 219
CountingMessageListenerFactory.java 220
ASFClient1.java. 220
Load2.java 222
LoggingMessageListenerFactory.java. 222
ASFClient2.java. 222
TopicLoad.java 223
ASFClient3.java. 224
ASFClient4.java. 225

Chapter 14. JMS interfaces and classes . . . 227
Sun Java Message Service classes and interfaces 227
MQSeries JMS classes 230
BytesMessage 232
Methods 232
Connection 240
Methods 240
ConnectionConsumer. 243
Methods 243
ConnectionFactory. 244
MQSeries constructor. 244
Methods 244
ConnectionMetaData 248
MQSeries constructor. 248
Methods 248
DeliveryMode 250
Fields 250
Destination 251

MQSeries constructors 251
Methods 251
ExceptionListener 253
Methods 253
MapMessage 254
Methods 254
Message 262
Fields 262
Methods 262
MessageConsumer 275
Methods 275
MessageListener 277
Methods 277
MessageProducer 278
MQSeries constructors 278
Methods 278
MQQueueEnumeration * 282
Methods 282
ObjectMessage 283
Methods 283
Queue. 284
MQSeries constructors 284
Methods 284
QueueBrowser 286
Methods 286
QueueConnection 288
Methods 288
QueueConnectionFactory 290
MQSeries constructor. 290
Methods 290
QueueReceiver 292
Methods 292
QueueRequestor 293
Constructors. 293
Methods 293
QueueSender 295
Methods 295
QueueSession 298
Methods 298
Session 301
Fields 301
Methods 301
StreamMessage 306
Methods 306
TemporaryQueue 314
Methods 314
TemporaryTopic 315
MQSeries constructor. 315
Methods 315
TextMessage. 316
Methods 316
Topic 317
MQSeries constructor. 317
Methods 317
TopicConnection 319
Methods 319
TopicConnectionFactory 321
MQSeries constructor. 321
Methods 321
TopicPublisher 324
Methods 324
TopicRequestor 327
Constructors. 327
Methods 327
TopicSession. 329
MQSeries constructor. 329
Methods 329
TopicSubscriber. 333
Methods 333
| XAConnection 334
| XAConnectionFactory 335
| XAQueueConnection 336
| Methods 336
| XAQueueConnectionFactory 337
| Methods 337
| XAQueueSession 339
| Methods 339
| XASession 340
| Methods 340
| XATopicConnection 342
| Methods 342
| XATopicConnectionFactory 344
| Methods 344
| XATopicSession. 346
| Methods 346

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. 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.

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. Also, it provides a place to hold the parameters that control how to connect to MQSeries. Examples of these parameters are the name of the queue manager, and the name of the remote host if you use 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. Therefore, the MessageProducers or MessageConsumers associated with a Session cannot be called concurrently. This is consistent with the JMS restriction of a single thread per Session.
- PUTs can use remote queues, but GETs can only be applied to queues on the local queue manager.

JMS model

The generic JMS interfaces are subclassed into more specific versions for 'Point-to-Point' and 'Publish/Subscribe' behavior.

The point-to-point versions are:
- QueueConnection
- QueueSession
- QueueSender
- QueueReceiver

A key idea in JMS is that it is possible, and strongly recommended, to write application programs that use only references to the interfaces in `javax.jms`. All vendor-specific information is encapsulated in implementations of:
- QueueConnectionFactory
- TopicConnectionFactory
- Queue
- Topic

These are known as 'administered objects', that is, objects that can be built using a vendor-supplied administration tool and can be stored in a JNDI namespace. A JMS application can retrieve these objects from the namespace and use them without needing to know which vendor provided the implementation.

Building a connection

Connections are not created directly, but are built using a connection factory. Factory objects can be stored in a JNDI namespace, thus insulating the JMS application from provider-specific information. Details of how to create and store factory objects are in "Chapter 5. Using the MQ JMS administration tool" on page 31.

If you do not have a JNDI namespace available, see "Creating factories at runtime" on page 171.

Retrieving the factory from JNDI

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

```
import javax.jms.*;
import javax.naming.*;
import javax.naming.directory.*;
 .
 .
 .
  java.util.Hashtable environment = new java.util.Hashtable();
  environment.put(Context.INITIAL_CONTEXT_FACTORY, icf);
  environment.put(Context.PROVIDER_URL, url);
  Context ctx = new InitialDirContext( environment );
```

where:

icf	defines a factory class for the initial context
url	defines a context specific URL

For more details about JNDI usage, see Sun's JNDI documentation.

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");
```

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

```
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 because it requires references to MQSeries specific classes.

The following code creates a QueueConnectionFactory with all default settings:

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

(You can omit the `com.ibm.mq.jms.` prefix if you import the `com.ibm.mq.jms` package instead.)

A connection created from the above factory uses the Java bindings to connect to the default queue manager on the local machine. The `set` methods shown in Table 14 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 starts, MessageConsumers that are associated with the connection cannot receive any messages. To start the connection, issue the following command:

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

Table 14. 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()`. We recommend that this is the name of a temporary dynamic queue, rather than a permanent dynamic queue.

Table 14. Set methods on MQQueueConnectionFactory (continued)

Method	Description
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 that implements the relevant exit methods. (See the MQSeries 5.1 product documentation for details.) Also, the class must implement a constructor with a single `String` parameter. This string provides any initialization data that may be required by the exit, and is set to the value provided in the corresponding `set*ExitInit` method.

Choosing client or bindings transport

MQ JMS can communicate with MQSeries using either the client or bindings transports. If you use the Java bindings, the JMS application and the MQSeries queue manager must be located on the same machine. If you use the client, the queue manager can be on a different machine to the application.

The contents of the connection factory object determine which transport to use. "Chapter 5. Using the MQ JMS administration tool" on page 31 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 is made, use the `createQueueSession` method on the QueueConnection to obtain a session.

The method takes two parameters:

1. A boolean that determines whether the session is 'transacted' or 'non-transacted'.
2. A parameter that determines the 'acknowledge' mode.

The simplest case is that of the 'non-transacted' session with `AUTO_ACKNOWLEDGE`, as shown in the following code fragment:

```
QueueSession session;

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

Note: A connection is thread safe, but sessions (and objects that are created from them) are not. The recommended practice for multi-threaded applications is to use a separate session for each thread.

Sending a message

Messages are sent using a MessageProducer. For point-to-point this is a `QueueSender` that is created using the `createSender` method on QueueSession. A `QueueSender` is normally created for a specific queue, so that all messages sent using that sender are sent to the same destination. The destination is specified using a Queue object. Queue objects can be either created at runtime, or built and stored in a JNDI namespace.

Queue objects are retrieved from JNDI in the following way:

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

MQ JMS provides an implementation of Queue in `com.ibm.mq.jms.MQQueue`. It contains properties that control the details of MQSeries specific behavior, but in many cases it is possible to use the default values. JMS defines a standard way to specify the destination that minimizes the 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). This form allows you to specify remote queues (queues on a queue manager other than the one to which you are connected). It also allows you to set the other properties contained in a `com.ibm.mq.jms.MQQueue` object.

  The URI for a queue begins with the sequence `queue://`, followed by the name of the queue manager on which the queue resides. This is followed by a further '/', the name of the queue, and optionally, a list of name-value pairs that set the remaining Queue properties. For example, the URI equivalent of the previous example is:

  ```
  ioQueue = session.createQueue("queue:///SYSTEM.DEFAULT.LOCAL.QUEUE");
  ```

  Note that the name of the queue manager is omitted. This is interpreted as the queue manager to which the owning QueueConnection is connected at the time when the Queue object is used.

Sending a message

The following example connects to queue ’Q1’ on queue manager ’HOST1.QM1’, and causes all messages to be sent as non-persistent and priority 5:

```
ioQueue = session.createQueue("queue://HOST1.QM1/Q1?persistence=1&priority=5");
```

Table 15 lists the names that can be used in the name-value part of the URI. A disadvantage of this format is that it does not support symbolic names for the values, so where appropriate, the table also indicates ’special’ values. Note that these special values may be subject to change. (See “Setting properties with the ‘set’ method” for an alternative method to set properties.)

Table 15. 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

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.

Once the Queue object is obtained (either using `createQueue` as above or from JNDI), it must be passed into the `createSender` method to create a QueueSender:

```
QueueSender queueSender = session.createSender(ioQueue);
```

The resulting `queueSender` object is used to send messages by using the `send` method:

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

Setting properties with the ‘set’ method

You can set Queue properties by first creating an instance of `com.ibm.mq.jms.MQQueue` using the default constructor. Then you can fill in the required values by using public `set` methods. This method means that you can use symbolic names for the property values. However, because these values are vendor-specific, and are embedded in the code, the applications become less portable.

The following code fragment shows the setting of a queue property with a `set` method.

```
com.ibm.mq.jms.MQQueue q1 = new com.ibm.mq.jms.MQQueue();
      q1.setBaseQueueManagerName("HOST1.QM1");
      q1.setBaseQueueName("Q1");
      q1.setPersistence(DeliveryMode.NON_PERSISTENT);
      q1.setPriority(5);
```

Table 16 shows the symbolic property values that are supplied with MQ JMS for use with the `set` methods.

Table 16. 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 42 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 are in "Chapter 14. JMS interfaces and classes" on page 227.

Receiving a message

Messages are received by using a QueueReceiver. This is created from a Session by using the `createReceiver()` method. This method takes a Queue parameter that defines where the messages are received from. See "Sending a message" on page 173 for details of how to create a Queue object.

Receiving a message

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. This parameter defines how long the method should wait if there is no message available immediately. You can omit this parameter, in which case, the call blocks indefinitely. If you do not want any delay, use the `receiveNoWait()` method.

The receive methods return a message of the appropriate type. For example, if a `TextMessage` is put on a queue, when the message is received, the object that is returned is an instance of `TextMessage`.

To extract the content from the body of the message, it is necessary to cast from the generic Message class (which is the declared return type of the receive methods) to the more specific subclass, such as `TextMessage`. If the received message type is not known, you can use the 'instanceof' operator to determine which type it is. It is good practice always to test the message class before casting, so that unexpected errors can be handled gracefully.

The following code illustrates the use of 'instanceof', and extraction of the content from a TextMessage:

```
if (inMessage instanceof TextMessage) {
  String replyString = ((TextMessage) inMessage).getText();
  .
  .
  .
} else {
  // Print error message if Message was not a TextMessage.
  System.out.println("Reply message was not a TextMessage");
}
```

Message selectors

JMS provides a mechanism to select a subset of the messages on a queue so that this subset is returned by a receive call. When creating a QueueReceiver, a string can be provided that contains an SQL (Structured Query Language) expression to determine which messages to retrieve. The selector can refer to fields in the JMS message header as well as fields in the message properties (these are effectively application-defined header fields). Details of the header field names, as well as the syntax for the SQL selector, are in "Chapter 12. JMS messages" on page 191.

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 is created, the selector is fixed for the lifetime of that receiver. This means that if you require different selectors, you must create new receivers.

Asynchronous delivery

An alternative to making calls to `QueueReceiver.receive()` is to register a method that is called automatically when a suitable message is available. The following fragment illustrates the mechanism:

```
import javax.jms.*;

public class MyClass implements MessageListener
{
  // The method that will be called by JMS when a message
  // is available.
  public void onMessage(Message message)
  {
    System.out.println("message is "+message);

    // application specific processing here
    .
    .
    .
  }
}

 .
 .
 .
  // In Main program (possibly of some other class)
  MyClass listener = new MyClass();
  queueReceiver.setMessageListener(listener);

  // main program can now continue with other application specific
  // behavior.
```

Note: Use of asynchronous delivery with a QueueReceiver marks the entire Session as asynchronous. It is an error to make an explicit call to the `receive` methods of a QueueReceiver that is associated with a Session that is using asynchronous delivery.

Closing down

Garbage collection alone cannot release all MQSeries resources in a timely manner. This is especially true if the application needs to create many short-lived JMS objects at the Session level or lower. It is therefore important to call the `close()` methods of the various classes (QueueConnection, QueueSession, QueueSender, and QueueReceiver) when the resources are no longer required.

Java Virtual Machine hangs at shutdown

If an MQ JMS application finishes without calling `Connection.close()`, some JVMs will appear to hang. If this problem occurs, either edit the application to include a call to `Connection.close()`, or terminate the JVM by using the `Ctrl-C` keys.

Handling errors

Any runtime errors in a JMS application are reported by exceptions. The majority of methods in JMS throw JMSExceptions to indicate errors. It is good programming practice to catch these exceptions and display them on a suitable output.

Unlike normal Java Exceptions, a JMSException may contain a further exception embedded in it. For JMS, this can be a valuable way to pass important detail from the underlying transport. In the case of MQ JMS, when MQSeries raises an MQException, this exception is usually included as the embedded exception in a JMSException.

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

```
try {
  .
  . code which may throw a JMSException
  .
} catch (JMSException je) {
  System.err.println("caught "+je);
  Exception e = je.getLinkedException();
  if (e != null) {
    System.err.println("linked exception: "+e);
  }
}
```

Exception listener

For asynchronous message delivery, the application code cannot catch exceptions raised by failures to receive messages. This is because the application code does not make explicit calls to `receive()` methods. To cope with this situation, it is possible to register an `ExceptionListener`, which is an instance of a class that implements the `onException()` method. When a serious error occurs, this method is called with the JMSException passed as its only parameter. Further details are in Sun’s JMS documentation.

Chapter 11. Programming Publish/Subscribe applications

This section introduces the programming model that is used to write Publish/Subscribe applications that use the MQSeries Classes for Java Message Service.

Writing a simple Publish/Subscribe application

This section provides a 'walkthrough' of a simple MQ JMS application.

Import required packages

An MQSeries classes for Java Message Service application starts with a number of import statements which should include at least 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 that are used for point-to-point, as shown in the following:

Obtain a TopicConnectionFactory

The preferred way to do this is to use JNDI lookup, so that portability of the application code is maintained. The following code initializes a JNDI context:

```
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 in a hashtable, which is passed to the InitialDirContext constructor. If this connection fails, an exception is thrown to indicate that the administered objects required later in the application are not available.

Now obtain a TopicConnectionFactory by using a lookup key that the administrator has defined:

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

| If a JNDI namespace is not available, you can create a
| TopicConnectionFactory at runtime. You create a new
| com.ibm.mq.jms.MQTopicConnectionFactory in a similar way to the
| method described for a QueueConnectionFactory in "Creating factories at
| runtime" on page 171.

Create a TopicConnection

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

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

Create a TopicSession

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

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

Obtain a Topic

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

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

| If a JNDI namespace is not available, you can create a Topic at runtime, as
| described in "Creating topics at runtime" on page 182.

Create consumers and producers of publications

Depending on the nature of the JMS client application that you write, a subscriber, a publisher, or both must be created. Use the `createPublisher` and `createSubscriber` methods as follows:

```
// 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 by 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 must 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 is available. Alternative versions of the receive call are available (such as ’receiveNoWait’). For details, see “TopicSubscriber” on page 333.

Close down unwanted resources

It is important to free up all the resources used by the Publish/Subscribe application when it terminates. Use the `close()` method on 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 applications.

Topic names

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

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

Topic names in MQ JMS are arranged in a tree-like hierarchy, an example of which is shown in Figure 3 on page 182.

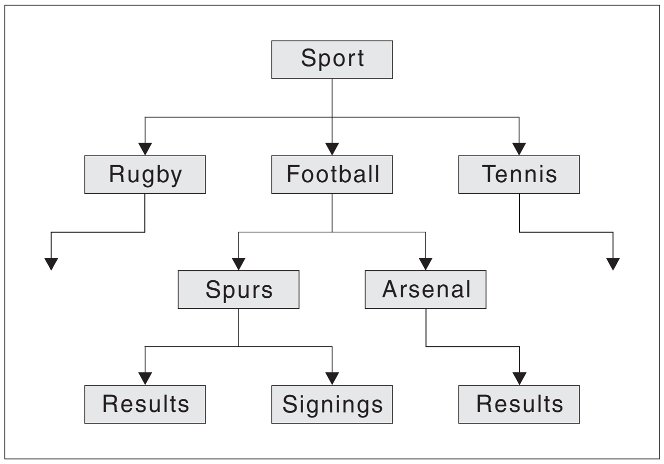

Figure 3. Topic name hierarchy

In a topic name, levels in the tree are separated by the '/' character. This means that the 'Signings' node is represented by the topic name:

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

A powerful feature of the topic system in MQSeries classes for Java Message Service 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, while the '?' wildcard matches a single character.

If a subscriber subscribes to the Topic represented by the following topic name:

```
Sport/Football/*/Results
```

it receives publications on topics including:
- Sport/Football/Spurs/Results
- Sport/Football/Arsenal/Results

If the subscription topic is:

```
Sport/Football/Spurs/*
```

it receives publications on topics including:
- Sport/Football/Spurs/Results
- Sport/Football/Spurs/Signings

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

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

Creating topics at runtime

There are four ways to create Topic objects at runtime:

1. Construct a topic by using the one-argument `MQTopic` constructor
2. Construct a topic by using the default `MQTopic` constructor, and then call the `setBaseTopicName(..)` method
3. Use the session's `createTopic(..)` method

4. Use the session’s `createTemporaryTopic()` method

Method 1: Using MQTopic(..)

This method requires a reference to the 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 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 173.

The following code creates a topic for non-persistent, priority 5 messages:

```
// Create a Topic using the one-argument MQTopic constructor
String tSpec = "Sport/Football/Spurs/Results?persistence=1&priority=5";
Topic rtTopic = new MQTopic( "topic://" + tSpec );
```

Method 2: Using MQTopic(), then setBaseTopicName(..)

This method uses the default `MQTopic` constructor, and therefore renders the code non-portable.

After the object is created, set the `baseTopicName` property by using the `setBaseTopicName` method, passing in the required topic name.

Note: The topic name used here is the non-URI form, and cannot include name-value pairs. Set these by using the ’set’ methods, as described in “Setting properties with the ‘set’ method” on page 174. The following code uses this method to create a topic:

```
// Create a Topic using the default MQTopic constructor
Topic rtTopic = new MQTopic();

// Set the object properties using the setter methods
((MQTopic)rtTopic).setBaseTopicName( "Sport/Football/Spurs/Results" );
((MQTopic)rtTopic).setPersistence(1);
((MQTopic)rtTopic).setPriority(5);
```

Method 3: Using session.createTopic(..)

A Topic object may also be created by using the `createTopic` method of TopicSession, which takes a topic URI as follows:

```
// Create a Topic using the session factory method
Topic rtTopic = session.createTopic( "topic://Sport/Football/Spurs/Results" );
```

Method 4: Using session.createTemporaryTopic()

A TemporaryTopic is a Topic that may be consumed only by subscribers that are created by the same TopicConnection. A TemporaryTopic is created as follows:

```
// Create a TemporaryTopic using the session factory method
Topic rtTopic = session.createTemporaryTopic();
```

Subscriber options

There are a number of different ways to use JMS subscribers. This section describes some examples of their use.

JMS provides two types of subscribers:

Subscriber options

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 that are published while the subscriber is inactive.

Creating non-durable subscribers

The subscriber created in "Create consumers and producers of publications" on page 180 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 you must also provide a name that uniquely identifies the subscriber:

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

Non-durable subscribers automatically deregister themselves when their `close()` method is called (or when they fall out of scope). However, if you wish to terminate a durable subscription, you must explicitly notify the system. To do this, use the session's `unsubscribe()` method and pass in the unique name that created the subscriber:

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

| A durable subscriber is created at the queue manager specified in the MQTopicConnectionFactory queue manager parameter. If there is a subsequent attempt to create a durable subscriber with the same name at a different queue manager, a new and completely independent durable subscriber is returned.

Using message selectors

You can use message selectors to filter out messages that do not satisfy given criteria. For details about message selectors, see "Message selectors" on page 176. 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

You can combine the subscriber variations, so that you can create a durable subscriber that applies a selector and ignores local publications, if you wish to. The following code fragment shows the use of the combined options:

```
// Create a durable, noLocal subscriber with a selector applied
String selector = "company = 'IBM'";
TopicSubscriber sub = session.createDurableSubscriber( topic, "D_SUB_000001",
                                                        selector, true );
```

Configuring the base subscriber queue

With MQ JMS V5.2, there are two ways in which you can configure subscribers:

- Multiple queue approach

  Each subscriber has an exclusive queue assigned to it, from which it retrieves all its messages. JMS creates a new queue for each subscriber. This is the only approach available with MQ JMS V1.1.
- Shared queue approach

  A subscriber uses a shared queue, from which it, and other subscribers, retrieve their messages. This approach requires only one queue to serve multiple subscribers. This is the default approach used with MQ JMS V5.2.

In MQ JMS V5.2, you can choose which approach to use, and configure which queues to use.

In general, the shared queue approach gives a modest performance advantage. For systems with a high throughput, there are also large architectural and administrative advantages, because of the significant reduction in the number of queues required.

In some situations, there are still good reasons for using the multiple queue approach:

- The theoretical physical capacity for message storage is greater.

  An MQSeries queue cannot hold more than 640000 messages, and in the shared queue approach, this must be divided between all the subscribers that share the queue. This issue is more significant for durable subscribers, because the lifetime of a durable subscriber is usually much longer than that of a non-durable subscriber. Therefore, more messages might accumulate for a durable subscriber.
- External administration of subscription queues is easier.

  For certain application types, administrators may wish to monitor the state and depth of particular subscriber queues. This task is much simpler when there is one to one mapping between a subscriber and a queue.

Default configuration

The default configuration uses the following shared subscription queues:

- SYSTEM.JMS.ND.SUBSCRIPTION.QUEUE for non-durable subscriptions
- SYSTEM.JMS.D.SUBSCRIPTION.QUEUE for durable subscriptions

These are created for you when you run the MQJMS_PSQ.MQSC script.

If required, you can specify alternative physical queues. You can also change the configuration to use the multiple queue approach.

Configuring non-durable subscribers

You can set the non-durable subscriber queue name property in either of the following ways:

- Use the MQ JMS administration tool (for JNDI retrieved objects) to set the BROKERSUBQ property
- Use the setBrokerSubQueue() method in your program

For non-durable subscriptions, the queue name you provide should start with the following characters:

SYSTEM.JMS.ND.

To select a shared queue approach, specify an explicit queue name, where the named queue is the one to use for the shared queue. The queue that you specify must already physically exist before you create the subscription.

To select the multiple queue approach, specify a queue name that ends with the * character. Subsequently, each subscriber that is created with this queue name creates an appropriate dynamic queue, for exclusive use by that particular subscriber. MQ JMS uses its own internal model queue to create such queues. Therefore, with the multiple queue approach, all required queues are created dynamically.

When you use the multiple queue approach, you cannot specify an explicit queue name. However, you can specify the queue prefix. This enables you to create different subscriber queue domains. For example, you could use:

SYSTEM.JMS.ND.MYDOMAIN.*

The characters that precede the * character are used as the prefix, so that all dynamic queues that are associated with this subscription will have queue names that start with SYSTEM.JMS.ND.MYDOMAIN.

Configuring durable subscribers

As discussed earlier, there may still be good reasons to use the multiple queue approach for durable subscriptions. Durable subscriptions are likely to have a longer life span, so it is possible that a large number of un-retrieved messages could accumulate on the queue.

Therefore, the durable subscriber queue name property is set in the Topic object (that is, at a more manageable level than TopicConnectionFactory). This enables you to specify a number of different subscriber queue names, without needing to re-create multiple objects starting from the TopicConnectionFactory.

You can set the durable subscriber queue name in either of the following ways:

- Use the MQ JMS administration tool (for JNDI retrieved objects) to set the BROKERDURSUBQ property
- Use the setBrokerDurSubQueue() method in your program:

  ```
  // Set the MQTopic durable subscriber queue name using
  // the multi-queue approach
  sportsTopic.setBrokerDurSubQueue("SYSTEM.JMS.D.FOOTBALL.*");
  ```

  Once the Topic object is initialized, it is passed into the TopicSession createDurableSubscriber() method to create the specified subscription:

  ```
  // Create a durable subscriber using our earlier Topic
  TopicSubscriber sub = new session.createDurableSubscriber
                                                (sportsTopic, "D_SUB_SPORT_001");
  ```

For durable subscriptions, the queue name you provide should start with the following characters:

SYSTEM.JMS.D.

To select a shared queue approach, specify an explicit queue name, where the named queue is the one to use for the shared queue. The queue that you specify must already physically exist before you create the subscription.

To select the multiple queue approach, specify a queue name that ends with the * character. Subsequently, each subscriber that is created with this queue name creates an appropriate dynamic queue, for exclusive use by that particular subscriber. MQ JMS uses its own internal model queue to create such queues. Therefore, with the multiple queue approach, all required queues are created dynamically.

When you use the multiple queue approach, you cannot specify an explicit queue name. However, you can specify the queue prefix. This enables you to create different subscriber queue domains. For example, you could use:

SYSTEM.JMS.D.MYDOMAIN.*

The characters that precede the * character are used as the prefix, so that all dynamic queues that are associated with this subscription will have queue names that start with SYSTEM.JMS.D.MYDOMAIN.

Re-creation and migration issues for durable subscribers

For a durable subscriber, do not try to reconfigure the subscriber queue name until the subscriber has been deleted. That is, perform an unsubscribe(), and then create the queue again from new (remember that any old subscriber messages are deleted).

However, if you created a subscriber using MQ JMS V1.1, that subscriber will be recognized when you migrate to the current level. You do not need to delete the subscription. The subscription continues to operate using a multiple queue approach.

Solving Publish/Subscribe problems

This section describes some problems that can occur when you develop JMS client applications that use the publish/subscribe domain. Note that this section discusses problems that are specific to the publish/subscribe domain. Refer to “Handling errors” on page 177 and “Solving problems” on page 28 for more general troubleshooting guidance.

Incomplete Publish/Subscribe close down

It is important that JMS client applications surrender all external resources when they terminate. To do this, call the `close()` method on all objects that can be closed once they are no longer required. For the publish/subscribe domain, these objects are:
- TopicConnection
- TopicSession
- TopicPublisher
- TopicSubscriber

The MQSeries classes for Java Message Service implementation eases this task through the use of a ’cascading close’. With this process, a call to ’close’ on a

TopicConnection results in calls to 'close' on each of the TopicSessions it created. This in turn results in calls to 'close' on all TopicSubscribers and TopicPublishers the sessions created.

Therefore, to ensure the proper release of external resources, it is important to call `connection.close()` for each of the connections that an application creates.

There are some circumstances where this 'close' procedure may not complete. These include:
- Loss of an MQSeries client connection
- Unexpected application termination

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

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

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

Subscriber cleanup utility

To avoid the problems that are associated with non-graceful closure of subscriber objects, MQ JMS includes a subscriber cleanup utility. This utility runs on a queue manager when the first TopicConnection to use that physical queue manager initializes. If all the TopicConnections on a given queue manager become closed, when the next TopicConnection initializes for that queue manager, the utility runs again.

The cleanup utility attempts to detect any earlier MQ JMS publish/subscribe problems that could have occurred from other applications. If it detects problems, it cleans up associated resources by:
- de-registering against the MQSeries Message Broker
- cleaning up any un-retrieved messages and queues associated with the subscription

The cleanup utility runs transparently in the background and only persists for a short time. It should not affect other MQ JMS operations. If a large number of problems are detected against a given queue manager, there might be a small delay at initialization time while resources are cleaned up.

Note: We still strongly recommend that whenever possible, you close all subscriber objects gracefully to avoid a build up of subscriber problems.

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 implementation, but under some error conditions, they may remain on the queue. These messages are sent to the `SYSTEM.JMS.REPORT.QUEUE` queue on the local queue manager.

A Java application, `PSReportDump`, is supplied with MQSeries classes for Java Message Service, which dumps the contents of this queue in plain text format. The information can then be analyzed, either by the user, or by IBM support staff. You can also use the application to clear the queue of messages after a problem is diagnosed or fixed.

The compiled form of the tool is installed in the `<MQ_JAVA_INSTALL_PATH>/bin` directory. To invoke the tool, change to this directory, then use the following command:

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

where:

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

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

Output is sent to the screen, or you can redirect it to a file.

Chapter 12. JMS messages

JMS Messages are composed of the following parts:

Header	All messages support the same set of header fields. Header fields contain values that are used by both clients and providers to identify and route messages.
Properties	Each message contains a built-in facility to support application-defined property values. Properties provide an efficient mechanism to filter application-defined messages.
Body	JMS defines several types of message body which cover the majority of messaging styles currently in use.

JMS defines five types of message body:

Stream	a stream of Java primitive values. It is filled and read sequentially.
Map	a set of name-value pairs, where names are Strings and values are Java primitive types. The entries can be accessed sequentially or randomly by name. The order of the entries is undefined.
Text	a message containing a java.util.String.
Object	a message that contains a Serializable java object
Bytes	a stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.

The JMSCorrelationID header field is used to link one message with another. It typically links a reply message with its requesting message. JMSCorrelationID can hold a provider-specific message ID, an application-specific String, or a provider-native byte[] value.

Message selectors

A Message contains a built-in facility to support application-defined property values. In effect, this provides a mechanism to add application-specific header fields to a message. Properties allow an application, via message selectors, to have a JMS provider select or filter messages on its behalf, using application-specific criteria. Application-defined properties must obey the following rules:

- Property names must obey the rules for a message selector identifier.
- Property values can be boolean, byte, short, int, long, float, double, and string.
- The following name prefixes are reserved: JMSX, JMS_.

Property values are set before sending a message. When a client receives a message, the message properties are read-only. If a client attempts to set properties at this point, a 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. JMS does not define a policy for what should or should not be made into a property.

Message selectors

However, application developers should note that JMS providers will probably handle data in a message’s body more efficiently than data in a message’s properties. For best performance, applications should only use message properties when they need to customize a message’s header. The primary reason for doing this is to support customized message selection.

A JMS message selector allows a client to specify the messages that it is interested in by using the message header. Only messages whose headers match the selector are delivered.

Message selectors cannot reference message body values.

A message selector matches a message when the selector evaluates to true when the message’s header field and property values are substituted for their corresponding identifiers in the selector.

A message selector is a String, whose syntax is based on a subset of the SQL92 conditional expression syntax. The order in which a message selector is evaluated is from left to right within a precedence level. You can use parentheses to change this order. Predefined selector literals and operator names are written here in upper case; however, they are not case-sensitive.

A selector can contain:

- Literals
  - A string literal is enclosed in single quotes. A doubled single quote represents a single quote. Examples are ’literal’ and ’literal’’s’. Like Java string literals, these use the Unicode character encoding.
  - An exact numeric literal is a numeric value without a decimal point, such as 57, -957, +62. Numbers in the range of Java long are supported.
  - An approximate numeric literal is a numeric value in scientific notation, such as 7E3 or -57.9E2, or a numeric value with a decimal, such as 7., -95.7, or +6.2. Numbers in the range of Java double are supported.
  - The boolean literals TRUE and FALSE.
- Identifiers:
  - An identifier is an unlimited length sequence of Java letters and Java digits, the first of which must be a Java letter. A letter is any character for which the method Character.isJavaLetter returns true. This includes ’_’ and ’$’. A letter or digit is any character for which the method Character.isJavaLetterOrDigit returns true.
  - Identifiers cannot be the names NULL, TRUE, or FALSE.
  - Identifiers cannot be NOT, AND, OR, BETWEEN, LIKE, IN, and IS.
  - Identifiers are either header field references or property references.
  - Identifiers are case-sensitive.
  - Message header field references are restricted to:
    - JMSDeliveryMode
    - JMSPriority
    - JMSMessageID
    - JMSTimestamp
    - JMSCorrelationID
    - JMSType

JMSMessageID, JMSTimestamp, JMSCorrelationID, and JMSType values may be null, and if so, are treated as a NULL value.
  - Any name beginning with 'JMSX' is a JMS-defined property name.
  - Any name beginning with 'JMS_' is a provider-specific property name.
  - Any name that does not begin with 'JMS' is an application-specific property name. If there is a reference to a property that does not exist in a message, its value is NULL. If it does exist, its value is the corresponding property value.
- White space is the same as it is defined for Java: space, horizontal tab, form feed, and line terminator.
- Expressions:
  - A selector is a conditional expression. A selector that evaluates to true does match, and a selector that evaluates to false or unknown does not match.
  - Arithmetic expressions are composed of themselves, arithmetic operations, identifiers (whose value is treated as a numeric literal), and numeric literals.
  - Conditional expressions are composed of themselves, comparison operations, and logical operations.
- Standard bracketing (), to set the order in which expressions are evaluated, is supported.
- Logical operators in precedence order: NOT, AND, OR.
- Comparison operators: =, >, >=, <, <=, <> (not equal).
  - Only values of the same type can be compared. One exception is that it is valid to compare exact numeric values and approximate numeric values. (The type conversion required is defined by the rules of Java numeric promotion.) If there is an attempt to compare different types, the selector is always false.
  - String and boolean comparison is restricted to = and <>. Two strings are equal if, and only if, they contain the same sequence of characters.
- Arithmetic operators in precedence order:
  - +, - unary.
  - *, /, multiplication, and division.
  - +, -, addition, and subtraction.
  - Arithmetic operations on a NULL value are not supported. If they are attempted, the complete selector is always false.
  - Arithmetic operations must use Java numeric promotion.
- arithmetic-expr1 [NOT] BETWEEN arithmetic-expr2 and arithmetic-expr3 comparison operator:
  - age BETWEEN 15 and 19 is equivalent to age >= 15 AND age <= 19.
  - age NOT BETWEEN 15 and 19 is equivalent to age < 15 OR age > 19.
  - If any of the exprs of a BETWEEN operation are NULL, the value of the operation is false. If any of the exprs of a NOT BETWEEN operation are NULL, the value of the operation is true.
- identifier [NOT] IN (string-literal1, string-literal2,...) comparison operator where identifier has a String or NULL value.
  - Country IN (' UK', 'US', 'France') is true for 'UK' and false for 'Peru'. It is equivalent to the expression (Country = ' UK') OR (Country = ' US') OR (Country = ' France').
  - Country NOT IN (' UK', 'US', 'France') is false for 'UK' and true for 'Peru'. It is equivalent to the expression NOT ((Country = ' UK') OR (Country = ' US') OR (Country = ' France')).

- If the identifier of an IN or NOT IN operation is NULL, the value of the operation is unknown.

- identifier [NOT] LIKE pattern-value [ESCAPE escape-character] comparison operator, where identifier has a String value. pattern-value is a string literal, where '_' stands for any single character and '%' stands for any sequence of characters (including the empty sequence). All other characters stand for themselves. The optional escape-character is a single character string literal, whose character is used to escape the special meaning of the '_' and '%' in pattern-value.
  - phone LIKE '12%3' is true for '123' '12993' and false for '1234'.
  - word LIKE 'l_se' is true for 'lose' and false for 'loose'.
  - underscored LIKE '_%' ESCAPE '\' is true for '_foo' and false for 'bar'.
  - phone NOT LIKE '12%3' is false for '123' '12993' and true for '1234'.
  - If the identifier of a LIKE or NOT LIKE operation is NULL, the value of the operation is unknown.
- identifier IS NULL comparison operator tests for a null header field value, or a missing property value.
  - prop_name IS NULL.
- identifier IS NOT NULL comparison operator tests for the existence of a non-null header field value or a property value.
  - prop_name IS NOT NULL.

The following message selector selects messages with a message type of car, color of blue, and weight greater than 2500 lbs:

```
"JMSType = 'car' AND color = 'blue' AND weight > 2500"
```

As noted above, property values may be NULL. The evaluation of selector expressions that contain NULL values is defined by SQL 92 NULL semantics. The following is a brief description of these semantics:

- SQL treats a NULL value as unknown.
- Comparison or arithmetic with an unknown value always yields an unknown value.
- The IS NULL and IS NOT NULL operators convert an unknown value into the respective TRUE and FALSE values.

Although SQL supports fixed decimal comparison and arithmetic, JMS message selectors do not. This is why exact numeric literals are restricted to those without a decimal. It is also why there are numerics with a decimal as an alternate representation for an approximate numeric value.

SQL comments are not supported.

Mapping JMS messages onto 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 who wish to transmit messages between JMS and traditional MQSeries applications. It is also of interest to people who wish to manipulate messages transmitted between two JMS applications - for example, in a message broker implementation.

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. You can set this flag using the MQSeries JMS administration tool. Because the MQRFH2 carries JMS-specific information, always include it in the message when the sender knows that the receiving destination is a JMS application. Normally, omit the MQRFH2 when sending a message directly to a non-JMS application (MQSeries Native application). This is because such an application does not expect an MQRFH2 in its MQSeries message. Figure 4 shows the transformation of the structures:

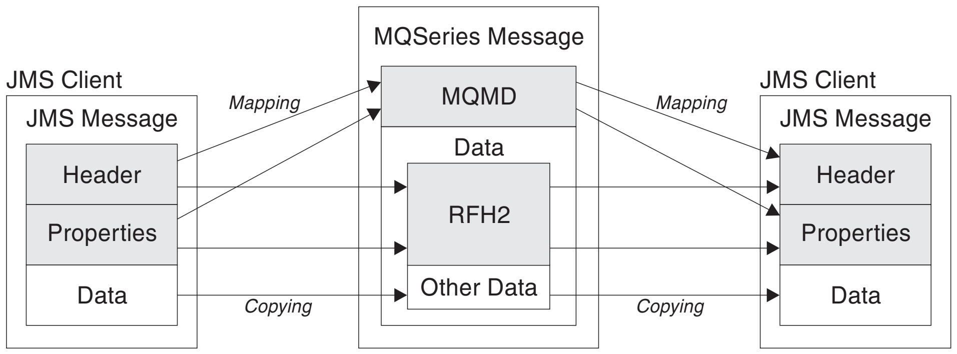

Figure 4. JMS to MQSeries mapping model

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, because a JMS application may need to get or set these fields when communicating with a non-JMS application.

Copying
: Where there is no MQMD equivalent, a JMS header field or property is passed, possibly transformed, as a field inside the MQRFH2.

The MQRFH2 header

This section describes the MQRFH Version 2 header, which carries JMS-specific data that is associated with the message content. The MQRFH2 Version 2 is an extensible header, and can also carry additional information that is not directly associated with JMS. However, but this section covers only its use by JMS.

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

Fixed portion

The fixed portion is modelled on the ’standard’ 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 value set into StrucLength must be a multiple of 4 (the data in the NameValueData fields may be padded with space characters to achieve this).

Encoding (MQLONG)

Data encoding.

Encoding of any numeric data in the portion of the message following the MQRFH2 (the next header, or the message data following this header).

CodedCharSetId (MQLONG)

Coded character set identifier.

Representation of any character data in the portion of the message following the MQRFH2 (the next header, or the message data following this header).

Format (MQCHAR8)

Format name.

Format name for the portion of the message following the MQRFH2.

Flags (MQLONG)

Flags.

MQRFH_NO_FLAGS =0. No flags set.

NameValueCCSID (MQLONG)

The coded character set identifier (CCSID) for the NameValueData character strings contained in this header. The NameValueData may be coded in a character set that differs from the other character strings that are contained in the header (StrucID and Format).

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 variable portion follows the fixed portion. The variable portion contains a variable number of MQRFH2 Folders. Each folder contains a variable number of elements or properties. Folders group together related properties. The MQRFH2 headers created by JMS can contain up to three folders:

The <mcd> folder

This contains properties that describe the 'shape' or 'format' of the message. For example the `msd` property identifies the message as being Text, Bytes, Stream. Map, Object, or 'Null'. This folder is always present in a JMS MQRFH2.

The <jms> folder

This is used to transport JMS header fields, and JMSX properties that cannot be fully expressed in the MQMD. This folder is always present in a JMS MQRFH2.

The <usr> folder

This is used to transport any application-defined properties associated with the message. This folder is only present if the application has set some application-defined properties.

Table 17 shows a full list of property names.

Table 17. 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	any

Table 17. MQRFH2 folders and properties used by JMS (continued)

JMS fields		MQRFH2 fields		
Name	Java type	Folder name	Property name	Type/values
		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 that immediately follows this length field (it does not include its own length). The value set into NameValueLength is always a multiple of 4 (the NameValueData field is padded with space characters to achieve this).

NameValueData (MQCHARn)
A single character string, whose length in bytes is given by the preceding NameValueLength field. It contains a ′folder′ holding a sequence of ′properties′. Each property is a ′name/type/value′ triplet, contained within an XML element whose name is the folder name, as follows:

```
<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, because their datatype is predefined. If it is included, one or more space characters must be included before the `dt=` tag.

`name` is the name of the property - see Table 17 on page 197.

`datatype` must match, after folding, one of the literal values in Table 18.

`value` is a string representation of the value to be conveyed, as shown in Table 18.

A null value is encoded using the following syntax:

```
<name/>
```

Table 18. 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 lay in the range -128 to 127 inclusive

Table 18. Property datatypes and values (continued)

Datatype	Value
i2	A number, expressed using digits `0..9`, with optional sign (no fractions or exponent). Must lay in the range -32768 to 32767 inclusive
i4	A number, expressed using digits `0..9`, with optional sign (no fractions or exponent). Must lay in the range -2147483648 to 2147483647 inclusive
i8	A number, expressed using digits `0..9`, with optional sign (no fractions or exponent). Must lay in the range -9223372036854775808 to 92233720368547750807 inclusive
int	A number, expressed using digits 0..9, with optional sign (no fractions or exponent). Must lay in the same range as 'i8'. This can be used in place of one of the 'i*' types if the sender does not wish to associate a particular precision with the property
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. You must use the following escape sequences in a string value:
`&amp;` for the `&` character
| `&lt;` for the `<` character

You can use the following escape sequences, but they are not required:
| `&gt;` for the `>` character
`&apos;` for the `'` character
`&quot;` for the `"` character

JMS fields and properties with corresponding MQMD fields

Table 19 lists the properties that are mapped directly to MQMD fields.

Table 19. 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

Table 19. JMS properties mapping to MQMD fields (continued)

JMS field		MQMD field	
Header	**Java type**	**Field**	**C type**
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_Report_Discard_Msg	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
JMS_IBM_Encoding	int	Encoding	MQLONG
JMS_IBM_Character_Set	String	CodedCharacterSetId	MQLONG

Mapping JMS fields onto MQSeries fields (outgoing messages)

Table 20 shows how the header/property fields are mapped into MQMD/RFH2 fields at send() or publish() time.

For fields marked 'Set by Message Object', the value transmitted is the value held in the JMS message immediately before the send/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). 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 20. Outgoing message field mapping

JMS fields	Transmitted 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

Table 20. Outgoing message field mapping (continued)

JMS fields	Transmitted in		Set by
Name	**MQMD field**	**Header**	
JMSTimestamp	PutDate/PutTime		Send Method
JMSCorrelationID	CorrelId	MQRFH2	Message Object
JMSReplyTo	ReplyToQ/ReplyToQMgr	MQRFH2	Message Object
JMSType		MQRFH2	Message Object
JMSRedelivered			Receive-only
Properties			
JMSXUserID	UserIdentifier		Send Method
JMSXAppID	PutApplName		Send Method
JMSXDeliveryCount			Receive-only
JMSXGroupID	GroupId	MQRFH2	Message Object
JMSXGroupSeq	MsgSeqNumber	MQRFH2	Message Object
Provider specific			
JMS_IBM_Report_Exception	Report		Message Object
JMS_IBM_Report_Expiration	Report		Message Object
JMS_IBM_Report_COA/COD	Report		Message Object
JMS_IBM_Report_NAN/PAN	Report		Message Object
JMS_IBM_Report_Pass_Msg_ID	Report		Message Object
JMS_IBM_Report_Pass_Correl_ID	Report		Message Object
JMS_IBM_Report_Discard_Msg	Report		Message Object
JMS_IBM_MsgType	MsgType		Message Object
JMS_IBM_Feedback	Feedback		Message Object
JMS_IBM_Format	Format		Message Object
JMS_IBM_PutApplType	PutApplType		Send Method
JMS_IBM_Encoding	Encoding		Message Object
JMS_IBM_Character_Set	CodedCharacterSetId		Message Object

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 “uniform resource identifiers” on page 173 for details of the URI notation).
- **JMSReplyTo to MQMD ReplyToQ, ReplyToQMgr, MQRFH2:** The Queue and QueueManager name are copied to the MQMD ReplyToQ and ReplyToQMgr fields respectively. The destination extension information (other ’useful’ details that are kept in the Destination Object) is copied into the MQRFH2 field. The MQRFH2 field is encoded as URI (see “uniform resource identifiers” on page 173 for details of the URI notation).

- **JMSDeliveryMode to MQMD Persistence:** The JMSDeliveryMode value is set by the send/publish() Method or MessageProducer, unless the Destination Object overrides it. The JMSDeliveryMode value is mapped to the MQMD Persistence field as follows:
  - JMS value PERSISTENT is equivalent to MQPER_PERSISTENT
  - JMS value NON_PERSISTENT is equivalent to MQPER_NOT_PERSISTENT

  If 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 in milliseconds, but MQMD.expiry is in centiseconds.
  - If the send() method sets an unlimited time to live, MQMD Expiry is set to MQEI_UNLIMITED, and no JMSExpiration is encoded in the MQRFH2.
  - If the send() method sets a time to live that is less than 214748364.7 seconds (about 7 years), the time to live is stored in MQMD. Expiry and the expiration time (in milliseconds) is encoded as an i8 value in the MQRFH2.
  - If the send() method sets a time to live greater than 214748364.7 seconds, MQMD.Expiry is set to MQEI_UNLIMITED. The true expiration time in milliseconds is encoded as an i8 value in the MQRFH2.
- **JMSPriority to MQMD Priority:** Directly map JMSPriority value (0-9) onto MQMD priority value (0-9). If JMSPriority is set to a non-default value, the priority level is also encoded in the MQRFH2.
- **JMSMessageID from MQMD MessageID:** All messages sent from JMS have unique message identifiers assigned by 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 that is set into the JMSMessageId field before 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 that is set into the JMSMessageId field before 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 is a message identifier from a message previously sent or received, and so should be a string of 48 hexadecimal digits that are prefixed with 'ID:'. The prefix is removed, the remaining characters are converted into binary, and then they are set into the MQMD CorrelId field. No correlid value is encoded in the MQRFH2.
  - **A provider-native byte[] value:** The value is copied into the MQMD CorrelId field - padded with nulls, or truncated to 24 bytes if necessary. No correlid value is encoded in the MQRFH2.
  - **An application-specific String:** The value is copied into the MQRFH2. The first 24 bytes of the string, in UTF8 format, are written into the MQMD CorrelID.

Mapping JMS property fields

These notes refer to the mapping of JMS property fields in 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

| **JMS_IBM_Report_Discard_Msg**
| MQRO_DISCARD_MSG

- **JMS_IBM_MsgType to MQMD MsgType:** Value maps directly onto MQMD MsgType. If the application has not set an explicit value of JMS_IBM_MsgType, then a default value is used. This default value is determined as follows:
  - If JMSReplyTo is set to 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.
- | **JMS_IBM_Encoding to MQMD Encoding:** If set, this property overrides the | numeric encoding of the Destination Queue or Topic.
- | **JMS_IBM_Character_Set to MQMD CodedCharacterSetId:** If set, this property | overrides the coded character set property of the Destination Queue or Topic.

Mapping MQSeries fields onto JMS fields (incoming messages)

Table 21 shows how the header/property fields are mapped into MQMD/MQRFH2 fields at send() or publish() time.

Table 21. 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	

Table 21. Incoming message field mapping (continued)

JMS fields	Retrieved from	
Name	**MQMD field**	**MQRFH2**
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_Report_Discard_Msg	Report	
JMS_IBM_MsgType	MsgType	
JMS_IBM_Feedback	Feedback	
JMS_IBM_Format	Format	
JMS_IBM_PutApplType	PutApplType	
JMS_IBM_Encoding[1]	Encoding	
JMS_IBM_Character_Set[1]	CodedCharacterSetId	
1. Only set if the incoming message is a Bytes Message.		

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 on page 206 is a diagram of the mapping.

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 that no MQRFH2 field is to be produced.

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

Mapping JMS messages

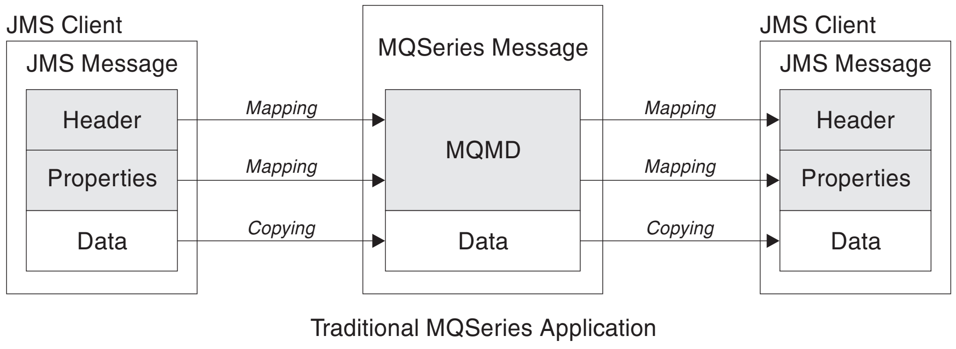

Figure 5. JMS to MQSeries mapping model

Message body

This section discusses the encoding of the message body itself. The encoding depends on the type of JMS message:

ObjectMessage
is an object serialized by the Java Runtime in the normal way.

TextMessage
is an encoded string. For an outgoing message, the string is encoded in the character set given by the Destination object. This defaults to UTF8 encoding (the UTF8 encoding starts with the first character of the message - there is no length field at the start). It is, however, possible to specify any other character set supported by MQ Java. Such character sets are used mainly when you send a message to a non-JMS application.

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

An incoming message is interpreted using the character set and encoding that are specified in the message itself. These specifications are in the rightmost 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 1.0.2 specification, and associated Java documentation.

For an outgoing message that was assembled by the application itself, the Destination object's encoding property may be used to override the encodings of integer and floating point fields contained in the message. For example, you can request that floating point values are stored in S/390 rather than IEEE format).

An incoming message is interpreted using the numeric encoding specified in the message itself. This specification is in the rightmost 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 is resent without modification, its body is transmitted byte for byte, as it was received. The Destination object's encoding property has no effect on the body. The only string-like entity that can be sent explicitly in a BytesMessage is a UTF8 string. This is encoded in Java UTF8 format, and starts with a 2-byte length field. The Destination object's character set property has no effect on the encoding of an outgoing BytesMessage. The character set value in an incoming 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. Therefore, for a JMS application to send a BytesMessage that contains text data, the application itself must convert its strings to byte arrays, and write these byte arrays into the BytesMessage.

MapMessage
is a string containing a set of XML name/type/value triplets, encoded as:

```
<map><elementName1 dt='datatype'>value</elementName1>
<elementName2 dt='datatype'>value</elementName2>.....
</map>
```

where:
`datatype` can take one of the values described in Table 18 on page 198.
`string` is the default datatype, so `dt='string'` is omitted.

The character set used to encode or interpret the XML string that makes up the MapMessage body is determined following the rules that apply to a TextMessage.

StreamMessage
is like a map, but without element names:

```
<stream><elt dt='datatype'>value</elt>
<elt dt='datatype'>value</elt>.....</stream>
```

Every element is sent using the same tagname (elt). The default type is string, so `dt='string'` is omitted for string elements.

The character set used to encode or interpret the XML string that makes up the StreamMessage body is determined following the rules that apply to a TextMessage.

The MQRFH2.format field is set as follows:

MQFMT_NONE
for ObjectMessage, BytesMessage, or messages with no body.

MQFMT_STRING
for TextMessage, StreamMessage, or MapMessage.

Chapter 13. MQ JMS Application Server Facilities

MQ JMS V5.2 supports the Application Server Facilities (ASF) that are specified in the Java Message Service 1.0.2 specification (see Sun's Java Web site at `http://java.sun.com`). This specification identifies three roles within this programming model:

- **The JMS provider** supplies ConnectionConsumer and advanced Session functionality.
- **The application server** supplies ServerSessionPool and ServerSession functionality.
- **The client application** uses the functionality that the JMS provider and application server supply.

The following sections contain details about how MQ JMS implements ASF:

- "ASF classes and functions" describes how MQ JMS implements the ConnectionConsumer class and advanced functionality in the Session class.
- "Application server sample code" on page 215 describes the sample ServerSessionPool and ServerSession code that is supplied with MQ JMS.
- "Examples of ASF use" on page 219 describes supplied ASF samples and examples of ASF use from the perspective of a client application.

Note: The Java Message Service 1.0.2 specification for ASF also describes JMS support for distributed transactions using the X/Open XA protocol. For details of the XA support that MQ JMS provides, see "Appendix E. JMS JTA/XA interface with WebSphere" on page 357.

ASF classes and functions

MQ JMS implements the ConnectionConsumer class and advanced functionality in the Session class. For details, see:

- "MQPoolServices" on page 123
- "MQPoolServicesEvent" on page 124
- "MQPoolToken" on page 126
- "MQPoolServicesEventListener" on page 153
- "ConnectionConsumer" on page 243
- "QueueConnection" on page 288
- "Session" on page 301
- "TopicConnection" on page 319

ConnectionConsumer

The JMS specification enables an application server to integrate closely with a JMS implementation by using the `ConnectionConsumer` interface. This feature provides concurrent processing of messages. Typically, an application server creates a pool of threads, and the JMS implementation makes messages available to these threads. A JMS-aware application server can use this feature to provide high-level messaging functionality, such as message processing beans.

Normal applications do not use the ConnectionConsumer, but expert JMS clients might use it. For such clients, the ConnectionConsumer provides a high-performance method to deliver messages concurrently to a pool of threads. When a message arrives on a queue or a topic, JMS selects a thread from the pool and delivers a batch of messages to it. To do this, JMS runs an associated MessageListener's `onMessage()` method.

You can achieve the same effect by constructing multiple Session and MessageConsumer objects, each with a registered MessageListener. However, the ConnectionConsumer provides better performance, less use of resources, and greater flexibility. In particular, fewer Session objects are required.

To help you develop applications that use ConnectionConsumers, MQ JMS provides a fully-functioning example implementation of a pool. You can use this implementation without any changes, or adapt it to suit the specific needs of the application.

Planning an application

General principles for point-to-point messaging

When an application creates a ConnectionConsumer from a QueueConnection object, it specifies a JMS Queue object and a selector string. The ConnectionConsumer then begins to receive messages (or, more accurately, to provide messages to Sessions in the associated ServerSessionPool). Messages arrive on the queue, and if they match the selector, they are delivered to Sessions in the associated ServerSessionPool.

In MQSeries terms, the Queue object refers to either a QLOCAL or a QALIAS on the local Queue Manager. If it is a QALIAS, that QALIAS must refer to a QLOCAL. The fully resolved MQSeries QLOCAL is known as the *underlying QLOCAL*. A ConnectionConsumer is said to be *active* if it is not closed and its parent QueueConnection is started.

It is possible for multiple ConnectionConsumers, each with different selectors, to run against the same underlying QLOCAL. To maintain performance, unwanted messages should not accumulate on the queue. Unwanted messages are those for which no active ConnectionConsumer has a matching selector. You can set the QueueConnectionFactory so that these unwanted messages are removed from the queue (for details, see "Removing messages from the queue" on page 213). You can set this behavior in one of two ways:

- Use the JMS Administration tool to set the QueueConnectionFactory to MRET(NO).
- In your program, use:
  ```
  MQQueueConnectionFactory.setMessageRetention(JMSC.MQJMS_MRET_NO)
  ```

If you do not change this setting, the default is to retain such unwanted messages on the queue.

It is possible that ConnectionConsumers that target the same underlying QLOCAL could be created from multiple QueueConnection objects. However, for performance reasons, we recommend that multiple JVMs do not create ConnectionConsumers against the same underlying QLOCAL.

When you set up the MQSeries Queue Manager, consider the following points:

- The underlying QLOCAL must be enabled for shared input. To do this, use the following MQSC command:
  ```
  ALTER QLOCAL(your.qlocal.name) SHARE GET(ENABLED)
  ```

- Your queue manager must have an enabled dead-letter queue. If a ConnectionConsumer experiences a problem when it puts a message on the dead-letter queue, message delivery from the underlying QLOCAL stops. To define a dead-letter queue, use:
  ```
  ALTER QMGR DEADQ(your.dead.letter.queue.name)
  ```
- The user that runs the ConnectionConsumer must have authority to perform MQOPEN with MQOO_SAVE_ALL_CONTEXT and MQOO_PASS_ALL_CONTEXT. For details, see the MQSeries documentation for your specific platform.
- If unwanted messages are left on the queue, they degrade the system performance. Therefore, plan your message selectors so that between them, the ConnectionConsumers will remove all messages from the queue.

For details about MQSC commands, see *MQSeries MQSC Command Reference*.

General principles for publish/subscribe messaging

When an application creates a ConnectionConsumer from a TopicConnection object, it specifies a Topic object and a selector string. The ConnectionConsumer then begins to receive messages on that Topic that match the selector.

Alternatively, an application can create a durable ConnectionConsumer that is associated with a specific name. This ConnectionConsumer receives messages that have been published on the Topic since the durable ConnectionConsumer was last active. It receives all such messages on the Topic that match the selector.

For non-durable subscriptions, a separate queue is used for ConnectionConsumer subscriptions. The CCSUB configurable option on the TopicConnectionFactory specifies the queue to use. Normally, the CCSUB should specify a single queue for use by all ConnectionConsumers that use the same TopicConnectionFactory. However, it is possible to make each ConnectionConsumer generate a temporary queue by specifying a queue name prefix followed by a '*'.

For durable subscriptions, the CCDSUB property of the Topic specifies the queue to use. Again, this may be a queue that already exists or a queue name prefix followed by a '*'. If you specify a queue that already exists, all durable ConnectionConsumers that subscribe to the Topic use this queue. If you specify a queue name prefix followed by a '*', a queue is generated the first time that a durable ConnectionConsumer is created with a given name. This queue is reused later when a durable ConnectionConsumer is created with the same name.

When you set up the MQSeries Queue Manager, consider the following points:

- Your queue manager must have an enabled dead-letter queue. If a ConnectionConsumer experiences a problem when it puts a message on the dead-letter queue, message delivery from the underlying QLOCAL stops. To define a dead-letter queue, use:
  ```
  ALTER QMGR DEADQ(your.dead.letter.queue.name)
  ```
- The user that runs the ConnectionConsumer must have authority to perform MQOPEN with MQOO_SAVE_ALL_CONTEXT and MQOO_PASS_ALL_CONTEXT. For details, see the MQSeries documentation for your specific platform.
- You can optimize performance for an individual ConnectionConsumer by creating a separate, dedicated, queue for it. This is at the cost of extra resource usage.

Handling poison messages

Sometimes, a badly-formatted message arrives on a queue. Such a message might make the receiving application fail and back out the receipt of the message. In this situation, such a message might be received, then returned to the queue, repeatedly. These messages are known as *poison messages*. The ConnectionConsumer must be able to detect poison messages and reroute them to an alternative destination.

When an application uses ConnectionConsumers, the circumstances in which a message is backed out depend on the Session that the application server provides:

- When the Session is non-transacted, with AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE, a message is backed out only after a system error, or if the application terminates unexpectedly.
- When the Session is non-transacted with CLIENT_ACKNOWLEDGE, unacknowledged messages can be backed out by the application server calling `Session.recover()`.

  Typically, the client implementation of MessageListener or the application server calls `Message.acknowledge()`. `Message.acknowledge()` acknowledges all messages delivered on the session so far.
- When the Session is transacted, typically, the application server commits the Session. If the application server detects an error, it may choose to back out one or more messages.
- If the application server supplies an XASession, messages are committed or backed out depending on a distributed transaction. The application server takes responsibility for completing the transaction.

The MQSeries Queue Manager keeps a record of the number of times that each message has been backed out. When this number reaches a configurable threshold, the ConnectionConsumer requeues the message on a named Backout Queue. If this requeue fails for any reason, the message is removed from the queue and either requeued to the dead-letter queue, or discarded. See "Removing messages from the queue" on page 213 for more details.

On most platforms, the threshold and requeue queue are properties of the MQSeries QLOCAL. For point-to-point messaging, this should be the underlying QLOCAL. For publish/subscribe messaging, this is the CCSUB queue defined on the TopicConnectionFactory, or the CCDSUB queue defined on the Topic. To set the threshold and requeue Queue properties, issue the following MQSC command:

```
ALTER QLOCAL(your.queue.name) BOTHRESH(threshold) BOQUEUE(your.requeue.queue.name)
```

For publish/subscribe messaging, if your system creates a dynamic queue for each subscription, these settings are obtained from the MQ JMS model queue. To alter these settings, you can use:

```
ALTER QMODEL(SYSTEM.JMS.MODEL.QUEUE) BOTHRESH(threshold) BOQUEUE(your.requeue.queue.name)
```

If the threshold is zero, poison message handling is disabled, and poison messages will remain on the input queue. Otherwise, when the backout count reaches the threshold, the message is sent to the named requeue queue. If the backout count reaches the threshold, but the message cannot go to the requeue queue, the message is sent to the dead-letter queue or discarded. This situation occurs if the requeue queue is not defined, or if the ConnectionConsumer cannot send the message to the requeue queue. On some platforms, you cannot specify the threshold and requeue queue properties. On these platforms, messages are sent to the dead-letter queue, or discarded, when the backout count reaches 20. See"Removing messages from the queue" for further details.

Removing messages from the queue

When an application uses ConnectionConsumers, JMS might need to remove messages from the queue in a number of situations:

Badly formatted message
: A message may arrive that JMS cannot parse.

Poison message
: A message may reach the backout threshold, but the ConnectionConsumer fails to requeue it on the backout queue.

No interested ConnectionConsumer
: For point-to-point messaging, when the QueueConnectionFactory is set so that it does not retain unwanted messages, a message arrives that is unwanted by any of the ConnectionConsumers.

In these situations, the ConnectionConsumer attempts to remove the message from the queue. The disposition options in the report field of the message's MQMD set the exact behavior. These options are:

MQRO_DEAD_LETTER_Q
: The message is requeued to the queue manager's dead-letter queue. This is the default.

MQRO_DISCARD_MSG
: The message is discarded.

The ConnectionConsumer also generates a report message, and this also depends on the report field of the message's MQMD. This message is sent to the message's ReplyToQ on the ReplyToQmgr. If there is an error while the report message is being sent, the message is sent to the dead-letter queue instead. The exception report options in the report field of the message's MQMD set details of the report message. These options are:

MQRO_EXCEPTION
: A report message is generated that contains the MQMD of the original message. It does not contain any message body data.

MQRO_EXCEPTION_WITH_DATA
: A report message is generated that contains the MQMD, any MQ headers, and 100 bytes of body data.

MQRO_EXCEPTION_WITH_FULL_DATA
: A report message is generated that contains all data from the original message.

default
: No report message is generated.

When report messages are generated, the following options are honoured:

- MQRO_NEW_MSG_ID
- MQRO_PASS_MSG_ID
- MQRO_COPY_MSG_ID_TO_CORREL_ID
- MQRO_PASS_CORREL_ID

If a ConnectionConsumer cannot follow the disposition options, or the exception report options, in the message's MQMD, its action depends on the persistence of the message. If the message is non-persistent, the message is discarded, and no report message is generated. If the message is persistent, delivery of all messages from the QLOCAL stops.

Therefore, it is important to define a dead-letter queue, and to check it regularly to ensure that no problems occur. Particularly, ensure that the dead-letter queue does not reach its maximum depth, and that its maximum message size is large enough for all messages.

When a message is requeued to the dead-letter queue, it is preceded by an MQSeries dead-letter header (MQDLH). See *MQSeries Application Programming Reference* for details about the format of the MQDLH. You can identify messages that a ConnectionConsumer has placed on the dead-letter queue, or report messages that a ConnectionConsumer has generated, by the following fields:

- PutApplType is `MQAT_JAVA (0x1C)`
- PutApplName is "`MQ JMS ConnectionConsumer`"

These fields are in the MQDLH of messages on the dead-letter queue, and the MQMD of report messages. The feedback field of the MQMD, and the Reason field of the MQDLH, contain a code describing the error. For details about these codes, see "Error handling". Other fields are as described in the *MQSeries Application Programming Reference*.

Error handling

Recovering from error conditions

If a ConnectionConsumer experiences a serious error, message delivery to all ConnectionConsumers with an interest in the same QLOCAL stops. Typically, this occurs if the ConnectionConsumer cannot requeue a message to the dead-letter queue, or it experiences an error when reading messages from the QLOCAL.

When this occurs, the application and application server are notified in the following way:

- Any ExceptionListener that is registered with the affected Connection is notified.

You can use these to identify the cause of the problem. In some cases, the system administrator must intervene to resolve the problem.

There are two ways in which an application can recover from these error conditions:

- Call `close()` on all affected ConnectionConsumers. The application can create new ConnectionConsumers only after all affected ConnectionConsumers are closed and any system problems are resolved.
- Call `stop()` on all affected Connections. Once all Connections are stopped and any system problems are resolved, the application should be able to `start()` all Connections successfully.

Reason and feedback codes

To determine the cause of an error, you can use:

- The feedback code in any report messages
- The reason code in the MQDLH of any messages in the dead-letter queue

ConnectionConsumers generate the following reason codes.

MQRC_BACKOUT_THRESHOLD_REACHED (0x93A; 2362)

Cause
The message reaches the Backout Threshold defined on the QLOCAL, but no Backout Queue is defined.

On platforms where you cannot define the Backout Queue, the message reaches the JMS-defined backout threshold of 20.

Action
To avoid this situation, ensure that ConnectionConsumers using the queue provide a set of selectors that deal with all messages, or set the QueueConnectionFactory to retain messages.

Alternatively, investigate the source of the message.

MQRC_MSG_NOT_MATCHED (0x93B; 2363)

Cause
In point-to-point messaging, there is a message that does not match any of the selectors for the ConnectionConsumers monitoring the queue. To maintain performance, the message is requeued to the dead-letter queue.

Action
To avoid this situation, ensure that ConnectionConsumers using the queue provide a set of selectors that deal with all messages, or set the QueueConnectionFactory to retain messages.

Alternatively, investigate the source of the message.

MQRC_JMS_FORMAT_ERROR (0x93C; 2364)

Cause
JMS cannot interpret the message on the queue.

Action
Investigate the origin of the message. JMS usually delivers messages of an unexpected format as a BytesMessage or TextMessage. Occasionally, this fails if the message is very badly formatted.

Other codes that appear in these fields are caused by a failed attempt to requeue the message to a Backout Queue. In this situation, the code describes the reason that the requeue failed. To diagnose the cause of these errors, refer to the *MQSeries Application Programming Reference*.

If the report message cannot be put on the ReplyToQ, it is put on the dead-letter queue. In this situation, the feedback field of the MQMD is filled in as described above. The reason field in the MQDLH explains why the report message could not be placed on the ReplyToQ.

Application server sample code

Figure 6 on page 216 summarises the principles of ServerSessionPool and ServerSession functionality.

Application server sample code

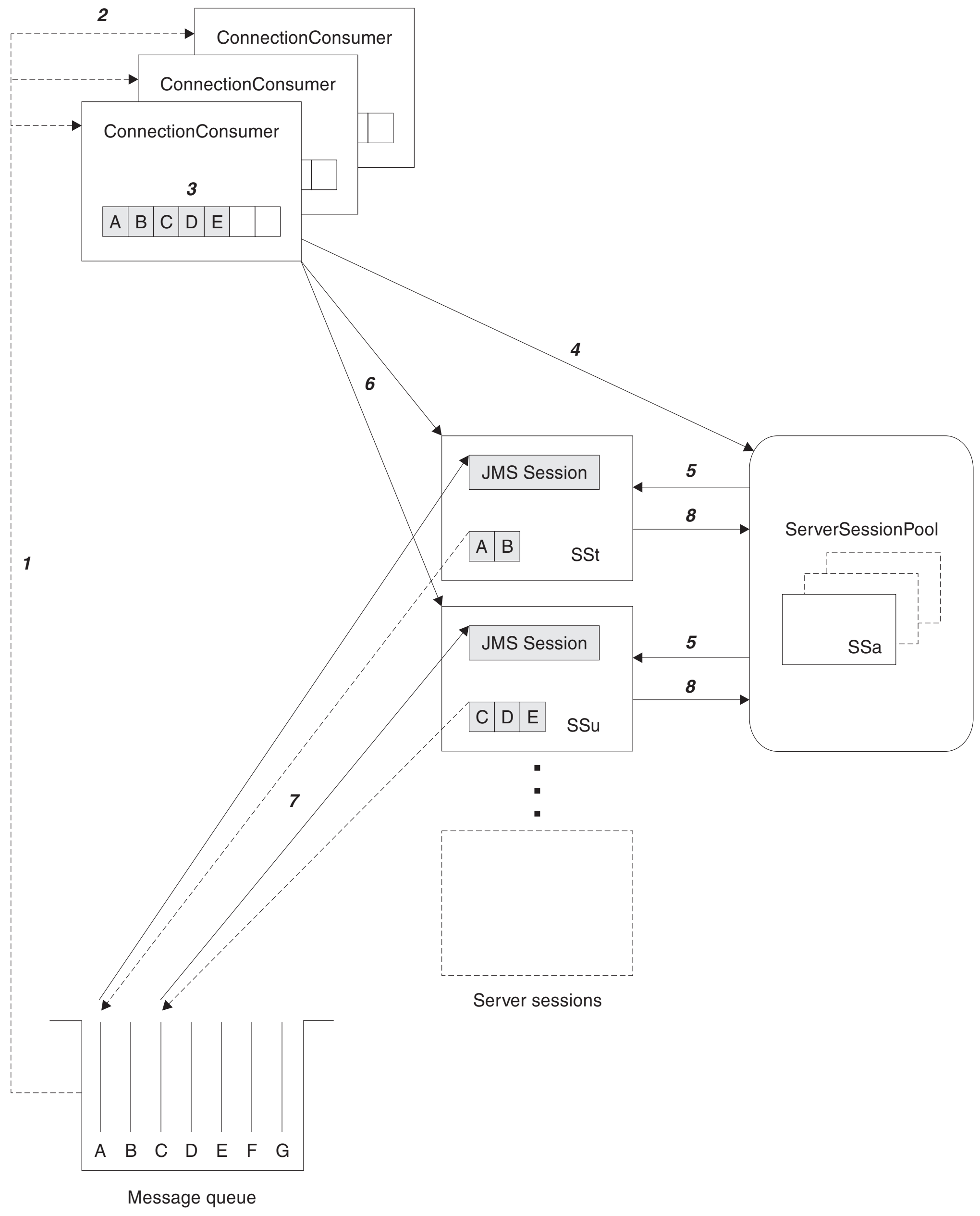

Figure 6. ServerSessionPool and ServerSession functionality

1. The ConnectionConsumers get message references from the queue.
2. Each ConnectionConsumer selects specific message references.
3. The ConnectionConsumer buffer holds the selected message references.
4. The ConnectionConsumer requests one or more ServerSessions from the ServerSessionPool.
5. ServerSessions are allocated from the ServerSessionPool.
6. The ConnectionConsumer assigns message references to the ServerSessions and starts the ServerSession threads running.
7. Each ServerSession retrieves its referenced messages from the queue. It passes them to the `onMessage` method from the MessageListener that is associated with the JMS Session.
8. After it completes its processing, the ServerSession is returned to the pool.

Normally, the application server supplies ServerSessionPool and ServerSession functionality. However, MQ JMS is supplied with a simple implementation of these interfaces, with program source. These samples are in the following directory, where <install_dir> is the installation directory for MQ JMS:

```
<install_dir>/samples/jms/asf
```

These samples enable you to use the MQ JMS ASF in a standalone environment (that is, you do not need a suitable application server). Also, they provide examples of how to implement these interfaces and take advantage of the MQ JMS ASF. These examples are intended to aid both MQ JMS users, and vendors of other application servers.

MyServerSession.java

This class implements the `javax.jms.ServerSession` interface. Its basic function is to associate a thread with a JMS Session. Instances of this class are pooled by a ServerSessionPool (see "MyServerSessionPool.java"). As a ServerSession, it must implement the following two methods:

- `getSession()`, which returns the JMS Session this associated with this ServerSession
- `start()`, which starts this ServerSession's thread and results in the JMS Session's `run()` method being invoked

MyServerSession also implements the `Runnable` interface. Therefore, the creation of the ServerSession's thread can be based on this class, and does not need a separate class.

The class uses a `wait()-notify()` mechanism that is based on the values of two boolean flags, ready and quit. This mechanism means that the ServerSession creates and starts its associated thread during its construction. However, it does not automatically execute the body of the `run()` method. The body of the `run()` method is executed only when the ready flag is set to true by the `start()` method. The ASF calls the `start()` method when it is necessary to deliver messages to the associated JMS Session.

For delivery, the `run()` method of the JMS Session is called. The MQ JMS ASF will have already loaded the `run()` method with messages.

After delivery completes, the ready flag is reset to false, and the owning ServerSessionPool is notified that delivery is complete. The ServerSession then remains in a wait state until either the `start()` method is called again, or the `close()` method is invoked and ends this ServerSession's thread.

MyServerSessionPool.java

This class implements the `javax.jms.ServerSessionPool` interface, and exists to create and control access to a pool of ServerSessions.

In this simple implementation, the pool consists of a static array of ServerSession objects that are created during the construction of the pool. The following four parameters are passed into the constructor:

- `javax.jms.Connection` *`connection`*

  The connection used to create JMS Sessions.
- `int` *`capacity`*

  The size of the array of MyServerSession objects.

- `int` *`ackMode`*

  The required acknowledge mode of the JMS Sessions.
- `MessageListenerFactory` *`mlf`*

  The MesssageListenerFactory that creates the message listener that is supplied to the JMS Sessions. See "MessageListenerFactory.java".

The pool's constructor uses these parameters to create an array of MyServerSession objects. The supplied connection is used to create JMS Sessions of the given acknowledge mode and correct domain (QueueSessions for point-to-point and TopicSessions for publish/subscribe). The Sessions are supplied with a message listener. Finally, the ServerSession objects, based on the JMS Sessions, are created.

This sample implementation is a static model. That is, all the ServerSessions in the pool are created when the pool is created, and after this, the pool cannot grow or shrink. This approach is just for simplicity. It is possible for a ServerSessionPool to use a sophisticated algorithm to create ServerSessions dynamically, as needed.

MyServerSessionPool keeps a record of which ServerSessions are currently in use by maintaining an array of boolean values called inUse. These booleans are all initialized to false. When the `getServerSession` method is invoked and requests a ServerSession from the pool, the inUse array is searched for the first false value. When one is found, the boolean is set to true and the corresponding ServerSession is returned. If there are no false values in the inUse array, the `getServerSession` method must `wait()` until notification occurs.

Notification occurs in either of the following circumstances:
- The pool's `close()` method is called, indicating that the pool should be shut down.
- A ServerSession that is currently in use completes its workload and calls the `serverSessionFinished` method. The `serverSessionFinished` method returns the ServerSession to the pool, and sets the corresponding inUse flag to false. The ServerSession then becomes eligible for re-use.

MessageListenerFactory.java

In this sample, a message listener factory object is associated with each ServerSessionPool instance. The MessageListenerFactory class represents a very simple interface that is used to obtain an instance of a class that implements the `javax.jms.MessageListener` interface. The class contains a single method:

```
javax.jms.MessageListener createMessageListener();
```

An implementation of this interface is supplied when the ServerSessionPool is constructed. This object is used to create message listeners for the individual JMS Sessions that back up the ServerSessions in the pool. This architecture means that each separate implementation of the `MessageListenerFactory` interface must have its own ServerSessionPool.

MQ JMS includes a sample MessageListenerFactory implementation, which is discussed in "CountingMessageListenerFactory.java" on page 220.

Examples of ASF use

There is a set of classes, with their source, in the directory `<install_dir>/samples/jms/asf` (where <install_dir> is the installation directory for MQ JMS). These classes use the MQ JMS application server facilities that are described in "ASF classes and functions" on page 209, within the sample standalone application server environment that is described in "Application server sample code" on page 215.

These samples provide three examples of ASF use from the perspective of a client application:

- A simple point-to-point example uses:
  - ASFClient1.java
  - Load1.java
  - CountingMessageListenerFactory.java
- A more complex point-to-point example uses:
  - ASFClient2.java
  - Load2.java
  - CountingMessageListenerFactory.java
  - LoggingMessageListenerFactory.java
- A simple publish/subscribe example uses:
  - ASFClient3.java
  - TopicLoad.java
  - CountingMessageListenerFactory.java
- A more complex publish/subscribe example uses:
  - ASFClient4.java
  - TopicLoad.java
  - CountingMessageListenerFactory.java
  - LoggingMessageListenerFactory.java

The following sections describe each class in turn.

Load1.java

This class is a simple generic JMS application that loads a given queue with a number of messages, then terminates. It can either retrieve the required administered objects from a JNDI namespace, or create them explicitly, using the MQ JMS classes that implement these interfaces. The administered objects that are required are a QueueConnectionFactory and a Queue. You can use the command line options to set the number of messages with which to load the queue, and the sleep time between individual message puts.

This application has two versions of the command line syntax.

For use with JNDI, the syntax is:

```
java Load1 [-icf jndiICF] [-url jndiURL] [-qcfLookup qcfLookup]
          [-qLookup qLookup] [-sleep sleepTime] [-msgs numMsgs]
```

For use without JNDI, the syntax is:

```
java Load1 -nojndi [-qm qMgrName] [-q qName]
                 [-sleep sleepTime] [-msgs numMsgs]
```

Table 22 describes the parameters and gives their defaults.

Table 22. Load1 parameters and defaults

Parameter	Meaning	Default
jndiICF	Initial context factory class used for JNDI	com.sun.jndi.ldap.LdapCtxFactory
jndiURL	Provider URL used for JNDI	ldap://localhost/o=ibm,c=us
qcfLookup	JNDI lookup key used for QueueConnectionFactory	cn=qcf
qLookup	JNDI lookup key used for Queue	cn=q
qMgrName	Name of queue manager to connect to	"" (use the default queue manager)
qName	Name of queue to load	SYSTEM.DEFAULT.LOCAL.QUEUE
sleepTime	Time (in milliseconds) to pause between message puts	0 (no pause)
numMsgs	Number of messages to put	1000

If there are any errors, an error message is displayed, and the application terminates.

You can use this application to simulate message load on an MQSeries queue. In turn, this message load could trigger the ASF-enabled applications described in the following sections. The messages put to the queue are simple JMS TextMessage objects. These objects do not contain user-defined message properties, which could be useful to make use of different message listeners. The source code is supplied so that you can modify this load application if necessary.

CountingMessageListenerFactory.java

This file contains definitions for two classes:
- CountingMessageListener
- CountingMessageListenerFactory

CountingMessageListener is a very simple implementation of the `javax.jms.MessageListener` interface. It keeps a record of the number of times its `onMessage` method has been invoked, but does nothing with the messages it is passed.

CountingMessageListenerFactory is the factory class for CountingMessageListener. It is an implementation of the `MessageListenerFactory` interface described in "MessageListenerFactory.java" on page 218. This factory keeps a record of all the message listeners that it produces. It also includes a method, `printStats()`, which displays usage statistics for each of these listeners.

ASFClient1.java

This application acts as a client of the MQ JMS ASF. It sets up a single ConnectionConsumer to consume the messages in a single MQSeries queue. It displays throughput statistics for each message listener that is used, and terminates after one minute.

The application can either retrieve the required administered objects from a JNDI namespace, or create them explicitly, using the MQ JMS classes that implement these interfaces. The administered objects that are required are a QueueConnectionFactory and a Queue.

This application has two versions of the command line syntax:

For use with JNDI, the syntax is:

```
java ASFClient1 [-icf jndiICF] [-url jndiURL] [-qcfLookup qcfLookup]
                [-qLookup qLookup] [-poolSize poolSize] [-batchSize batchSize]
```

For use without JNDI, the syntax is:

```
java ASFClient1 -nojndi [-qm qMgrName] [-q qName]
                        [-poolSize poolSize] [-batchSize batchSize]
```

Table 23 describes the parameters and gives their defaults.

Table 23. ASFClient1 parameters and defaults

Parameter	Meaning	Default
jndiICF	Initial context factory class used for JNDI	com.sun.jndi.ldap.LdapCtxFactory
jndiURL	Provider URL used for JNDI	ldap://localhost/o=ibm,c=us
qcfLookup	JNDI lookup key used for QueueConnectionFactory	cn=qcf
qLookup	JNDI lookup key used for Queue	cn=q
qMgrName	Name of queue manager to connect to	"" (use the default queue manager)
qName	Name of queue to consume from	SYSTEM.DEFAULT.LOCAL.QUEUE
poolSize	The number of ServerSessions created in the ServerSessionPool being used	5
batchSize	The maximum number of message that can be assigned to a ServerSession at a time	10

The application obtains a QueueConnection from the QueueConnectionFactory.

A ServerSessionPool, in the form of a MyServerSessionPool, is constructed using:
- the QueueConnection that was created previously
- the required poolSize
- an acknowledge mode, AUTO_ACKNOWLEDGE
- an instance of a CountingMessageListenerFactory, as described in “CountingMessageListenerFactory.java” on page 220

Then, the connection's `createConnectionConsumer` method is invoked, passing in:

- the Queue that was obtained earlier
- a null message selector (indicating that all messages should be accepted)
- the ServerSessionPool that was just created
- the batchSize that is required

The consumption of messages is then started through invocation of the connection's `start()` method.

The client application displays throughput statistics for each message listener that is used, displaying statistics every 10 seconds. After one minute, the connection is closed, the server session pool is stopped, and the application terminates.

Load2.java

This class is a JMS application that loads a given queue with a number of messages, then terminates, in a similar way to Load1.java. The command line syntax is also similar to that for Load1.java (substitute Load2 for Load1 in the syntax). For details, see "Load1.java" on page 219.

The difference is that each message contains a user property called value, which takes a randomly selected integer value between 0 and 100. This property means that you can apply message selectors to the messages. Consequently, the messages can be shared between the two consumers that are created in the client application described in "ASFClient2.java".

LoggingMessageListenerFactory.java

This file contains definitions for two classes:

- LoggingMessageListener
- LoggingMessageListenerFactory

LoggingMessageListener is an implementation of the `javax.jms.MessageListener` interface. It takes the messages that are passed to it and writes an entry to the log file. The default log file is `./ASFClient2.log`. You can inspect this file and check the messages that are sent to the connection consumer that is using this message listener.

LoggingMessageListenerFactory is the factory class for LoggingMessageListener. It is an implementation of the `MessageListenerFactory` interface described in "MessageListenerFactory.java" on page 218.

ASFClient2.java

ASFClient2.java is a slightly more complicated client application than ASFClient1.java. It creates two ConnectionConsumers that feed off the same queue, but that apply different message selectors. The application uses a CountingMessageListenerFactory for one consumer, and a LoggingMessageListenerFactory for the other. Use of two different message listener factories means that each consumer must have its own server session pool.

The application displays statistics that relate to one ConnectionConsumer on screen, and writes statistics that relate to the other ConnectionConsumer to a log file.

The command line syntax is similar to that for “ASFClient1.java” on page 220 (substitute ASFClient2 for ASFClient1 in the syntax). Each of the two server session pools contains the number of ServerSessions set by the poolSize parameter.

There should be an uneven distribution of messages. The messages loaded onto the source queue by Load2 contain a user property, where the value should be between 0 and 100, evenly and randomly distributed. The message selector `value>75` is applied to highConnectionConsumer, and the message selector `value≤75` is applied to normalConnectionConsumer. The highConnectionConsumer’s messages (approximately 25% of the total load) are sent to a LoggingMessageListener. The normalConnectionConsumer’s messages (approximately 75% of the total load) are sent to a CountingMessageListener.

When the client application runs, statistics that relate to the normalConnectionConsumer, and its associated CountingMessageListenerFactories, are printed to screen every 10 seconds. Statistics that relate to the highConnectionConsumer, and its associated LoggingMessageListenerFactories, are written to the log file.

You can inspect the screen and the log file to see the real destination of the messages. Add the totals for each of the CountingMessageListeners. As long as the client application does not terminate before all the messages are consumed, this should account for approximately 75% of the load. The number of log file entries should account for the remainder of the load. (If the client application terminates before all the messages are consumed, you can increase the application timeout.)

TopicLoad.java

This class is a JMS application that is a publish/subscribe version of the Load2 queue loader described in “Load2.java” on page 222. It publishes the required number of messages under the given topic, then it terminates. Each message contains a user property called value, which takes a randomly selected integer value between 0 and 100.

To use this application, ensure that the broker is running and that the required setup is complete. For details, see “Additional setup for Publish/Subscribe mode” on page 20.

This application has two versions of the command line syntax.

For use with JNDI, the syntax is:

```
java TopicLoad [-icf jndiICF] [-url jndiURL] [-tcfLookup tcfLookup]
               [-tLookup tLookup] [-sleep sleepTime] [-msgs numMsgs]
```

For use without JNDI, the syntax is:

```
java TopicLoad -nojndi [-qm qMgrName] [-t tName]
                       [-sleep sleepTime] [-msgs numMsgs]
```

Table 24 describes the parameters and gives their defaults.

Table 24. TopicLoad parameters and defaults

Parameter	Meaning	Default
jndiICF	Initial context factory class used for JNDI	com.sun.jndi.ldap.LdapCtxFactory
jndiURL	Provider URL used for JNDI	ldap://localhost/o=ibm,c=us

Table 24. TopicLoad parameters and defaults (continued)

Parameter	Meaning	Default
tcfLookup	JNDI lookup key used for TopicConnectionFactory	cn=tcf
tLookup	JNDI lookup key used for Topic	cn=t
qMgrName	Name of queue manager to connect to, and broker queue manager to publish messages to	"" (use the default queue manager)
tName	Name of topic to publish to	MQJMS/ASF/TopicLoad
sleepTime	Time (in milliseconds) to pause between message puts	0 (no pause)
numMsgs	Number of messages to put	200

If there are any errors, an error message is displayed, and the application terminates.

ASFClient3.java

ASFClient3.java is a client application that is a publish/subscribe version of "ASFClient1.java" on page 220. It sets up a single ConnectionConsumer to consume the messages published on a single Topic. It displays throughput statistics for each message listener that is used, and terminates after one minute.

This application has two versions of the command line syntax.

For use with JNDI, the syntax is:

```
java ASFClient3 [-icf jndiICF] [-url jndiURL] [-tcfLookup tcfLookup]
                [-tLookup tLookup] [-poolsize poolSize] [-batchsize batchSize]
```

For use without JNDI, the syntax is:

```
java ASFClient3 -nojndi [-qm qMgrName] [-t tName]
                        [-poolsize poolSize] [-batchsize batchSize]
```

Table 25 describes the parameters and gives their defaults.

Table 25. ASFClient3 parameters and defaults

Parameter	Meaning	Default
jndiICF	Initial context factory class used for JNDI	com.sun.jndi.ldap.LdapCtxFactory
jndiURL	Provider URL used for JNDI	ldap://localhost/o=ibm,c=us
tcfLookup	JNDI lookup key used for TopicConnectionFactory	cn=tcf
tLookup	JNDI lookup key used for Topic	cn=t
qMgrName	Name of queue manager to connect to, and broker queue manager to publish messages to	"" (use the default queue manager)
tName	Name of topic to consume from	MQJMS/ASF/TopicLoad
poolSize	The number of ServerSessions created in the ServerSessionPool being used	5

Table 25. ASFClient3 parameters and defaults (continued)

Parameter	Meaning	Default
batchSize	The maximum number of message that can be assigned to a ServerSession at a time	10

Like ASFClient1, the client application displays throughput statistics for each message listener that is used, displaying statistics every 10 seconds. After one minute, the connection is closed, the server session pool is stopped, and the application terminates.

ASFClient4.java

ASFClient4.java is a more complex publish/subscribe client application. It creates three ConnectionConsumers that all feed off the same topic, but each one applies different message selectors.

The first two consumers use ‘high’ and ‘normal’ message selectors, in the same way as the application “ASFClient2.java” on page 222. The third consumer does not use any message selector. The application uses two CountingMessageListenerFactories for the two selector-based consumers, and a LoggingMessageListenerFactory for the third consumer. Because the application uses different message listener factories, each consumer must have its own server session pool.

The application displays statistics that relate to the two selector-based consumers on screen. It writes statistics that relate to the third ConnectionConsumer to a log file.

The command line syntax is similar to that for “ASFClient3.java” on page 224 (substitute ASFClient4 for ASFClient3 in the syntax). Each of the three server session pools contains the number of ServerSessions set by the poolSize parameter.

When the client application runs, statistics that relate to the normalConnectionConsumer and the highConnectionConsumer, and their associated CountingMessageListenerFactories, are printed to screen every 10 seconds. Statistics that relate to the third ConnectionConsumer, and its associated LoggingMessageListenerFactories, are written to the log file.

You can inspect the screen and the log file to see the real destination of the messages. Add the totals for each of the CountingMessageListeners and inspect the number of log file entries.

The distribution of messages should be different from the distribution obtained by a point-to-point version of the same application (ASFClient2.java). This is because, in the publish/subscribe domain, each consumer of a topic obtains its own copy of each message published on that topic. In this application, for a given topic load, the ‘high’ and ‘normal’ consumers will receive approximately 25% and 75% of the load, respectively. The third consumer will still receive 100% of the load. Therefore, the total number of messages received is greater than 100% of the load originally published on the topic.

Chapter 14. JMS interfaces and classes

MQSeries classes for Java Message Service consists of a number of java classes and interfaces that are based on the Sun javax.jms package of interfaces and classes. Clients should be written using the Sun interfaces and classes that are listed below, and that are described in detail in the following sections. The names of the MQSeries objects that implement the Sun interfaces and classes have a prefix of 'MQ' (unless stated otherwise in the object description). The descriptions include details about any deviations of the 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 implemented by applications. Interfaces marked with '**' are implemented by application servers.

Table 26. 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 an administrator has defined.
ConnectionMetaData	ConnectionMetaData provides information that describes 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 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 MessageProducer to send messages to a Destination.
ObjectMessage	An ObjectMessage is used to send a message that contains a serializable Java object.
Queue	A Queue object encapsulates a provider-specific queue name.

Table 26. Interface Summary (continued)

Interface	Description
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 point-to-point provider.
QueueReceiver	A client uses a QueueReceiver to receive 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 to create 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 Publish/Subscribe provider.
TopicPublisher	A client uses a TopicPublisher for publishing messages on a topic.
TopicSession	A TopicSession provides methods to create TopicPublishers, TopicSubscribers and TemporaryTopics.
TopicSubscriber	A client uses a TopicSubscriber to receive messages that have been published to a topic.
XAConnection	XAConnection extends the capability of Connection by providing an XASession.
XAConnectionFactory	Some application servers provide support for grouping Java Transaction Service (JTS)-capable resource use into a distributed transaction.
XAQueueConnection	XAQueueConnection provides the same create options as QueueConnection.

Table 26. Interface Summary (continued)

Interface	Description
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 the Java Transaction API (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 27. 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 28. 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
MQSession	Session
MQTemporaryQueue	TemporaryQueue
MQTemporaryTopic	TemporaryTopic
MQTopic	Topic
MQTopicConnection	TopicConnection
MQTopicConnectionFactory	TopicConnectionFactory
MQTopicPublisher	TopicPublisher
MQTopicSession	TopicSession
MQTopicSubscriber	TopicSubscriber
MQXAConnection	XAConnection
MQXAConnectionFactory	XAConnectionFactory
MQXAQueueConnection	XAQueueConnection
MQXAQueueConnectionFactory	XAQueueConnectionFactory
MQXAQueueSession	XAQueueSession
MQXASession	XASession
MQXATopicConnection	XATopicConnection
MQXATopicConnectionFactory	XATopicConnectionFactory
MQXATopicSession	XATopicSession

Table 29. Package 'com.ibm.jms' class summary

Class	Implements
JMSBytesMessage	BytesMessage
JMSMapMessage	MapMessage
JMSMessage	Message
JMSObjectMessage	ObjectMessage
JMSStreamMessage	StreamMessage
JMSTextMessage	TextMessage

A sample implementation of the following JMS interfaces is supplied in this release of MQSeries classes for Java Message Service.
- ServerSession
- ServerSessionPool

See "Application server sample code" on page 215.

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 the message is in write-only mode.
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if it is the end of the 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 the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

readUnsignedByte

```
public int readUnsignedByte() throws JMSException
```

Read an unsigned 8-bit number from the bytes message.

Returns:
the next byte from the bytes message, interpreted as an unsigned 8-bit number.

Throws:
- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

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 the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

readUnsignedShort

```
public int readUnsignedShort() throws JMSException
```

Read an unsigned 16-bit number from the bytes message.

Returns:
the next two bytes from the bytes message, interpreted as an unsigned 16-bit integer.

Throws:
- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

readChar

```
public char readChar() throws JMSException
```

Read a Unicode character value from the bytes message.

Returns:
the next two bytes from the bytes message as a Unicode character.

Throws:

- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

readInt

```
public int readInt() throws JMSException
```

Read a signed 32-bit integer from the bytes message.

Returns:
the next four bytes from the bytes message, interpreted as an `int`.

Throws:

- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

readLong

```
public long readLong() throws JMSException
```

Read a signed 64-bit integer from the bytes message.

Returns:
the next eight bytes from the bytes message, interpreted as a `long`.

Throws:

- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

readFloat

```
public float readFloat() throws JMSException
```

Read a `float` from the bytes message.

Returns:
the next four bytes from the bytes message, interpreted as a `float`.

Throws:

- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

readDouble

```
public double readDouble() throws JMSException
```

Read a `double` from the bytes message.

Returns:
the next eight bytes from the bytes message, interpreted as a `double`.

Throws:
- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

readUTF

```
public java.lang.String readUTF() throws JMSException
```

Read in a string that has been encoded using a modified UTF-8 format from the bytes message. The first two bytes are interpreted as a 2-byte length field.

Returns:
a Unicode string from the bytes message.

Throws:
- MessageNotReadableException - if the message is in write-only mode.
- MessageEOFException - if it is the end of the message bytes.
- JMSException - if JMS fails to read the message because of an internal JMS error.

readBytes

```
public int readBytes(byte[] value) throws JMSException
```

Read a byte array from the bytes message. If there are sufficient bytes remaining in the stream the entire buffer is filled, if not, the buffer is partially filled.

Parameters:
`value` - the buffer into which the data is read.

Returns:
the total number of bytes read into the buffer, or -1 if there is no more data because the end of the bytes has been reached.

Throws:
- MessageNotReadableException - if the message is in write-only mode.
- JMSException - if JMS fails to read the message because of an internal JMS error.

readBytes

```
public int readBytes(byte[] value, int length)
                                    throws JMSException
```

Read a portion of the bytes message.

Parameters:

- `value` - the buffer into which the data is read.
- `length` - the number of bytes to read.

Returns:

the total number of bytes read into the buffer, or -1 if there is no more data because the end of the bytes has been reached.

Throws:

- MessageNotReadableException - if the message is in write-only mode.
- IndexOutOfBoundsException - if `length` is negative, or is less than the length of the array `value`
- JMSException - if JMS fails to read the message because of an internal JMS error.

writeBoolean

```
public void writeBoolean(boolean value) throws JMSException
```

Write a `boolean` to the bytes message as a 1-byte value. The value `true` is written out as the value `(byte)1`; the value `false` is written out as the value `(byte)0`.

Parameters:

`value` - the `boolean` value to be written.

Throws:

- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.

writeByte

```
public void writeByte(byte value) throws JMSException
```

Write out a `byte` to the bytes message as a 1-byte value.

Parameters:

`value` - the `byte` value to be written.

Throws:

- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.

writeShort

```
public void writeShort(short value) throws JMSException
```

Write a `short` to the bytes message as two bytes.

Parameters:

`value` - the `short` to be written.

Throws:

- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.

writeChar

```
public void writeChar(char value) throws JMSException
```

Write a `char` to the bytes message as a 2-byte value, high byte first.

Parameters:
`value` - the `char` value to be written.

Throws:
- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.

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 the message because of an 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 the message because of an 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 write that `int` value to the bytes message as a 4-byte quantity.

Parameters:
`value` - the `float` value to be written.

Throws:
- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.

writeDouble

```
public void writeDouble(double value) throws JMSException
```

Convert the double argument to a `long` using `doubleToLongBits` method in class `Double`, and then write that `long` value to the bytes message as an 8-byte quantity.

Parameters:

`value` - the `double` value to be written.

Throws:

- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.

writeUTF

```
public void writeUTF(java.lang.String value)
                              throws JMSException
```

Write a string to the bytes message using UTF-8 encoding in a machine-independent manner. The UTF-8 string written to the buffer starts with a 2-byte length field.

Parameters:

`value` - the `String` value to be written.

Throws:

- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.

writeBytes

```
public void writeBytes(byte[] value) throws JMSException
```

Write a byte array to the bytes message.

Parameters:

`value` - the byte array to be written.

Throws:

- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.

writeBytes

```
public void writeBytes(byte[] value,
                    int length) throws JMSException
```

Write a portion of a byte array to the bytes message.

Parameters:

- `value` - the byte array value to be written.
- `offset` - the initial offset within the byte array.
- `length` - the number of bytes to use.

Throws:

- MessageNotWriteableException - if message in read-only mode.
- JMSException - if JMS fails to write the message because of an internal JMS error.

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 (such as Integer, Double, and Long), Strings, and byte arrays.

Parameters:
`value` - the Java object to be written.

Throws:

- MessageNotWriteableException - if message in read-only mode.
- MessageFormatException - if object is invalid type.
- JMSException - if JMS fails to write the message because of an internal JMS error.

reset

```
public void reset() throws JMSException
```

Put the message body in read-only mode, and reposition the bytes of bytes to the beginning.

Throws:

- JMSException - if JMS fails to reset the message because of an internal JMS error.
- MessageFormatException - if message has an invalid format

Connection

public interface **Connection**
Subinterfaces: **QueueConnection**, **TopicConnection**, **XAQueueConnection**, and **XATopicConnection**

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**, **TopicConnection**, **XAQueueConnection**, and **XATopicConnection**

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 the JMS implementation fails to return the client ID for this Connection because of an 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 the JMS implementation fails to set the client ID for this Connection because of an internal error.
- InvalidClientIDException - if the 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 the 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 the 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 the JMS implementation fails to set the Exception listener for this Connection.

start

```
public void start() throws JMSException
```

Start (or restart) a Connection's delivery of incoming messages. Starting a started session is ignored.

Throws:
JMSException - if the JMS implementation fails to start the message delivery because of an internal error.

See also:
stop

stop

```
public void stop() throws JMSException
```

Used to stop a Connection's delivery of incoming messages temporarily. It can be restarted using its `start` method. When stopped, delivery to all the Connection's message consumers is inhibited. Synchronous receives are blocked, and messages are not delivered to message listeners.

Stopping a session has no affect on its ability to send messages. Stopping a stopped session is ignored.

Throws:
JMSException - if the JMS implementation fails to stop the message delivery because of an internal error.

See also:
start

close

```
public void close() throws JMSException
```

Because a provider may allocate some resources outside the JVM on behalf of a Connection, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough. There is no need to close the sessions, producers, and consumers of a closed connection.

Closing a connection causes any of its sessions' in-process transactions to be rolled back. In the case where a session's work is coordinated by an external transaction manager, when using XASession, a session's commit and rollback methods are not used and the result of a closed session's work is determined later by a transaction manager. Closing a connection does NOT force an acknowledge of client acknowledged sessions.

MQ JMS keeps a pool of MQSeries hConns available for use by Sessions. Under some circumstances, Connection.close() clears this pool. If an application uses multiple Connections sequentially, it is possible to force the pool to remain active between JMS Connections. To do this, register an MQPoolToken with com.ibm.mq.MQEnvironment for the lifetime of your JMS application. For details, see "Connection pooling" on page 64 and "MQEnvironment" on page 88.

Throws:
JMSException - if the JMS implementation fails to close the connection because of an internal error. Examples are a failure to release resources or to close a socket connection.

ConnectionConsumer

public interface **ConnectionConsumer**

MQSeries class: **MQConnectionConsumer**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQConnectionConsumer
```

For application servers, Connections provide a special facility to create a ConnectionConsumer. A Destination and a Property Selector specify the messages that it is to consume. Also, a ConnectionConsumer must be given a ServerSessionPool to use to process its messages.

See also: **QueueConnection**, and **TopicConnection**.

Methods

close()

```
public void close() throws JMSException
```

Because a provider may allocate some resources outside of the JVM on behalf of a ConnectionConsumer, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Throws:
JMSException - if a JMS implementation fails to release resources on behalf of ConnectionConsumer, or if it fails to close the connection consumer.

getServerSessionPool()

```
public ServerSessionPool getServerSessionPool()
                                          throws JMSException
```

Get the server session associated with this connection consumer.

Returns:
the server session pool used by this connection consumer.

Throws:
JMSException - if a JMS implementation fails to get the server session pool associated with this connection consumer because of an internal error.

ConnectionFactory

public interface **ConnectionFactory**
Subinterfaces: **QueueConnectionFactory**, **TopicConnectionFactory**,
XAQueueConnectionFactory, and **XATopicConnectionFactory**

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**, **TopicConnectionFactory**,
XAQueueConnectionFactory, and **XATopicConnectionFactory**

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. It can be either JMSC.MQJMS_TP_BINDINGS_MQ, or JMSC.MQJMS_TP_CLIENT_MQ_TCPIP.

getTransportType *

```
public int getTransportType()
```

Retrieve the transport type.

setClientId *

```
public void setClientId(String x)
```

Sets the client Identifier to be used for all connections created using this Connection.

getClientId *

```
public String getClientId()
```

Get the client Identifier that is used for all connections that are 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 the 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 to be used when connecting to the queue manager. See
| Table 13 on page 105 for a list of allowed values. We recommend that you
| use the default value (819) for most situations.

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 that describes 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 an 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 an internal error occurs in JMS implementation during the metadata retrieval.

getJMSMinorVersion

```
public int getJMSMinorVersion() throws JMSException
```

Get the JMS minor version number.

Returns:
the JMS minor version number.

Throws:
JMSException - if an internal error occurs in JMS implementation during the metadata retrieval.

getJMSXPropertyNames

```
public java.util.Enumeration getJMSXPropertyNames()
                                             throws JMSException
```

Get an enumeration of the names of the JMSX Properties supported by this connection.

Returns:
an Enumeration of JMSX PropertyNames.

Throws:
JMSException - if an internal error occurs in JMS implementation during the property names retrieval.

getJMSProviderName

```
public java.lang.String getJMSProviderName()
                                            throws JMSException
```

Get the JMS provider name.

Returns:
the JMS provider name.

Throws:
JMSException - if an internal error occurs in JMS implementation during the metadata retrieval.

getProviderVersion

```
public java.lang.String getProviderVersion()
                                             throws JMSException
```

Get the JMS provider version.

Returns:
the JMS provider version.

Throws:
JMSException - if an internal error occurs in JMS implementation during the metadata retrieval.

getProviderMajorVersion

```
public int getProviderMajorVersion() throws JMSException
```

Get the JMS provider major version number.

Returns:
the JMS provider major version number.

Throws:
JMSException - if an internal error occurs in JMS implementation during the metadata retrieval.

getProviderMinorVersion

```
public int getProviderMinorVersion() throws JMSException
```

Get the JMS provider minor version number.

Returns:
the JMS provider minor version number.

Throws:
JMSException - if an internal error occurs in JMS implementation during the metadata retrieval.

toString *

```
public String toString()
```

Overrides:
toString in class Object.

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**, **TemporaryQueue**, **TemporaryTopic**, and **Topic**

MQSeries class: **MQDestination**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQDestination
```

The Destination object encapsulates provider-specific addresses.

See also: **Queue**, **TemporaryQueue**, **TemporaryTopic**, 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 13 on page 105 for a list of allowed values. The default value is 1208 (UTF8).

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 Table 13 on page 105 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 that describes the problem.

This allows a client to be asynchronously notified of a problem. Some Connections only consume messages so they would have no other way to learn their Connection has failed.

Exceptions are delivered when:
- There is a failure in receiving an asynchronous message
- A message throws a runtime exception

Methods

onException

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

Notify user of a JMS exception.

Parameters:

exception - the JMS exception. These are exceptions that result from asynchronous message delivery. Typically, they indicate a problem with receiving a message from the queue manager, or possibly an internal error in the JMS implementation.

MapMessage

public interface **MapMessage**
extends **Message**

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 the message because of an 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 the message because of an 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 the message because of an 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 the message because of an 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 the message because of an 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 the message because of an 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 the message because of an 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 the message because of an 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 the message because of an 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 the message because of an 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 the message because of an 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 the message because of an 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 the message is 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 the message is 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 the message is in read-only mode.

setChar

```
public void setChar(java.lang.String name,
                    char value) throws JMSException
```

Set a Unicode character value with the given name into the Map.

Parameters:

- name - the name of the Unicode character.
- value - the Unicode character value to set in the Map.

Throws:

- JMSException - if JMS fails to write message due to some internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

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 the message is 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 the message is 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 the message is 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 the message is 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 the message is 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 the message is in read-only mode.

setBytes

```
public void setBytes(java.lang.String name,
                     byte[] value,
                     int offset,
                     int length) throws JMSException
```

Set a portion of the byte array value with the given name into the Map.

The array is copied, so the value in the map is not altered by subsequent modifications to the array.

Parameters:

- name - the name of the byte array.
- value - the byte array value to set in the Map.
- offset - the initial offset within the byte array.
- length - the number of bytes to be copied.

Throws:

- JMSException - if JMS fails to write message due to some internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

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 the message is 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 because of an 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 because of an internal JMS error.

See also:
: getJMSMessageID()

getJMSTimestamp

```
public long getJMSTimestamp() throws JMSException
```

Get the message timestamp.

Returns:
: the message timestamp.

Throws:
: JMSException - if JMS fails to get the Timestamp because of an internal JMS error.

See also:
: setJMSTimestamp()

setJMSTimestamp

```
public void setJMSTimestamp(long timestamp)
                                         throws JMSException
```

Set the message timestamp.

| Any value set using this method is ignored when the message is sent, but
| this method can be used to change the value in a received message.

Parameters:
: timestamp - the timestamp for this message.

Throws:
: JMSException - if JMS fails to set the timestamp because of an internal JMS error.

See also:
: getJMSTimestamp()

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 correlation ID because of an 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 the correlation ID because of an 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 the correlation ID because of an internal JMS error.

See also:
setJMSCorrelationID(), getJMSCorrelationIDAsBytes(), setJMSCorrelationIDAsBytes()

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 typical use is to link a response message with its request message.

Note: The use of a byte[] value for JMSCorrelationID is non-portable.

Parameters:
correlationID - the message ID of a message being referred to.

Throws:
JMSException - if JMS fails to set the correlation ID because of an internal JMS error.

See also:
getJMSCorrelationID(), 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 because of an internal JMS error.

See also:
setJMSReplyTo()

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 because of an internal JMS error.

See also:
getJMSReplyTo()

getJMSDestination

```
public Destination getJMSDestination() throws JMSException
```

Get the destination for this message.

Returns:
the destination of this message.

Throws:
JMSException - if JMS fails to get JMS Destination because of an internal JMS error.

See also:
setJMSDestination()

setJMSDestination

```
public void setJMSDestination(Destination destination)
                                                throws JMSexception
```

Set the destination for this message.

| Any value set using this method is ignored when the message is sent, but
| this method can be used to change the value in a received message.

Parameters:
destination - the destination for this message.

Throws:
JMSException - if JMS fails to set JMS Destination because of an internal JMS error.

See also:
getJMSDestination()

getJMSDeliveryMode

```
public int getJMSDeliveryMode() throws JMSException
```

Get the delivery mode for this message.

Returns:
the delivery mode of this message.

Throws:
JMSException - if JMS fails to get JMS DeliveryMode because of an internal JMS error.

See also:
setJMSDeliveryMode(), DeliveryMode

setJMSDeliveryMode

```
public void setJMSDeliveryMode(int deliveryMode)
                                          throws JMSException
```

Set the delivery mode for this message.

Any value set using this method is ignored when the message is sent, but this method can be used to change the value in a received message.

To alter the delivery mode when a message is sent, use the setDeliveryMode method on the QueueSender or TopicPublisher (this method is inherited from MessageProducer).

Parameters:
deliveryMode - the delivery mode for this message.

Throws:
JMSException - if JMS fails to set JMS DeliveryMode because of an internal JMS error.

See also:
getJMSDeliveryMode(), DeliveryMode

getJMSRedelivered

```
public boolean getJMSRedelivered() throws JMSException
```

Get an indication of whether this message is being redelivered.

If a client receives a message with the redelivered indicator set, it is likely, but not guaranteed, that this message was delivered to the client earlier but the client did not acknowledge its receipt at that earlier time.

Returns:
set to true if this message is being redelivered.

Throws:
JMSException - if JMS fails to get JMS Redelivered flag because of an internal JMS error.

See also:
setJMSRedelivered()

setJMSRedelivered

```
public void setJMSRedelivered(boolean redelivered)
                                              throws JMSException
```

Set to indicate whether this message is being redelivered.

| Any value set using this method is ignored when the message is sent, but
| this method can be used to change the value in a received message.

Parameters:
redelivered - an indication of whether this message is being redelivered.

Throws:
JMSException - if JMS fails to set JMSRedelivered flag because of an internal JMS error.

See also:
getJMSRedelivered()

getJMSType

```
public java.lang.String getJMSType() throws JMSException
```

Get the message type.

Returns:
the message type.

Throws:
JMSException - if JMS fails to get JMS message type because of an internal JMS error.

See also:
setJMSType()

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 ensures that it is properly set for those providers that require it.

Parameters:
type - the class of message.

Throws:
JMSException - if JMS fails to set JMS message type because of an internal JMS error.

See also:
getJMSType()

getJMSExpiration

```
public long getJMSExpiration() throws JMSException
```

Get the message's expiration value.

Returns:
the time the message expires. It is the sum of the time-to-live value specified by the client, and the GMT at the time of the send.

Throws:
JMSException - if JMS fails to get JMS message expiration because of an internal JMS error.

See also:
setJMSExpiration()

setJMSExpiration

```
public void setJMSExpiration(long expiration)
                                          throws JMSException
```

Set the message's expiration value.

Any value set using this method is ignored when the message is sent, but this method can be used to change the value in a received message.

Parameters:
expiration - the message's expiration time.

Throws:
JMSException - if JMS fails to set JMS message expiration because of an internal JMS error.

See also:
getJMSExpiration()

getJMSPriority

```
public int getJMSPriority() throws JMSException
```

Get the message priority.

Returns:
the message priority.

Throws:
JMSException - if JMS fails to get JMS message priority because of an internal JMS error.

See also:
setJMSPriority() for priority levels

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 because of an 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 because of an 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 whether a property exists because of an internal JMS error.

getBooleanProperty

```
public boolean getBooleanProperty(java.lang.String name)
                                                    throws JMSException
```

Return the boolean property value with the given name.

Parameters:
name - the name of the boolean property.

Returns:
the boolean property value with the given name.

Throws:
- JMSException - if JMS fails to get the property because of an internal JMS error.
- MessageFormatException - if this type conversion is 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 the property because of an 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 the property because of an 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 the property because of an 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 the property because of an 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 the property because of an 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 the property because of an 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 the property because of an 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, an Integer is returned). If there is no property by this name, a null value is returned.

Throws:
JMSException - if JMS fails to get the property because of an internal JMS error.

getPropertyNames

```
public java.util.Enumeration getPropertyNames()
                                          throws JMSException
```

Return an Enumeration of all the property names.

Returns:
an enumeration of all the names of property values.

Throws:
JMSException - if JMS fails to get the property names because of an internal JMS error.

setBooleanProperty

```
public void setBooleanProperty(java.lang.String name,
                               boolean value) throws JMSException
```

Set a boolean property value with the given name into the Message.

Parameters:
- name - the name of the boolean property.
- value - the boolean property value to set in the Message.

Throws:
- JMSException - if JMS fails to set Property because of an internal JMS error.
- MessageNotWriteableException - if the properties are read-only.

setByteProperty

```
public void setByteProperty(java.lang.String name,
                            byte value) throws JMSException
```

Set a byte property value with the given name into the Message.

Parameters:
- name - the name of the byte property.
- value - the byte property value to set in the Message.

Throws:
- JMSException - if JMS fails to set Property because of an internal JMS error.
- MessageNotWriteableException - if the properties are read-only.

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 because of an internal JMS error.
- MessageNotWriteableException - if the 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 because of an internal JMS error.
- MessageNotWriteableException - if the 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 because of an internal JMS error.
- MessageNotWriteableException - if the 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 the property because of an internal JMS error.
- MessageNotWriteableException - if the properties are read-only.

setDoubleProperty

```
public void setDoubleProperty(java.lang.String name,
                              double value) throws JMSException
```

Set a double property value with the given name into the Message.

Parameters:

- name - the name of the double property.
- value - the double property value to set in the Message.

Throws:

- JMSException - if JMS fails to set the property because of an internal JMS error.
- MessageNotWriteableException - if the 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 the property because of an internal JMS error.
- MessageNotWriteableException - if the 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 because of an internal JMS error.
- MessageFormatException - if the object is invalid.
- MessageNotWriteableException - if the 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 because of an internal JMS error.

clearBody

```
public void clearBody() throws JMSException
```

Clear out the message body. All other parts of the message are left untouched.

Throws:

JMSException - if JMS fails to because of an internal JMS error.

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 the message selector because of a 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 the message listener because of a 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 because of a 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 because of an 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 because of an error.

receiveNoWait

```
public Message receiveNoWait() throws JMSException
```

Receive the next message if one is immediately available.

Returns:
the next message produced for this message consumer, or null if one is not available.

Throws:
JMSException - if JMS fails to receive the next message because of an error.

close

```
public void close() throws JMSException
```

Because a provider may allocate some resources outside of the JVM on behalf of a MessageConsumer, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

This call blocks until a receive or message listener in progress has completed.

Throws:
JMSException - if JMS fails to close the consumer because of an error.

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 implementation.

Parameters:
value - indicates whether message IDs are disabled.

Throws:
JMSException - if JMS fails to set the disabled message ID because of an internal error.

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 the disabled message ID because of an 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 implementation.

Parameters:
value - indicates whether message timestamps are disabled.

Throws:
JMSException - if JMS fails to set the disabled message timestamp because of an internal error.

getDisableMessageTimestamp

```
public boolean getDisableMessageTimestamp()
                                           throws JMSException
```

Get an indication of whether message timestamps are disabled.

Returns:
an indication of whether message IDs are disabled.

Throws:
JMSException - if JMS fails to get the disabled message timestamp because of an internal error.

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 the delivery mode because of an 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 the delivery mode because of an 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 the priority because of an 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 the priority because of an 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 the Time to Live because of an 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 the Time to Live because of an internal error.

See also:
setTimeToLive

close

```
public void close() throws JMSException
```

Because a provider may allocate some resources outside of the JVM on behalf of a MessageProducer, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Throws:

JMSException - if JMS fails to close the producer because of an error.

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 MQQueue instance to hold the browse cursor. The queue is closed once the cursor has moved off the end of the queue.

There is no way to reset an instance of this class - it acts as a 'one-shot' mechanism.

See also: **MQQueueBrowser**

Methods

hasMoreElements

```
public boolean hasMoreElements()
```

Indicate whether another message can be returned.

nextElement

```
public Object nextElement() throws NoSuchElementException
```

Return the current message.

If `hasMoreElements()` returns 'true', `nextElement()` always returns a message. It is possible for the returned message to pass its expiry date between the `hasMoreElements()` and the `nextElement` calls.

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 the object because of an internal JMS error.
- MessageFormatException - if object serialization fails.
- MessageNotWriteableException - if the message is 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 the object because of an 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 173.

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 the queue name because of an 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 because of a 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 because of a JMS error.

getEnumeration

```
public java.util.Enumeration getEnumeration() throws JMSException
```

Get an enumeration for browsing the current queue messages in the order that they would be received.

Returns:
an enumeration for browsing the messages.

Throws:
JMSException - if JMS fails to get the enumeration for this browser because of a JMS error.

Note: If the browser is created for a nonexistent queue, this is not detected until the first call to getEnumeration.

close

```
public void close() throws JMSException
```

Because a provider may allocate some resources outside of the JVM on behalf of a QueueBrowser, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Throws:

JMSException - if a JMS fails to close this Browser because of a JMS error.

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.

See also: **Connection**, **QueueConnectionFactory**, and **XAQueueConnection**

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 because of an 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. This is an expert facility that is not used by regular JMS clients.

Parameters:

- queue - the queue to access.
- messageSelector - only messages with properties that match the message selector expression are delivered.
- sessionPool - the server session pool to associate with this connection consumer.
- maxMessages - the maximum number of messages that can be assigned to a server session at one time.

Returns:
the connection consumer.

Throws:

- JMSException - if the JMS Connection fails to create a connection consumer because of an internal error, or invalid arguments for sessionPool and messageSelector.
- InvalidSelectorException - if the message selector is invalid.

See Also:
ConnectionConsumer

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 point-to-point provider.

| See also: **ConnectionFactory** and **XAQueueConnectionFactory**

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 because of an internal error.
- JMSSecurityException - if client authentication fails because of an 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 because of an internal error.
- JMSSecurityException - if client authentication fails because of an 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.

setMessage Retention*

```
public void setMessageRetention(int x) throws JMSException
```

Set method for messageRetention attribute.

Parameters:
Valid values are:
- JMSC.MQJMS_MRET_YES - unwanted messages remain on the input queue.
- JMSC.MQJMS_MRET_NO - uwanted messages are dealt with according to their disposition options.

getMessage Retention*

```
public void getMessageRetention()
```

Get method for messageRetention attribute.

Returns:
- JMSC.MQJMS_MRET_YES - unwanted messages remain on the input queue.
- JMSC.MQJMS_MRET_NO - uwanted messages are dealt with according to their disposition options.

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 methods from **MQMessageConsumer**.
- receive
- receiveNoWait
- close
- getMessageListener
- setMessageListener

Methods

getQueue

```
public Queue getQueue() throws JMSException
```

Get the queue associated with this queue receiver.

Returns:
the queue.

Throws:
JMSException - if JMS fails to get queue for this queue receiver because of an internal error.

QueueRequestor

public class **QueueRequestor**
extends **java.lang.Object**

```
java.lang.Object
    |
    +----javax.jms.QueueRequestor
```

JMS provides this QueueRequestor helper class to simplify making service requests. The QueueRequestor constructor is given a non-transacted QueueSession and a destination Queue. It creates a TemporaryQueue for the responses, and provides a request() method that sends the request message and waits for its reply. Users are free to create more sophisticated versions.

See also: **TopicRequestor**

Constructors

QueueRequestor

```
public QueueRequestor(QueueSession session,
                      Queue queue)
                              throws JMSException
```

This implementation assumes that the session parameter is non-transacted and either AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE.

Parameters:
- session - the queue session the queue belongs to.
- queue - the queue to perform the request/reply call on.

Throws:
JMSException - if a JMS error occurs.

Methods

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
```

Because a provider may allocate some resources outside of the JVM on behalf of a QueueRequestor, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Note: This method closes the Session object passed to the QueueRequestor constructor.

Throws:
JMSException - if a JMS error occurs.

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 that is 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 because of an internal error.

send

```
public void send(Message message) throws JMSException
```

Send a message to the queue. Use the QueueSender's default delivery mode, time to live, and priority.

Parameters:
message - the message to be sent.

Throws:
- JMSException - if JMS fails to send the message because of an error.
- MessageFormatException - if an invalid message is 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 because of an internal error.
- MessageFormatException - if an invalid message is 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, time to live, and priority.

Note: This method can only be used with unidentified QueueSenders.

Parameters:
- queue - the queue that this message should be sent to.
- message - the message to be sent.

Throws:
- JMSException - if JMS fails to send the message because of an internal error.
- MessageFormatException - if an invalid message is 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 because of an internal error.
- MessageFormatException - if an invalid message is specified.
- InvalidDestinationException - if a client uses this method with an invalid queue.

close *

```
public void close() throws JMSException
```

Because a provider may allocate some resources outside of the JVM on behalf of a QueueSender, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Throws:

JMSException if JMS fails to close the producer due to some error.

Overrides:

close in class MQMessageProducer.

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 to create 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 173.

Note: Clients that depend on this ability are not portable.

Parameters:
queueName - the name of this queue.

Returns:
a Queue with the given name.

Throws:
JMSException - if a session fails to create a queue because of a JMS error.

createReceiver

```
public QueueReceiver createReceiver(Queue queue)
                                              throws JMSException
```

Create a QueueReceiver to receive messages from the specified queue.

Parameters:
queue - the queue to access.

Throws:
- JMSException - if a session fails to create a receiver because of a JMS error.
- InvalidDestinationException - if an invalid Queue is 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 that match the message selector expression are delivered.

Throws:
- JMSException - if a session fails to create a receiver because of a JMS error.
- InvalidDestinationException - if an invalid Queue is 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 because of a JMS error.
- InvalidDestinationException - if an invalid Queue is 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 because of a JMS error.
- InvalidDestinationException - if an invalid Queue is specified.

createBrowser

```
public QueueBrowser createBrowser(Queue queue,
                                  java.lang.String messageSelector)
                                  throws JMSException
```

Create a QueueBrowser to peek at the messages on the specified queue.

Parameters:
- queue - the queue to access.
- messageSelector - only messages with properties that match the message selector expression are delivered.

Throws:

- JMSException - if a session fails to create a browser because of a JMS error.
- InvalidDestinationException - if an invalid Queue is specified.
- InvalidSelectorException - if the message selector is invalid.

createTemporaryQueue

```
public TemporaryQueue createTemporaryQueue()
                                    throws JMSException
```

Create a temporary queue. Its lifetime will be that of the QueueConnection unless deleted earlier.

Returns:

a temporary queue.

Throws:

JMSException - if a session fails to create a Temporary Queue because of a JMS error.

Session

public interface **Session**
extends **java.lang.Runnable**
Subinterfaces: **QueueSession**, **TopicSession**, **XAQueueSession**, **XASession**, and **XATopicSession**

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**, **TopicSession**, **XAQueueSession**, **XASession**, and **XATopicSession**

Fields

AUTO_ACKNOWLEDGE

```
public static final int AUTO_ACKNOWLEDGE
```

With this acknowledgement mode, the session automatically acknowledges a message when it has either successfully returned from a call to receive, or the message listener it has called to process the message successfully returns.

CLIENT_ACKNOWLEDGE

```
public static final int CLIENT_ACKNOWLEDGE
```

With this acknowledgement mode, the client acknowledges a message by calling a message's acknowledge method.

DUPS_OK_ACKNOWLEDGE

```
public static final int DUPS_OK_ACKNOWLEDGE
```

This acknowledgement mode instructs the session to lazily acknowledge the delivery of messages.

Methods

createBytesMessage

```
public BytesMessage createBytesMessage()
                              throws JMSException
```

Create a BytesMessage. A BytesMessage is used to send a message containing a stream of uninterpreted bytes.

Throws:
JMSException - if JMS fails to create this message because of an internal error.

createMapMessage

```
public MapMessage createMapMessage() throws JMSException
```

Create a MapMessage. A MapMessage is used to send a self-defining set of name-value pairs, where names are Strings, and values are Java primitive types.

Throws:
JMSException - if JMS fails to create this message because of an internal error.

createMessage

```
public Message createMessage() throws JMSException
```

Create a Message. The Message interface is the root interface of all JMS messages. It holds all the standard message header information. It can be sent when a message containing only header information is sufficient.

Throws:
JMSException - if JMS fails to create this message because of an internal error.

createObjectMessage

```
public ObjectMessage createObjectMessage()
                            throws JMSException
```

Create an ObjectMessage. An ObjectMessage is used to send a message that contains a serializable Java object.

Throws:
JMSException - if JMS fails to create this message because of an internal error.

createObjectMessage

```
public ObjectMessage createObjectMessage
                           (java.io.Serializable object)
                            throws JMSException
```

Create an initialized ObjectMessage. An ObjectMessage is used to send a message that contains a serializable Java object.

Parameters:
object - the object to use to initialize this message.

Throws:
JMSException - if JMS fails to create this message because of an internal error.

createStreamMessage

```
public StreamMessage createStreamMessage()
                             throws JMSException
```

Create a StreamMessage. A StreamMessage is used to send a self-defining stream of Java primitives.

Throws:
JMSException if JMS fails to create this message because of an internal error.

createTextMessage

```
public TextMessage createTextMessage() throws JMSException
```

Create a TextMessage. A TextMessage is used to send a message containing a String.

Throws:
JMSException - if JMS fails to create this message because of an internal error.

createTextMessage

```
public TextMessage createTextMessage
                        (java.lang.String string)
                        throws JMSException
```

Create an initialized TextMessage. A TextMessage is used to send a message containing a String.

Parameters:
string - the string used to initialize this message.

Throws:
JMSException - if JMS fails to create this message because of an internal error.

getTransacted

```
public boolean getTransacted() throws JMSException
```

Is the session in transacted mode?

Returns:
true if the session is in transacted mode.

Throws:
JMSException - if JMS fails to return the transaction mode because of an internal error in JMS Provider.

commit

```
public void commit() throws JMSException
```

Commit all messages done in this transaction and release any locks currently held.

Throws:
- JMSException - if JMS implementation fails to commit the transaction because of an internal error.
- TransactionRolledBackException - if the transaction gets rolled back because of an internal error during commit.

rollback

```
public void rollback() throws JMSException
```

Roll back any messages done in this transaction and release any locks currently held.

Throws:
JMSException - if the JMS implementation fails to roll back the transaction because of an internal error.

Session

close

```
public void close() throws JMSException
```

Because a provider may allocate some resources outside of the JVM on behalf of a Session, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Closing a transacted session rolls back any in-progress transaction. Closing a session automatically closes its message producers and consumer, so there is no need to close them individually.

Throws:
JMSException - if the JMS implementation fails to close a Session because of an internal error.

recover

```
public void recover() throws JMSException
```

Stop message delivery in this session, and restart sending messages with the oldest unacknowledged message.

Throws:
JMSException - if the JMS implementation fails to stop message delivery and restart message send because of an internal error.

getMessageListener

```
public MessageListener getMessageListener()
                                 throws JMSException
```

| Return the session’s distinguished message listener.

Returns:
the message listener associated with this session.

Throws:
JMSException - if JMS fails to get the message listener because of an internal error in the JMS Provider.

See also:
setMessageListener

setMessageListener

```
public void setMessageListener(MessageListener listener)
                                                throws JMSException
```

Set the session’s distinguished message listener. When it is set, no other form of message receipt in the session can be used. However, all forms of sending messages are still supported.

| This is an expert facility that is not used by regular JMS clients.

Parameters:
listener - the message listener to associate with this session.

Throws:
JMSException - if JMS fails to set the message listener because of an internal error in the JMS Provider.

See also:
| getMessageListener, ServerSessionPool, ServerSession

run

```
public void run()
```

This method is intended for use only by application servers.

Specified by:
`run` in the interface java.lang.Runnable

See also:
ServerSession

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 the message because of an internal JMS error.
- MessageEOFException - if an end of message stream is received.
- MessageFormatException - if this type conversion is invalid.
- MessageNotReadableException - if the message is in write-only mode.

readByte

```
public byte readByte() throws JMSException
```

Read a byte value from the stream message.

Returns:
the next byte from the stream message as an 8-bit byte.

Throws:
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if an end of message stream is received.
- MessageFormatException - if this type conversion is invalid.
- MessageNotReadableException - if the message is 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 the message because of an internal JMS error.
- MessageEOFException - if an end of message stream is received.
- MessageFormatException - if this type conversion is invalid.
- MessageNotReadableException - if the message is 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 the message because of an internal JMS error.
- MessageEOFException - if an end of message stream is received.
- MessageFormatException if this type conversion is invalid.
- MessageNotReadableException if the message is in write-only mode.

readInt

```
public int readInt() throws JMSException
```

Read a 32-bit integer from the stream message.

Returns:
a 32-bit integer value from the stream message, interpreted as an int.

Throws:
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if an end of message stream is received.
- MessageFormatException if this type conversion is invalid.
- MessageNotReadableException if the message is in write-only mode.

readLong

```
public long readLong() throws JMSException
```

Read a 64-bit integer from the stream message.

Returns:
a 64-bit integer value from the stream message, interpreted as a long.

Throws:

- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if an end of message stream
- MessageFormatException if this type conversion is invalid.
- MessageNotReadableException if the message is in write-only mode.

readFloat

```
public float readFloat() throws JMSException
```

Read a float from the stream message.

Returns:
a float value from the stream message.

Throws:

- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if an end of message stream
- MessageFormatException if this type conversion is invalid.
- MessageNotReadableException - if the message is 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 the message because of an internal JMS error.
- MessageEOFException - if an end of message stream is received.
- MessageFormatException - if this type conversion is invalid.
- MessageNotReadableException - if the message is 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 the message because of an internal JMS error.
- MessageEOFException - if an end of message stream is received.
- MessageFormatException - if this type conversion is invalid.
- MessageNotReadableException - if the message is 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[] object (the read buffer). If the buffer size is less than, or equal to, the size of the data in the message field, an application must make further calls to this method to retrieve the remainder of the data. Once the first readBytes call on a byte[] field value has been done, the full value of the field must be read before it is valid to read the next field. An attempt to read the next field before that has been done will throw a MessageFormatException.

Parameters:
value - the buffer into which the data is read.

Returns:
the total number of bytes read into the buffer, or -1 if there is no more data because the end of the byte field has been reached.

Throws:
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if an end of message stream is received.
- MessageFormatException - if this type conversion is invalid.
- MessageNotReadableException - if the message is 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, an Integer is returned).

Throws:
- JMSException - if JMS fails to read the message because of an internal JMS error.
- MessageEOFException - if an end of message stream is received.
- NotReadableException - if the message is 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 the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

writeByte

```
public void writeByte(byte value) throws JMSException
```

Write out a byte to the stream message.

Parameters:
value - the byte value to be written.

Throws:

- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

writeShort

```
public void writeShort(short value) throws JMSException
```

Write a short to the stream message.

Parameters:
value - the short to be written.

Throws:

- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

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 the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

writeInt

```
public void writeInt(int value) throws JMSException
```

Write an int to the stream message.

Parameters:
value - the int to be written.

Throws:

- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

writeLong

```
public void writeLong(long value) throws JMSException
```

Write a long to the stream message.

Parameters:
value - the long to be written.

Throws:
- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

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 the message because of an internal JMS error.
- MessageNotWriteableException - if the message is 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 the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

writeString

```
public void writeString(java.lang.String value)
                                          throws JMSException
```

Write a string to the stream message.

Parameters:
value - the String value to be written.

Throws:
- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

writeBytes

```
public void writeBytes(byte[] value) throws JMSException
```

Write a byte array to the stream message.

Parameters:
value - the byte array to be written.

Throws:
- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

writeBytes

```
public void writeBytes(byte[] value,
                       int offset,
                       int length) throws JMSException
```

Write a portion of a byte array to the stream message.

Parameters:
- value - the byte array value to be written.
- offset - the initial offset within the byte array.
- length - the number of bytes to use.

Throws:
- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.

writeObject

```
public void writeObject(java.lang.Object value)
                                     throws JMSException
```

Write a Java object to the stream message. This method only works for object primitive types (Integer, Double, Long, for example), Strings, and byte arrays.

Parameters:
value - the Java object to be written.

Throws:
- JMSException - if JMS fails to write the message because of an internal JMS error.
- MessageNotWriteableException - if the message is in read-only mode.
- MessageFormatException - if the object is 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 because of an internal JMS error.
- MessageFormatException - if the 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 that is 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, a JMSException will be thrown.

Throws:

JMSException - if JMS implementation fails to delete a TemporaryQueue because of an internal error.

TemporaryTopic

public interface **TemporaryTopic**
extends **Topic**

MQSeries class: **MQTemporaryTopic**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQDestination
           |
           +----com.ibm.mq.jms.MQTopic
                   |
                   +----com.ibm.mq.jms.MQTemporaryTopic
```

A TemporaryTopic is a unique Topic object created for the duration of a TopicConnection and can only be consumed by consumers of that connection.

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, a JMSException will be thrown.

Throws:

JMSException - if JMS implementation fails to delete a TemporaryTopic because of an 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 because of an internal JMS error.
- MessageNotWriteableException - if the message is 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 the text because of an 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 in “Creating topics at runtime” on page 182.)

Note: Clients that depend upon the name are not portable.

Returns:
the topic name.

Throws:
JMSException - if JMS implementation for Topic fails to return the topic name because of an internal error.

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 topic.

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.

setBrokerDurSubQueue *

```
public void setBrokerDurSubQueue(String x) throws JMSException
```

Set method for brokerDurSubQueue attribute.

Parameters:
brokerDurSubQueue - the name of the durable subscription queue to use.

getBrokerDurSubQueue *

```
public String getBrokerDurSubQueue()
```

Get method for brokerDurSubQueue attribute.

Returns:
the name of the durable subscription queue (the brokerDurSubQueue) to use.

setBrokerCCDurSubQueue *

```
public void setBrokerCCDurSubQueue(String x) throws JMSException
```

Set method for brokerCCDurSubQueue attribute.

Parameters:
brokerCCDurSubQueue - the name of the durable subscription queue to use for a ConnectionConsumer.

getBrokerCCDurSubQueue *

```
public String getBrokerCCDurSubQueue()
```

Get method for brokerCCDurSubQueue attribute.

Returns:
the name of the durable subscription queue (the brokerCCDurSubQueue) to use for a ConnectionConsumer.

TopicConnection

public interface **TopicConnection**
extends **Connection**
| Subinterfaces: **XATopicConnection**

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 Publish/Subscribe provider.

| See also: **Connection**, **TopicConnectionFactory**, and **XATopicConnection**

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 because of an internal error, or a lack of support for the 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
| that is not used by regular JMS clients.

Parameters:

- topic - the topic to access.
- messageSelector - only messages with properties that match the message selector expression are delivered.
- sessionPool - the server session pool to associate with this connection consumer.
- maxMessages - the maximum number of messages that can be assigned to a server session at one time.

Returns:
the connection consumer.

Throws:

- JMSException - if the JMS Connection fails to create a connection consumer because of an internal error, or because of 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 that is not used by regular JMS clients.

Parameters:

- topic - the topic to access.
- subscriptionName - name of the durable subscription.
- messageSelector - only messages with properties that match the message selector expression are delivered.
- sessionPool - the server session pool to associate with this durable connection consumer.
- maxMessages - the maximum number of messages that can be assigned to a server session at one time.

Returns:
the durable connection consumer.

Throws:

- JMSException - if the JMS Connection fails to create a connection consumer because of an internal error, or because of invalid arguments for sessionPool and messageSelector.
- InvalidSelectorException - if the message selector is invalid.

See also:
ConnectionConsumer

TopicConnectionFactory

public interface **TopicConnectionFactory**
extends **ConnectionFactory**
Subinterfaces: **XATopicConnectionFactory**

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** and **XATopicConnectionFactory**

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 because of an internal error.
- JMSSecurityException - if client authentication fails because of an 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 because of an internal error.
- JMSSecurityException - if client authentication fails because of an 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.

setBrokerSubQueue *

```
public void setBrokerSubQueue(String x) throws JMSException
```

Set method for brokerSubQueue attribute.

Parameters:
brokerSubQueue - the name of the non-durable subscription queue to use.

getBrokerSubQueue *

```
public String getBrokerSubQueue()
```

Get method for brokerSubQueue attribute.

Returns:
the name of the non-durable subscription queue to use.

setBrokerCCSubQueue *

```
public void setBrokerCCSubQueue(String x) throws JMSException
```

Set method for brokerCCSubQueue attribute.

Parameters:
brokerSubQueue - the name of the non-durable subscription queue to use for a ConnectionConsumer.

getBrokerCCSubQueue *

```
public String getBrokerCCSubQueue()
```

Get method for brokerCCSubQueue attribute.

Returns:
the name of the non-durable subscription queue to use for a ConnectionConsumer.

setBrokerVersion *

```
public void setBrokerVersion(int x) throws JMSException
```

Set method for brokerVersion attribute.

Parameters:
brokerVersion - the broker's version number.

getBrokerVersion *

```
public int getBrokerVersion()
```

Get method for brokerVersion attribute.

Returns:
the broker's version number.

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 the topic for this topic publisher because of an internal error.

publish

```
public void publish(Message message) throws JMSException
```

Publish a Message to the topic Use the topic's default delivery mode, time to live, and priority.

Parameters:
message - the message to publish

Throws:
- JMSException - if JMS fails to publish the message because of an internal error.
- MessageFormatException - if an invalid message is 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 because of an internal error.
- MessageFormatException - if an invalid message is 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, time to live, and priority.

Parameters:

- topic - the topic to publish this message to.
- message - the message to send.

Throws:

- JMSException - if JMS fails to publish the message because of an internal error.
- MessageFormatException - if an invalid message is 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 because of an internal error.
- MessageFormatException - if an invalid message is specified.
- InvalidDestinationException - if a client uses this method with an invalid topic.

close *

```
public void close() throws JMSException
```

Because a provider may allocate some resources outside of the JVM on behalf of a TopicPublisher, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Throws:
JMSException if JMS fails to close the producer because of an error.

Overrides:
close in class MQMessageProducer.

TopicRequestor

public class **TopicRequestor**
extends **java.lang.Object**

```
java.lang.Object
    |
    +----javax.jms.TopicRequestor
```

JMS provides this TopicRequestor class to assist with making service requests.

The TopicRequestor constructor is given a non-transacted TopicSession and a destination Topic. It creates a TemporaryTopic for the responses, and provides a request() method that sends the request message and waits for its reply. Users are free to create more sophisticated versions.

Constructors

TopicRequestor

```
public TopicRequestor(TopicSession session,
                      Topic topic) throws JMSException
```

Constructor for the TopicRequestor class. This implementation assumes that the session parameter is non-transacted, and either AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE.

Parameters:
- session - the topic session the topic belongs to.
- topic - the topic to perform the request/reply call on.

Throws:
JMSException - if a JMS error occurs.

Methods

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
```

Because a provider may allocate some resources outside of the JVM on behalf of a TopicRequestor, clients should close them when they are not needed. You cannot rely on garbage collection to reclaim these resources eventually, because this may not occur soon enough.

Note: This method closes the Session object passed to the TopicRequestor constructor.

Throws:
JMSException - if a JMS error occurs.

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 in "Creating topics at runtime" on page 182.) This allows the creation of a topic with a provider specific name.

Note: Clients that depend on this ability are not portable.

Parameters:
topicName - the name of this topic.

Returns:
a Topic with the given name.

Throws:
JMSException - if a session fails to create a topic because of a JMS error.

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 because of a JMS error.
- InvalidDestinationException - if an invalid Topic is 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 that match the message selector expression are delivered. This value may be null.
- noLocal - if set, inhibits the delivery of messages published by its own connection.

Throws:

- JMSException - if a session fails to create a subscriber because of a JMS error or invalid selector.
- InvalidDestinationException - if an invalid Topic is 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 because of a JMS error.
- InvalidDestinationException - if an invalid Topic is 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 that match the message selector expression are delivered. This value may be null.
- noLocal - if set, inhibits the delivery of messages published by its own connection.

Throws:

- JMSException - if a session fails to create a subscriber because of a JMS error or invalid selector.
- InvalidDestinationException - if an invalid Topic is 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 because of a JMS error.
- InvalidDestinationException - if an invalid Topic is 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 because of a JMS error.

unsubscribe

```
public void unsubscribe(java.lang.String name)
                                        throws JMSException
```

Unsubscribe a durable subscription that has been created by a client.

| **Note:** Do not use this method while an active subscription exists. You
| must close() your subscriber first.

Parameters:

name - the name used to identify this subscription.

Throws:

- JMSException - if JMS fails to unsubscribe the durable subscription because of a JMS error.
- InvalidDestinationException - if an invalid Topic is 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 because of an 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 because of an internal error.

XAConnection

public interface **XAConnection**
Subinterfaces: **XAQueueConnection** and **XATopicConnection**

MQSeries class: **MQXAConnection**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQXAConnection
```

XAConnection extends the capability of Connection by providing an XASession. Refer to “Appendix E. JMS JTA/XA interface with WebSphere” on page 357 for details about how MQ JMS uses XA classes.

See also: **XAQueueConnection** and **XATopicConnection**

XAConnectionFactory

public interface **XAConnectionFactory**
Subinterfaces: **XAQueueConnectionFactory** and **XATopicConnectionFactory**

MQSeries class: **MQXAConnectionFactory**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQXAConnectionFactory
```

Some application servers provide support to group JTS-capable resource use into a distributed transaction. To include JMS transactions in a JTS transaction, an application server requires a JTS-aware JMS provider. A JMS provider exposes its JTS support by using a JMS XAConnectionFactory, which an application server uses to create XASessions. XAConnectionFactories are JMS-administered objects just like ConnectionFactories. It is expected that application servers use JNDI to find them.

Refer to “Appendix E. JMS JTA/XA interface with WebSphere” on page 357 for details about how MQ JMS uses XA classes.

See also: **XAQueueConnectionFactory** and **XATopicConnectionFactory**

XAQueueConnection

public interface **XAQueueConnection**
extends **QueueConnection** and **XAConnection**

MQSeries class: **MQXAQueueConnection**

```
java.lang.Object
    |
    +----com.ibm.mq.jms.MQConnection
            |
            +----com.ibm.mq.jms.MQQueueConnection
                    |
                    +----com.ibm.mq.jms.MQXAQueueConnection
```

XAQueueConnection provides the same create options as QueueConnection. The only difference is that, by definition, an XAConnection is transacted. Refer to "Appendix E. JMS JTA/XA interface with WebSphere" on page 357 for details about how MQ JMS uses XA classes.

See also: **XAConnection** and **QueueConnection**

Methods

createXAQueueSession

```
public XAQueueSession createXAQueueSession()
```

Create an XAQueueSession.

Throws:
JMSException - if JMS Connection fails to create an XA queue session because of an internal error.

createQueueSession

```
public QueueSession createQueueSession(boolean transacted,
                                       int acknowledgeMode)
                                                throws JMSException
```

Create a QueueSession.

Parameters:

- transacted - if true, the session is transacted.
- acknowledgeMode - indicates whether the consumer or the client will acknowledge any messages it receives. Possible values are:
  Session.AUTO_ACKNOWLEDGE
  Session.CLIENT_ACKNOWLEDGE
  Session.DUPS_OK_ACKNOWLEDGE

  This parameter is ignored if the session is transacted.

Returns:
a newly created queue session (note that this is not an XA queue session).

Throws:
JMSException - if JMS Connection fails to create a queue session because of an internal error.

XAQueueConnectionFactory

public interface **XAQueueConnectionFactory**
extends **QueueConnectionFactory** and **XAConnectionFactory**

MQSeries class: **MQXAQueueConnectionFactory**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQConnectionFactory
           |
           +----com.ibm.mq.jms.MQQueueConnectionFactory
                   |
                   +----com.ibm.mq.jms.MQXAQueueConnectionFactory
```

An XAQueueConnectionFactory provides the same create options as a QueueConnectionFactory. Refer to "Appendix E. JMS JTA/XA interface with WebSphere" on page 357 for details about how MQ JMS uses XA classes.

See also: **QueueConnectionFactory** and **XAConnectionFactory**

Methods

createXAQueueConnection

```
public XAQueueConnection createXAQueueConnection()
                                          throws JMSException
```

Create an XAQueueConnection using the default user identity. The connection is created in stopped mode. No messages are delivered until the Connection.start method is called explicitly.

Returns:
a newly created XA queue connection.

Throws:

- JMSException - if the JMS Provider fails to create an XA queue connection because of an internal error.
- JMSSecurityException - if client authentication fails because of an invalid user name or password.

createXAQueueConnection

```
public XAQueueConnection createXAQueueConnection
                                 (java.lang.String userName,
                                  java.lang.String password)
                                                      throws JMSException
```

Create an XA queue connection using a specific user identity. The connection is created in stopped mode. No messages are delivered until the Connection.start method is called explicitly.

Parameters:

- userName - the user name of the caller.
- password - the password for the caller.

Returns:
a newly created XA queue connection.

Throws:

- JMSException - if the JMS Provider fails to create an XA queue connection because of an internal error.
- JMSSecurityException - if client authentication fails because of an invalid user name or password.

XAQueueSession

public interface **XAQueueSession**
extends **XASession**

MQSeries class: **MQXAQueueSession**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQXASession
           |
           +----com.ibm.mq.jms.MQXAQueueSession
```

An XAQueueSession provides a regular QueueSession that can be used to create QueueReceivers, QueueSenders and QueueBrowsers. Refer to "Appendix E. JMS JTA/XA interface with WebSphere" on page 357 for details about how MQ JMS uses XA classes.

The XAResource that corresponds to the QueueSession can be obtained by calling the getXAResource method, which is is inherited from XASession.

See also: **XASession**

Methods

getQueueSession

```
public QueueSession  getQueueSession()
                                throws JMSException
```

Get the queue session associated with this XAQueueSession.

Returns:
the queue session object.

Throws:
JMSException - if a JMS error occurs.

XASession

public interface **XASession**
extends **Session**
Subinterfaces: **XAQueueSession** and **XATopicSession**

MQSeries class: **MQXASession**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQXASession
```

XASession extends the capability of Session by adding access to a JMS provider's support for JTA. This support takes the form of a javax.transaction.xa.XAResource object. The functionality of this object closely resembles that defined by the standard X/Open XA Resource interface.

An application server controls the transactional assignment of an XASession by obtaining its XAResource. It uses the XAResource to assign the session to a transaction, prepare and commit work on the transaction, and so on.

An XAResource provides some fairly sophisticated facilities such as interleaving work on multiple transactions and recovering a list of transactions in progress.

A JTA-aware JMS provider must fully implement this functionality. To do this, a JMS provider could either use the services of a database that supports XA, or implement this functionality from scratch.

A client of the application server is given what appears to be a regular JMS Session. Behind the scenes, the application server controls the transaction management of the underlying XASession.

Refer to "Appendix E. JMS JTA/XA interface with WebSphere" on page 357 for details about how MQ JMS uses XA classes.

See also: **XAQueueSession** and **XATopicSession**

Methods

getXAResource

```
public javax.transaction.xa.XAResource getXAResource()
```

Return an XA resource to the caller.

Returns:
an XA resource to the caller.

getTransacted

```
public boolean getTransacted()
                      throws JMSException
```

Always returns true.

Specified by:
getTransacted in the Session interface.

Returns:
true - if the session is in transacted mode.

Throws:
JMSException - if JMS fails to return the transaction mode because of an internal error in the JMS Provider.

commit

```
public void commit()
            throws JMSException
```

This method should not be called for an XASession object. If it is called, it throws a TransactionInProgressException.

Specified by:
commit in the Session interface.

Throws:
TransactionInProgressException - if this method is called on an XASession.

rollback

```
public void rollback()
              throws JMSException
```

This method should not be called for an XASession object. If it is called, it throws a TransactionInProgressException.

Specified by:
rollback in the Session interface.

Throws:
TransactionInProgressException - if this method is called on an XASession.

XATopicConnection

public interface **XATopicConnection**
extends **TopicConnection** and **XAConnection**

MQSeries class: **MQXATopicConnection**

```
java.lang.Object
     |
     +----com.ibm.mq.jms.MQConnection
              |
              +----com.ibm.mq.jms.MQTopicConnection
                       |
                       +----com.ibm.mq.jms.MQXATopicConnection
```

An XATopicConnection provides the same create options as TopicConnection. The only difference is that, by definition, an XAConnection is transacted. Refer to "Appendix E. JMS JTA/XA interface with WebSphere" on page 357 for details about how MQ JMS uses XA classes.

See also: **TopicConnection** and **XAConnection**

Methods

createXATopicSession

```
public XATopicSession createXATopicSession()
                                          throws JMSException
```

Create an XATopicSession.

Throws:
JMSException - if the JMS Connection fails to create an XA topic session because of an internal error.

createTopicSession

```
public TopicSession createTopicSession(boolean transacted,
                                       int acknowledgeMode)
                                  throws JMSException
```

Create a TopicSession.

Specified by:
createTopicSession in interface TopicConnection.

Parameters:
- transacted - if true, the session is transacted.
- acknowledgeMode - one of:
  Session.AUTO_ACKNOWLEDGE
  Session.CLIENT_ACKNOWLEDGE
  Session.DUPS_OK_ACKNOWLEDGE

  Indicates whether the consumer or the client will acknowledge any messages it receives. This parameter will be ignored if the session is transacted.

Returns:
a newly created topic session (note that this is not an XA topic session).

Throws:

JMSException - if JMS Connection fails to create a topic session because of an internal error.

XATopicConnectionFactory

public interface **XATopicConnectionFactory**
extends **TopicConnectionFactory** and **XAConnectionFactory**

MQSeries class: **MQXATopicConnectionFactory**

```
java.lang.Object
    |
    +----com.ibm.mq.jms.MQConnectionFactory
            |
            +----com.ibm.mq.jms.MQTopicConnectionFactory
                    |
                    +----com.ibm.mq.jms.MQXATopicConnectionFactory
```

An XATopicConnectionFactory provides the same create options as TopicConnectionFactory. Refer to “Appendix E. JMS JTA/XA interface with WebSphere” on page 357 for details about how MQ JMS uses XA classes.

See also: **TopicConnectionFactory** and **XAConnectionFactory**

Methods

createXATopicConnection

```
public XATopicConnection createXATopicConnection()
                                     throws JMSException
```

Create an XA topic connection using the default user identity. The connection is created in stopped mode. No messages are delivered until the Connection.start method is called explicitly.

Returns:
a newly created XA topic connection.

Throws:

- JMSException - if the JMS Provider fails to create an XA topic connection because of an internal error.
- JMSSecurityException - if client authentication fails because of an invalid user name or password.

createXATopicConnection

```
public XATopicConnection createXATopicConnection(java.lang.String userName,
                                                 java.lang.String password)
                                     throws JMSException
```

Create an XA topic connection using the specified user identity. The connection is created in stopped mode. No messages are delivered until the Connection.start method is called explicitly.

Parameters:

- userName - the user name of the caller
- password - the password of the caller

Returns:
a newly created XA topic connection.

Throws:

- JMSException - if the JMS Provider fails to create an XA topic connection because of an internal error.

- JMSSecurityException - if client authentication fails because of an invalid user name or password.

XATopicSession

public interface **XATopicSession**
extends **XASession**

MQSeries class: **MQXATopicSession**

```
java.lang.Object
   |
   +----com.ibm.mq.jms.MQXASession
           |
           +----com.ibm.mq.jms.MQXATopicSession
```

An XATopicSession provides a TopicSession, which can be used to create TopicSubscribers and TopicPublishers. Refer to “Appendix E. JMS JTA/XA interface with WebSphere” on page 357 for details about how MQ JMS uses XA classes.

The XAResource that corresponds to the TopicSession can be obtained by calling the getXAResource method, which is is inherited from XASession.

See also: **TopicSession** and **XASession**

Methods

getTopicSession

```
public TopicSession getTopicSession()
                             throws JMSException
```

Get the topic session associated with this XATopicSession.

Returns:
the topic session object.

Throws:
- JMSException - if a JMS error occurs.

Part 4. Appendixes

Appendix A. Mapping between Administration tool properties and programmable properties

MQSeries Classes for Java Message Service provides facilities to set and query the properties of administered objects either using the MQ JMS administration tool, or in an application program. Table 30 shows the mapping between each property name used with the administration tool and the corresponding member variable it refers to. It also shows the mapping between symbolic property values used in the tool and their programmable equivalents.

Table 30. 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		
MSGRETENTION	messageRetention	• YES • NO	JMSC.MQJMS_MRET_YES JMSC.MQJMS_MRET_NO
BROKERVER	brokerVersion	• V1	JMSC.MQJMS_BROKER_V1
BROKERPUBQ	brokerPubQueue		
BROKERSUBQ	brokerSubQueue		
BROKERDURSUBQ	brokerDurSubQueue		
BROKERCCSUBQ	brokerCCSubQueue		
BROKERCCDSUBQ	brokerCCDurSubQueue		
BROKERQMGR	brokerQueueManager		
BROKERCONQ	brokerControlQueue		
EXPIRY	expiry	• APP • UNLIM	JMSC.MQJMS_EXP_APP JMSC.MQJMS_EXP_UNLIMITED

Properties

Table 30. Comparison of representations of properties within the administration tool, and the programmable equivalents. (continued)

Property	Member variable name	Property value mapping	
		Tool	**Program**
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

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 31 lists the scripts and their uses.

Table 31. Utilities supplied with MQSeries classes for Java Message Service

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 21.
PSIVTRun.bat	Used to run the Pub/Sub installation verification test program described in “The Publish/Subscribe Installation Verification Test” on page 24.
formatLog.bat	Used to convert binary log files to plain text, described in “Logging” on page 28.
JMSAdmin.bat	Used to run the administration tool, described in “Chapter 5. Using the MQ JMS administration tool” on page 31.
JMSAdmin.config	Configuration file for the administration tool, described in “Configuration” on page 32.
runjms.bat	A utility script to assist with the running of JMS applications, described in “Running your own MQ JMS programs” on page 27.
PSReportDump.class	Used to view broker report messages, described in “Handling broker reports” on page 188.
Note: On UNIX systems, the extension ’.bat’ is omitted from the filenames.	

Scripts

Appendix C. LDAP server configuration for Java objects

If you use JNDI to store objects that are administered by MQ JMS, and you use an LDAP server as your JNDI service provider, the server must be LDAP v3 (for example, 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 31).

Attempt to create and display a test object using the following commands:

```
DEFINE QCF(ldapTest)
DISPLAY QCF(ldapTest)
```

If no exception occurs, your server is properly configured and you can proceed to store JMS objects.

If a 'SchemaViolationException' is returned, or if the message 'Unable to bind to object' occurs, your server is not properly configured. Either your server is not configured to store Java objects, or permissions on the objects or the suffix are not correct. The following procedures should 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 about the use of 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'
  DESC 'Container for a Java object'
  SUP top
  STRUCTURAL
  MUST ( cn )
)
( 1.3.6.1.4.1.42.2.27.4.2.4
  NAME 'javaObject'
  DESC 'Java object representation'
  SUP top
  ABSTRACT
  MUST ( javaClassName )
  MAY ( javaClassNames $
        javaCodebase $
        javaDoc $
        description )
)
```

```
( 1.3.6.1.4.1.42.2.27.4.2.5
  NAME 'javaSerializedObject'
  DESC 'Java serialized object'
  SUP javaObject
  AUXILIARY
  MUST ( javaSerializedData )
)
( 1.3.6.1.4.1.42.2.27.4.2.7
  NAME 'javaNamingReference'
  DESC 'JNDI reference'
  SUP javaObject
  AUXILIARY
  MAY ( javaReferenceAddress $
        javaFactory )
)
```

Also, ensure that the schema contains the following attribute definitions, updating the schema if necessary:

```
( 1.3.6.1.4.1.42.2.27.4.1.11
        NAME 'javaReferenceAddress'
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )

( 1.3.6.1.4.1.42.2.27.4.1.10
        NAME 'javaFactory'
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )

( 1.3.6.1.4.1.42.2.27.4.1.7
        NAME 'javaCodebase'
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
```

When the updates are complete, stop and restart the LDAP server, then repeat the configuration checking procedure described in “Checking your LDAP server configuration” on page 353.

Appendix D. Connecting to MQSeries Integrator V2

You can use MQSeries Integrator V2:

- as the publish/subscribe broker for MQ JMS
- to route or transform messages that are created by a JMS client application, and to send or publish messages to a JMS client

Publish/subscribe

| You can use MQSeries Integrator V2 as the publish/subscribe broker for MQ JMS.
| This requires the following setup activities:

- Base MQSeries

  First, you must create a broker publication queue. This is an MQSeries queue on the broker queue manager; it is used to submit publications to the broker. You can choose your own name for this queue but it must match the queue name in your TopicConnectionFactory's BROKERPUBQ property. By default, a TopicConnectionFactory's BROKERPUBQ property is set to the value SYSTEM.BROKER.DEFAULT.STREAM so, unless you want to configure a different name in the TopicConnectionFactory, you should name the queue SYSTEM.BROKER.DEFAULT.STREAM.
- MQSeries Integrator V2

  The next step is to set up a *message flow* within an execution group for the broker. The purpose of this message flow is to read messages from the broker publication queue. (If you want, you can set up multiple publication queues; each will need its own TopicConnectionFactory and message flow.)

| The basic message flow consists of an MQInput node (configured to read from
| the SYSTEM.BROKER.DEFAULT.STREAM queue) whose output is connected to
| the input of a Publication (or MQOutput) node.

| The message flow diagram would therefore look similar to the following:
|

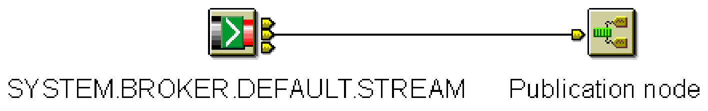

Figure 7. MQSeries Integrator message flow

When this message flow is deployed and the broker is started, from the JMS application's perspective the MQSeries Integrator V2 broker behaves like an MQSeries Publish/Subscribe broker. The current subscription state can be viewed using the MQSeries Integrator Control Center.

Notes:

1. No modifications are required to MQSeries classes for Java Message Service.
2. MQSeries Publish/Subscribe and MQSeries Integrator V2 brokers cannot coexist on the same queue manager.
3. Details of the MQSeries Integrator V2 installation and setup procedure are described in the *MQSeries Integrator for Windows NT Version 2.0 Installation Guide*.

Transformation and routing

You can use MQSeries Integrator V2 to route or transform messages that are created by a JMS client application, and to send or publish messages to a JMS client.

The MQSeries JMS implementation uses the mcd folder of the MQRFH2 to carry information about the message, as described in "The MQRFH2 header" on page 196. By default, the Message Domain (Msd) property is used to identify whether the message is a text, bytes, stream, map, or object message.

When a JMS application creates a text or bytes message, the application can override this Msd property, and can set other mcd folder fields. It does this by setting a JMS Type property that has a special URI format, for example:

```
mcd://domain/set/type[?format=fmt]
```

The values in the *domain*, *set*, *type*, and *fmt* fields (where *fmt* is optional) are copied to the outgoing MQRFH2. This means that the application can set these fields to values that are recognized by an MQSeries Integrator V2 message flow.

Appendix E. JMS JTA/XA interface with WebSphere

MQSeries classes for Java Message Service includes the JMS XA interfaces. These allow MQ JMS to participate in a two-phase commit that is coordinated by a transaction manager that complies with the Java Transaction API (JTA).

This section describes how to use these features with the WebSphere Application Server, Advanced Edition, so that WebSphere can coordinate JMS send and receive operations, and database updates, in a global transaction.

Before you use MQ JMS and the XA classes with WebSphere, there might be additional installation or configuration steps. Refer to the `Readme.txt` file on the MQSeries Using Java SupportPac Web page for the latest information (`www.ibm.com/software/ts/mqseries/txppacs/ma88.html`).

Using the JMS interface with WebSphere

This section provides guidance on using the JMS interface with the WebSphere Application Server, Advanced Edition.

You must already understand the basics of JMS programs, MQSeries, and EJB beans. These details are in the JMS specification, the EJB V2 specification (both available from Sun), this manual, the samples provided with MQ JMS, and other manuals for MQSeries and WebSphere.

Administered objects

JMS uses administered objects to encapsulate vendor-specific information. This minimizes the impact of vendor-specific details on end-user applications. Administered objects are stored in a JNDI namespace, and can be retrieved and used in a portable manner without knowledge of the vendor-specific contents.

For standalone use, MQ JMS provides the following classes:
- MQQueueConnectionFactory
- MQQueue
- MQTopicConnectionFactory
- MQTopic

WebSphere provides an additional pair of administered objects so that MQ JMS can integrate with WebSphere:
- JMSWrapXAQueueConnectionFactory
- JMSWrapXATopicConnectionFactory

You use these objects in exactly the same way as the MQQueueConnectionFactory and MQTopicConnectionFactory. However, behind the scenes they use the XA versions of the JMS classes, and enlist the MQ XAResource in the WebSphere transaction.

Container-managed versus bean-managed transactions

Container-managed transactions are transactions in EJB beans that are demarcated automatically by the EJB container. Bean-managed transactions are transactions in EJB beans that are demarcated by the program (via the UserTransaction interface).

Two-phase commit versus one-phase optimization

The WebSphere coordinator only invokes a true two-phase commit if more than one XAResource is used in a particular transaction. Transactions that involve a single resource are committed using a one-phase optimization. This largely removes the need to use different ConnectionFactories for distributed and non-distributed transactions.

Defining administered objects

You can use the MQ JMS administration tool to define the WebSphere-specific connection factories and store them in a JNDI namespace. The admin.config file in *MQ_install_dir*/bin should contain the following lines:

```
 INITIAL_CONTEXT_FACTORY=com.ibm.ejs.ns.jndi.CNInitialContextFactory
 PROVIDER_URL=iiop://hostname/
```

MQ_install_dir is the installation directory for MQ JMS, and *hostname* is the name or IP address of the machine that is running WebSphere.

To access the com.ibm.ejs.ns.jndi.CNInitialContextFactory, you must add the file ejs.jar from the WebSphere lib directory to the CLASSPATH.

To create the new factories, use the define verb with the following two new types:

```
def WSQCF(name) [properties]
def WSTCF(name) [properties]
```

These new types use the same properties as the equivalent QCF or TCF types, except that only the BIND transport type is allowed (and therefore, client properties cannot be configured). For details, see “Administering JMS objects” on page 35.

Retrieving administration objects

In an EJB bean, you retrieve the JMS-administered objects using the InitialContext.lookup() method, for example:

```
InitialContext ic = new InitialContext();
TopicConnectionFactory tcf = (TopicConnectionFactory) ic.lookup("jms/Samples/TCF1");
```

The objects can be cast to, and used as, the generic JMS interfaces. Normally, there is no need to program to the MQSeries specific classes in the application code.

Samples

There are three samples that illustrate the basics of using MQ JMS with WebSphere Application Server Advanced Edition. These are in subdirectories of *MQ_install_dir*/samples/ws, where *MQ_install_dir* is the installation directory for MQ JMS.

- Sample1 demonstrates a simple put and get for a message in a queue by using container-managed transactions.
- Sample2 demonstrates a simple put and get for a message in a queue by using bean-managed transactions.

- Sample3 illustrates the use of the publish/subscribe API.

For details about how to build and deploy the EJB beans, please refer to the WebSphere Application Server documentation.

The readme.txt files in each sample directory include example output from each EJB bean. The scripts provided assume that a default queue manager is available on the local machine. If your installation varies from the default, you can edit these scripts as required.

Sample1

Sample1EJB.java, in the sample1 directory, defines two methods that use JMS:

- putMessage() sends a TextMessage to a queue, and returns the MessageID of the sent message
- getMessage() reads the message with the specified MessageID back from the queue

Before you run the sample, you must store two administered objects in the WebSphere JNDI namespace:

QCF1	a WebSphere-specific queue connection factory
Q1	a queue

Both objects must be bound in the jms/Samples sub-context.

To set up the administered objects, you can either use the MQ JMS administration tool and set them up manually, or you can use the script provided.

The MQ JMS administration tool must be configured to access the WebSphere namespace. For details about how to configure the administration tool, refer to "Configuring for WebSphere" on page 33.

To set up the administered objects with typical default settings, you can enter the following command to run the script `admin.scp`:

```
JMSAdmin < admin.scp
```

The bean must be deployed with the getMessage and putMessage methods marked as TX_REQUIRED. This ensures that the container starts a transaction before entering each method, and commits the transaction when the method completes. Within the methods, you do not need any application code that relates to the transactional state. However, remember that the message sent from putMessage occurs under syncpoint, and will not become available until the transaction is committed.

In the sample1 directory, there is a simple client program, Sample1Client.java, to call the EJB bean. There is also a script, runClient, to simplify running this program.

The client program (or script) takes a single parameter, which is used as the body of a TextMessage that will be sent by the EJB bean putMessage method. Then, the getMessage is called to read the message back off the queue and return the body to the client for display. The EJB bean sends progress messages to the standard output (stdout) of the application server, so you might wish to monitor that output during the run.

If the application server is on a machine that is remote from the client, you might need to edit Sample1Client.java. If you do not use the defaults, you may need to edit the runClient script to match the local installation path and name of the deployed jar file.

Sample2

Sample2EJB.java, in the sample2 directory, performs the same task as sample1, and requires the same administered objects. Unlike sample1, sample2 uses bean-managed transactions to control the transactional boundaries.

If you have not already run sample1, ensure that you set up the administered objects QCF1 and Q1, as described in “Sample1” on page 359.

The putMessage methods and getMessage methods start by obtaining an instance of UserTransaction. They use this instance to create a transaction via the UserTransaction.begin() method. After that, the main body of the code is the same as sample1 until the end of each method. At the end of each method, the transaction is completed by the UserTransaction.commit() call.

In the sample2 directory, there is a simple client program, Sample2Client.java, to call the EJB bean. There is also a script, runClient, to simplify running this program. You can use these in the same way as described for “Sample1” on page 359.

Sample3

Sample3EJB.java, in the sample3 directory, demonstrates the use of the publish/subscribe API with WebSphere. Publishing a message is very similar to the point to point case. However, there are differences when receiving messages via a TopicSubscriber.

Publish/subscribe programs commonly use nondurable subscribers. These nondurable subscribers exist only for the lifetime of their owning sessions (or less if the subscriber is closed explicitly). Also, they can receive messages from the broker only during that lifetime.

To convert sample1 to publish/subscribe, you might replace the QueueSender in putMessage with a TopicPublisher, and the QueueReceiver in getMessage with a nondurable TopicSubscriber. However, this would fail, because when the message is sent, the broker would not know of any subscribers to the topic. Therefore, the message would be discarded.

The solution is to create a durable subscriber before the message is published. Durable subscribers persist as a deliverable end-point beyond the lifetime of the session. Therefore, the message is available for retrieval during the call to getMessage().

The EJB bean includes two additional methods:

- createSubscription creates a durable subscription
- destroySubscription deletes a durable subscription

These methods (along with putMessage and getMessage) must be deployed with the TX_REQUIRED attribute.

Before you run sample3, you must store two administered objects in the WebSphere JNDI namespace:

TCF1

T1

Both objects must be bound in the jms/Samples sub-context.

To set up the administered objects, you can either use the MQ JMS administration tool and set them up manually, or you can use a script. The script `admin.scp` is provided in the sample3 directory.

The MQ JMS administration tool must be configured to access the WebSphere namespace. For details about how to configure the administration tool, refer to "Configuring for WebSphere" on page 33.

To set up the administered objects with typical default settings, you can enter the following command to run the script `admin.scp`:

```
JMSAdmin < admin.scp
```

If you have already run admin.scp to set up objects for sample1 or sample2, there will be error messages when you run admin.scp for sample3. (These occur when you attempt to create the jms and Samples sub-contexts.) You can safely ignore these error messages.

Also, before you run sample3, ensure that the MQSeries publish/subscribe broker (SupportPac MA0C) is installed and running.

In the sample3 directory, there is a simple client program, Sample3Client.java, to call the EJB bean. There is also a script, runClient, to simplify running this program. You can use these in the same way as described for "Sample1" on page 359.

Appendix F. Notices

Notices

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	AS/400	BookManager
CICS	IBM	IBMLink
Language Environment	MQSeries	MVS/ESA
OS/2	OS/390	OS/400
SecureWay	SupportPac	System/390
S/390	VisualAge	VSE/ESA
WebSphere		

Java, HotJava, JDK, and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

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.

Abstract Window Toolkit for Java (AWT). A collection of Graphical User Interface (GUI) components that are implemented using native-platform versions of the components.

applet. A Java program which is designed to run only on a Web page.

API. Application Programming Interface.

Application Programming Interface (API). An Application Programming Interface consists of the functions and variables that programmers are allowed to use in their applications.

AWT. Abstract Window Toolkit for Java.

casting. A term used in Java to describe the explicit conversion of the value of an object or primitive type into another type.

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 runtime component that provides access to queuing services on a server for local user applications.

EJB. Enterprise JavaBeans.

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.

Enterprise JavaBeans (EJB). A server-side component architecture, distributed by Sun Microsystems, for writing reusable business logic and portable enterprise applications. Enterprise JavaBean components are written entirely in Java and run on any EJB compliant server.

HTML. Hypertext Markup Language

Hypertext Markup Language (HTML). A language used to define information that is to be displayed on the World Wide Web.

IEEE. Institute of Electrical and Electronics Engineers.

IIOP. Internet Inter-ORB Protocol.

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 (Transmission Control Protocol/Internet Protocol).

JAAS. Java Authentication and Authorization Service.

Java Authentication and Authorization Service (JAAS). A Java service that provides entity authentication and access control.

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.

Java Runtime Environment (JRE). A subset of the Java Development Kit (JDK) that contains the core executables and files that constitute the standard Java platform. The JRE includes the Java Virtual Machine, core classes, and supporting files.

Java Transaction API (JTA). An API that allows applications and J2EE servers to access transactions.

Java Transaction Service (JTS). A transaction manager that supports JTA and implements the Java mapping of the OMG Object Transaction Service 1.1 specification below the level of the API.

Java Virtual Machine (JVM). A software implementation of a central processing unit (CPU) that runs compiled Java code (applets and applications).

Java 2 Platform, Enterprise Edition (J2EE). A set of services, APIs, and protocols that provide the functionality to develop multi-tiered, Web-based applications.

JDK. Java Developers Kit.

JNDI. Java Naming and Directory Service.

JMS. Java Message Service.

JRE. Java Runtime Environment.

JTA. Java Transaction API.

JTS. Java Transaction Service.

JVM. Java Virtual Machine.

J2EE. Java 2 Platform, Enterprise Edition.

LDAP. Lightweight Directory Access Protocol.

Lightweight Directory Access Protocol (LDAP). 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.

MQDLH. MQSeries dead letter header. See *MQSeries Application Programming Reference*.

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.

MQMD. MQSeries Message Descriptor.

MQSC. MQSeries commands.

MQSeries. MQSeries is a family of IBM licensed programs that provide message queuing services.

MQSeries commands (MQSC). Human-readable commands, uniform across all platforms, that are used to manipulate MQSeries objects.

MQSeries Message Descriptor (MQMD). Control information that describes the message format and properties, that is carried as part of an MQSeries message.

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.

Object Management Group (OMG). A consortium that sets standards in object-oriented programming.

OMG. Object Management Group.

ORB. Object Request Broker.

overloading. The situation where one identifier refers to multiple items in the same scope. In Java, methods can be overloaded, but not variables or operators.

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.

Red Hat Package Manager (RPM). A software packaging system for use on Red Hat Linux platforms, and other Linux and UNIX platforms.

RPM. Red Hat Package Manager.

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.

TCP/IP. Transmission Control Protocol/Internet Protocol.

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.

| **Uniform Resource Locator (URL).** A sequence of
| characters that represent information resources on a
| computer or in a network such as the Internet.

| **URL.** Uniform Resource Locator.

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, V5.2
- MQSeries for AS/400, V5.2
- MQSeries for AT&T GIS UNIX, V2.2
- MQSeries for Compaq (DIGITAL) OpenVMS, V2.2.1.1
- MQSeries for Compaq Tru64 UNIX, V5.1
- MQSeries for HP-UX, V5.2
- MQSeries for Linux, V5.2
- MQSeries for OS/2 Warp, V5.1
- MQSeries for OS/390, V5.2
- MQSeries for SINIX and DC/OSx, V2.2
- MQSeries for Sun Solaris, V5.2
- MQSeries for Sun Solaris, Intel Platform Edition, V5.1
- MQSeries for Tandem NonStop Kernel, V2.2.0.1
- MQSeries for VSE/ESA, V2.1.1
- MQSeries for Windows, V2.0
- MQSeries for Windows, V2.1
- MQSeries for Windows NT and Windows 2000, V5.2

The MQSeries cross-platform publications are:

- *MQSeries Brochure*, G511-1908
- *An Introduction to Messaging and Queuing*, GC33-0805
- *MQSeries Intercommunication*, SC33-1872
- *MQSeries Queue Manager Clusters*, SC34-5349
- *MQSeries Clients*, GC33-1632
- *MQSeries System Administration*, SC33-1873
- *MQSeries MQSC Command Reference*, SC33-1369
- *MQSeries Event Monitoring*, SC34-5760
- *MQSeries Programmable System Management*, SC33-1482
- *MQSeries Administration Interface Programming Guide and Reference*, SC34-5390
- *MQSeries Messages*, GC33-1876
- *MQSeries Application Programming Guide*, SC33-0807
- *MQSeries Application Programming Reference*, SC33-1673
- *MQSeries Programming Interfaces Reference Summary*, SX33-6095
- *MQSeries Using C++*, SC33-1877
- *MQSeries Using Java*, SC34-5456
- *MQSeries Application Messaging Interface*, SC34-5604

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, V5.2

MQSeries for AIX Quick Beginnings, GC33-1867

MQSeries for AS/400, V5.2

MQSeries for AS/400 Quick Beginnings, GC34-5557

MQSeries for AS/400 System Administration, SC34-5558

MQSeries for AS/400 Application Programming Reference (ILE RPG), SC34-5559

MQSeries for AT&T GIS UNIX, V2.2

MQSeries for AT&T GIS UNIX System Management Guide, SC33-1642

MQSeries for Compaq (DIGITAL) OpenVMS, V2.2.1.1

MQSeries for Compaq (DIGITAL) OpenVMS System Management Guide, GC33-1791

MQSeries for Compaq Tru64 UNIX, V5.1

MQSeries for Compaq Tru64 UNIX Quick Beginnings, GC34-5684

MQSeries for HP-UX, V5.2

MQSeries for HP-UX Quick Beginnings, GC33-1869

MQSeries for Linux, V5.2

MQSeries for Linux Quick Beginnings, GC34-5691

MQSeries for OS/2 Warp, V5.1

MQSeries for OS/2 Warp Quick Beginnings, GC33-1868

MQSeries for OS/390, V5.2

MQSeries for OS/390 Concepts and Planning Guide, GC34-5650

MQSeries for OS/390 System Setup Guide, SC34-5651

MQSeries for OS/390 System Administration Guide, SC34-5652

MQSeries for OS/390 Problem Determination Guide, GC34-5892

MQSeries for OS/390 Messages and Codes, GC34-5891

MQSeries for OS/390 Licensed Program Specifications, GC34-5893

MQSeries for OS/390 Program Directory

MQSeries link for R/3, Version 1.2

MQSeries link for R/3 User's Guide, GC33-1934

MQSeries for SINIX and DC/OSx, V2.2

MQSeries for SINIX and DC/OSx System Management Guide, GC33-1768

MQSeries for Sun Solaris, V5.2

MQSeries for Sun Solaris Quick Beginnings, GC33-1870

MQSeries for Sun Solaris, Intel Platform Edition, V5.1

MQSeries for Sun Solaris, Intel Platform Edition Quick Beginnings, GC34-5851

MQSeries for Tandem NonStop Kernel, V2.2.0.1

MQSeries for Tandem NonStop Kernel System Management Guide, GC33-1893

MQSeries for VSE/ESA, V2.1.1

MQSeries for VSE/ESA™ Licensed Program Specifications, GC34-5365

MQSeries for VSE/ESA System Management Guide, GC34-5364

MQSeries for Windows, V2.0

MQSeries for Windows User's Guide, GC33-1822

MQSeries for Windows, V2.1

MQSeries for Windows User's Guide, GC33-1965

MQSeries for Windows NT and Windows 2000, V5.2

MQSeries for Windows NT 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.

HTML format

Relevant MQSeries documentation is provided in HTML format with these MQSeries products:
- MQSeries for AIX, V5.2
- MQSeries for AS/400, V5.2
- MQSeries for Compaq Tru64 UNIX, V5.1
- MQSeries for HP-UX, V5.2
- MQSeries for Linux, V5.2
- MQSeries for OS/2 Warp, V5.1
- MQSeries for OS/390, V5.2
- MQSeries for Sun Solaris, V5.2
- MQSeries for Sun Solaris, Intel Platform Edition, V5.1
- MQSeries for Windows NT and Windows 2000, V5.2 (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/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, V5.2
- MQSeries for AS/400, V5.2
- MQSeries for Compaq Tru64 UNIX, V5.1
- MQSeries for HP-UX, V5.2
- MQSeries for Linux, V5.2
- MQSeries for OS/2 Warp, V5.1
- MQSeries for OS/390, V5.2

- MQSeries for Sun Solaris, V5.2
- MQSeries for Sun Solaris, Intel Platform Edition, V5.1
- MQSeries for Windows NT and Windows 2000, V5.2
- 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/mqseries/
```

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

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/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 an MQSeries SupportPac.

Index

A

accessing queues and processes 59
administered objects 36, 170
  with WebSphere 357
administering JMS objects 35
administration
  commands 34
  verbs 34
administration tool
  configuration file 32
  configuring 32
  overview 31
  property mapping 349
  starting 31
advantages of Java interface 47
AIX, installing MQ Java 8
applet viewer
  using 5, 13
  with sample applet 14
applets
  example code 52
  running 70
  versus applications 51
application example 56
Application Server Facilities 209
  classes and functions 209
  sample client applications 219
  sample code 215
applications
  closing 177
  Publish/Subscribe, writing 179
  running 70
  unexpected termination 188
  versus applets 51
AS/400, installing MQ base Java 9
ASF (Application Server Facilities) 209
ASFClient1.java 220
ASFClient2.java 222
ASFClient3.java 224
ASFClient4.java 225
asynchronous message delivery 177

B

bean-managed transactions 358
  sample application 360
behavior in different environments 73
benefits of JMS 3
bibliography 369
bindings
  connection 6
  connection, programming 52
  example application 56
  verifying 15
bindings transport, choosing 172
body, message 191
BookManager 371
broker reports 189
BROKERCCDSUBQ object property 39, 211, 351
BROKERCCSUBQ object property 39, 211, 351
BROKERCONQ object property 39, 351
BROKERDURSUBQ object property 39, 351
BROKERPUBQ object property 39, 351
BROKERQMGR object property 39, 351
BROKERSUBQ object property 39, 351
BROKERVER object property 39, 351
building a connection 170
bytes message 191
BytesMessage
  interface 232
  type 175

C

CCSID object property 39, 351
CHANGE (administration verb) 34
CHANNEL object property 39, 351
choosing transport 172
CICS Transaction Server
  running applications 70
  using 16
class library 49
classes, Application Server Facilities 209
classes, core 73
classes, JMS 227
classes, MQSeries classes for Java 79
  ManagedConnection 161
  ManagedConnectionFactory 164
  ManagedConnectionMetaData 166
  MQC 152
  MQChannelDefinition 80
  MQChannelExit 82
  MQConnectionManager 154
  MQDistributionList 85
  MQDistributionListItem 87
  MQEnvironment 88
  MQException 93
  MQGetMessageOptions 95
  MQManagedObject 99
  MQMessage 102
  MQMessageTracker 121
  MQPoolServices 123
  MQPoolServicesEvent 124
  MQPoolServicesEventListener 153
  MQPoolToken 126
  MQProcess 127
  MQPutMessageOptions 129
  MQQueue 132
  MQQueueManager 140
  MQReceiveExit 155
  MQSecurityExit 157
  MQSendExit 159
  MQSimpleConnectionManager 150
classpath
  configuring 19
  settings 10
client properties 41
client transport, choosing 172
CLIENTID object property 39, 351
clients
  configuring queue manager 13
  connection 5
  programming 51
  verifying 15
closing
  applications 177
  JMS resources in Publish/Subscribe mode 181
  resources 177
code examples 52
com.ibm.jms package 231
com.ibm.mq.iiop.jar 7
com.ibm.mq.jar 7
com.ibm.mq.jms package 230
com.ibm.mqbind.jar 7
com.ibm.mqjms.jar 7
combinations, valid, of objects and properties 41
commands, administration 34
compiling MQSeries classes for Java programs 69
configuration file, for administration tool 32
configuring
  environment variables 19
  for Publish/Subscribe 20
  for WebSphere 33
  LDAP server 353
  queue manager for clients 13
  the administration tool 32
  Web server 12
  your classpath 19
  your installation 19
confirm on arrival report options, message 103
confirm on delivery report options, message 103
connecting to a queue manager 58
connecting to MQSeries Integrator V2 355
connection
  building 170
  creating 171
  interface 169
  MQSeries, losing 188
  options 4
  starting 171
Connection interface 240
connection pooling 64
  example 64
connection type, defining 52
ConnectionConsumer class 209
ConnectionConsumer interface 243
ConnectionFactory interface 244
ConnectionMetaData interface 248
connector.jar 7
container-managed transactions 358
  sample application 359
converting the log file 31

COPY (administration verb) 34
core classes 73
  exceptions 74
  extensions for V5 76
CountingMessageListenerFactory.java 220
createQueueSession method 172
createReceiver method 175
createSender method 173
creating
  a connection 171
  factories at runtime 171
  JMS objects 37
  Topics at runtime 182
customizing the sample applet 15

D

default connection pool 64
  multiple components 66
default trace and log output locations 27
DEFINE (administration verb) 34
defining connection type 52
defining transport 172
DELETE (administration verb) 34
DeliveryMode interface 250
dependencies, property 41
DESCRIPTION object property 39, 351
Destination interface 251
differences between applets and applications 51
differences due to environment 73
directories, installation 10
disconnecting from a queue manager 58
DISPLAY (administration verb) 34
disposition options, message 103, 213
durable subscribers 184

E

ENCODING object property 42
END (administration verb) 34
environment differences 73
environment variables 10
  configuring 19
error
  conditions for object creation 43
  handling 61
  logging 28
  recovery, IVT 24
  recovery, PSIVT 27
  runtime, handling 177
error messages 18
  LDAP server 353
example code 52
exception listener 178
exception report options, message 103, 213
ExceptionListener interface 253
exceptions
  JMS 177
  MQSeries 177
  to core classes 74
exit string properties 42
expiration report options, message 103
EXPIRY object property 39, 351
extensions to core classes for V5 76
extra function provided over MQ Java 3

F

factories, creating at runtime 171
formatLog utility 31, 351
fscontext.jar 7
function, extra provided over MQ Java 3
functions, Application Server Facilities 209

G

getting started 3
glossary 365

H

handling
  errors 61
  JMS runtime errors 177
  messages 60
headers, message 191
HOSTNAME object property 39, 351
HP-UX, installing MQ Java 8
HTML (Hypertext Markup Language) 370
Hypertext Markup Language (HTML) 370

I

IIOP connection, programming 51
import statements 179
INITIAL_CONTEXT_FACTORY parameter 32
inquire and set 61
installation
  directories 10
  Installation Verification Test program for Publish/Subscribe (PSIVT) 24
  IVT error recovery 24
  PSIVT error recovery 27
  setup 19
  verifying 19
Installation Verification Test program (IVT) 21
installing
  MQ base Java on AS/400 9
  MQ Java on Linux 9
  MQ Java on UNIX 8
  MQ Java on Windows 10
  MQSeries classes for Java 7
  MQSeries classes for Java Message Service 7
interface, programming 48
interfaces
  JMS 169, 227
  MQSeries 169
introduction
  for programmers 47
  MQSeries classes for Java 3
  MQSeries classes for Java Message Service 3
IVT (Installation Verification Test program) 21
IVTrun utility 351
IVTRun utility 21, 23
IVTSetup utility 22, 351
IVTTidy utility 24, 351

J

J2EE connector architecture 64
JAAS (Java Authentication and Authorization Service) 64, 154
jar files 7
Java 2 Platform Enterprise Edition (J2EE) 64
Java Authentication and Authorization Service (JAAS) 64, 154
Java classes 49, 79
Java Developers Kit (JDK) 48
Java interface, advantages 47
Java Transaction API (JTA) 340, 357
javax.jms package 227
JDK (Java Developers Kit) 48
JMS
  administered objects 170
  benefits 3
  classes 227
  exception listener 178
  exceptions 177
  interfaces 169, 227
  introduction 3
  mapping of fields at send/publish 201
  mapping with MQMD 199
  message types 175
  messages 191
  model 169
  objects, administering 35
  objects, creating 37
  objects, properties 38
  objects for Publish/Subscribe 179
  programs, writing 169
  resources, closing in Publish/Subscribe mode 181
jms.jar 7
JMS JTA/XA Interface 357
JMSAdmin.config utility 351
JMSAdmin utility 351
JMSBytesMessage class 232
JMSCorrelationID header field 191
JMSMapMessage class 254
JMSMessage class 262
JMSStreamMessage class 306
JMSTextMessage class 316
JNDI
  retrieving 170
  security considerations 33
jndi.jar 7
JTA (Java Transaction API) 340, 357

L

ldap.jar 7
LDAP naming considerations 37
LDAP server 22
  configuration 353
  schema 353
library, Java classes 49
Linux, installing MQ Java 9
listener, JMS exception 178
Load1.java 219

Load2.java 222
local publications, suppressing 184
log file
  converting 31
  default output location 27
logging errors 28
LoggingMessageListenerFactory.java 222

M

ManagedConnection 161
ManagedConnectionFactory 164
ManagedConnectionMetaData 166
manipulating subcontexts 35
map message 191
MapMessage
  interface 254
  type 175
mapping properties between admin. tool and programs 349
message
  body 191
  delivery, asynchronous 177
  error 18
  handling 60
  headers 191
  message body 206
  properties 191
  selectors 176, 191
  selectors and SQL 192
  selectors in Publish/Subscribe mode 184
  sending 173
  types 175, 191
Message interface 262
MessageConsumer interface 169, 275
MessageListener interface 277
MessageListenerFactory.java 218
MessageProducer interface 169, 278
MessageProducer object 173
messages
  JMS 191
  mapping between JMS and MQSeries 195
  poison 212
  publishing 181
  receiving 175
  receiving in Publish/Subscribe mode 181
  selecting 176, 191
model, JMS 169
MOVE (administration verb) 34
MQC 152
MQChannelDefinition 80
MQChannelExit 82
MQConnection class 240
MQConnectionConsumer class 209, 243
MQConnectionFactory class 244
MQConnectionManager 154
MQConnectionMetaData class 248
MQDeliveryMode class 250
MQDestination class 251
MQDistributionList 85
MQDistributionListItem 87
MQEnvironment 52, 58, 88
MQException 93
MQGetMessageOptions 95
MQIVP
  listing 15
MQIVP *(continued)*
  sample application 15
  tracing 17
mqjavac
  tracing 17
  using to verify 13
MQManagedObject 99
MQMD (MQSeries Message Descriptor) 195
MQMessage 60, 102
MQMessageConsumer class 275
MQMessageProducer interface 278
MQMessageTracker 121
MQObjectMessage class 283
MQPoolServices 123
MQPoolServicesEvent 124
MQPoolServicesEventListener 153
MQPoolToken 126
MQProcess 127
MQPutMessageOptions 129
MQQueue 60, 132
  (JMS object) 36
  class 284
  for verification 22
MQQueueBrowser class 286
MQQueueConnection class 288
MQQueueConnectionFactory
  (JMS object) 36
  class 290
  for verification 22
  interface 290
  object 170
  set methods 172
MQQueueEnumeration class 282
MQQueueManager 59, 140
MQQueueReceiver class 292
MQQueueSender interface 295
MQQueueSession class 298
MQReceiveExit 155
MQRFH2 header 196
MQSecurityExit 157
MQSendExit 159
MQSeries
  connection, losing 188
  exceptions 177
  interfaces 169
  messages 195
MQSeries classes for Java classes 79
MQSeries Integrator V2, connecting to MQ JMS 355
MQSeries Message Descriptor (MQMD) 195
  mapping with JMS 199
MQSeries publications 369
MQSeries supported verbs 48
MQSeriesV5 extensions 76
MQSession class 209, 301
MQSimpleConnectionManager 150
MQTemporaryQueue class 314
MQTemporaryTopic class 315
MQTopic
  (JMS object) 36
  class 317
MQTopicConnection class 319
MQTopicConnectionFactory
  (JMS object) 36
  class 321
MQTopicConnectionFactory *(continued)*
  object 170
MQTopicPublisher class 324
MQTopicSession class 329
MQTopicSubscriber class 333
MQXAConnection class 334
MQXAConnectionFactory class 335
MQXAQueueConnection class 336
MQXAQueueConnectionFactory class 337
MQXAQueueSession class 339
MQXASession class 340
MQXATopicConnection class 342
MQXATopicConnectionFactory class 344
MQXATopicSession class 346
MSGRETENTION object property 39, 351
multithreaded programs 62
MyServerSession.java 217
MyServerSessionPool.java 217

N

names, of Topics 181
naming considerations, LDAP 37
Netscape Navigator, using 6
non-durable subscribers 184

O

object creation, error conditions 43
ObjectMessage
  interface 283
  type 175
objects
  administered 170
  JMS, administering 35
  JMS, creating 37
  JMS, properties 38
  message 191
  retrieving from JNDI 170
objects and properties, valid combinations 41
obtaining a session 172
one-phase optimization, with WebSphere 358
operations on queue managers 58
options
  connection 4
  subscribers 183
overview 3

P

package
  com.ibm.jms 231
  com.mq.ibm.jms 230
  javax.jms 227
PDF (Portable Document Format) 370
PERSISTENCE object property 39, 351
platform differences 73
point-to-point installation verification 21
poison messages 212
PORT object property 39, 351
Portable Document Format (PDF) 370
PostScript format 371
prerequisite software 6

PRIORITY object property 39, 351
problems, solving 17, 28
problems, solving in Publish/Subscribe mode 187
processes, accessing 59
programmers, introduction 47
programming
  bindings connection 52
  client connections 51
  compiling 69
  connections 51
  multithreaded 62
  tracing 70
  writing 51
programming interface 48
programs
  JMS, writing 169
  Publish/Subscribe, writing 179
  running 27, 70
  tracing 28
properties
  client 41
  dependencies 41
  mapping between admin. tool and programs 349
  message 191
  of exit strings 42
  of JMS objects 38
  queue, setting 173
properties and objects, valid combinations 41
PROVIDER_URL parameter 32
providerutil.jar 7
PSIVT (Installation Verification Test program) 24
PSIVTRun utility 25, 351
PSReportDump application 189
publications
  MQSeries 369
publications (Publish/Subscribe), local suppressing 184
publish/subscribe, sample application 360
Publish/Subscribe Installation Verification Test program (PSIVT) 24
publishing messages 181

Q

QMANAGER object property 39, 351
Queue
  interface 284
  object 170
queue manager
  configuring for clients 13
  connecting to 58
  disconnecting from 58
  operations on 58
QUEUE object property 39, 351
queue properties
  setting 173
  setting with set methods 174
QueueBrowser interface 286
QueueConnection interface 288
QueueReceiver interface 292
QueueRequestor class 293
queues, accessing 59
QueueSender interface 295
QueueSession interface 298

R

reading strings 61
receiving
  messages 175
  messages in Publish/Subscribe mode 181
RECEXIT object property 39, 351
RECEXITINIT object property 39, 351
report options, message 102, 213
reports, broker 189
resources, closing 177
retrieving objects from JNDI 170
runjms utility 27, 351
running
  applets 70
  applications under CICS Transaction Server 70
  in a Web browser 5
  MQSeries classes for Java programs 70
  programs 27
  standalone program 5
  the IVT 21
  the PSIVT 24
  with applet viewer 5
  your own programs 16
runtime
  creating factories 171
  creating Topics 182
  errors, handling 177

S

sample applet
  customizing 15
  tracing 17
  using to verify 13
  with applet viewer 14
sample application
  bean-managed transactions 360
  bindings mode 56
  container-managed transactions 359
  MQ JMS with WebSphere 358
  publish/subscribe 179, 360
  tracing 17
  using Application Server Facilities 219
  using to verify 15
sample classpath settings 10
sample code
  applet 52
  ServerSession 215
  ServerSessionPool 215
Sample1EJB.java 359
Sample2EJB.java 360
Sample3EJB.java 360
schema, LDAP server 353
scripts provided with MQSeries classes for Java Message Service 351
SECEXIT object property 39, 351
SECEXITINIT object property 39, 351
SECURITY_AUTHENTICATION parameter 32
security considerations, JNDI 33
selecting a subset of messages 176, 191
selectors
  message 176, 191
  message, and SQL 192
  message in Publish/Subscribe mode 184
SENDEXIT object property 39, 351
SENDEXITINIT object property 39, 351
sending a message 173
ServerSession sample code 215
ServerSessionPool sample code 215
session, obtaining 172
Session class 209
Session interface 169, 301
set and inquire 61
set methods
  on MQQueueConnectionFactory 172
  using to set queue properties 174
setting
  queue properties 173
  queue properties with set methods 174
shutting down applications 177
softcopy books 370
software, prerequisites 6
Solaris
  installing MQ Java 8
solving problems 17
  general 28
  in Publish/Subscribe mode 187
SQL for message selectors 192
standalone program, running 5
starting a connection 171
starting the administration tool 31
stream message 191
StreamMessage
  interface 306
  type 175
strings, reading and writing 61
subcontexts, manipulating 35
subscriber options 183
subscriptions, receiving 181
subset of messages, selecting 176, 191
Sun JMS interfaces and classes 227
Sun Solaris
  installing MQ Java 8
Sun Web site 3
SupportPac 371
suppressing local publications 184

T

TARGCLIENT object property 39, 351
TCP/IP
  client verifying 15
  connection, programming 51
TEMPMODEL object property 39, 351
TemporaryQueue interface 314
TemporaryTopic interface 315
termination, unexpected 188
testing MQSeries classes for Java programs 70
text message 191
TextMessage
  interface 316
  type 175
tokens, connection pooling 64
Topic
  interface 179, 317

Topic *(continued)*
  names 181
  names, wildcards 182
  object 170
TOPIC object property 39, 351
TopicConnection 179
  interface 319
TopicConnectionFactory 179
  interface 321
TopicLoad.java 223
TopicPublisher 180
  interface 324
TopicRequestor class 327
TopicSession 179
  interface 329
TopicSubscriber 180
  interface 333
trace, default output location 27
tracing
  MQSeries for Java Message Service 28
  programs 70
  sample applet 17
  the sample application 17
transactions
  bean-managed 358
  container-managed 358
  sample application 359, 360
transport, choosing 172
TRANSPORT object property 39, 351
two-phase commit, with WebSphere 358
types of JMS message 175, 191

U

unexpected application termination 188
uniform resource identifier (URI) for queue properties 173
UNIX, installing MQ Java 8
URI for queue properties 173
user exits, writing 63
uses for MQSeries 3
using
  applet viewer 13
  CICS Transaction server 16
  MQ base Java 13
utilities provided with MQSeries classes for Java Message Service 351

V

V5 extensions 76
valid combinations of objects and properties 41
verbs, MQSeries supported 48
verification
  with JNDI (point-to-point) 22
  with JNDI (Publish/Subscribe) 26
  without JNDI (point-to-point) 21
  without JNDI (Publish/Subscribe) 25
verifying
  client mode installation 13
  TCP/IP clients 15
  with the sample applet 13
  with the sample application 15
  your installation 19
versions of software required 6
VisiBroker
  configuring the queue manager 14
  connection 4, 52, 55
  using 4, 6, 16

W

Web browser
  using 5
Web server, configuring 12
WebSphere
  configuration 33
  CosNaming namespace 32
  CosNaming repository 32
WebSphere Application Server 215, 357
  using with JMS 357
widcards in topic names 182
Windows
  installing MQ Java 10
Windows Help 371
writing
  JMS programs 169
  programs 51
  Publish/Subscribe applications 179
  strings 61
  user exits 63

X

XAConnection interface 334
XAConnectionFactory interface 335
XAQueueConnection interface 288, 336
XAQueueConnectionFactory interface 290, 337
XAQueueSession interface 339
XAResource 340
XASession interface 340
XATopicConnection interface 342
XATopicConnectionFactory interface 344
XATopicSession interface 346

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44–1962–870229
  - From within the U.K., use 01962–870229
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC34-5456-06