MQSeries®

Using Java®

SC34-5456-00

MQSeries®

Using Java®

SC34-5456-00

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix A, “Notices” on page 119.

First edition (March 1999)

This edition applies to the following products:

- MQSeries for AIX® Version 5 Release 1
- MQSeries for AS/400®. Version 4 Release 2 Modification 1
- MQSeries for AT&T GIS UNIX® Version 2 Release 2
- MQSeries for HP-UX Version 5 Release 1
- MQSeries for OS/2® Warp Version 5 Release 1
- MQSeries for OS/390® Version 2 Release 1
- MQSeries for SINIX and DC/OSx Version 2 Release 2
- MQSeries for Sun Solaris Version 5 Release 1
- MQSeries for MVS/ESA™ Version 1 Release 2
- MQSeries for Windows NT® Version 5 Release 1

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

Order publications through your IBM® representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

At the back of this publication is a page titled “Sending your comments to IBM”. If you want to make comments, but the methods described are not available to you, please address them to:

IBM United Kingdom Laboratories,
Information Development,
Mail Point 095,
Hursley Park,
Winchester,
Hampshire,
England,
SO21 2JN

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Tables

About this book

This book describes the MQSeries classes for Java which can be used to access MQSeries systems.

Part 1 describes the use of MQSeries classes for Java, Part 2 provides assistance for programmers, and Part 3 contains detailed information about the MQSeries classes for Java.

Who this book is for

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

What you need to know

You should have:

- Knowledge of the Java programming language
- Understanding of the purpose of the Message Queue Interface (MQI) as described in Chapter 6, “Introducing the Message Queue Interface” in the *MQSeries Application Programming Guide* and in Chapter 3, “Call descriptions” in the *MQSeries Application Programming Reference* book
- Experience of MQSeries programs in general, or familiarity with the content of the other MQSeries publications

How to use this book

First read the sections of Part 1 that introduce you to the MQSeries classes for Java. Then read the programming guidance in Part 2 to understand how to use the classes to send and receive MQSeries messages. Refer to Part 3 for detailed information about the syntax of the classes.

Changes in this version

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

Programmable transport options

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

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

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

MQSeries publications

This section describes the documentation available for all current MQSeries products.

MQSeries cross-platform publications

Most of these publications, which are sometimes referred to as the MQSeries "family" books, apply to all MQSeries Level 2 products. The latest MQSeries Level 2 products are:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V4R2M1
- MQSeries for AT&T GIS UNIX V2.2
- MQSeries for Digital OpenVMS V2.2
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for OS/390 V2.1
- MQSeries for SINIX and DC/OSx V2.2
- MQSeries for Sun Solaris V5.1
- MQSeries for Tandem NonStop Kernel V2.2
- MQSeries for VSE/ESA V2.1
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1
- MQSeries for Windows NT V5.1

Any exceptions to this general rule are indicated. (Publications that support the MQSeries Level 1 products are listed in "MQSeries Level 1 product publications" on page xii. For a functional comparison of the Level 1 and Level 2 MQSeries products, see the *MQSeries Planning Guide*.)

MQSeries Brochure
The *MQSeries Brochure*, G511-1908, gives a brief introduction to the benefits of MQSeries. It is intended to support the purchasing decision, and describes some authentic customer use of MQSeries.

MQSeries: An Introduction to Messaging and Queuing
MQSeries: An Introduction to Messaging and Queuing, GC33-0805, describes briefly what MQSeries is, how it works, and how it can solve some classic interoperability problems. This book is intended for a more technical audience than the *MQSeries Brochure*.

MQSeries Planning Guide
The *MQSeries Planning Guide*, GC33-1349, describes some key MQSeries concepts, identifies items that need to be considered before MQSeries is installed, including storage requirements, backup and recovery, security, and migration from

earlier releases, and specifies hardware and software requirements for every MQSeries platform.

MQSeries Intercommunication
The *MQSeries Intercommunication* book, SC33-1872, defines the concepts of distributed queuing and explains how to set up a distributed queuing network in a variety of MQSeries environments. In particular, it demonstrates how to (1) configure communications to and from a representative sample of MQSeries products, (2) create required MQSeries objects, and (3) create and configure MQSeries channels. The use of channel exits is also described.

MQSeries Clients
The *MQSeries Clients* book, GC33-1632, describes how to install, configure, use, and manage MQSeries client systems.

MQSeries System Administration
The *MQSeries System Administration* book, SC33-1873, supports day-to-day management of local and remote MQSeries objects. It includes topics such as security, recovery and restart, transactional support, problem determination, and the dead-letter queue handler. It also includes the syntax of the MQSeries control commands.

This book applies to the following MQSeries products only:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries Command Reference
The *MQSeries Command Reference*, SC33-1369, contains the syntax of the MQSC commands, which are used by MQSeries system operators and administrators to manage MQSeries objects.

MQSeries Programmable System Management
The *MQSeries Programmable System Management* book, SC33-1482, provides both reference and guidance information for users of MQSeries events, Programmable Command Format (PCF) messages, and installable services.

MQSeries Messages
The *MQSeries Messages* book, GC33-1876, which describes “AMQ” messages issued by MQSeries, applies to these MQSeries products only:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1

This book is available in softcopy only.

MQSeries Application Programming Guide
The *MQSeries Application Programming Guide*, SC33-0807, provides guidance information for users of the message queue interface (MQI). It describes how to design, write, and build an MQSeries application. It also includes full descriptions of the sample programs supplied with MQSeries.

MQSeries Application Programming Reference
The *MQSeries Application Programming Reference*, SC33-1673, provides comprehensive reference information for users of the MQI. It includes: data-type descriptions; MQI call syntax; attributes of MQSeries objects; return codes; constants; and code-page conversion tables.

MQSeries Application Programming Reference Summary
The *MQSeries Application Programming Reference Summary*, SX33-6095, summarizes the information in the *MQSeries Application Programming Reference* manual.

MQSeries Using C++
MQSeries Using C++, SC33-1877, provides both guidance and reference information for users of the MQSeries C++ programming-language binding to the MQI. MQSeries C++ is supported by these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V4R2M1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for OS/390 V2.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries C++ is also supported by MQSeries clients supplied with these products and installed in the following environments:

- AIX
- HP-UX
- OS/2
- Sun Solaris
- Windows NT
- Windows 3.1
- Windows 95 and Windows 98

MQSeries Using Java
MQSeries Using Java, SC34-5456, provides both guidance and reference information for users of the MQSeries Bindings for Java and the MQSeries Client for Java. MQSeries Java is supported by these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries Administration Interface Programming Guide and Reference
The *MQSeries Administration Interface Programming Guide and Reference*, SC34-5390, provides information for users of the MQAI. The MQAI is a programming interface that simplifies the way in which applications manipulate Programmable Command Format (PCF) messages and their associated data structures.

This book applies to the following MQSeries products only:

MQSeries for AIX V5.1
MQSeries for HP-UX V5.1
MQSeries for OS/2 Warp V5.1
MQSeries for Sun Solaris V5.1

MQSeries for Windows NT V5.1

MQSeries Queue Manager Clusters
MQSeries Queue Manager Clusters, SC34-5349, describes MQSeries clustering. It explains the concepts and terminology and shows how you can benefit by taking advantage of clustering. It details changes to the MQI, and summarizes the syntax of new and changed MQSeries commands. It shows a number of examples of tasks you can perform to set up and maintain clusters of queue managers.

This book applies to the following MQSeries products only:

MQSeries for AIX V5.1
MQSeries for HP-UX V5.1
MQSeries for OS/2 Warp V5.1
MQSeries for OS/390 V2.1
MQSeries for Sun Solaris V5.1
MQSeries for Windows NT V5.1

MQSeries platform-specific publications

Each MQSeries product is documented in at least one platform-specific publication, in addition to the MQSeries family books.

MQSeries for AIX

MQSeries for AIX Version 5 Release 1 Quick Beginnings, GC33-1867

MQSeries for AS/400

MQSeries for AS/400 Version 4 Release 2.1 Administration Guide, GC33-1956

MQSeries for AS/400 Version 4 Release 2 Application Programming Reference (RPG), SC33-1957

MQSeries for AT&T GIS UNIX

MQSeries for AT&T GIS UNIX Version 2 Release 2 System Management Guide, SC33-1642

MQSeries for Digital OpenVMS

MQSeries for Digital OpenVMS Version 2 Release 2 System Management Guide, GC33-1791

MQSeries for Digital UNIX

MQSeries for Digital UNIX Version 2 Release 2.1 System Management Guide, GC34-5483

MQSeries for HP-UX

MQSeries for HP-UX Version 5 Release 1 Quick Beginnings, GC33-1869

MQSeries for OS/2 Warp

MQSeries for OS/2 Warp Version 5 Release 1 Quick Beginnings, GC33-1868

MQSeries for OS/390

MQSeries for OS/390 Version 2 Release 1 Licensed Program Specifications, GC34-5377

MQSeries for OS/390 Version 2 Release 1 Program Directory

MQSeries for OS/390 Version 2 Release 1 System Management Guide, SC34-5374

MQSeries for OS/390 Version 2 Release 1 Messages and Codes, GC34-5375

MQSeries for OS/390 Version 2 Release 1 Problem Determination Guide, GC34-5376

MQSeries link for R/3

MQSeries link for R/3 Version 1 Release 2 User's Guide, GC33-1934

MQSeries for SINIX and DC/OSx

MQSeries for SINIX and DC/OSx Version 2 Release 2 System Management Guide, GC33-1768

MQSeries for Sun Solaris

MQSeries for Sun Solaris Version 5 Release 1 Quick Beginnings, GC33-1870

MQSeries for Tandem NonStop Kernel

MQSeries for Tandem NonStop Kernel Version 2 Release 2 System Management Guide, GC33-1893

MQSeries for VSE/ESA

MQSeries for VSE/ESA Version 2 Release 1 Licensed Program Specifications, GC34-5365

MQSeries for VSE/ESA Version 2 Release 1 System Management Guide, GC34-5364

MQSeries for Windows

MQSeries for Windows Version 2 Release 0 User's Guide, GC33-1822
MQSeries for Windows Version 2 Release 1 User's Guide, GC33-1965

MQSeries for Windows NT

MQSeries for Windows NT Version 5 Release 1 Quick Beginnings, GC34-5389
MQSeries for Windows NT Using the Component Object Model Interface, SC34-5387
MQSeries LotusScript Extension, SC34-5404

MQSeries Level 1 product publications

For information about the MQSeries Level 1 products, see the following publications:

MQSeries: Concepts and Architecture, GC33-1141

MQSeries Version 1 Products for UNIX Operating Systems Messages and Codes, SC33-1754

MQSeries for UnixWare Version 1 Release 4.1 User's Guide, SC33-1379

Softcopy books

Most of the MQSeries books are supplied in both hardcopy and softcopy formats.

BookManager® format

The MQSeries library is supplied in IBM BookManager format on a variety of online library collection kits, including the *Transaction Processing and Data* collection kit, SK2T-0730. You can view the softcopy books in IBM BookManager format using the following IBM licensed programs:

BookManager READ/2
BookManager READ/6000
BookManager READ/DOS
BookManager READ/MVS
BookManager READ/VM
BookManager READ for Windows

HTML format

Relevant MQSeries documentation is provided in HTML format with these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1 (compiled HTML)
- MQSeries link for R/3 V1.2

The MQSeries books are also available in HTML format from the MQSeries product family Web site at:

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

Portable Document Format (PDF)

PDF files can be viewed and printed using the Adobe Acrobat Reader.

If you need to obtain the Adobe Acrobat Reader, or would like up-to-date information about the platforms on which the Acrobat Reader is supported, visit the Adobe Systems Inc. Web site at:

`http://www.adobe.com/`

PDF versions of relevant MQSeries books are supplied with these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1
- MQSeries link for R/3 V1.2

PDF versions of all current MQSeries books are also available from the MQSeries product family Web site at:

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

PostScript format

The MQSeries library is provided in PostScript (.PS) format with many MQSeries Version 2 products. Books in PostScript format can be printed on a PostScript printer or viewed with a suitable viewer.

Windows Help format

The *MQSeries for Windows User's Guide* is provided in Windows Help format with MQSeries for Windows Version 2.0 and MQSeries for Windows Version 2.1.

MQSeries information available on the Internet

MQSeries Web site

The MQSeries product family Web site is at:

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

By following links from this Web site you can:

- Obtain latest information about the MQSeries product family.
- Access the MQSeries books in HTML and PDF formats.
- Download MQSeries SupportPacs.

Part 1. Guidance for users

Chapter 1. Getting started with the MQSeries classes for Java

This chapter gives an overview of the MQSeries classes for Java, their uses, and installation requirements.

What are the MQSeries classes for Java?

The MQSeries classes for Java allow a program written in the Java programming language to connect to MQSeries as an MQSeries client, or directly to an MQSeries server. It enables Java applets, applications, and servlets to issue calls and queries to MQSeries giving access to mainframe and legacy applications, typically over the Internet, without necessarily having any other MQSeries code on the client machine. With the MQSeries classes for Java the user of an Internet terminal can become a true participant in transactions, rather than just a giver and receiver of information.

Who should use the MQSeries classes for Java?

If your enterprise fits any of the following scenarios, you can gain significant advantage by using the MQSeries classes for Java:

- A medium or large enterprise that is introducing intranet-based client/server solutions. Here Internet technology provides low cost easy access to global communications, while MQSeries connectivity provides high integrity with assured delivery and time independence.
- A medium or large enterprise with a need for reliable business-to-business communications with partner enterprises. Here again, the Internet provides low-cost easy access to global communications, while MQSeries connectivity provides high integrity with assured delivery and time independence.
- A medium or large enterprise that wishes to provide access from the public Internet to some of its enterprise applications. Here the Internet provides global reach at a low cost, while MQSeries connectivity provides high integrity through the queuing paradigm. In addition to low cost, the business can achieve improved customer satisfaction through 24 hour a day availability, fast response, and improved accuracy.
- An Internet Service provider, or other Value Added Network provider. These companies can exploit the low cost and easy communications provided by the Internet and add the value of high integrity provided by MQSeries connectivity. An Internet Service provider that exploits MQSeries can immediately acknowledge receipt of input data from a Web browser, guarantee delivery, and provide an easy way for the user of the Web browser to monitor the status of the message.

MQSeries provides an excellent infrastructure for access to enterprise applications and for development of complex Web applications. A service request from a Web browser can be queued and processed when possible, thus allowing a timely response to be sent to the end user regardless of system loading. By placing this queue 'close' to the user in network terms, the timeliness of the response is not impacted by network loading. In addition, the transactional nature of MQSeries

messaging means that a simple request from the browser can be expanded safely into a sequence of individual back-end processes in a transactional manner.

The MQSeries classes for Java also enable application developers to exploit the power of the Java programming language to create applets and applications that can run on any platform that supports the Java run-time environment. These factors combine to reduce significantly the development time for multi-platform MQSeries applications, and future enhancements to applets are automatically picked up by end users as the applet code is downloaded.

Connection options

Programmable options allow the MQSeries classes for Java to connect to MQSeries in three distinct ways:

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

These options are described in more detail below.

Client connection

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

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

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

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

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

Using VisiBroker for Java

Connection through Visibroker is provided as an alternative to connection 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 the MQSeries classes for Java.

Bindings connection

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

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

Prerequisites

The following software is required to run the MQSeries classes for Java:

- MQSeries for the server platform you wish to use. Table 1 shows the connection modes that can be used for each platform.

Table 1. Platforms and connection modes

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

- 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 4.)

  Note: To run MQSeries classes for 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 with a VisiBroker connection.)

Installing the MQSeries classes for Java

The MQSeries classes for Java can be installed from either the MQSeries Version 5.1 Software Server CD or the MQSeries Version 5.1 Software Client CD. Follow the installation instructions provided with the CD. If you choose the typical installation, the MQSeries classes for Java are included in the installation. If you choose to customize your installation, make sure that MQSeries classes for Java is checked.

Notes:

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

The MQSeries classes for Java files, documentation, and samples are installed in the directories shown in Table 2.

Table 2. Installation directories

Platform	Files	Directory
AIX	code Documentation Samples	usr/lpp/mqm/java/lib usr/lpp/mqm/html/mqjava usr/lpp/mqm/samp/javaclnt/*langdir*
HP-UX Solaris	code Documentation Samples	opt/mqm/java/lib opt/mqm/html/mqjava opt/mqm/samp/javaclnt/*langdir*
OS/2 Windows NT	code Documentation Samples	*install_dir*\java\lib *install_dir*\html\mqjava\ *install_dir*\tools\javaclnt\samples*langdir*
Note: *install_dir* is the directory in which you chose to install the MQSeries classes for Java. *langdir* is the language directory for your installation		

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

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

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

com.ibm.mqbind.jar This code supports only the bindings connection.

After installation, you will need to update your CLASSPATH environment variable to include the MQSeries Java code and samples directories. Table 3 on page 7 shows typical CLASSPATH settings for the various platforms:

Table 3. Sample CLASSPATH statements	
Platform	**Sample CLASSPATH**
AIX	CLASSPATH=/usr/lpp/jdk1.1.6/lib/classes.zip: /usr/lpp/mqm/java/lib/com.ibm.mq.jar: /usr/lpp/mqm/java/lib/com.ibm.mqbind.jar: /usr/lpp/mqm/samp/javaclnt/en_us:
HP-UX Solaris	CLASSPATH=/opt/jdk1.1.1/lib/classes.zip: /opt/mqm/java/lib/com.ibm.mq.jar: /opt/mqm/java/lib/com.ibm.mqbind.jar: /opt/mqm/samp/javaclnt/en_us:
OS/2	CLASSPATH=C:\jdk1.1.6\lib\classes.zip; *install_dir*\java\lib\com.ibm.mq.jar; *install_dir*\java\lib\com.ibm.mqbind.jar; *install_dir*\tools\javaclnt\samples\en_us;
Windows NT	CLASSPATH=C:\jdk1.1.6\lib\classes.zip; *install_dir*\java\lib\com.ibm.mq.jar; *install_dir*\java\lib\com.ibm.mqbind.jar; *install_dir*\java\lib\com.ibm.mq.iiop.jar; *install_dir*\tools\javaclnt\samples\en_us;
Note: *install_dir* is the directory in which you chose to install the MQSeries classes for Java.	

Additional environment variables need to be updated on Solaris, AIX, and HP-UX, as shown in Table 4.

Table 4. AIX and HP-UX environment variables	
Platform	**Environment variable**
AIX	LD_LIBRARY_PATH=/usr/lpp/mqm/lib
Solaris	LD_LIBRY_PATH=/opt/mqm/lib
HP-UX	SHLIB_PATH=/opt/mqm/lib

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.

Chapter 2. Using the MQSeries classes for Java

This chapter describes how to configure your system to run the sample applet and application programs to verify your installation of the MQSeries classes for Java, and how to modify the procedures to run your own programs.

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

Using the sample applet to verify the TCP/IP client

An installation verification applet, `mqjavac.html`, is provided with the MQSeries classes for Java. The applet can be used to verify the TCP/IP connected client mode of the MQSeries classes for Java. (See also "Verifying with the sample application" on page 11.)

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

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

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

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

Configuring your queue manager to accept client connections

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

TCP/IP client

1. Define a server connection channel using the following procedure:
   a. Start your queue manager using the strmqm command
   b. Type

      ```
      runmqsc
      ```

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

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

2. Start a listener program with the following commands:

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

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

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

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

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

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

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

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

Running from appletviewer

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

Local installation procedure

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

   ```
   appletviewer mqjavac.html
   ```

Web server installation procedure:
Enter the command:

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

Notes:

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

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

Running from a Web browser

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

Local installation procedure

Note: This method requires a Java 1.1 capable browser.

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

Web server installation procedure

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

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

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

Customizing the verification applet

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

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

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

hostname
Prefills the hostname edit box with the supplied value.

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

channel
Prefills the channel edit box with the supplied value.

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

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

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

trace
Causes the MQSeries classes for 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 the MQSeries classes for Java. You can use this application to test all the connection modes of the MQSeries classes for Java. The program prompts for a number of choices and data to determine which connection mode you want to verify. Use the following procedure to verify your installation:

1. **If you want to test a client connection:**
   Configure your queue manager as described in “Configuring your queue manager to accept client connections” on page 9.

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

2. Change to your samples directory.
3. Type

   ```
   java MQIVP
   ```

   The program tries to:

   a. Connect to, and disconnect from the named queue manager

   b. Open, put, get, and close the system default local queue

   c. Return a message if the operations are successful

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

   ```
   (1)Please enter the type of connection (MQSeries)               : (MQSeries)
   (2)Please enter the IP address of the MQSeries server          : myhost
   (3)Please enter the port to connect to                         : (1414)
   (3)Please enter the server connection channel name             : JAVA.CHANNEL
     Please enter the queue manager name                          :
     Success: Connected to queue manager.
     Success: Opened SYSTEM.DEFAULT.LOCAL.QUEUE
     Success: Put a message to SYSTEM.DEFAULT.LOCAL.QUEUE
     Success: Got a message from SYSTEM.DEFAULT.LOCAL.QUEUE
     Success: Closed SYSTEM.DEFAULT.LOCAL.QUEUE
     Success: Disconnected from queue manager

     Tests complete -
     SUCCESS: This transport is functioning correctly.
     Press Enter to continue...
   ```

4. At prompt (1):

   Leave the default 'MQSeries'.

5. At prompt (2):

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

Note: If you choose server connection, you do not see the prompts marked (3).

Using VisiBroker connectivity

If you are running with VisiBroker, the procedures described in “Configuring your queue manager to accept client connections” on page 9 are not required.

To test an installation using VisBroker, use the procedures described in “Verifying with the sample application” on page 11, but at prompt (1), type `VisiBroker` using the exact case.

Running your own programs

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

For information on writing MQSeries Java applications and applets, see Part 2, “Programming with the MQSeries classes for Java” on page 15.

Chapter 3. Solving MQSeries classes for Java problems

If a program does not complete successfully, try running the installation verification applet or installation verification program, both of which are described in Chapter 2, “Using the MQSeries classes for Java” on page 9, and follow the advice given in the diagnostic messages.

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

Tracing the sample applet

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

In the line

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

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

The line should then read:

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

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

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

Tracing the sample application

To trace the `MQIVP` program enter the following:

```
java MQIVP -trace n
```

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

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

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 MQSeries classes for Java from MQSeries Version 5.1 are installed and retry.

NO_IMPLEMENT
There is a communications problem involving VisiBroker Smart Agents.

Recommended Action: Consult your VisiBroker documentation.

COMM_FAILURE
There is a communications problem involving VisiBroker Smart Agents.

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

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

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

Part 2. Programming with the MQSeries classes for Java

Chapter 4. Introduction for programmers

This chapter contains general information for programmers. For more detailed information about writing programs see Chapter 5, “Writing Java programs for MQSeries” on page 21.

Why should I use the Java interface?

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

- The Java programming language is **easy to use**. There is no need for header files, pointers, structures, unions, and operator overloading. Programs written in Java are easier to develop and debug than their C and C++ equivalents.
- Java is **object-oriented**. The object-oriented features of Java are comparable to those of C++, but there is no multiple inheritance. Instead, Java uses the concept of an interface.
- Java is inherently **distributed**. The Java class libraries contain a library of routines for coping with TCP/IP protocols like HTTP and FTP. Java programs can access URLs as easily as accessing a file system.
- Java is **robust**. Java puts a lot of emphasis on early checking for possible problems, dynamic (runtime) checking, and the elimination of situations that are error prone. Java uses a concept of references that eliminates the possibility of overwriting memory and corrupting data.
- Java is **secure**. Java is intended to be run in networked/distributed environments, and a lot of emphasis has been placed on security. Java programs cannot overrun their run-time stack, cannot corrupt memory outside of their process space, and when downloaded from the Internet cannot even read or write local files.
- Java programs are **portable**. There are no "implementation-dependent" aspects of the Java specification. The Java compiler generates an architecture neutral object file format. The compiled code is executable on many processors, as long as the Java run-time system is present.

If you write your application using thge MQSeries classes for Java, users can download the Java byte codes for your program (called *applets*) from the Internet and run them on their own machines. This means that users with access to your Web server can load and run your application with no prior installation needed on their machines. When an update to the program is required, you update the copy on the Web server and users automatically receive the latest version the next time they access the applet. This can significantly reduce the costs involved in installing and updating traditional client applications where a large number of desktops are involved. If you place your applet on a Web server that is accessible outside the corporate firewall, anyone on the Internet can download and use your application. This means that you can get messages into your MQSeries system from anywhere on the internet. This opens the door to building a whole new set of Internet accessible service, support and electronic commerce applications.

The MQSeries classes for Java interface

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

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

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

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

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

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

Java Developer's Kit

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

The MQSeries classes for Java require JDK 1.1.6 or higher.

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

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

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

The MQSeries classes for Java class library

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

The following classes are provided:

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

and the following Java interfaces:

- MQC
- MQReceiveExit
- MQSecurityExit
- MQSendExit

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

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

Chapter 5. Writing Java programs for MQSeries

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

This chapter provides information to assist with writing Java applets, servlets, and applications to interact with MQSeries systems. For details of individual classes, see Chapter 7, “The Java classes and interfaces for MQSeries” on page 45.

Should I write applets or applications?

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

The main differences between applets and applications are:

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

The following general rules apply:

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

Client connections

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

- It supports only TCP/IP.
- It does not support connection tables.
- It does not read any MQSeries environment variables at startup.
- Information that would be stored in a channel definition and in environment variables is stored in a class called MQEnvironment, or can be passed as parameters when the connection is made.
- Error and exception conditions are written to a log specified in the MQException class. The default error destination is the Java console.

The MQSeries classes for Java clients do not support the MQBEGIN verb or fast bindings.

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

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

Bindings mode

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

- Most of the parameters provided by the MQEnvironment class are ignored.
- The bindings support the MQBEGIN verb and fast bindings into the MQSeries queue manager.

Defining which connection to use

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

MQEnvironment.properties
This can contain the following key/value pairs:

- For client and bindings connections:

  ```
  MQC.TRANSPORT_PROPERTY, MQC.TRANSPORT_MQSERIES
  ```

- For VisiBroker connections:

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

MQEnvironment.hostname
Set the value of this variable follows:

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

Example code fragments

Two example code fragments are included in this section; Figure 1 on page 23 and Figure 2 on page 26. Each is written to use a particular connection with notes to describe the changes needed to use alternative connections.

Example applet code

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

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

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

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

public class MQSample extends java.applet.Applet
{

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

  private MQQueueManager qMgr;                    // define a queue manager object

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

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

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

  } // end of init
```

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

```
public void start()
{

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

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

    int openOptions = MQC.MQOO_INPUT_AS_Q_DEF |
                      MQC.MQOO_OUTPUT ;

    // Now specify the queue that we wish to open, and the open options...

    MQQueue system_default_local_queue =
            qMgr.accessQueue("SYSTEM.DEFAULT.LOCAL.QUEUE",
                             openOptions,
                             null,           // default q manager
                             null,           // no dynamic q name
                             null);          // no alternate user id

    // Define a simple MQSeries message, and write some text in UTF format..

    MQMessage hello_world = new MQMessage();
    hello_world.writeUTF("Hello World!");

    // specify the message options...

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

    // put the message on the queue

    system_default_local_queue.put(hello_world,pmo);

    // get the message back again...
    // First define a MQSeries message buffer to receive the message into..

    MQMessage retrievedMessage = new MQMessage();
    retrievedMessage.messageId = hello_world.messageId;

    // Set the get message options..

    MQGetMessageOptions gmo = new MQGetMessageOptions();  // accept the defaults
                                                          // same as
                                                          // MQGMO_DEFAULT

    // get the message off the queue..

    system_default_local_queue.get(retrievedMessage, gmo);
```

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

```
      // And prove we have the message by displaying the UTF message text

      String msgText = retrievedMessage.readUTF();
      System.out.println("The message is: " + msgText);

      // Close the queue

      system_default_local_queue.close();

      // Disconnect from the queue manager

      qMgr.disconnect();

    }

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

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

  } // end of start

} // end of sample
```

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

Changing the connection to use VisiBroker for Java

Modify the line

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

to

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

and add the following lines to initialize the ORB:

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

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

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

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

Example application code

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

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

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

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

import java.util.Hashtable      // Required for properties

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

      java.util.Hashtable properties;

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

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

      int openOptions = MQC.MQOO_INPUT_AS_Q_DEF |
                        MQC.MQOO_OUTPUT ;

      // Now specify the queue that we wish to open,
      // and the open options...

      MQQueue system_default_local_queue =
              qMgr.accessQueue("SYSTEM.DEFAULT.LOCAL.QUEUE",
                               openOptions,
                               null,           // default q manager
                               null,           // no dynamic q name
                               null);          // no alternate user id

      // Define a simple MQSeries message, and write some text in UTF format..

      MQMessage hello_world = new MQMessage();
      hello_world.writeUTF("Hello World!");
```

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

```
      // Define a simple MQSeries message, and write some text in UTF format..

      MQMessage hello_world = new MQMessage();
      hello_world.writeUTF("Hello World!");

      // specify the message options...

      MQPutMessageOptions pmo = new MQPutMessageOptions(); // accept the // defaults,
                                                           // same as MQPMO_DEFAULT
      // put the message on the queue

      system_default_local_queue.put(hello_world,pmo);

      // get the message back again...
      // First define a MQSeries message buffer to receive the message into..

      MQMessage retrievedMessage = new MQMessage();
      retrievedMessage.messageId = hello_world.messageId;

      // Set the get message options...

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

      system_default_local_queue.get(retrievedMessage, gmo);

      // And prove we have the message by displaying the UTF message text

      String msgText = retrievedMessage.readUTF();
      System.out.println("The message is: " + msgText);

      // Close the queue...

      system_default_local_queue.close();

      // Disconnect from the queue manager

      qMgr.disconnect();

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

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

} // end of sample
```

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

Running MQSeries classes for Java applets

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

```
<html>
<body>
<applet code="MyClass.class" width=200 height=400>
</applet>
</body>
</html>
```

Operations on queue managers

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

Setting up the MQSeries environment

Note: This step is not necessary when using the MQSeries classes for Java in bindings mode. In that case, go directly to “Connecting to a queue manager” on page 29. Before connecting to a queue manager using the client connection, you must take care to set up the MQEnvironment.

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

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

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

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

This is equivalent to an MQSERVER environment variable setting of:

```
"java.client.channel/TCP/host.domain.com".
```

By default, the MQSeries classes for Java 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 25.

Connecting to a queue manager

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

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

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

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

Calling the disconnect method causes all open queues and processes that you have accessed through that queue manager to be closed. It is good programming practice, however, to close these resources yourself when you have finished using them. You do this with the close() method.

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

Accessing queues and processes

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

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

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

The accessQueue method returns a new object of class MQQueue.

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

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

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

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

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

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

The accessProcess method returns a new object of class MQProcess.

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

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

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

Handling messages

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

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

When the new message instance is created, all the MQMD parameters are automatically set to their default values, as defined in the *MQSeries Application Programming Reference*. The put() method of MQQueue also takes an instance of the MQPutMessageOptions class as a parameter. This class represents the MQPMO structure.

The following example shows the creation of a message and putting it onto a queue:

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

String name = "Wendy Ling";
myMessage.writeInt(name.length());
myMessage.writeBytes(name);

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

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

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

There is no need to specify a maximum message size because get() method automatically adjusts the size of its internal buffer to fit the incoming message. Use the readXXX methods of the MQMessage class to access the data in the returned message.

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

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

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

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

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

See “MQMessage” on page 68 for more details.

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

Handling errors

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

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

Getting and setting attribute values

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

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

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

The inquire and set methods take three parameters:

- selectors array
- intAttrs array
- charAttrs array

There is no need for the SelectorCount, IntAttrCount and CharAttrLength parameters found in MQINQ, because the length of an array in Java is always known.

The following example shows how to make an inquiry on a queue:

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

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

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

queue.inquire(selectors,intAttrs,charAttrs);

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

Multithreaded programs

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

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

With the "C" based MQSeries client this would cause a problem, since handles cannot be shared across multiple threads. The MQSeries classes for Java relax this constraint, allowing a queue manager object (and its associated queue and process objects) to be shared across multiple threads.

The implementation of the MQSeries classes for Java ensures that, for a given connection (queue manager object instance), all access to the target MQSeries queue manager is synchronized. This means that a thread wishing to issue a call to a queue manager is blocked until all other calls in progress for that connection have completed. If you require simultaneous access to the same queue manager from within your program, create a new queue manager object for each thread requiring concurrent access. (This is equivalent to issuing a separate MQCONN call for each thread.)

Writing user exits

The MQSeries classes for Java allow you to provide your own send, receive, and security exits.

To implement an exit, you define a new Java class that implements the appropriate interface. There are three exit interfaces defined in the MQSeries package:

- MQSendExit
- MQReceiveExit
- MQSecurityExit

The following sample defines a class that implements all three:

```
class MyMQExits implements MQSendExit, MQReceiveExit, MQSecurityExit {

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


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


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

}
```

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

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

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

The simplest possible exit body is:

```
{
  return agentBuffer;
}
```

If your program is to run as a downloaded Java applet, note that under the security restrictions placed on it you will not be able to read or write any local files. If your exit needs a configuration file, you can place the file on the web and use the java.net.URL class to download it and examine its contents.

Compiling and testing MQSeries classes for Java programs

Before compiling MQSeries classes for Java programs you must ensure that your MQSeries classes for Java installation directory is in your CLASSPATH environment variable, as described in “Installing the MQSeries classes for Java” on page 5.

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

```
javac MyClass.java
```

Running MQSeries classes for Java programs

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

```
java MyClass
```

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

If you are writing MQSeries classes for Java applets, run your program 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
```

Tracing MQSeries Java programs

The MQSeries classes for Java include a trace facility, which can be used to produce diagnostic messages if you suspect there might be a problem with the code. (You will normally need to use this facility only at the request of IBM service.)

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

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

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

If your program is an application, or you are running it from your local disk using the appletviewer command, you also have the option of redirecting the trace output to a file of your choice.

The following code fragment shows an example of how to make the redirection to a file called `myapp.trc`:

```
import java.io.*;

try {
  FileOutputStream
  traceFile = new FileOutputStream("myapp.trc");
  MQEnvironment.enableTracing(2,traceFile);
}
catch (IOException ex) {
  // couldn't open the file,
  // trace to System.err instead
  MQEnvironment.enableTracing(2);
}
```

There are 5 different levels of tracing:

1 Provides entry, exit and exception tracing.
2 Provides parameter information in addition to 1.
3 Provides transmitted and received MQSeries headers and data blocks in addition to 2.
4 Provides transmitted and received user message data in addition to 3.
5 Provides tracing of methods in the Java Virtual Machine in addition to 4.

To trace methods in the Java Virtual Machine with trace level 5, issue the command `java_g` in place of `java` to run an application, or `appletviewer_g` instead of `appletviewer` to run an applet.

Part 3. MQSeries classes for Java reference

Chapter 6. Environment dependent behavior

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

1. MQSeries Client for Java connected to an MQSeries V2.x server
2. MQSeries Client for Java connected to an MQSeries for OS/390 V 2.5 server
3. MQSeries Client for Java connected to an MQSeries for OS/400 server
4. MQSeries Client for Java connected to an MQSeries V5 server
5. MQSeries Bindings for Java executing on an MQSeries V5 server
6. MQSeries Bindings for Java executing on an MQSeries for OS/390 V2.4 or higher server

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

The MQSeries classes for Java provide a 'core' of classes, which provide consistent function and behavior in all the environments, and 'V5 extensions', which are designed for use only in environments 4 and 5. The core and extensions are documented below.

Core details

The MQSeries classes for Java contain 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 40.

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

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

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.

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 5.

Apart from these documented variations, the core classes give consistent behavior across all environments, even if the equivalent MQSeries classes normally have environment differences. In general, the behavior will be that expected in environments 4 and 5.

Table 5 (Page 1 of 2). Core classes restrictions and variations

MQGMO_LOCK MQGMO_UNLOCK	Cause MQRC_OPTIONS_ERROR when used in environments 2 or 6.
MQPMO_NEW_MSG_ID MQPMO_NEW_CORREL_ID MQPMO_LOGICAL_ORDER	Give errors except in environments 4 and 5. (See V5 extensions.)
MQGMO_SYNCPOINT_IF_PERSISTENT MQGMO_LOGICAL_ORDER MQGMO_COMPLETE_MESSAGE MQGMO_ALL_MSGS_AVAILABLE MQGMO_ALL_SEGMENTS_AVAILABLE	Give errors except in environments 4 and 5. (See V5 extensions.)
MQGMO_MARK_SKIP_BACKOUT	Causes MQRC_OPTIONS_ERROR except in environment 2 and 6.
MQCNO_FASTPATH_BINDING	Supported only in environment 5. (See V5 extensions.)
MQPMRF_* fields	Supported only in environments 4 and 5.
Putting a message with MQQueue.priority > MaxPriority	Rejected with MQCC_FAILED and MQRC_PRIORITY_ERROR in environments 2 and 6. 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.

Table 5 (Page 2 of 2). Core classes restrictions and variations	
BackoutCount	Environments 2 and 6 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 2 and 6. 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.

Version 5 extensions operating in other environments

The MQSeries classes for Java contain the following functions specifically designed to use the API extensions introduced in MQSeries V5. These functions operate as designed only in environments 4 and 5. This section describes how they can be expected to behave in other environments.

MQQueueManager constructor option

An optional integer argument is included in the MQQueueManager constructor. This maps onto the MQI's MQCNO.options field, and is used to switch between normal and fastpath connection. This extended form of the constructor is accepted in all environments, provided that the only options used are MQCNO_STANDARD_BINDING or MQCNO_FASTPATH_BINDING. Any other options cause the constructor to fail with MQRC_OPTIONS_ERROR. The fastpath option MQC.MQCNO_FASTPATH_BINDING is only honored when used in the MQSeries V5 bindings (environment 5). If it is used in any other environment, it is ignored.

MQQueueManager.begin() method

This can be used only in environment 5. In any other environment it fails with MQRC_ENVIRONMENT_ERROR.

MQPutMessageOptions options.

The following flags may be set into the MQPutMessageOptions options fields in any environment, but if used with a subsequent MQQueue.put() in any environment other than 4 or 5, the put() fails with MQRC_OPTIONS_ERROR:

- MQPMO_NEW_MSG_ID
- MQPMO_NEW_CORREL_ID
- MQPMO_LOGICAL_ORDER

MQGetMessageOptions options.

The following flags may be set into the MQGetMessageOptions options fields in any environment, but if used with a subsequent MQQueue.get() in any environment other than 4 or 5, the get() fails with MQRC_OPTIONS_ERROR:

- MQGMO_SYNCPOINT_IF_PERSISTENT
- MQGMO_LOGICAL_ORDER
- MQGMO_COMPLETE_MESSAGE
- MQGMO_ALL_MSGS_AVAILABLE
- MQGMO_ALL_SEGMENTS_AVAILABLE

MQGetMessageOptions fields

Values may be set into the following fields, regardless of the environment, but if the MQGetMessageOptions used on a subsequent MQQueue.get() is found to contain non-default values when running in any environment other than 4 or 5, the get() fails with MQRC_GMO_ERROR. This means that in environments other than 4 or 5, these fields will always be set to their initial values after every successful get().

- MatchOptions
- GroupStatus
- SegmentStatus
- Segmentation

Distribution Lists

The following classes are used to create Distribution Lists:

- MQDistributionList
- MQDistributionListItem
- MQMessageTracker

You can be create and populate MQDistributionList and MQDistributionListItems in any environment, but you can only create and open MQDistributionList successfully in environments 4 and 5. An attempt to create and open one in any other environment is rejected with MQRC_OD_ERROR.

MQPutMessageOptions fields

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

- knownDestCount
- unknownDestCount
- invalidDestCount
- recordFields

Although primarily intended for use with distribution lists, the MQSeries V5 server also fills in the DestCount fields after an MQPUT to a single queue. For example, if the queue resolves to a local queue, then knownDestCount is set to 1 and the other two fields to 0. In environments 4 and 5, the values set by the V5 server are returned in the MQPutMessageOptions class. In the other environments return values are simulated as follows:

- If the put() succeeds, unknownDestCount is set to 1, and the others are set to 0.
- If the put() fails, invalidDestCount is set to 1, and the others to 0.

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

MQPutMessage options are used on a subsequent MQQueue.put(), rather than MQDistributionList.put().

MQMD fields

The following MQMD fields are largely concerned with message segmentation:

- GroupId
- MsgSeqNumber
- Offset MsgFlags
- OriginalLength

If an application sets any of these MQMD fields to non-default values, and then does a put() to or get() in an environment other than 4 or 5, the put() or get() raises an exception (MQRC_MD_ERROR). A successful put() or get() in an environment other than 4 or 5, always leaves the new MQMD fields set to their default values. A grouped or segmented message should not normally be sent to a Java application running against a queue manager that is not MQSeries Version 5 or higher. If such an application does issue a get, and the physical message to be retrieved happens to be part of a group or segmented message (it has non-default values for the MQMD fields), it is retrieved without error. However, the MQMD fields in the MQMessage are not updated. The MQMessage format property is set to MQFMT_MD_EXTENSION, and the true message data is prefixed with an MQMDE structure containing the values for the new fields.

Chapter 7. The Java classes and interfaces for MQSeries

This chapter describes all the classes and interfaces contained in MQSeries. It includes details of the variables, constructors, and methods in each class and interface.

The following classes are described:

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

and the following interfaces:

- MQC
- MQReceiveExit
- MQSecurityExit
- MQSendExit

MQChannelDefinition

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

public class **MQChannelDefinition**
extends **Object**

The MQChannelDefinition class is used to pass information concerning the connection to the queue manager to the send, receive and security exits.

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

Variables

channelName
`public String channelName`

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

queueManagerName
`public String queueManagerName`

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

maxMessageLength
`public int maxMessageLength`

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

securityUserData
`public String securityUserData`

A storage area for the security exit to use. Information placed here is preserved across invocations of the security exit, and is also available to the send and receive exits.

sendUserData
`public String sendUserData`

A storage area for the send exit to use. Information placed here is preserved across invocations of the send exit, and is also available to the security and receive exits.

receiveUserData
`public String receiveUserData`

A storage area for the receive exit to use. Information placed here is preserved across invocations of the receive exit, and is also available to the send and security exits.

connectionName
`public String connectionName`

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

remoteUserId

```
public String remoteUserId
```

The user id used to establish the connection.

remotePassword

```
public String remotePassword
```

The password used to establish the connection.

Constructors

MQChannelDefinition

```
public MQChannelDefinition()
```

MQChannelExit

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

public class **MQChannelExit**
extends **Object**

This class defines context information passed to the send, receive, and security exits when they are invoked. The exitResponse member variable should be set by the exit to indicate what action the MQSeries classes 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 MQSeries classes for Java across exit invocations with the same exitID. (That is to say, the send, receive, and security exits each have their own, independent, user areas.)

capabilityFlags
```
public static final int capabilityFlags
```

Indicates the capability of the queue manager. Only the MQC.MQCF_DIST_LISTS flag is supported.

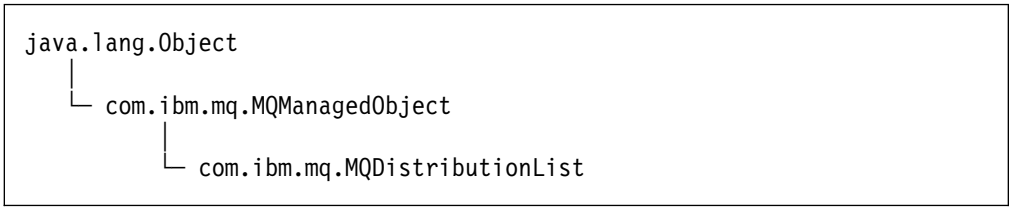
fapLevel
```
public static final int fapLevel
```

The negotiated Format and Protocol (FAP) level.

Constructors

MQChannelExit
```
public MQChannelExit()
```

MQDistributionList



public class **MQDistributionList**
extends **MQManagedObject** (See page 65.)

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

An MQDistributionList is created with MQDistributionList constructor or with the accessDistributionList method for MQQueueManager.

A distribution list represents a set of open queues to which messages can be sent using a single call to the put() method. (See“Distribution lists” in the *MQSeries Application Programming Guide*.)

Constructors

MQDistributionList

```
public MQDistributionList(MQQueueManager qMgr,
              MQDistributionListItem[] litems,
              int openOptions,
              String alternateUserId)
```

Throws MQException.

`qMgr` is the queue manager where the list is to be opened.

`litems` are the items to be included in the distribution list.

See "accessDistributionList" on page 107 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 90 for details.)

Throws MQException if the put fails.

getFirstDistributionListItem

```
public MQDistributionListItem getFirstDistributionListItem()
```

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

getValidDestinationCount

```
public int getValidDestinationCount()
```

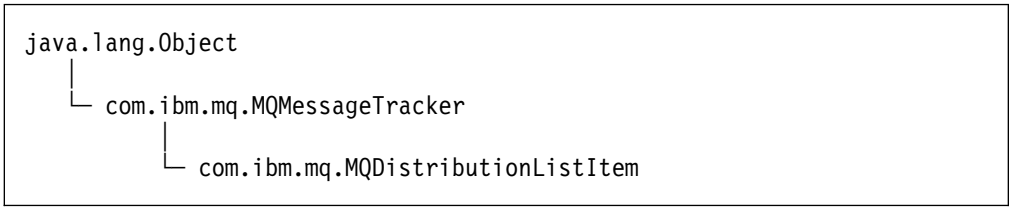
Returns the number of items in the distribution list that were opened successfully.

getInvalidDestinationCount

```
public int getInvalidDestinationCount()
```

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

MQDistributionListItem



public class **MQDistributionListItem**
extends **MQMessage** (See page 68.)

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

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

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

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

Values set in the MQEnvironment class take effect when the MQQueueManager constructor is called so you should set the values in the MQEnvironment class before constructing an MQQueueManager instance.

Variables

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 112.

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 114.

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 110.

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 statis 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 bindingd 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 68.)

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 classes for Java trace facility.

enableTracing

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

Turns on the MQSeries classes 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)
```

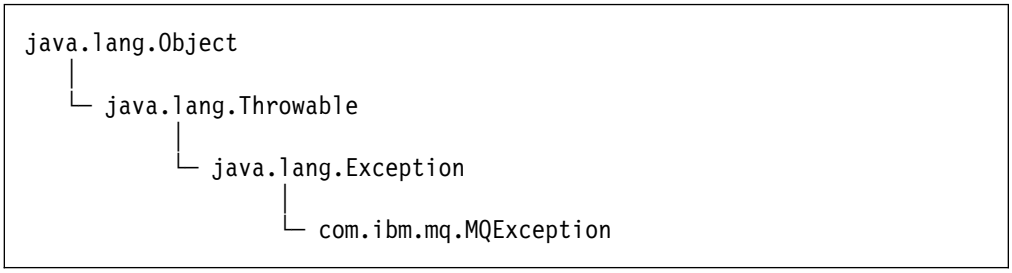
Turns on the MQSeries classes for Java trace facility.

Parameters:

level
The level of tracing required, from 1 to 5 (5 being the most detailed)

stream
The stream to which the trace is written

MQException



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

An MQException is thrown whenever an MQSeries error occurs. You can change the output stream for the exceptions that are logged by setting the value of MQException.log. The default value is System.err. This class contains definitions of completion code and error code constants. Constants beginning MQCC_ are MQSeries completion codes, and constants beginning MQRC_ are MQSeries reason codes. The *MQSeries Application Programming Reference* contains a full description of these errors and their probable causes.

Variables

log

```
public static java.io.outputStreamWriter log
```

Stream to which exceptions are logged. (The default is System.err.) If you set this to null no logging occurs.

completionCode

```
public int completionCode
```

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

- MQException.MQCC_WARNING
- MQException.MQCC_FAILED

reasonCode

```
public int reasonCode
```

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

exceptionSource

```
public Object exceptionSource
```

The object instance that threw the exception. You can use this as part of your diagnostics when determining the cause of an error.

Constructors

MQException

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

Construct a new MQException object.

Parameters

completionCode
The MQSeries completion code

reasonCode
The MQSeries reason code

source
The object in which the error occurred

MQGetMessageOptions

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

public class **MQGetMessageOptions**
extends **Object**

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

Note: The behavior of some of the options available in this class depends on the environment in which they are used. These elements are marked with a *****. See Chapter 6, “Environment dependent behavior” on page 39 for details.

Variables

options

```
public int options
```

Options that control the action of MQQueue.get. Any or none of the following values can be specified. If more than one option is required the values can be added together or combined using the bitwise OR operator.

MQC.MQGMO_NONE

MQC.MQGMO_WAIT
Wait for a message to arrive.

MQC.MQGMO_NO_WAIT
Return immediately if there is no suitable message.

MQC.MQGMO_SYNCPOINT
Get the message under syncpoint control; the message is marked as being unavailable to other applications, but it is deleted from the queue only when the unit of work is committed. The message is made available again if the unit of work is backed out.

MQC.MQGMO_NO_SYNCPOINT
Get message without syncpoint control.

MQC.MQGMO_BROWSE_FIRST
Browse from start of queue.

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

MQC.MQGMO_BROWSE_MSG_UNDER_CURSOR
Browse message under browse cursor.

MQC.MQGMO_MSG_UNDER_CURSOR
Get message under browse cursor.

MQC.MQGMO_LOCK*
Lock the message that is browsed.

MQC.MQGMO_UNLOCK*
Unlock a previously locked message.

MQC.MQGMO_ACCEPT_TRUNCATED_MSG
Allow truncation of message data.

MQC.MQGMO_FAIL_IF_QUIESCING
Fail if the queue manager is quiescing.

MQC.MQGMO_CONVERT
Request the application data to be converted, to conform to the characterSet and encoding attributes of the MQMessage, before the data is copied into the message buffer. Because data conversion is also applied as the data is retrieved from the message buffer, applications do not usually set this option.

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

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

Segmenting and grouping
MQSeries messages can be sent or received as a single entity, can be split into several segments for sending and receiving, and can also be linked to other messages in a group. Each piece of data that is sent is known as a *physical* message which can be a complete *logical* message, or a segment of a longer logical message. Each physical message usually has a different `MsgId`. All the segments of a single logical message have the same `groupId` value, and `MsgSeqNumber` value, but the `Offset` value is different for each segment. The `Offset` field gives the offset of the data in the physical message from the start of the logical message. The segments usually have different `MsgId` values as they are individual physical messages. Logical messages which form part of a group, have the same `groupId` value, but each message in the group has a different `MsgSeqNumber` value. Messages in a group can also be segmented.

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

MQC.MQGMO_LOGICAL_ORDER*
Return messages in groups, and segments of logical messages, in logical order.

MQC.MQGMO_COMPLETE_MSG*
Retrieve only complete logical messages.

MQC.MQGMO_ALL_MSGS_AVAILABLE*
Retrieve messages from a group only when all the messages in the group are available.

MQC.MQGMO_ALL_SEGMENTS_AVAILABLE*
Retrieve the segments of a logical message only when all the segments in the group are available.

waitInterval
```
public int waitInterval
```

The maximum time (in milliseconds) that an MQQueue.get call waits for a suitable message to arrive (used in conjunction with MQC.MQGMO_WAIT). A value of MQC.MQWI_UNLIMITED indicates that an unlimited wait is required.

resolvedQueueName

```
public String resolvedQueueName
```

This is an output field set by the queue manager to the local name of the queue from which the message was retrieved. This will be different from the name used to open the queue if an alias queue or model queue was opened.

matchOptions*

```
public int matchOptions
```

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

MQC.MQMO_MATCH_MSG_ID
Message id to be matched

MQC.MQMO_MATCH_CORREL_ID
Correlation id to be matched

MQC.MQMO_MATCH_GROUP_ID
Group id to be matched

MQC.MQMO_MATCH_MSG_SEQ_NUMBER
Match message sequence number

MQC.MQMO_NONE
No matching required

groupStatus*

```
public char groupStatus
```

This is an output field which indicates whether the retrieved message is in a group, and if it is, whether it is the last in the group. Possible values are:

MQC.MQGS_NOT_IN_GROUP
Message is not in a group.

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

MQC.MQGS_LAST_MSG_IN_GROUP
Message is the last in the group. This is also the value returned if the group consists of only one message.

segmentStatus*

```
public char segmentStatus
```

This is an output field that indicates whether the retrieved message is a segment of a logical message. If the message is a segment, the flag indicates whether or not it is the last segment. Possible values are:

MQC.MQSS_NOT_A_SEGMENT
Message is not a segment.

MQC.MQSS_SEGMENT
Message is a segment, but is not the last segment of the logical message.

MQC.MQSS_LAST_SEGMENT
Message is the last segment of the logical message. This is also the value returned if the logical message consists of only one segment.

segmentation*

```
public char segmentation
```

This is an output field that indicates whether or not segmentation is allowed for the retrieved message is a segment of a logical message. Possible values are:

MQC.MQSEG_INHIBITED
Segmentation not allowed.

MQC.MQSEG_ALLOWED
Segmentation allowed.

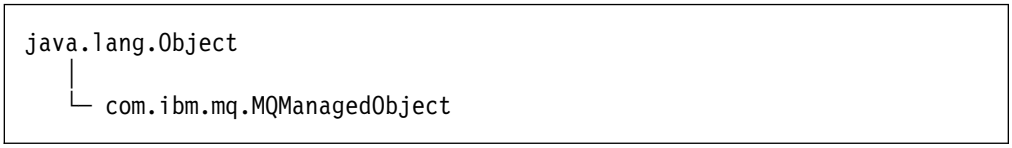
Constructors

MQGetMessageOptions

```
public MQGetMessageOptions()
```

Construct a new MQGetMessageOptions object with options set to MQC.MQGMO_NO_WAIT, a wait interval of zero, and a blank resolved queue name.

MQManagedObject



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

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

Variables

alternateUserId
`public String alternateUserId`

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

name
`public String name`

The name of this resource (either the name supplied on the access method, or the name allocated by the queue manager for a dynamic queue). Setting this attribute has no effect.

openOptions
`public int openOptions`

The options specified when this resource was opened. Setting this attribute has no effect.

isOpen
`public boolean isOpen`

Indicates whether this resource is currently open. Setting this attribute 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. 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 IOException.

Return the description of this resource as held at the queue manager.

If this method is called after the resource has been closed, an IOException 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 attribute values 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 attribute values can be set using the setXXX() methods defined in MQQueue.

Parameters

selectors
Integer array identifying the attributes with values to be set.

intAttrs
The array of integer attribute values to be set. These values must be in the same order as the integer attribute selectors in the selectors array.

charAttrs
The buffer in which the character attributes to be set are concatenated. These values must be in the same order as the character attribute selectors in the selectors array. The length of each character attribute is fixed.

Throws MQException if the set fails.

close

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

Throws MQException.

Close the object. No further operations against this resource are permitted after this method has been called. The behavior of the close method may be altered by setting the closeOptions attribute.

Throws MQException if the MQSeries call fails

MQMessage

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

public class **MQMessage**
implements **DataInput**, **DataOutput**

MQMessage represents both the message descriptor and the data for an MQSeries message. There is group of readXXX methods for reading data from a message, and a group of writeXXX data for writing data into a message. The format of numbers and strings used by these read and write methods can be controlled by the encoding and characterSet member variables. The remaining member variables contain control information that accompanies the application message data when a message travels between sending and receiving applications. The application can set values into the member variable before putting a message to a queue and can read values after retrieving a message from a queue.

Variables

report

```
public int report
```

A report is a message about another message. This member variable enables the application sending the original message to specify which report messages are required, whether the application message data is to be included in them, and also how the message and correlation identifiers in the report or reply are to be set. Any, all or none of the following report types can be requested:

- Exception
- Expiration
- Confirm on arrival
- Confirm on delivery

For each type, only one of the three corresponding values below should be specified, depending on whether the application message data is to be included in the report message.

Note: Values marked with ****** in the following list are not supported by MVS queue managers and should not be used if your application is likely to access an MVS queue manager, regardless of the platform on which the application is running.

The valid values are:

- MQC.MQRO_EXCEPTION
- MQC.MQRO_EXCEPTION_WITH_DATA
- MQC.MQRO_EXCEPTION_WITH_FULL_DATA******
- MQC.MQRO_EXPIRATION
- MQC.MQRO_EXPIRATION_WITH_DATA
- MQC.MQRO_EXPIRATION_WITH_FULL_DATA******
- MQC.MQRO_COA
- MQC.MQRO_COA_WITH_DATA
- MQC.MQRO_COA_WITH_FULL_DATA******

- MQC.MQRO_COD
- MQC.MQRO_COD_WITH_DATA
- MQC.MQRO_COD_WITH_FULL_DATA**

You can specify one of the following to control how the message Id is generated for the report or reply message:

- MQC.MQRO_NEW_MSG_ID
- MQC.MQRO_PASS_MSG_ID

You can specify one of the following to control how the correlation Id of the report or reply message is to be set:

- MQC.MQRO_COPY_MSG_ID_TO_CORREL_ID
- MQC.MQRO_PASS_CORREL_ID

You can specify one of the following to control the disposition of the original message when it cannot be delivered to the destination queue:

- MQC.MQRO_DEAD_LETTER_Q
- MQC.MQRO_DISCARD_MSG **

If no report options are specified, the default is:

```
MQC.MQRO_NEW_MSG_ID |
MQC.MQRO_COPY_MSG_ID_TO_CORREL_ID |
MQC.MQRO_DEAD_LETTER_Q
```

You can specify one or both of the following to request that the receiving application send a positive action or negative action report message.

- MQRO_PAN
- MQRO_NAN

messageType

```
public int messageType
```

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

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

Application-defined values can also be used. These should be in the range MQC.MQMT_APPL_FIRST to MQC.MQMT_APPL_LAST.

The default value of this field is MQC.MQMT_DATAGRAM.

expiry

```
public int expiry
```

An expiry time expressed in tenths of a second, set by the application that puts the message. After a message's expiry time has elapsed, it is eligible to be discarded by the queue manager. If the message specified one of the MQC.MQRO_EXPIRATION flags, a report is generated when the message is discarded.

The default value is MQC.MQEI_UNLIMITED, meaning that the message never expires.

feedback

```
public int feedback
```

This is used with a message of type MQC.MQMT_REPORT to indicate the nature of the report. The following feedback codes are defined by the system:

- MQC.MQFB_EXPIRATION
- MQC.MQFB_COA
- MQC.MQFB_COD
- MQC.MQFB_QUIT
- MQC.MQFB_PAN
- MQC.MQFB_NAN
- MQC.MQFB_DATA_LENGTH_ZERO
- MQC.MQFB_DATA_LENGTH_NEGATIVE
- MQC.MQFB_DATA_LENGTH_TOO_BIG
- MQC.MQFB_BUFFER_OVERFLOW
- MQC.MQFB_LENGTH_OFF_BY_ONE
- MQC.MQFB_IIH_ERROR

Application-defined feedback values in the range MQC.MQFB_APPL_FIRST to MQC.MQFB_APPL_LAST can also be used.

The default value of this field is MQC.MQFB_NONE, indicating that no feedback is provided.

encoding

```
public int encoding
```

This member variable specifies the representation used for numeric values in the application message data; this applies to binary, packed decimal, and floating point data. The behavior of the read and write methods for these numeric formats is altered accordingly.

The following encodings are defined for binary integers:

MQC.MQENC_INTEGER_NORMAL
Big-endian integers, as in Java

MQC.MQENC_INTEGER_REVERSED
Little-endian integers, as used by PCs

The following encodings are defined for packed-decimal integers:

MQC.MQENC_DECIMAL_NORMAL
Big-endian packed-decimal, as used by System/390

MQC.MQENC_DECIMAL_REVERSED
Little-endian packed-decimal

The following encodings are defined for floating-point numbers:

MQC.MQENC_FLOAT_IEEE_NORMAL
Big-endian IEEE floats, as in Java

MQC.MQENC_FLOAT_IEEE_REVERSED
Little-endian IEEE floats, as used by PCs

MQC.MQENC_FLOAT_S390
System/390 format floating points

A value for the encoding field should be constructed by adding together one value from each of these three sections (or using the bitwise OR operator). The default value is:

```
MQC.MQENC_INTEGER_NORMAL |
MQC.MQENC_DECIMAL_NORMAL |
MQC.MQENC_FLOAT_IEEE_NORMAL
```

For convenience, this value is also represented by MQC.MQENC_NATIVE. This setting causes writeInt() to write a big-endian integer, and readInt() to read a big-endian integer. If the flag MQC.MQENC_INTEGER_REVERSED flag had been set instead, writeInt() would write a little-endian integer, and readInt() would read a little-endian integer.

Note that a loss in precision can occur when converting from IEEE format floating points to System/390 format floating points.

characterSet

```
public int characterSet
```

This specifies the coded character set identifier of character data in the application message data. The behavior of the readString, readLine and writeString methods is altered accordingly.

The default value for this field is MQC.MQCCSI_Q_MGR, which specifies that character data in the application message data is in the queue manager's character set. The additional character set values shown in Table 6 are supported.

Table 6 (Page 1 of 2). Character set identifiers

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

Table 6 (Page 2 of 2). Character set identifiers

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

format

```
public String format
```

A format name used by the sender of the message to indicate to the receiver the nature of the data in the message. You can use your own format names, but names beginning with the letters "MQ" have meanings that are defined by the queue manager. The queue manager built-in formats are:

MQC.MQFMT_NONE
No format name

MQC.MQFMT_ADMIN
Command server request/reply message

MQC.MQFMT_COMMAND_1
Type 1 command reply message

MQC.MQFMT_COMMAND_2
Type 2 command reply message

MQC.MQFMT_DEAD_LETTER_HEADER
Dead letter header

MQC.MQFMT_EVENT
Event message

MQC.MQFMT_PCF
User-defined message in programmable command format

MQC.MQFMT_STRING
Message consisting entirely of characters

MQC.MQFMT_TRIGGER
Trigger message

MQC.MQFMT_XMIT_Q_HEADER
Transmission queue header

The default value is MQC.MQFMT_NONE.

priority

```
public int priority
```

The message priority. The special value MQC.MQPRI_PRIORITY_AS_Q_DEF can also be set in outbound messages, in which case the priority for the message is taken from the default priority attribute of the destination queue.

The default value is MQC.MQPRI_PRIORITY_AS_Q_DEF.

persistence

```
public int persistence
```

Message persistence. The following values are defined:

- MQC.MQPER_PERSISTENT
- MQC.MQPER_NOT_PERSISTENT
- MQC.MQPER_PERSISTENCE_AS_Q_DEF

The default value is MQC.MQPER_PERSISTENCE_AS_Q_DEF, which indicates that the persistence for the message should be taken from the default persistence attribute of the destination queue.

messageId

```
public byte messageId[]
```

For an MQQueue.get() call, this field specifies the message identifier of the message to be retrieved. Normally the queue manager returns the first message with a message identifier and correlation identifier match those specified. The special value MQC.MQMI_NONE allows *any* message identifier to match.

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

The default value is MQC.MQMI_NONE.

correlationId

```
public byte correlationId[]
```

For an MQQueue.get() call, this field specifies the correlation identifier of the message to be retrieved. Normally the queue manager returns the first message with a message identifier and correlation identifier that match those specified. The special value MQC.MQCI_NONE allows *any* correlation identifier to match.

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

The default value is MQC.MQCI_NONE.

backoutCount

```
public int backoutCount
```

A count of the number of times the message has previously been returned by an MQQueue.get() call as part of a unit of work, and subsequently backed out.

The default value is zero.

replyToQueueName

```
public String replyToQueueName
```

The name of the message queue to which the application that issued the get request for the message should send MQC.MQMT_REPLY and MQC.MQMT_REPORT messages.

The default value is "".

replyToQueueManagerName

```
public String replyToQueueManagerName
```

The name of the queue manager to which reply or report messages should be sent.

The default value is "".

If the value is "" on an MQQueue.put() call, the QueueManager fills in the value.

userId

```
public String userId
```

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

The default value is "".

accountingToken

```
public byte accountingToken[]
```

Part of the identity context of the message; it allows an application to cause work done as a result of the message to be appropriately charged.

The default value is "MQC.MQACT_NONE".

applicationIdData

```
public String applicationIdData
```

Part of the identity context of the message; it is information that is defined by the application suite, and can be used to provide additional information about the message or its originator.

The default value is "".

putApplicationType

```
public int putApplicationType
```

The type of application that put the message. This may be a system or user defined value. The following values are defined by the system:

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

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

putApplicationName

```
public String putApplicationName
```

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

putDateTime

```
public GregorianCalendar putDateTime
```

The time and date that the message was put.

applicationOriginData

```
public String applicationOriginData
```

Information defined by the application that can be used to provide additional information about the origin of the message.

The default value is "".

groupId

```
public byte[] groupId
```

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

The default value is "MQC.MQGI_NONE".

messageSequenceNumber

```
public int messageSequenceNumber
```

The sequence number of a logical message within a group.

offset

```
public int offset
```

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

messageFlags

```
public int messageFlags
```

Flags controlling the segmentation and status of a message.

originalLength

```
public int originalLength
```

The original length of a segmented message.

Constructors

MQMessage

```
public MQMessage()
```

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

Methods

getTotalMessageLength

```
public int getTotalMessageLength()
```

The total number of bytes in the message as stored on the message queue from which this message was retrieved (or attempted to be retrieved). When an MQQueue.get() method fails with a message-truncated error code, this method tells you the total size of the message on the queue.

See also "MQQueue.get" on page 94.

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 class about the size of buffer that may be required for subsequent get operations. If the message currently contains message data, and the new size is less than the current size, the message data is truncated.

readBoolean

```
public boolean readBoolean()
```

Throws IOException.

Read a (signed) byte from the current position in the message buffer.

readChar

```
public char readChar()
```

Throws IOException, EOFException.

Read a Unicode character from the current position in the message buffer.

readDouble

```
public double readDouble()
```

Throws IOException, EOFException.

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

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

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

readFloat

```
public float readFloat()
```

Throws IOException, EOFException.

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

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

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

readFully

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

Throws Exception, EOFException.

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

readFully

```
public void readFully(byte b[],
                      int off,
                      int len)
```

Throws IOException, EOFException.

Fill *len* elements of the byte array b with data from the message buffer, starting at offset *off*.

readInt

```
public int readInt()
```

Throws IOException, EOFException.

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

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

readInt4

```
public int readInt4()
```

Throws IOException, EOFException.

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

readLine

```
public String readLine()
```

Throws IOException.

Converts from the codeset identified in the characterSet member variable to Unicode, and then reads in a line that has been terminated by \n, \r, \r\n, EOF, or the end of a UTF string.

Note that if you use readLine to read part of a UTF string, you are not able to use readString() or readLine() to read the remainder of that string. This is because the Java UTF format requires a length of string prefix in front of the character data that is not present if you try to read from midway to a string.

readLong

```
public long readLong()
```

Throws IOException, EOFException.

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

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

readInt8

```
public long readInt8()
```

Throws IOException, EOFException.

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

readObject

```
public Object readObject()
```

Throws OptionalDataException, ClassNotFoundException, IOException.

Read an object from the message buffer. The class of the object, the signature of the class, and the value of the non-transient and non-static fields of the class are all read.

readShort

```
public short readShort()
```

Throws IOException, EOFException.

readInt2

```
public short readInt2()
```

Throws IOException, EOFException.

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

readUTF

```
public String readUTF()
```

Throws IOException.

Read a UTF format String from the current position in the message buffer.

readUnsignedByte

```
public int readUnsignedByte()
```

Throws IOException, EOFException.

Read an unsigned byte from the current position in the message buffer.

readUnsignedShort

```
public int readUnsignedShort()
```

Throws IOException, EOFException.

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

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

readUInt2

```
public int readUInt2()
```

Throws IOException, EOFException.

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

readString

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

Throws IOException, EOFException.

Read a string in the codeset identified by the characterSet member variable, and convert it into Unicode.

Parameters:

length The number of characters to read (which may differ from the number of bytes according to the codeset, because some codesets use more than one byte per character).

readDecimal2

```
public short readDecimal2()
```

Throws IOException, EOFException.

Read a 2-byte packed decimal number (-999..999). The behavior of this method is controlled by the value of the encoding member variable. A value of MQC.MQENC_DECIMAL_NORMAL reads a big-endian packed decimal number, and a value of MQC.MQENC_DECIMAL_REVERSED reads a little-endian packed decimal number.

readDecimal4

```
public int readDecimal4()
```

Throws IOException, EOFException.

Read a 4-byte packed decimal number (-9999999..9999999). The behavior of this method is controlled by the value of the encoding member variable. A value of MQC.MQENC_DECIMAL_NORMAL reads a big-endian packed decimal number, and a value of MQC.MQENC_DECIMAL_REVERSED reads a little-endian packed decimal number.

readDecimal8

```
public long readDecimal8()
```

Throws IOException, EOFException.

Read an 8-byte packed decimal number (-999999999999999 to 999999999999999). The behavior of this method is controlled by the encoding member variable. A value of MQC.MQENC_DECIMAL_NORMAL reads a big-endian packed decimal number, and MQC.MQENC_DECIMAL_REVERSED reads a little-endian packed decimal number.

setVersion

```
public void setVersion(int version)
```

Specifies which version of the structure to use. Possible values are:

- MQC.MQMD_VERSION_1
- MQC.MQMD_VERSION_2

You should not normally need to call this method unless you wish to force the client to use a version 1 structure when connected to a queue manager that is capable of handling version 2 structures. In all other situations, the client determines the correct version of the structure to use by querying the queue manager's capabilities.

skipBytes

```
public int skipBytes(int n)
```

Throws IOException, EOFException.

Move forward n bytes in the message buffer.

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.

Write a string as a sequence of bytes into the message buffer at the current position.

writeChar

```
public void writeChar(int v)
```

Throws IOException.

Write a Unicode character into the message buffer at the current position.

writeChars

```
public void writeChars(String s)
```

Throws IOException.

Write a string as a sequence of Unicode characters into the message buffer at the current position.

writeDouble

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

Throws IOException

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

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

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

writeFloat

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

Throws IOException.

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

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

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

writeInt

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

Throws IOException.

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

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

writeInt4

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

Throws IOException.

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

writeLong

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

Throws IOException.

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

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

writeInt8

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

Throws IOException.

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

writeObject

```
public void writeObject(Object obj)
```

Throws IOException.

Write the specified object to the message buffer. The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all its supertypes are all written.

writeShort

```
public void writeShort(int v)
```

Throws IOException.

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

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

writeInt2

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

Throws IOException.

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

writeDecimal2

```
public void writeDecimal2(short v)
```

Throws IOException.

Write a 2-byte packed decimal format number into the message buffer at the current position. The behavior of this method is determined by the value of the encoding member variable.

A value of MQC.MQENC_DECIMAL_NORMAL writes a Big-endian packed decimal, a value of MQC.MQENC_DECIMAL_REVERSED writes a Little-endian packed decimal.

Parameters

v can be in the range -999 to 999.

writeDecimal4

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

Throws IOException.

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

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

Parameters

v can be in the range -9999999 to 9999999.

writeDecimal8

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

Throws IOException.

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

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

Parameters:

v can be in the range -999999999999999 to 999999999999999.

writeUTF

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

Throws IOException.

Write a string in UTF format into the message buffer at the current position.

writeString

```
public void writeStringString(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 53) where it is used to tailor message parameters for a given destination in a distribution list.

Variables

feedback

```
public int feedback
```

This is used with a message of type MQC.MQMT_REPORT to indicate the nature of the report. The following feedback codes are defined by the system:

- MQC.MQFB_EXPIRATION
- MQC.MQFB_COA
- MQC.MQFB_COD
- MQC.MQFB_QUIT
- MQC.MQFB_PAN
- MQC.MQFB_NAN
- MQC.MQFB_DATA_LENGTH_ZERO
- MQC.MQFB_DATA_LENGTH_NEGATIVE
- MQC.MQFB_DATA_LENGTH_TOO_BIG
- MQC.MQFB_BUFFER_OVERFLOW
- MQC.MQFB_LENGTH_OFF_BY_ONE
- MQC.MQFB_IIH_ERROR

Application defined feedback values in the range MQC.MQFB_APPL_FIRST to MQC.MQFB_APPL_LAST can also be used.

The default value of this field is MQC.MQFB_NONE, indicating that no feedback is provided.

messageId

```
public byte messageId[]
```

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

The default value is MQC.MQMI_NONE.

correlationId

```
public byte correlationId[]
```

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

The default value is MQC.MQCI_NONE.

accountingToken

```
public byte accountingToken[]
```

This is part of the identity context of the message. It allows an application to cause work done as a result of the message to be appropriately charged.

The default value is "MQC.MQACT_NONE".

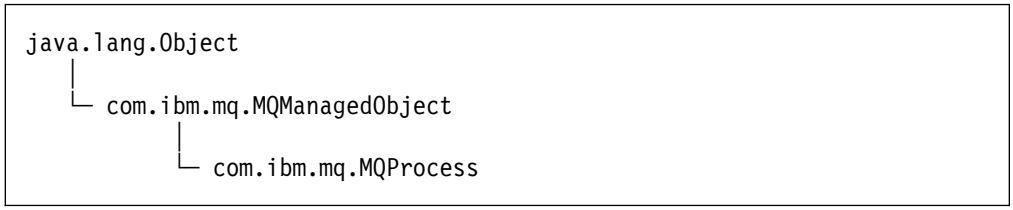
groupId

```
public byte[] groupId
```

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

The default value is "MQC.MQGI_NONE".

MQProcess



public class **MQProcess**
extends **MQManagedObject**. (on page 65.)

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 101 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 59)

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 67

Throws MQException if you have already closed the queue when you call this method.

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 41 for details.

Variables

options

```
public int options
```

Options that control the action of MQQueue.put. Any or none of the following values can be specified. If more than one option is required the values can be added together or combined using the bitwise OR operator.

MQC.MQPMO_SYNCPOINT
Put a message with syncpoint control. The message is not visible outside the unit of work until the unit of work is committed. If the unit of work is backed out, the message is deleted.

MQC.MQPMO_NO_SYNCPOINT
Put a message without syncpoint control.

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

MQC.MQPMO_DEFAULT_CONTEXT
Associate default context with the message.

MQC.MQPMO_SET_IDENTITY_CONTEXT
Set identity context from the application.

MQC.MQPMO_SET_ALL_CONTEXT
Set all context from the application.

MQC.MQPMO_FAIL_IF_QUIESCING
Fail if the queue manager is quiescing.

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

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

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

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

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

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

MQC.MQPMO_ALTERNATE_USER_AUTHORITY
Validate with specified user identifier.

contextReference
`public MQQueue ContextReference`

This is an input field which indicates the source of the context information.

If the `options` field includes MQC.MQPMO_PASS_IDENTITY_CONTEXT, or MQC.MQPMO_PASS_ALL_CONTEXT, set this field to refer to the MQQueue from which the context information should be taken.

The initial value of this field is null.

recordFields *
`public int recordFields`

Flags indicating which fields are to be customized on a per-queue basis when putting a message to a distribution list. One or more of the following flags can be specified:

MQC.MQPMRF_MSG_ID
Use the messageId attribute in the MQDistributionListItem.

MQC.MQPMRF_CORREL_ID
Use the correlationId attribute in the MQDistributionListItem.

MQC.MQPMRF_GROUP_ID
Use the groupId attribute in the MQDistributionListItem.

MQC.MQPMRF_FEEDBACK
Use the feedback attribute in the MQDistributionListItem.

MQC.MQPMRF_ACCOUNTING_TOKEN
Use the accountingToken attribute in the MQDistributionListItem.

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

resolvedQueueName.
`public String resolvedQueueName`

This is an output field that is set by the queue manager to the name of the queue on which the message is placed. This may be different from the name used to open the queue if the opened queue was an alias or model queue.

resolvedQueueManagerName
`public String resolvedQueueManagerName`

This is an output field set by the queue manager to the name of the queue manager that owns the queue specified by the remote queue name. This may be different from the name of the queue manager from which the queue was accessed if the queue is a remote queue.

knownDestCount *

```
public int knownDestCount
```

This is an output field set by the queue manager to the number of messages that the current call has sent successfully to queues that resolve to local queues. This field is also set when opening a single queue that is not part of a distribution list.

unknownDestCount *

```
public int unknownDestCount
```

This is an output field set by the queue manager to the number of messages that the current call has sent successfully to queues that resolve to remote queues. This field is also set when opening a single queue that is not part of a distribution list.

invalidDestCount *

```
public int invalidDestCount
```

This is an output field set by the queue manager to the number of messages that could not be sent to queues in a distribution list. The count includes queues that failed to open as well as queues that were opened successfully, but for which the put operation failed. This field is also set when opening a single queue that is not part of a distribution list.

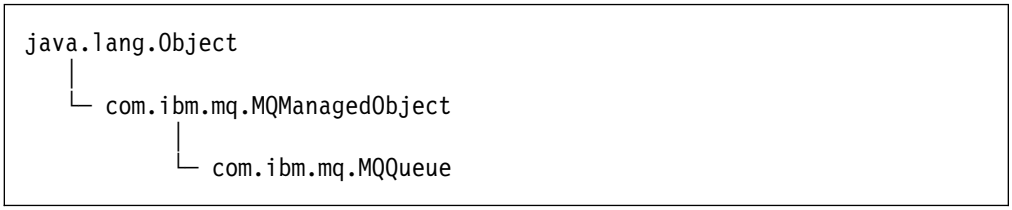
Constructors

MQPutMessageOptions

```
public MQPutMessageOptions()
```

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

MQQueue



public class **MQQueue**
extends **MQManagedObject**. (See page 65.)

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 104.

Constructors

MQQueue:

```
public MQQueue(MQQueueManager qMgr, String queueName, int openOptions,
               String queueManagerName, String dynamicQueueName,
```

Throws MQException.

Access a queue on the queue manager `qMgr`.

See "MQQueueManager.accessQueue" on page 104 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 the MQMessage documentation for details.)

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 the MQGetMessageOptions documentation for details.)

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, then 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 then the message is left on the queue and an MQException is raised with completion code MQException.MQCC_FAILED 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 the MQMessage documentation for details.)

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 61 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.

Note that performing a put updates the messageId and correlationId, which must be taken into consideration when making further calls to put/get using the same MQMessage object. Also note that 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 the MQPutMessageOptions documentation for details.)

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 GregorianCalander 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_MODEL
- 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

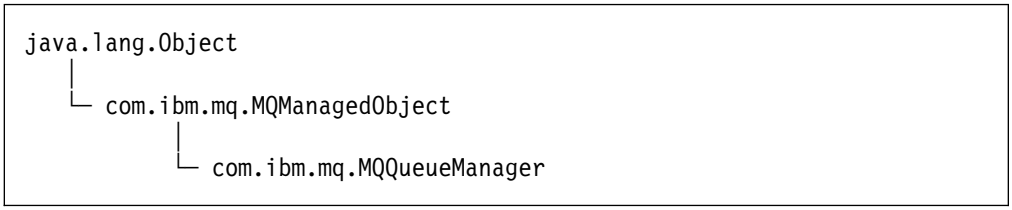
close

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

Throws MQException.

Override of "MQManagedObject.close" on page 67

MQQueueManager



public class **MQQueueManager**
extends **MQManagedObject**. (See page 65.)

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 6, “Environment dependent behavior” on page 39 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 the 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 55.

MQQueueManager

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

Throws MQException.

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

- MQC.MQCNO_FASTPATH_BINDING for fast bindings *****
- MQC.MQCNO_STANDARD_BINDING for normal bindings

MQQueueManager

```
public MQQueueManager(String queueManagerName,
java.util.Hashtable properties)
```

The properties parameter takes a series of key/value pairs that describe the MQSeries environment for this particular queue manager. These properties, where specified, override the values set by the MQEnvironment class, and allow the individual properties to be set on a queue manager by queue manager basis. See "MQEnvironment.properties" on page 56.

Methods

getCharacterSet

```
public int getCharacterSet()
```

Throws MQException.

Returns the CCSID (Coded Character Set Identifier) of the queue manager's codeset. This defines the character set used by the queue manager for all character string fields in the application programming interface.

Throws MQException if you call this method after disconnecting from the queue manager.

getMaximumMessageLength

```
public int getMaximumMessageLength()
```

Throws MQException.

Returns the maximum length of a message (in bytes) that can be handled by the queue manager. No queue can be defined with a maximum message length greater than this.

Throws MQException if you call this method after disconnecting from the queue manager.

getCommandLevel

```
public int getCommandLevel()
```

Throws MQException.

Indicates the level of system control commands supported by the queue manager. The set of system control commands that correspond to a particular command level varies according to the architecture of the platform on which the queue manager is running. See the MQSeries documentation for your platform for further details.

Throws MQException if you call this method after disconnecting from the queue manager.

Returns

One of the MQC.MQCMDL_LEVEL_xxx constants

getCommandInputQueueName

```
public String getCommandInputQueueName()
```

Throws MQException.

Returns the name of the command input queue defined on the queue manager. This is a queue to which applications can send commands, if authorized to do so.

Throws MQException if you call this method after disconnecting from the queue manager.

getMaximumPriority

```
public int getMaximumPriority()
```

Throws MQException.

Returns the maximum message priority supported by the queue manager. Priorities range from zero (lowest) to this value.

Throws MQException if you call this method after disconnecting from the queue manager.

getSyncpointAvailability

```
public int getSyncpointAvailability()
```

Throws MQException.

Indicates whether the queue manager supports units of work and syncpointing with the MQQueue.get and MQQueue.put methods.

Returns

- MQC.MQSP_AVAILABLE if syncpointing is available
- MQC.MQSP_NOT_AVAILABLE if syncpointing is not available

Throws MQException if you call this method after disconnecting from the queue manager.

getDistributionListCapable

```
public boolean getDistributionListCapable()
```

Indicates whether the queue manager supports distribution lists

disconnect

```
public synchronized void disconnect()
```

Throws MQException.

Terminates the connection to the queue manager. All open queues and processes accessed by this queue manager are closed, and hence become unusable. When you have disconnected from a queue manager the only way to reconnect is to create a new MQQueueManager object.

commit

```
public synchronized void commit()
```

Throws MQException.

Calling this method indicates to the queue manager that the application has reached a syncpoint, and that all of the message gets and puts that have occurred since the last syncpoint are to be made permanent. Messages put as part of a unit of work (with the MQC.MQPMO_SYNCPOINT flag set in the options field of MQPutMessageOptions) are made available to other applications. Messages retrieved as part of a unit of work (with the MQC.MQGMO_SYNCPOINT flag set in the options field of MQGetMessageOptions) are deleted.

See also the description of "backout" that follows.

backout

```
public synchronized void backout()
```

Throws MQException.

Calling this method indicates to the queue manager that all the message gets and puts that have occurred since the last syncpoint are to be backed out. Messages put as part of a unit of work (with the MQC.MQPMO_SYNCPOINT flag set in the options field of MQPutMessageOptions) are deleted; messages retrieved as part of a unit of work (with the MQC.MQGMO_SYNCPOINT flag set in the options field of MQGetMessageOptions) are reinstated on the queue.

See also the decription of "commit" above.

accessQueue

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

Throws MQException.

Establishes access to an MQSeries queue on this queue manager to get or browse messages, put messages, inquire about the attributes of the queue or set the attributes of the queue.

If the queue named is a model queue, then a dynamic local queue is created. The name of the created queue can be determined by inspecting the `name` attribute of the returned MQQueue object.

Parameters

queueName
Name of queue to open

openOptions
Options that control the opening of the queue. Valid options are:

MQC.MQOO_BROWSE
Open to browse message.

MQC.MQOO_INPUT_AS_Q_DEF
Open to get messages using queue-defined default.

MQC.MQOO_INPUT_SHARED
Open to get messages with shared access.

MQC.MQOO_INPUT_EXCLUSIVE
Open to get messages with exclusive access.

MQC.MQOO_OUTPUT
Open to put messages.

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

MQC.MQOO_SET
Open to set attributes.

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

MQC.MQOO_SET_IDENTITY_CONTEXT
Allows identity context to be set.

MQC.MQOO_SET_ALL_CONTEXT
Allows all context to be set.

MQC.MQOO_ALTERNATE_USER_AUTHORITY
Validate with the specified user identifier.

MQC.MQOO_FAIL_IF_QUIESCING
Fail if the queue manager is quiescing.

MQC.MQOO_BIND_AS_QDEF
Use default binding for queue.

MQC.MQOO_BIND_ON_OPEN
Bind handle to destination when queue is opened.

MQC.MQOO_BIND_NOT_FIXED
Do not bind to a specific destination.

If more than one option is required the values can be added together or combined using the bitwise OR operator. See the MQSeries *Application Programming Reference* for a fuller description of these options.

queueManagerName
Name of the queue manager on which the queue is defined. A name which is entirely blank, or which is null, denotes the queue manager to which this MQQueueManager object is connected.

dynamicQueueName
This parameter is ignored unless queueName specifies the name of a model queue. If it does, this parameter specifies the name of the dynamic queue to be created. A blank or null name is not valid if queueName specifies the name of a model queue. If the last non-blank character in the name is an asterisk (*), the queue manager replaces the asterisk with a string of characters that guarantees that the name generated for the queue is unique on this queue manager.

alternateUserId
If MQOO_ALTERNATE_USER_AUTHORITY is specified in the openOptions parameter this parameter specifies the alternate user

identifier that is to be used to check the authorization for the open. If MQOO_ALTERNATE_USER_AUTHORITY is not specified this parameter can be left blank (or null).

Returns
MQQueue that has been successfully opened

Throws MQException if the open fails.

See also "accessProcess" on page 106.

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 "accessQueue" on page 104 for details of the parameters.

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

accessProcess

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

Throws MQException.

Establishes access to an MQSeries process on this queue manager to inquire about the process attributes.

Parameters

processName
Name of process to open.

openOptions
Options that control the opening of the process. Inquire is automatically added to the options specified, so there is no need to specify it explicitly.

Valid options are:

MQC.MQOO_ALTERNATE_USER_AUTHORITY
Validate with the specified user id.

MQC.MQOO_FAIL_IF_QUIESCING
Fail if the queue manager is quiescing.

If more than one option is required,the values can be added together or combined using the bitwise OR operator. See the *MQSeries Application Programming Reference* for a fuller description of these options.

queueManagerName
Name of the queue manager on which the process is defined. Applications should leave this parameter blank or null.

alternateUserId
If MQOO_ALTERNATE_USER_AUTHORITY is specified in the openOptions parameter this parameter specifies the alternate user identifier that is to be used to check the authorization for the open. If MQOO_ALTERNATE_USER_AUTHORITY is not specified this parameter can be left blank (or null).

Returns
MQProcess that has been successfully opened.

Throws MQException if the open fails.

See also "accessQueue" on page 104

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 106 for details of the options.

queueManagerName and *alternateUserId* are set to "".

accessDistributionList

```
public synchronized MQDistributionList accessDistributionList
          (
           MQDistributionListItem[] litems, int openOptions,
           String alternateUserId
          )
```

Throws MQException.

Parameters

litems
The items to be included in the distribution list.

openOptions
Options that control the opening of the distribution list.

alternateUserId
If MQOO_ALTERNATE_USER_AUTHORITY is specified in the openOptions parameter this parameter specifies the alternate user identifier that is to be used to check the authorization for the open. If MQOO_ALTERNATE_USER_AUTHORITY is not specified this parameter can be left blank (or null).

Returns
A newly created MQDistributionList which is open and ready for put operations.

Throws MQException if the open fails.

See also "accessQueue" on page 104.

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 107 for details of the parameters.

alternateUserId is set to "".

begin* (bindings connection only)

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

Throws MQException.

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

isConnected

```
public boolean void isConnected()
```

Returns the value of the isConnected variable.

MQC

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

The MQC interface defines all the constants used by the MQSeries classes for Java. To refer to one of these constants from within your programs, prefix the constant name with "MQC.". For example, you can set the close options for a queue as follows:

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

A full description of these constants can be found inChapter 6, “MQSeries constants” in the *MQSeries Application Programming Reference* book.

MQReceiveExit

public interface **MQReceiveExit**
extends **Object**

The receive exit interface allows you to examine and possibly alter the data received from the queue manager by the MQSeries classes for Java.

Note: This class 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 receive 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 48 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 109
- “MQChannelDefinition” on page 46

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 class 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 classes for Java what action to take next. See the “MQChannelExit” on page 48 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 109
- “MQChannelDefinition” on page 46

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 classes for Java.

Note: This class 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 wish 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 48 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 109
- “MQChannelDefinition” on page 46

Part 4. Appendix

Appendix A. Notices

This information was developed for products and services offered in the United States. IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM documentation or non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those documents or Web sites. The materials for those documents or Web sites are not part of the materials for this IBM product and use of those documents or Web sites is at your own risk.

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.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	AS/400	BookManager
IBM	IBMLink	MQSeries
MVS/ESA	OS/2	OS/390
OS/400	System/390	

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and/or other countries licensed exclusively through X/Open Company Limited.

Other company, product, or service names, may be the trademarks or service marks of others.

Part 5. Glossary and Index

Glossary of terms and abbreviations

This glossary describes terms used in this book and words used with other than their everyday meaning. In some cases, a definition may not be the only one applicable to a term, but it gives the particular sense in which the word is used in this book.

If you do not find the term you are looking for, see the index or the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

applet. A Java program which is designed to run only on a web page.

Application Programming Interface (API). An Application Programming Interface consists of the functions and variables that programmers are allowed to use in their applications.

channel. See MQI channel.

class. A class is an encapsulated collection of data and methods to operate on the data. A class may be instantiated to produce an object that is an instance of the class.

client. In MQSeries, a client is a run-time component that provides access to queuing services on a server for local user applications.

encapsulation. Encapsulation is an object-oriented programming technique that makes an object's data private or protected and allows programmers to access and manipulate the data only through method calls.

HTML. HTML (Hypertext Markup Language) is a language used to define information that is to be displayed on the World Wide Web.

IIOP. Internet Inter-ORB Protocol. A standard for TCP/IP communications between ORBs from different vendors.

instance. An instance is an object. When a class is instantiated to produce an object, we say that the object is an instance of the class.

interface. An interface is a class that contains only abstract methods and no instance variables. An interface provides a common set of methods that can be implemented by subclasses of a number of different classes.

Internet. The Internet is a cooperative public network of shared information. Physically, the Internet uses a subset of the total resources of all the currently existing public telecommunication networks. Technically, what distinguishes the Internet as a cooperative public network is its use of a set of protocols called TCP/IP (Transport Control Protocol/Internet Protocol).

Java Developers Kit (JDK). A package of software distributed by Sun Microsystems for Java developers. It includes the Java interpreter, Java classes and Java development tools: compiler, debugger, disassembler, appletviewer, stub file generator, and documentation generator.

message. In message queuing applications, a message is a communication sent between programs.

message queue. See queue

message queuing. A programming technique in which each program within an application communicates with the other programs by putting messages on queues.

method. Method is the object-oriented programming term for a function or procedure.

MQI channel. An MQI channel connects an MQSeries client to a queue manager on a server system and transfers MQI calls and responses in a bidirectional manner.

MQSeries. MQSeries is a family of IBM licensed programs that provide message queuing services.

object. (1) In Java, an object is an instance of a class. A class models a group of things; an object models a particular member of that group. (2) In MQSeries, an object is a queue manager, a queue, or a channel.

Object Request Broker (ORB). An application framework that provides interoperability between objects, built in different languages, running on different machines, in heterogeneous distributed environments.

package. A package in Java is a way of giving a piece of Java code access to a specific set of classes. Java code that is part of a particular package has access to all the classes in the package and to all non-private methods and fields in the classes.

private. A private field is not visible outside its own class.

protected. A protected field is visible only within its own class, within a subclass, or within packages of which the class is a part

public. A public class or interface is visible everywhere. A public method or variable is visible everywhere that its class is visible

queue. A queue is an MQSeries object. Message queueing applications can put messages on, and get messages from, a queue

queue manager. a queue manager is a system program the provides message queuing services to applications.

server. (1) An MQSeries a server is a queue manager that provides message queuing services to client applications running on a remote workstation. (2) More generally, a server is a program that responds to requests for information in the particular two-program information flow model of client/server. (3) The computer on which a server program runs.

servlet. A Java program which is designed to run only on a web server.

subclass. A subclass is a class that extends another. The subclass inherits the public and protected methods and variables of its superclass.

superclass. A superclass is a class that is extended by some other class. The superclass's public and protected methods and variables are available to the subclass.

TCP/IP. Transmission Control Protocol/Internet Protocol. A set of communication protocols that support peer-to-peer connectivity functions for both local and wide area networks.

Visibroker for Java. An Object Request Broker (ORB) written in Java

Web. See World Wide Web.

Web browser. A program that formats and displays information that is distributed on the World Wide Web.

World Wide Web (Web). The World Wide Web is an Internet service, based on a common set of protocols, which allows a particularly configured server computer to distribute documents across the Internet in a standard way.

Index

S

T

U

V

W

Sending your comments to IBM

MQSeries®

Using Java®

SC34-5456-00

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 request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should 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, use the Readers' Comment Form
- 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 you use, ensure that you include:

- The publication number and title
- The page number or topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Readers' Comments

MQSeries®

Using Java®

SC34-5456-00

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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.

Be sure to print your name and address below if you would like a reply.

Name	Address
Company or Organization	
Telephone	Email

You can send your comments POST FREE on this form from any one of these countries:

Australia	Finland	Iceland	Netherlands	Singapore	United States
Belgium	France	Israel	New Zealand	Spain	of America
Bermuda	Germany	Italy	Norway	Sweden	
Cyprus	Greece	Luxembourg	Portugal	Switzerland	
Denmark	Hong Kong	Monaco	Republic of Ireland	United Arab Emirates	

If your country is not listed here, your local IBM representative will be pleased to forward your comments to us. Or you can pay the postage and send the form direct to IBM (this includes mailing in the U.K.).

2 Fold along this line

By air mail
Par avion

IBRS/CCRI NUMBER: PHQ - D/1348/SO

NE PAS AFFRANCHIR

NO STAMP REQUIRED

REPONSE PAYEE
GRANDE-BRETAGNE

IBM United Kingdom Laboratories
Information Development Department (MP095)
Hursley Park,
WINCHESTER, Hants
SO21 2ZZ United Kingdom

3 Fold along this line

From: Name _____

Company or Organization _____

Address _____

_____

EMAIL _____

Telephone _____

4 Fasten here with adhesive tape

1 Cut along this line

1 Cut along this line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC34-5456-00