WebSphere MQ

Using Java

Version 7.0

SC34-6935-01

WebSphere MQ

IBM

Using Java

Version 7.0

SC34-6935-01

Note

Before using this information and the product it supports, be sure to read the general information under notices at the back of this book.

Second edition (January 2009)

This edition of the book applies to the following products:

- IBM WebSphere MQ, Version 7.0
- IBM WebSphere MQ for z/OS, Version 7.0

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

Contents

Figures

Tables

Chapter 1. Using Java

Note: Before using this information and the product it supports, be sure to read the general information under notices.

Terms used in this book

The following terms are used in this collection of topics, with the following meanings.

The term *i5/OS*® means any release of i5/OS or OS/400® supported by the current version of WebSphere® MQ for i5/OS.

The term *z/OS*® means any release of z/OS supported by the current version of WebSphere MQ for z/OS.

Linux® is used as a general term for any of the following platforms:
- Linux (POWER™ platform)
- Linux (x86 platform)
- Linux (x86-64 platform)
- Linux (zSeries® s390x platform)

UNIX® system is used as a general term for any of the following platforms:
- AIX®
- HP-UX
- Linux
- Solaris

Windows® system, or just *Windows*, is used as a general term for any of the following platforms:
- Windows 2000
- Windows Server 2003
- Windows XP
- Windows Vista
- Windows Server 2008

The term *real-time connection to a broker* refers to a connection between an application and a broker of WebSphere Event Broker or WebSphere Message Broker in which the application and the broker exchange messages using WebSphere MQ Real-Time Transport. Depending on the configuration, messages might also be delivered to the application using WebSphere MQ Multicast Transport.

The term *IP address* means either an Internet Protocol Version 4 (IPv4) address, expressed as a sequence of decimal numbers separated by dots, or an Internet Protocol Version 6 (IPv6) address, expressed as a sequence of hexadecimal numbers separated by colons.

What is new in Version 7.0?

WebSphere MQ classes for Java™ and WebSphere MQ classes for JMS, as supplied in WebSphere MQ Version 7.0, contain a number of enhancements compared to previous releases.

Enhancements to WebSphere MQ classes for JMS include:

- A layered architecture for generic IBM® JMS providers such as Lotus® Expeditor micro broker, with specific features for Websphere MQ classes for JMS
- Embedded publish/subscribe function
- Asynchronous message consumption
- Message selection performed within the queue manager
- Multiple connections to the queue manager using the same MQI channel can share a single TCP connection
- Read ahead for nonpersistent messages on a client connection
- Sending messages on a client connection without determining whether the queue manager has received the message safely
- New channel exit interfaces are provided, which offer improved functionality and performance
- Better access to JMS properties in a WebSphere MQ application
- Serviceability improvements

In WebSphere MQ Version 7.0, the implementation of WebSphere MQ classes for JMS is no longer dependent on WebSphere MQ classes for Java. WebSphere MQ classes for Java and WebSphere MQ classes for JMS are now peers that use a common Java interface to the MQI.

Enhancements to WebSphere MQ classes for Java include:

- Extensions to the API to support publish/subscribe applications
- Asynchronous message consumption
- Asynchronous message put
- Message selection performed within the queue manager
- Multiple connections to the queue manager using the same MQI channel can share a single TCP connection
- Read ahead for nonpersistent messages on a client connection
- Java classes to process various types of message header
- Java classes to process PCF-structured messages
- Properties can be added to any message.
- New channel exit interfaces are provided, which offer improved functionality and performance
- A client configuration file can be used to specify client configuration options.

Should I use WebSphere MQ classes for Java or WebSphere MQ classes for JMS?

A Java application can use either WebSphere MQ classes for Java or WebSphere MQ classes for JMS to access WebSphere MQ resources. Each approach has its advantages.

WebSphere MQ classes for Java encapsulates the Message Queue Interface (MQI), the native WebSphere MQ API, and uses the same object model as other object-oriented interfaces, whereas WebSphere MQ classes for Java Message Service implements Sun's Java Message Service (JMS) interfaces.

If you are familiar with WebSphere MQ in environments other than Java, using either procedural or object-oriented languages, you can transfer your existing knowledge to the Java environment by using WebSphere MQ classes for Java. You can also exploit the full range of features of WebSphere MQ, not all of which are available in WebSphere MQ classes for JMS.

If you are not familiar with Websphere MQ, or already have JMS experience, you might find it easier to use the familiar JMS API to access WebSphere MQ resources, by using WebSphere MQ classes for JMS. JMS is also an integral part of the Java Platform, Enterprise Edition (Java EE) platform. Java EE applications can use message-driven beans (MDBs) to process messages asynchronously, and MDBs can process only JMS messages. JMS is also the standard mechanism for Java EE to interact with asynchronous messaging systems such as WebSphere MQ. Every application server that is Java EE compliant must include a JMS provider, therefore you can use JMS to communicate between different application servers or you can port an application from one JMS provider to another without any change to the application.

See the linked topics for more information about the advantages of each approach.

Chapter 2. WebSphere MQ classes for JMS

This collection of topics contains the documentation for WebSphere MQ classes for JMS.

Getting started with WebSphere MQ classes for JMS

This topic provides an overview of WebSphere MQ classes for JMS and tells you what you need to know before using WebSphere MQ classes for JMS.

What is WebSphere MQ classes for JMS?

WebSphere MQ classes for Java Message Service (WebSphere MQ classes for JMS) is the JMS provider that is supplied with WebSphere MQ. As well as implementing the interfaces defined in the javax.jms package, WebSphere MQ classes for JMS provides two sets of extensions to the JMS API.

The JMS specification defines a set of interfaces that applications can use to perform messaging operations. The latest version of the specification is Version 1.1. The javax.jms package defines the JMS interfaces, and a JMS provider implements these interfaces for a specific messaging product. WebSphere MQ classes for JMS is a JMS provider that implements the JMS interfaces for WebSphere MQ.

The JMS specification expects ConnectionFactory and Destination objects to be administered objects. An administrator creates and maintains administered objects in a central repository, and a JMS application retrieves these objects using the Java Naming and Directory Interface (JNDI). WebSphere MQ classes for JMS supports the use of administered objects, and an administrator can use either the WebSphere MQ JMS administration tool or WebSphere MQ Explorer to create and maintain administered objects.

WebSphere MQ classes for JMS also provides two sets of extensions to the JMS API. The main focus of these extensions concerns creating and configuring connection factories and destinations dynamically at run time, but the extensions also provide function that is not directly related to messaging, such as function for problem determination.

The WebSphere MQ JMS extensions
: Previous releases of WebSphere MQ classes for JMS contain extensions that are implemented in objects such as MQConnectionFactory, MQQueue, and MQTopic objects. These objects have properties and methods that are specific to WebSphere MQ. The objects can be administered objects, or an application can create the objects dynamically at run time. This release of WebSphere MQ classes for JMS maintains these extensions, which are now known as the WebSphere MQ JMS extensions. You can continue to use, without change, any applications that use these extensions.

The IBM JMS extensions
: This release of WebSphere MQ classes for JMS provides a more generic set of extensions to the JMS API, which are not specific to WebSphere MQ as the messaging system. These extensions are known as the IBM JMS extensions and have the following broad objectives:

- To provide a greater level of consistency across IBM JMS providers

- To make it easier to write a bridge application between two IBM messaging systems
- To make it easier to port an application from one IBM JMS provider to another

The IBM JMS extensions are also implemented in Lotus Expeditor micro broker.

The extensions provide function that is similar to that provided in Message Service Client for C/C++ and Message Service Client for .NET.

Why should I use WebSphere MQ classes for JMS?

A Java application can use either WebSphere MQ classes for Java or WebSphere MQ classes for JMS to access WebSphere MQ resources. Using WebSphere MQ classes for JMS has a number of advantages.

Consider the following advantages:

- You can reuse JMS skills.

  WebSphere MQ classes for JMS is a JMS provider that implements the JMS interfaces for WebSphere MQ as the messaging system. If your organization is new to WebSphere MQ, but already has JMS application development skills, you might find it easier to use the familiar JMS API to access WebSphere MQ resources rather than one of the other APIs provided with WebSphere MQ.
- JMS is an integral part of Java Platform, Enterprise Edition (Java EE).

  JMS is the natural API to use for messaging on the Java EE platform. Every application server that is Java EE compliant must include a JMS provider. You can use JMS in application clients, servlets, JavaServer pages (JSPs), enterprise Java beans (EJBs), and message driven beans (MDBs). Note in particular that Java EE applications use MDBs to process messages asynchronously, and all messages are delivered to MDBs as JMS messages.
- An administrator can create and maintain JMS administered objects in a central repository, and WebSphere MQ classes for JMS applications can retrieve these objects using the Java Naming and Directory Interface (JNDI).

  JMS connection factories and destinations encapsulate WebSphere MQ specific information such as queue manager names, channel names, connection options, queue names, and topic names. If connection factories and destinations are stored as administered objects, this information is not hard coded into an application. This arrangement therefore provides the application with a degree of independence from the underlying WebSphere MQ configuration.
- JMS is an industry standard API that can provide application portability.

  A JMS application can use JNDI to retrieve connection factories and destinations that are stored as administered objects, and use only the interfaces defined in the javax.jms package to perform messaging operations. The application is then entirely independent of any JMS provider, such as WebSphere MQ classes for JMS, and can be ported from one JMS provider to another without any change to the application.

  If JNDI is not available in a particular application environment, a WebSphere MQ classes for JMS application can use extensions to the JMS API to create and configure connection factories and destinations dynamically at run time. The application is then completely self contained, but is tied to WebSphere MQ classes for JMS as the JMS provider.
- Bridge applications might be easier to write using JMS.

A bridge application is an application that receives messages from one messaging system and sends them to another messaging system. Writing a bridge application can be complicated using product specific APIs and message formats. Instead, you can write a bridge application using two JMS providers, one for each messaging system. The application then uses only one API, the JMS API, and processes only JMS messages.

Prerequisites for WebSphere MQ classes for JMS

To develop and run WebSphere MQ classes for JMS applications, you need certain software components as prerequisites.

For the latest information about the prerequisites for WebSphere MQ classes for JMS, see the WebSphere MQ readme file.

To develop WebSphere MQ classes for JMS applications, you need a Java 2 Software Development Kit (SDK). Details of the JDKs supported with your operating system can be found on the WebSphere MQ System requirements page. See http://www-01.ibm.com/software/integration/wmq/requirements/index.html

To run WebSphere MQ classes for JMS applications, you need the following software components:
- A WebSphere MQ queue manager
- A Java Runtime Environment (JRE), for each system on which you run applications
- For i5/OS, QShell, which is option 30 of the operating system
- For z/OS, UNIX System Services (USS)

If you require SSL connections to use cryptographic modules that are FIPS 140-2 certified, you need the IBM Java JSSE FIPS provider (IBMJSSEFIPS). Every IBM Java 2 SDK and JRE at Version 1.4.2 or later contains IBMJSSEFIPS.

You can use Internet Protocol Version 6 (IPv6) addresses in your WebSphere MQ classes for JMS applications provided IPv6 addresses are supported by your Java virtual machine (JVM) and the TCP/IP implementation on your operating system. The WebSphere MQ JMS administration tool (see “Using the WebSphere MQ JMS administration tool” on page 167) also accepts IPv6 addresses.

The WebSphere MQ JMS administration tool and WebSphere MQ Explorer use the Java Naming and Directory Interface (JNDI) to access a directory service, which stores administered objects. WebSphere MQ classes for JMS applications can also use JNDI to retrieve administered objects from a directory service. A service provider is code that provides access to a directory service by mapping JNDI calls to calls to the directory service. The following service providers are supplied with WebSphere MQ classes for JMS:
- A Lightweight Directory Access Protocol (LDAP) service provider in the files ldap.jar and providerutil.jar. The LDAP service provider provides access to a directory service based on an LDAP server.
- A file system service provider in the files fscontext.jar and providerutil.jar. The file system service provider provides access to a directory service based on the local file system.

If you intend to use a directory service based on an LDAP server, you must install and configure an LDAP server, or have access to an existing LDAP server. In particular, you must configure the LDAP server to store Java objects. For information about how to install and configure your LDAP server, see the documentation that is supplied with the server.

Installation and configuration of WebSphere MQ classes for JMS

This topic describes the directories and files that are created when you install WebSphere MQ classes for JMS and tells you how to configure WebSphere MQ classes for JMS after installation.

What is installed for WebSphere MQ classes for JMS

A number of files and directories are created when you install WebSphere MQ classes for JMS. On Windows some configuration is performed during installation by automatically setting environment variables. On other platforms, and in certain Windows environments, you must set environment variables before you can run WebSphere MQ classes for JMS applications.

WebSphere MQ classes for JMS is installed as an optional component when you install WebSphere MQ. See the following documentation for information about installing WebSphere MQ:

WebSphere MQ for AIX Quick Beginnings
WebSphere MQ for HP-UX Quick Beginnings
WebSphere MQ for i5/OS Quick Beginnings
WebSphere MQ for Linux Quick Beginnings
WebSphere MQ for Solaris Quick Beginnings
WebSphere MQ for Windows Quick Beginnings
WebSphere MQ for z/OS Program Directory

"Installation directories for WebSphere classes for JMS" on page 9 lists the installation directories by platform into which the WebSphere MQ classes for JMS are installed. Do not move or copy the .jar files to other directories.

The installation provides some additional scripts, JAR files and libraries in addition to the JAR files and native libraries that implement the WebSphere MQ classes for JMS. The scripts are described in "Scripts provided with WebSphere MQ classes for JMS" on page 57 and are located in the java\bin directory. Support for Open Services Grid Initiative (OGSi) is installed in the java\lib\OGSi directory and described in "Support for OSGi" on page 58. Additional resources for SOAP, HTTP, JCA and symbols for debugging applications on Windows are installed in other java\lib subdirectories.

After installation you might need to perform some configuration tasks to compile and run applications. "Environment variables used by WebSphere MQ classes for JMS" on page 9 describes the classpath required to run simple WebSphere MQ classes for JMS applications. It also describes additional JAR files that need to be referenced in special circumstances and the environment variables that you must set to run the scripts provided with WebSphere MQ classes for JMS. Should you need your WebSphere MQ classes for JMS application to link to code written in languages other than Java (for example to use the bindings mode connection to the Queue Manager), then "The Java Native Interface (JNI) libraries required by WebSphere MQ classes for JMS applications" on page 11 explains where to find the location of the Java Native Interface (JNI) libraries to specify as a parameter of the java command. To control properties such as tracing and logging of an application you need to provide a configuration properties file. The WebSphere MQ classes for JMS configuration properties file is described in "The WebSphere MQ classes for

JMS configuration file" on page 12. On z/OS you need to configure the STEPLIB used at runtime: see "STEPLIB configuration on z/OS" on page 14.

The Javadoc tool has been used to generate the HTML pages containing the specifications of the WebSphere MQ classes for JMS API. The HTML pages are in the `doc/WMQJMSClasses` subdirectory of the WebSphere MQ classes for JMS installation directory. On UNIX systems and Windows, this subdirectory contains the individual HTML pages but, on i5/OS and z/OS, the HTML pages are in a file called wmqjms_javadoc.jar.

Installation directories for WebSphere classes for JMS

On each platform, the WebSphere MQ classes for JMS files are installed in one directory. The sample applications, which include the installation verification programs (IVPs), are installed in a different directory.

Table 1 shows where the WebSphere MQ classes for JMS files are installed on each platform.

Table 1. WebSphere MQ classes for JMS installation directories

Platform	**Directory**
AIX	/usr/mqm/java
HP-UX, Linux, and Solaris	/opt/mqm/java
i5/OS	/QIBM/ProdData/mqm/java
Windows	*install_dir*\java
z/OS	*install_dir*/mqm/V7R0M0/java
Note: *install_dir* is the directory where you installed WebSphere MQ. On Windows, this directory is normally C:\Program Files\IBM\WebSphere MQ. On z/OS, this directory is likely to be /usr/lpp.	

Some sample applications, such as the installation verification programs (IVPs), are supplied with WebSphere MQ classes for JMS. Table 2 shows where the sample applications are installed on each platform.

Table 2. Samples directories

Platform	**Directory**
AIX	/usr/mqm/samp/jms
HP-UX, Linux, and Solaris	/opt/mqm/samp/jms
i5/OS	/QIBM/ProdData/mqm/java/samples/jms
Windows	*install_dir*\tools\jms
z/OS	*install_dir*/mqm/V7R0M0/java/samples/jms
Note: *install_dir* is the directory where you installed WebSphere MQ. On Windows, this directory is normally C:\Program Files\IBM\WebSphere MQ. On z/OS, this directory is likely to be /usr/lpp.	

Environment variables used by WebSphere MQ classes for JMS

Before you can compile and run WebSphere MQ classes for JMS applications, the setting for your CLASSPATH environment variable must include the WebSphere MQ classes for JMS Java archive (JAR) file. Depending on your requirements, you might need to add other JAR files to your class path. To run the scripts provided with WebSphere MQ classes for JMS, other environment variables must be set.

To compile and run WebSphere MQ classes for JMS applications, use the CLASSPATH setting for your platform as shown in Table 3. The setting includes the samples directory, so that you can compile and run the WebSphere MQ classes for JMS sample applications. Alternatively, you can specify the class path on the **java** command instead of using the environment variable.

Table 3. CLASSPATH setting to compile and run WebSphere MQ classes for JMS applications, including the sample applications

Platform	CLASSPATH setting
AIX	CLASSPATH=/usr/mqm/java/lib/com.ibm.mqjms.jar: /usr/mqm/samp/jms:
HP-UX, Linux, and Solaris	CLASSPATH=/opt/mqm/java/lib/com.ibm.mqjms.jar: /opt/mqm/samp/jms:
i5/OS	CLASSPATH=/QIBM/ProdData/mqm/java/lib/com.ibm.mqjms.jar: /QIBM/ProdData/mqm/java/samples/jms:
Windows	CLASSPATH=*install_dir*\java\lib\com.ibm.mqjms.jar; *install_dir*\tools\jms;
z/OS	CLASSPATH=*install_dir*/mqm/V7R0M0/java/lib/com.ibm.mqjms.jar: *install_dir*/mqm/V6R0M0/java/samples/jms:
Note: *install_dir* is the directory where you installed WebSphere MQ. On Windows, this directory is normally C:\Program Files\IBM\WebSphere MQ. On z/OS, this directory is likely to be /usr/lpp.	

The manifest of the JAR file com.ibm.mqjms.jar contains references to most of the other JAR files required by WebSphere MQ classes for JMS applications, and so you do not need to add these JAR files to your class path. These JAR files include those required by applications that use the Java Naming and Directory Interface (JNDI) to retrieve administered objects from a directory service and by applications that use the Java Transaction API (JTA).

However, you must include additional JAR files in your class path in the following circumstances:

- If your application performs XA operations in client mode, you must add the extended transactional client JAR file, com.ibm.mqetclient.jar, to your class path.
- If you are using channel exit classes that implement the channel exit interfaces defined in the com.ibm.mq package, instead of those defined in the com.ibm.mq.exits package, you must add the Websphere MQ classes for Java JAR file, com.ibm.mq.jar, to your class path.
- If you compile your Java code using a Java 2 Software Development Kit (SDK) at Version 1.4.2, you must add the following JAR files to your class path:
  - jms.jar
  - com.ibm.mq.jmqi.jar

  Additionally, if your application uses JNDI to retrieve administered objects from a directory service, you must also add the following JAR files to your class path:
  - fscontext.jar
  - jndi.jar
  - ldap.jar
  - providerutil.jar

  And if your application uses the JTA, you must also add jta.jar to your class path.

Note that these additional JAR files are required only for compiling your applications, not for running them.

The scripts provided with WebSphere MQ classes for JMS use the following environment variables:

MQ_JAVA_DATA_PATH
This environment variable specifies the directory for log and trace output.

MQ_JAVA_INSTALL_PATH
This environment variable specifies the directory where WebSphere MQ classes for JMS is installed, as shown in Table 1 on page 9.

MQ_JAVA_LIB_PATH
This environment variable specifies the directory where the WebSphere MQ classes for JMS libraries are stored, as shown in Table 4.

On Windows, all the environment variables are set automatically during installation. On any other platform, you must set them yourself. On a UNIX system, you can use the script setjmsenv (if you are using a 32-bit JVM) or setjmsenv64 (if you are using a 64-bit JVM) to set the environment variables. On AIX, these scripts are in the /usr/mqm/java/bin directory and, on HP-UX, Linux, and Solaris, they are in the /opt/mqm/java/bin directory.

On i5/OS, you must set the environment variable QIBM_MULTI_THREADED to `Y`. You can then run multithreaded applications in the same way that you run single threaded applications.

The Java Native Interface (JNI) libraries required by WebSphere MQ classes for JMS applications

When you start a WebSphere MQ classes for JMS application that connects in bindings mode, or one that connects in client mode and uses channel exit programs written in languages other than Java, you must specify the location of the Java Native Interface (JNI) libraries as a parameter on the **java** command.

To specify the location of the Java Native Interface (JNI) libraries, start your application using a **java** command with the following format:

```
java -Djava.library.path=library_path application_name
```

where *library_path* is the path to the directory containing the WebSphere MQ classes for JMS libraries. The JNI libraries are in the same directory. Table 4 shows the location of the WebSphere MQ classes for JMS libraries for each platform.

Table 4. The location of the WebSphere MQ classes for JMS libraries for each platform

Platform	Directory containing the WebSphere MQ classes for JMS libraries
AIX	/usr/mqm/java/lib (32-bit libraries) /usr/mqm/java/lib64 (64-bit libraries)
HP-UX Linux (POWER, x86-64 and zSeries s390x platforms) Solaris (x86-64 and Sparc platforms)	/opt/mqm/java/lib (32-bit libraries) /opt/mqm/java/lib64 (64-bit libraries)
Linux (x86 platform) Linux (zSeries platform)	/opt/mqm/java/lib

Table 4. The location of the WebSphere MQ classes for JMS libraries for each platform (continued)

Platform	Directory containing the WebSphere MQ classes for JMS libraries
Windows	*install_dir*\java\lib (32-bit libraries) *install_dir*\java\lib64 (64-bit libraries)
z/OS	*install_dir*/mqm/V7R0M0/java/lib (31-bit and 64-bit libraries)
Note: *install_dir* is the directory where you installed WebSphere MQ. On Windows, this directory is normally C:\Program Files\IBM\WebSphere MQ. On z/OS, this directory is likely to be /usr/lpp.	

Notes:

1. On AIX, HP-UX, Linux (POWER platform), or Solaris, use either the 32-bit libraries or the 64-bit libraries. Use the 64-bit libraries only if you are running your application in a 64-bit Java virtual machine (JVM) on a 64-bit platform. Otherwise, use the 32-bit libraries.
2. On Windows, you can use the PATH environment variable to specify the location of the WebSphere MQ classes for JMS libraries instead of specifying their location on the **java** command. The directory containing the WebSphere MQ classes for JMS libraries is automatically added to the system path during the installation of WebSphere MQ classes for JMS.
3. To use WebSphere MQ classes for JMS in bindings mode on i5/OS, make sure that the library QMQMJAVA is in your library list.
4. On z/OS, you can use either a 31-bit or 64-bit Java virtual machine (JVM) when running applications in WebSphere Application Server. In other environments on z/OS, you can use only a 31-bit JVM. However, in any environment on z/OS, you do not have to specify which JNI libraries to use; WebSphere MQ classes for JMS can determine for itself which JNI libraries to load.

The WebSphere MQ classes for JMS configuration file

A WebSphere MQ classes for JMS configuration file specifies properties that are used to configure WebSphere MQ classes for JMS.

The format of a WebSphere MQ classes for JMS configuration file is that of a standard Java properties file. A sample configuration file called jms.config is supplied in the `bin` subdirectory of the WebSphere MQ classes for JMS installation directory. This file documents all the supported properties and their default values.

You can choose the name and location of a WebSphere MQ classes for JMS configuration file. When you start your application, use a **java** command with the following format:

```
java -Dcom.ibm.msg.client.config.location=config_file_url application_name
```

In the command, *config_file_url* is a uniform resource locator (URL) that specifies the name and location of the WebSphere MQ classes for JMS configuration file. URLs of the following types are supported: http, file, ftp, and jar.

Here is an example of a **java** command:

```
java -Dcom.ibm.msg.client.config.location=file:/D:/mydir/myjms.config MyAppClass
```

This command identifies the WebSphere MQ classes for JMS configuration file as the file D:\mydir\mjms.config on the local Windows system.

When an application starts, WebSphere MQ classes for JMS reads the contents of the configuration file and stores the specified properties in an internal property store. If the **java** command does not identify a configuration file, or if the configuration file cannot be found, WebSphere MQ classes for JMS uses the default values for all the properties. If required, you can override any property in the configuration file by specifying it as a system property on the **java** command.

A WebSphere MQ classes for JMS configuration file can be used with any of the supported transports between an application and a queue manager or broker.

Note that you cannot specify startup trace by setting a property in the WebSphere MQ classes for JMS configuration file. You can specify startup trace only by setting a system property on the **java** command, as shown in the following example:

```
java -Dcom.ibm.msg.client.commonservices.trace.startup=true
     -Dcom.ibm.msg.client.config.location=file:/D:/mydir/myjms.config
      MyAppClass
```

Overriding properties specified in a WebSphere MQ client configuration file

A WebSphere MQ client configuration file can also specify properties that are used to configure WebSphere MQ classes for JMS. However, properties specified in a WebSphere MQ client configuration file apply only when an application connects to a queue manager in client mode.

If required, you can override any attribute in a WebSphere MQ configuration file by specifying it as a property in a WebSphere MQ classes for JMS configuration file. To override an attribute in a WebSphere MQ client configuration file, use an entry with the following format in the WebSphere MQ classes for JMS configuration file:

```
com.ibm.mq.cfg.stanza.propName=propValue
```

The variables in the entry have the following meanings:

stanza
The name of the stanza in the WebSphere MQ client configuration file that contains the attribute

propName
The name of the attribute as specified in the WebSphere MQ client configuration file

propValue
The value of the property that overrides the value of the attribute specified in the WebSphere MQ client configuration file

Alternatively, you can override an attribute in a WebSphere MQ client configuration file by specifying the property as a system property on the **java** command. Use the preceding format to specify the property as a system property.

Only the following attributes in a WebSphere MQ client configuration file are relevant to WebSphere MQ classes for JMS. If you specify or override other attributes, it has no effect.

Stanza	Attribute
ClientExitPath	ExitsDefaultPath
ClientExitPath	ExitsDefaultPath64

Stanza	Attribute
ClientExitPath	JavaExitsClasspath
MessageBuffer	MaximumSize
MessageBuffer	PurgeTime
MessageBuffer	UpdatePercentage
TCP	ClntRcvBufSize
TCP	ClntSndBufSize
TCP	Connect_Timeout
TCP	KeepAlive

STEPLIB configuration on z/OS

On z/OS, the STEPLIB used at runtime must contain the WebSphere MQ SCSQAUTH and SCSQANLE libraries. From UNIX System Services, you can add these using a line in your `.profile` as shown below, replacing `thlqual` with the high level data set qualifier that you chose when installing WebSphere MQ:

```
export STEPLIB=thlqual.SCSQAUTH:thlqual.SCSQANLE:$STEPLIB
```

In other environments, you typically need to edit the startup JCL to include SCSQAUTH and SCSQANLE on the STEPLIB concatenation:

```
STEPLIB DD DSN=thlqual.SCSQAUTH,DISP=SHR
        DD DSN=thlqual.SCSQANLE,DISP=SHR
```

Running WebSphere MQ classes for JMS applications under the Java security manager

WebSphere MQ classes for JMS can run with the Java security manager enabled. To run applications successfully with the security manager enabled, you must configure your Java virtual machine (JVM) with a suitable policy configuration file.

The simplest way to do this is to change the policy configuration file supplied with your Java Runtime Environment (JRE). On most systems, this file is in the directory lib/security/java.policy relative to your JRE directory. You can edit the policy configuration file using your preferred editor or the `policytool` program supplied with your JRE.

Here is an example of two entries in a policy configuration file that allow WebSphere MQ classes for JMS to run successfully under the default security manager:

```
grant codeBase "file:/opt/mqm/java/lib/*" {
  permission java.io.FilePermission "/var/mqm/-", "read, execute";
  permission java.io.FilePermission "/var/mqm/trace/-", "read, write";
  permission java.io.FilePermission "/opt/mqm/-", "read, execute";
  permission java.io.FilePermission "/opt/mqtest/-", "read";
  permission java.util.PropertyPermission "*", "read, write";
  permission java.util.logging.LoggingPermission "control";
  permission java.net.SocketPermission "*", "connect, resolve";
  permission java.lang.RuntimePermission "loadLibrary.mqjbnd";
  permission java.lang.RuntimePermission "loadLibrary.libqmqjexitstub02";
  permission java.lang.RuntimePermission "loadLibrary.mqjxs_r";
  permission java.lang.RuntimePermission "loadLibrary.mqjx_r";
  permission java.lang.RuntimePermission "getenv.*";
  permission java.lang.RuntimePermission "createClassLoader";
  permission java.io.FilePermission "FFDC/-", "read, write";
  permission java.io.FilePermission "mqclient.ini", "read";
  permission java.io.FilePermission "*", "read, write";
};

grant codeBase "file:/opt/mqtest/base/lib/*"  {
  permission java.util.PropertyPermission "*", "read";
  permission java.io.FilePermission "/var/mqm/-", "read, execute";
  permission java.io.FilePermission "/opt/mqm/-", "read, execute";
  permission java.io.FilePermission "/opt/mqtest/-", "read, execute";
  permission java.util.PropertyPermission "APIJMS_PROVIDER_VERSION", "write";
  permission java.util.PropertyPermission "com.ibm.jsse2.JSSEFIPS", "write";
};
```

In the example, the first `grant` statement contains the permissions required by WebSphere MQ classes for JMS, and the second `grant` statement contains the permissions required by a WebSphere MQ classes for JMS application. In the first `grant` statement, the permission:

```
permission java.io.FilePermission "/opt/mqtest/-", "read";
```

is needed to allow WebSphere MQ classes for JMS to access the Java archive (JAR) files of an application. To use these grant statements in your policy configuration file, you might need to modify the path names depending on where you have installed WebSphere MQ classes for JMS and where you store your applications.

The sample applications supplied with WebSphere MQ classes for JMS, and scripts to run them, do not enable the security manager.

The WebSphere MQ resource adapter

The Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) provides a standard way of connecting applications running in a Java EE environment to an Enterprise Information System (EIS) such as WebSphere MQ or DB2®. The WebSphere MQ resource adapter implements the JCA 1.5 interfaces, and allows JMS applications and message driven beans (MDBs), running in an application server, to access the resources of a WebSphere MQ queue manager. The resource adapter supports both the point-to-point domain and the publish/subscribe domain.

The WebSphere MQ resource adapter supports two types of communication between an application and a queue manager:

Outbound communication
An application starts a connection to a queue manager, and then sends JMS messages to JMS destinations and receives JMS messages from JMS destinations in a synchronous manner.

Inbound communication
A JMS message arriving at a JMS destination is delivered to an MDB, which processes the message asynchronously.

The WebSphere MQ resource adapter is supported on all WebSphere MQ Version 7.0 platforms except z/OS. You can install it on any application server that is certified as compliant with the J2EE 1.4 specification. Using the resource adapter, an application can connect to a WebSphere MQ Version 7.0 queue manager in either client mode or bindings mode, or to a WebSphere MQ Version 6.0 or WebSphere MQ Version 5.3 queue manager in client mode only.

Other required documentation

Every application server provides its own set of administration interfaces. Some application servers provide graphical user interfaces to define JCA resources, but others require the administrator to write XML deployment plans. It is therefore beyond the scope of this documentation to provide information about how to configure the WebSphere MQ resource adapter for each application server. This documentation focuses only on what you need to configure, and you must refer to your application server's own documentation for information about how to configure a JCA resource adapter.

To understand this documentation, you must be familiar with JMS and WebSphere MQ classes for JMS, as described in the chapters from "Writing WebSphere MQ classes for JMS applications" on page 76 through to "WebSphere MQ classes for JMS Application Server Facilities" on page 159. Many of the properties used to configure the WebSphere MQ resource adapter are equivalent to properties of WebSphere MQ classes for JMS objects and have the same function.

Installation of the WebSphere MQ resource adapter

The WebSphere MQ resource adapter is supplied as a resource archive (RAR) file called wmq.jmsra.rar. This file is installed with WebSphere MQ classes for JMS in the directory shown in Table 5.

Table 5. The directory containing wmq.jmsra.rar for each platform

Platform	**Directory**
AIX	/usr/mqm/java/lib/jca
HP-UX, Linux, and Solaris	/opt/mqm/java/lib/jca
i5/OS	/QIBM/ProdData/mqm/java/lib/jca
Windows	*install_dir*\java\lib\jca
Note: *install_dir* is the directory where you installed the WebSphere MQ server or WebSphere MQ client. The default directory is C:\Program Files\IBM\WebSphere MQ, but you might have chosen a different directory.	

The RAR file contains WebSphere MQ classes for JMS and the WebSphere MQ implementation of the JCA interfaces.

You must install the WebSphere MQ resource adapter RAR file in your application server, but the way you do this depends on the application server. See the documentation for your application server for information about how to install a resource adapter RAR file.

For non-transacted client connections, no other files are required.

For bindings connections on UNIX systems, you must ensure that the directory containing the Java Native Interface (JNI) libraries is in the system path. For the location of this directory, which also contains the WebSphere MQ classes for JMS libraries, see Table 4 on page 11. On Windows, this directory is automatically added to the system path during the installation of WebSphere MQ classes for JMS.

Distributed transactions are supported by default in bindings mode, but in client mode they are supported only if the extended transactional client JAR file, com.ibm.mqetclient.jar, is in the class path.

Table 6 summarizes the support for non-transacted and transacted connections. For an explanation of client and bindings modes, see "Connection modes for WebSphere MQ classes for JMS" on page 44.

Table 6. Support for non-transacted and transacted connections

Type of connection	**Non-transacted connections**	**Transacted connections**
Client mode	Supported by default	Supported if com.ibm.mqetclient.jar is in the class path
Bindings mode	Supported if the JNI libraries are in the system path	Supported if the JNI libraries are in the system path

The WebSphere MQ resource adapter and the version of WebSphere MQ classes for JMS used by the resource adapter must be at the same release level.

WebSphere Application Server, Version 6 and the WebSphere MQ resource adapter:

The WebSphere MQ resource adapter should not be used within WebSphere Application Server, Version 6. To access the resources of a WebSphere MQ queue manager from within WebSphere Application Server, JMS applications should use the WebSphere MQ messaging provider, which contains a version of the WebSphere MQ classes for JMS.

Configuration of the WebSphere MQ resource adapter

To configure the WebSphere MQ resource adapter, you define various JCA resources and system properties.

Define JCA resources in the following categories:

- The properties of the ResourceAdapter object, which represent the global properties of the resource adapter, such as the level of diagnostic tracing. These properties are described in "Configuration of the ResourceAdapter object" on page 18.
- The properties of an ActivationSpec object, which determine how an MDB is activated for inbound communication. These properties are described in "Configuration for inbound communication" on page 21.
- The properties of a ConnectionFactory object, which the application server uses to create a JMS ConnectionFactory object for outbound communication. These properties are described in "Configuration for outbound communication" on page 31.
- The properties of an administered destination object, which the application server uses to create a JMS Queue object or JMS Topic object for outbound communication. These properties are also described in "Configuration for outbound communication" on page 31.

The WebSphere MQ resource adapter RAR file contains a file called META-INF/ra.xml, which contains a deployment descriptor for the resource adapter. This deployment descriptor is defined by the XML schema at http://java.sun.com/xml/ns/j2ee/connector_1_5.xsd and contains information about the resource adapter and the services that it provides. An application server might also require a deployment plan for the resource adapter. This deployment plan is specific to the application server. For example, WebSphere Application Server Community Edition requires a deployment plan called geronimo-ra.xml.

If you are using Secure Sockets Layer (SSL), specify the locations of the key store file and trust store file as JVM system properties, as in the following example:

```
java ... -Djavax.net.ssl.keyStore=key_store_location
         -Djavax.net.ssl.trustStore=trust_store_location
         -Djavax.net.ssl.keyStorePassword=key_store_password
```

These properties cannot be properties of an ActivationSpec or ConnectionFactory object, and you cannot specify more than one key store for an application server. The properties apply to the whole JVM, and might therefore affect the application server if other applications, running in the application server, are using SSL connections. The application server might also reset these properties to different values. For more information about using SSL with WebSphere MQ classes for JMS, see "Using Secure Sockets Layer (SSL) with WebSphere MQ classes for JMS" on page 141.

An installation verification test (IVT) program is supplied with the WebSphere MQ resource adapter, but you must configure the resource adapter before you can run the program. For information about what you need to configure in order to run the IVT program, see "The installation verification test program for the WebSphere MQ resource adapter" on page 54.

Configuration of the ResourceAdapter object:

The ResourceAdapter object encapsulates the global properties of the WebSphere MQ resource adapter. The object has two sets of properties:
- Properties associated with diagnostic tracing
- Properties associated with the connection pool managed by the resource adapter

The way you define these properties depends on the administration interfaces provided by your application server.

Table 7 lists the properties of the ResourceAdapter object that are associated with diagnostic tracing.

Table 7. Properties of the ResourceAdapter object that are associated with diagnostic tracing

Name of property	Type	Default value	Description
traceEnabled	String	false	A flag to enable or disable diagnostic tracing. If the value is false, tracing is turned off. If the value is true, a trace is sent to the location specified by the traceDestination property.
traceDestination	String	wmq_jca.trc	The location to where a diagnostic trace is sent. If the value is System.err, the trace is directed to the system error stream instead of a file. Similarly, if the value is System.out, the trace is directed to the system output stream.
traceLevel	String	3	The level of detail in a diagnostic trace. The value can be in the range 0, which produces no trace, to 10, which provides the most detail. See Table 8 on page 19 for a description of each level.

Table 7. Properties of the ResourceAdapter object that are associated with diagnostic tracing (continued)

Name of property	Type	Default value	Description
timestampsEnabled	String	true	A flag to enable or disable time stamps in a diagnostic trace. If the value is true, time stamps are added. If the value is false, time stamps are not added. An application server might add time stamps to a stream automatically. In this case, set the value of the property to false to avoid the duplication of time stamps.
logWriterEnabled	String	true	A flag to enable or disable the sending of a diagnostic trace to a LogWriter object provided by the application server. If the value is true, the trace is sent to a LogWriter object instead of the location specified by the traceDestination property. If the value is false, any LogWriter object provided by the application server is not used.

Table 8 describes the levels of detail for diagnostic tracing.

Table 8. The levels of detail for diagnostic tracing

Level number	Level of detail
0	No trace.
1	The trace contains error messages.
3	The trace contains error and warning messages.
6	The trace contains error, warning, and information messages.
8	The trace contains error, warning, and information messages, and entry and exit information for methods.
9	The trace contains error, warning, and information messages, entry and exit information for methods, and diagnostic data.
10	The trace contains all trace information.
Note: Any level that is not included in this table is equivalent to the next lowest level. For example, specifying a trace level of 4 is equivalent to specifying a trace level of 3. However, the levels that are not included might be used in future releases of the WebSphere MQ resource adapter, so it is better to avoid using these levels.	

If diagnostic tracing is turned off, error and warning messages are written to the system error stream. If diagnostic tracing is turned on, error messages are written to the system error stream and to the trace destination, but warning messages are written only to the trace destination. However, the trace contains warning messages only if the trace level is 3 or higher.

The resource adapter manages an internal connection pool of JMS connections that are used to deliver messages to MDBs. Table 9 on page 20 lists the properties of the ResourceAdapter object that are associated with the connection pool.

Table 9. Properties of the ResourceAdapter object that are associated with the connection pool

Name of property	Type	Default value	Description
maxConnections	String	10	The maximum number of connections to a WebSphere MQ queue manager.
connectionConcurrency	String	5	The maximum number of MDBs that can be supplied by each connection.
reconnectionRetryCount	String	5	The maximum number of attempts made by the resource adapter to reconnect to a WebSphere MQ queue manager if a connection fails.
reconnectionRetryInterval	String	300 000	The time, in milliseconds, that the resource adapter waits before making another attempt to reconnect to a WebSphere MQ queue manager.

When an MDB is deployed in the application server, the resource adapter attempts to use an existing JMS connection from the connection pool. Each connection can supply more than one MDB up to the maximum specified by the connectionConcurrency property. If there are no connections in the pool, or if all the connections are fully utilized, a new connection is created provided the maximum number of connections specified by the maxConnections property is not exceeded. The maximum number of MDBs that can be deployed is therefore equal to the product of the maxConnections and connectionConcurrency properties, which is 50 by default. If the number of deployed MDBs reaches the maximum, any attempt to deploy another MDB fails. If an MDB is stopped, its connection can be used by another MDB.

If MDBs are likely to receive a high volume of messages, you might need to reduce the value of the connectionConcurrency property. If you need to limit the number of connections, because of restrictions imposed by a firewall for example, you might need to increase the value of the connectionConcurrency property. In general, if many MDBs are to be deployed, increase the value of the maxConnections property.

The reconnectionRetryCount and reconnectionRetryInterval properties govern the behavior of the resource adapter when connections to a WebSphere MQ queue manager fail, because of a network failure for example. When a connection fails, the resource adapter suspends the delivery of messages to all MDBs supplied by that connection for an interval specified by the reconnectionRetryInterval property. The resource adapter then attempts to reconnect to the queue manager. If the attempt fails, the resource adapter makes further attempts to reconnect at intervals specified by the reconnectionRetryInterval property until the limit imposed by the reconnectionRetryCount property is reached. If all attempts fail, delivery is stopped permanently until the MDBs are restarted manually.

In general, the ResourceAdapter object requires no administration. However, to enable diagnostic tracing on a UNIX system for example, you can set the following properties:

```
traceEnabled:       true
traceDestination:   /tmp/wmq_jca.trace
traceLevel:         10
```

These properties have no effect if the resource adapter has not been started, which is the case, for example, when applications using WebSphere MQ resources are running only in the client container. In this situation, you can set the properties for diagnostic tracing as Java Virtual Machine (JVM) system properties. You can do this by using the -D flag on the **java** command, as in the following example:

```
java ... -DtraceEnabled=true -DtraceDestination=System.err -DtraceLevel=6
```

You do not need to define all the properties of the ResourceAdapter object. Any properties left unspecified take their default values. In a managed environment, it is better not to mix the two ways of specifying properties. If you do mix them, the JVM system properties take precedence over the properties of the ResourceAdapter object.

Configuration for inbound communication:

To configure inbound communication, define the properties of one or more ActivationSpec objects.

The properties of an ActivationSpec object determine how a message drive bean (MDB) receives JMS messages from a WebSphere MQ queue. The transactional behavior of the MDB is defined in its deployment descriptor.

An ActivationSpec object has two sets of properties:
- Properties that are used to create a JMS connection to a WebSphere MQ queue manager
- Properties that are used to create a JMS connection consumer that delivers messages asynchronously as they arrive on a specified queue

The way in which you define the properties of an ActivationSpec object depends on the administration interfaces provided by your application server.

Table 10 lists the properties of an ActivationSpec object that are used to create a JMS connection to a WebSphere MQ queue manager.

Table 10. Properties of an ActivationSpec object that are used to create a JMS connection

Name of property	Type	Valid values (default value in bold)	Description
brokerCCDurSubQueue[1]	String	• **SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a connection consumer receives durable subscription messages
brokerCCSubQueue[1]	String	• **SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a connection consumer receives nondurable subscription messages
brokerControlQueue[1]	String	• **SYSTEM.BROKER.CONTROL.QUEUE** • A queue name	The name of the broker control queue
brokerQueueManager[1]	String	• **"" (empty string)** • A queue manager name	The name of the queue manager on which the broker is running
brokerSubQueue[1]	String	• **SYSTEM.JMS.ND.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a nondurable message consumer receives messages

Table 10. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
brokerVersion[1]	String	• **unspecified** - After the broker has been migrated from V6 to V7, set this property so that RFH2 headers are no longer used. After migration this property is no longer relevant. • **V1** - To use a WebSphere MQ Publish/Subscribe broker, or to use a broker of WebSphere MQ Integrator, WebSphere Event Broker, WebSphere Business Integration Event Broker, or WebSphere Business Integration Message Broker in compatibility mode. This is the default value if TRANSPORT is set to BIND or CLIENT. • **V2** - To use a broker of WebSphere MQ Integrator, WebSphere Event Broker, WebSphere Business Integration Event Broker, or WebSphere Business Integration Message Broker in native mode. This is the default value if TRANSPORT is set to DIRECT or DIRECTHTTP.	The version of the broker being used
ccdtURL	String	• **null** • A uniform resource locator (URL)	A URL that identifies the name and location of the file containing the client channel definition table and specifies how the file can be accessed
CCSID	String	• **819** • A coded character set identifier supported by the Java virtual machine (JVM)	The coded character set identifier for a connection
channel	String	• **SYSTEM.DEF.SVRCONN** • The name of an MQI channel	The name of the MQI channel to use
cleanupInterval[1]	int	• **3 600 000** • A positive integer	The interval, in milliseconds, between background runs of the publish/subscribe cleanup utility
cleanupLevel[1]	String	• **SAFE** • NONE • STRONG • FORCE • NONDUR	The cleanup level for a broker based subscription store
clientID	String	• **null** • A client identifier	The client identifier for a connection
cloneSupport	String	• **DISABLED** - Only one instance of a durable topic subscriber can run at a time. • ENABLED - Two or more instances of the same durable topic subscriber can run simultaneously, but each instance must run in a separate Java virtual machine (JVM).	Whether two or more instances of the same durable topic subscriber can run simultaneously
failIfQuiesce	boolean	• **true** • false	Whether calls to certain methods fail if the queue manager is in a quiescing state

Table 10. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	**Type**	**Valid values (default value in bold)**	**Description**
headerCompression	String	• **NONE** • SYSTEM - RLE message header compression is performed	A list of the techniques that can be used for compressing header data on a connection
hostName	String	• **localhost** • A host name • An IP address	The host name or IP address of the system on which the queue manager resides
localAddress	String	• **null** • A string in the format: `[host_name][(low_port[,high_port])]` where *host_name* is a host name or IP address, *low_port* and *high_port* are TCP port numbers, and brackets denote an optional component	For a connection to a queue manager, this property specifies either or both of the following: • The local network interface to be used • The local port, or range of local ports, to be used
messageBatchSize[1]	int	• **10** • Any positive integer	The maximum number of messages to be taken from a queue in one packet when using asynchronous message delivery
messageCompression	String	• **NONE** • A list of one or more of the following values separated by blank characters: RLE ZLIBFAST ZLIBHIGH	A list of the techniques that can be used for compressing message data on a connection
messageRetention[1]	boolean	• **true** - Unwanted messages remain on the input queue • false - Unwanted messages are dealt with according to their disposition options	Whether or not the connection consumer keeps unwanted messages on the input queue
messageSelection[1]	String	• **CLIENT** • BROKER	Determines whether message selection is done by WebSphere MQ classes for JMS or by the broker. Message selection by the broker is not supported when brokerVersion has the value 1.
password	String	• **null** • A password	The default password to use when creating a connection to the queue manager

Table 10. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
pollingInterval[1]	int	• **5000** • Any positive integer	If each message listener within a session has no suitable message on its queue, this is the maximum interval, in milliseconds, that elapses before each message listener tries again to get a message from its queue. If it frequently happens that no suitable message is available for any of the message listeners in a session, consider increasing the value of this property. This property is relevant only if TRANSPORT has the value BIND or CLIENT.
port	int	• **1414** • A TCP port number	The port on which the queue manager listens
providerVersion	string	• **unspecified** • A string in one of the following formats – V.R.M.F – V.R.M – V.R – V where V, R, M and F are integer values greater than or equal to zero.	The version, release, modification level and fix pack of the queue manager to which the MDB intends to connect.
queueManager	String	• ″″ **(empty string)** • A queue manager name	The name of the queue manager to connect to
receiveExit	String	• **null** • A string comprising one or more items separated by commas, where each item is the fully qualified name of a class that implements the WebSphere MQ classes for Java interface, MQReceiveExit	Identifies a channel receive exit program, or a sequence of receive exit programs to be run in succession
receiveExitInit	String	• **null** • A string comprising one or more items of user data separated by commas	The user data that is passed to channel receive exit programs when they are called

Table 10. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
rescanInterval[1]	int	• **5000** • Any positive integer	When a message consumer in the point-to-point domain uses a message selector to select which messages it wants to receive, WebSphere MQ classes for JMS searches the WebSphere MQ queue for suitable messages in the sequence determined by the `MsgDeliverySequence` attribute of the queue. When WebSphere MQ classes for JMS finds a suitable message and delivers it to the consumer, WebSphere MQ classes for JMS resumes the search for the next suitable message from its current position in the queue. WebSphere MQ classes for JMS continues to search the queue in this way until it reaches the end of the queue, or until the interval of time in milliseconds, as determined by the value of this property, has expired. In each case, WebSphere MQ classes for JMS returns to the beginning of the queue to continue its search, and a new time interval commences.
securityExit	String	• **null** • The fully qualified name of a class that implements the WebSphere MQ classes for Java interface, MQSecurityExit	Identifies a channel security exit program
securityExitInit	String	• **null** • A string of user data	The user data that is passed to a channel security exit program when it is called
sendExit	String	• **null** • A string comprising one or more items separated by commas, where each item is the fully qualified name of a class that implements the WebSphere MQ classes for Java interface, MQSendExit	Identifies a channel send exit program, or a sequence of send exit programs to be run in succession
sendExitInit	String	• **null** • A string comprising one or more items of user data separated by commas	The user data that is passed to channel send exit programs when they are called

Table 10. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
shareConvAllowed	boolean	• **false** - A client connection cannot share its socket. • true - A client connection can share its socket.	Whether a client connection can share its socket with other top-level JMS connections from the same process to the same queue manager, if the channel definitions match
sparseSubscriptions[1]	boolean	• **false** - Subscriptions receive frequent matching messages. • true - Subscriptions receive infrequent matching messages. This value requires that the subscription queue can be opened for browse.	Controls the message retrieval policy of a TopicSubscriber object
sslCertStores	String	• **null** • A string of one or more LDAP URLs separated by blanks. Each LDAP URL has the format: `ldap://`*`host_name`*`[:`*`port`*`]` where *host_name* is a host name or IP address, *port* is a TCP port number, and brackets denote an optional component.	The Lightweight Directory Access Protocol (LDAP) servers that hold certificate revocation lists (CRLs) for use on an SSL connection
sslCipherSuite	String	• **null** • The name of a CipherSuite	The CipherSuite to use for an SSL connection
sslFipsRequired[2]	boolean	• **false** • true	Whether an SSL connection must use a CipherSuite that is supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS)
sslPeerName	String	• **null** • A template for distinguished names	For an SSL connection, a template that is used to check the distinguished name in the digital certificate provided by the queue manager
sslResetCount	int	• **0** • An integer in the range 0 to 999 999 999	The total number bytes sent and received by an SSL connection before the secret keys used by SSL are renegotiated
sslSocketFactory	String	A string representing the fully-qualified class name of a class providing an implementation of the javax.net.ssl.SSLSocketFactory interface, optionally including an argument to be passed to the constructor method, enclosed in parentheses.	Any connections established in the scope of the administered object use sockets obtained from this implementation of the SSLSocketFactory interface.

Table 10. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
statusRefreshInterval[1]	int	• **60000** • Any positive integer	The interval, in milliseconds, between refreshes of the long running transaction that detects when a subscriber loses its connection to the queue manager. This property is relevant only if subscriptionStore has the value QUEUE.
subscriptionStore[1]	String	• **BROKER** • MIGRATE • QUEUE	Determines where WebSphere MQ classes for JMS stores persistent data about active subscriptions
transportType	String	• **CLIENT** • BINDINGS • BINDINGS_THEN_CLIENT	Whether a connection to a queue manager uses client mode or bindings mode. If the value BINDINGS_THEN_CLIENT is specified, the resource adapter first tries to make a connection in bindings mode, and if this fails then tries to make a client mode connection.
username	String	• **null** • A user name	The default user name to use when creating a connection to a queue manager
wildcardFormat	int	• CHAR- Recognizes character wildcards only, as used in broker version 1 • **TOPIC** - Recognizes topic level wildcards only, as used in broker version 2	Which version of wildcard syntax is to be used

Notes:

1. This property can be used with Version 7.0 of WebSphere MQ classes for JMS but has no effect for an application connected to a Version 7.0 queue manager unless the providerVersion property is set to a version number less than 7.
2. For important information about using the sslFipsRequired property, see "Limitations of the WebSphere MQ resource adapter" on page 42.

Table 11 lists the properties of an ActivationSpec object that are used to create a JMS connection consumer.

Table 11. Properties of an ActivationSpec object that are used to create a JMS connection consumer

Name of property	Type	Valid values (default value in bold)	Description
destination	String	A destination name	The destination from which to receive messages. The useJNDI property determines how the value of this property is interpreted.

Table 11. Properties of an ActivationSpec object that are used to create a JMS connection consumer (continued)

Name of property	Type	Valid values (default value in bold)	Description
destinationType	String	• javax.jms.Queue • javax.jms.Topic	The type of destination, a queue or a topic
maxMessages	int	• **1** • A positive integer	The maximum number of messages that can be assigned to a server session at one time
maxPoolDepth	int	• **10** • A positive integer	The maximum number of server sessions in the server session pool used by the connection consumer
messageSelector	String	• **null** • An SQL92 message selector expression	A message selector expression specifying which messages are to be delivered
poolTimeout	int	• **300 000** • A positive integer	The period of time, in milliseconds, that an unused server session is held open in the server session pool before being closed due to inactivity
readAheadAllowed	int	• **DESTINATION** - Determine whether read ahead is allowed by referring to the queue or topic definition. • DISABLED - Read ahead is not allowed. • ENABLED - Read ahead is allowed. • QUEUE - Determine whether read ahead is allowed by referring to the queue definition. • TOPIC - Determine whether read ahead is allowed by referring to the topic definition.	Whether the MDB is allowed to use read ahead to get nonpersistent messages from the destination into an internal buffer before receiving them
readAheadClosePolicy	int	• **ALL** - All messages in the internal read ahead buffer are delivered to the MDB before it stops. • CURRENT - Only the current MDB invocation completes, potentially leaving messages in the internal read ahead buffer, which are then discarded.	What happens to messages in the internal read ahead buffer when the MDB is stopped by the administrator.
startTimeout	int	• **10 000** • A positive integer	The period of time, in milliseconds, within which delivery of a message to an MDB must start after the work to deliver the message has been scheduled. If this period of time elapses, the message is rolled back onto the queue.
subscriptionDurability	String	• **NonDurable** - A nondurable subscription is used to deliver messages to an MDB subscribing to the topic. • Durable - A durable subscription is used to deliver messages to an MDB subscribing to the topic.	Whether a durable or nondurable subscription is used to deliver messages to an MDB subscribing to the topic
subscriptionName	String	• **"" (empty string)** • A subscription name	The name of the durable subscription

Table 11. Properties of an ActivationSpec object that are used to create a JMS connection consumer (continued)

Name of property	Type	Valid values (default value in bold)	Description
useJNDI	boolean	• **false** - The property called destination is interpreted as the name of a WebSphere MQ queue or a topic. • true - The property called destination is interpreted as the name of a javax.jms.Queue object or javax.jms.Topic object in the application server's JNDI namespace.	Determines how the value of the property called destination is interpreted

The ActivationSpec properties called destination and destinationType must be defined explicitly. All the other properties are optional.

An ActivationSpec object can have conflicting properties. For example, you can specify SSL properties for a connection in bindings mode. In this case, the behavior is determined by the transport type and the messaging domain, which is either point-to-point or publish/subscribe as determined by the destinationType property. Any properties that are not applicable to the specified transport type or messaging domain are ignored.

If you define a property that requires other properties to be defined, but you do not define these other properties, the ActivationSpec object throws an InvalidPropertyException exception when its validate() method is called during the deployment of an MDB. The exception is reported to the administrator of the application server in a manner that depends on the application server. For example, if you set the subscriptionDurability property to Durable, indicating that you want use durable subscriptions, you must also define the subscriptionName property.

If the properties called ccdtURL and channel are both defined, an InvalidPropertyException exception is thrown. However, if you define the ccdtURL property only, leaving the property called channel with its default value of SYSTEM.DEF.SVRCONN, no exception is thrown, and the client channel definition table identified by the ccdtURL property is used to start a JMS connection.

Most of the properties of an ActivationSpec object are equivalent to properties of WebSphere MQ classes for JMS objects or parameters of WebSphere MQ classes for JMS methods. However, three tuning properties, and one usability property, have no equivalents in WebSphere MQ classes for JMS:

startTimeout
: The time, in milliseconds, that the work manager of the application server waits for resources to become available after the resource adapter schedules a Work object to deliver a message to an MDB. If this period of time elapses before delivery of the message starts, the Work object times out, the message is rolled back onto the queue, and the resource adapter can then make another attempt to deliver the message. A warning is written to diagnostic trace, if enabled, but this does not otherwise affect the process of delivering messages. You might expect this condition to occur only at times when the application server is experiencing a very high load. If the condition occurs regularly, consider increasing the value of this property to give the work manager longer to schedule message delivery.

maxPoolDepth
: The maximum number of server sessions in the server session pool used by a connection consumer. The connection consumer uses a server session to deliver a message to an MDB. A larger pool depth allows more messages to be delivered concurrently in high volume situations, but uses more resources of the application server. If many MDBs are to be deployed, consider making the pool depth smaller in order to maintain the load on the application server at a manageable level. Note that each connection consumer uses its own server session pool, so that this property does not define the total number of server sessions available to all connection consumers.

poolTimeout
: The period of time, in milliseconds, that an unused server session is held open in the server session pool before being closed due to inactivity. A transient increase in the message workload causes additional server sessions to be created in order to distribute the load but, after the message workload returns to normal, the additional server sessions remain in the pool and are not used.

  Every time a server session is used, it is marked with a timestamp. Periodically a scavenger thread checks that each server session has been used within the period specified by this property. If a server session has not been used, it is closed and removed from the server session pool. A server session might not be closed immediately after the specified period has elapsed, this property represents the minimum period of inactivity before removal.

useJNDI
: For a description of this property, see Table 11 on page 27.

To deploy an MDB, first define the properties of an ActivationSpec object, specifying the properties that the MDB requires. The following example is a typical set of properties that you might define explicitly:

```
channel:          SYSTEM.DEF.SVRCONN
destination:      SYSTEM.DEFAULT.LOCAL.QUEUE
destinationType:  javax.jms.Queue
hostName:         192.168.0.42
messageSelector:  color='red'
port:             1414
queueManager:     ExampleQM
transportType:    CLIENT
```

The application server uses the properties to create an ActivationSpec object, which is then associated with an MDB. The properties of the ActivationSpec object determine how messages are delivered to the MDB. Deployment of the MDB fails if the MDB requires distributed transactions but the resource adapter does not support distributed transactions. For information about how to install the resource adapter so that distributed transactions are supported, see “Installation of the WebSphere MQ resource adapter” on page 16.

If more than one MDB is receiving messages from the same destination, then a message sent in the point-to-point domain is received by only one MDB, even if other MDBs are eligible to receive the message. In particular, if two MDBs are using different message selectors, and an incoming message matches both message selectors, only one of the MDBs receives the message. The MDB chosen to receive a message is undefined, and you cannot rely on a specific MDB receiving the message. Messages sent in the publish/subscribe domain are received by all eligible MDBs.

Poison messages:

In some circumstances, a message delivered to an MDB might be rolled back onto a WebSphere MQ queue. This can happen, for example, if a message is delivered within a unit of work that is subsequently rolled back. A message that is rolled back is generally delivered again, but a badly formatted message might repeatedly cause an MDB to fail and therefore cannot be delivered. Such a message is called a *poison message*. You can configure WebSphere MQ so that WebSphere MQ classes for JMS automatically transfers a poison message to another queue for further investigation or discards the message. For information about how to configure WebSphere MQ in this way, see "Handling poison messages in ASF" on page 162.

Configuration for outbound communication:

To configure outbound communication, define the properties of a ConnectionFactory object and an administered destination object.

When using outbound communication, an application running in an application server starts a connection to a queue manager, and then sends messages to its queues and receives messages from its queues in a synchronous manner. For example, the following servlet method, doGet(), uses outbound communication:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {

...

// Look up ConnectionFactory and Queue objects from the JNDI namespace

     InitialContext ic = new InitialContext();
     ConnectionFactory cf = (javax.jms.ConnectionFactory) ic.lookup("myCF");
     Queue q = (javax.jms.Queue) ic.lookup("myQueue");

// Create and start a connection

     Connection c = cf.createConnection();
     c.start();

// Create a session and message producer

     Session s = c.createSession(false, Session.AUTO_ACKNOWLEDGE);
     MessageProducer pr = s.createProducer(q);

// Create and send a message

     Message m = s.createTextMessage("Hello, World!");
     pr.send(m);

// Create a message consumer and receive the message just sent

     MessageConsumer co = s.createConsumer(q);
     Message mr = co.receive(5000);

// Close the connection

      c.close();
}
```

When the servlet receives an HTTP GET request, it retrieves a ConnectionFactory object and a Queue object from the JNDI namespace, and uses the objects to send a message to a WebSphere MQ queue. The servlet then receives the message that it has just sent.

To configure outbound communication, define JCA resources in the following categories:

- The properties of a ConnectionFactory object, which the application server uses to create a JMS ConnectionFactory object.
- The properties of an administered destination object, which the application server uses to create a JMS Queue object or JMS Topic object.

The way you define these properties depends on the administration interfaces provided by your application server. ConnectionFactory, Queue, and Topic objects created by the application server are bound into a JNDI namespace from where they can be retrieved by an application.

Typically, you define one ConnectionFactory object for each queue manager that applications might need to connect to, one Queue object for each queue that applications might need to access in the point-to-point domain, and one Topic object for each topic that applications might want to publish or subscribe to. A ConnectionFactory object can be domain independent. Alternatively, it can be domain specific, a QueueConnectionFactory object for the point-to-point domain or a TopicConnectionFactory object for the publish/subscribe domain.

Table 12 lists the properties of a ConnectionFactory object.

Table 12. Properties of a ConnectionFactory object

Name of property	Type	Valid values (default value in bold)	Description
brokerCCSubQueue[1]	String	• **SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a connection consumer receives nondurable subscription messages
brokerControlQueue[1]	String	• **SYSTEM.BROKER.CONTROL.QUEUE** • A queue name	The name of the broker control queue
brokerPubQueue[1]	String	• **SYSTEM.BROKER.DEFAULT.STREAM** • A queue name	The name of the queue where published messages are sent (the stream queue)
brokerQueueManager[1]	String	• **"" (empty string)** • A queue manager name	The name of the queue manager on which the broker is running
brokerSubQueue[1]	String	• **SYSTEM.JMS.ND.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a nondurable message consumer receives messages

Table 12. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
brokerVersion[1]	String	• **unspecified** - After the broker has been migrated from V6 to V7, set this property so that RFH2 headers are no longer used. After migration this property is no longer relevant. • **V1** - To use a WebSphere MQ Publish/Subscribe broker, or to use a broker of WebSphere MQ Integrator, WebSphere Event Broker, WebSphere Business Integration Event Broker, or WebSphere Business Integration Message Broker in compatibility mode. This is the default value if TRANSPORT is set to BIND or CLIENT. • **V2** - To use a broker of WebSphere MQ Integrator, WebSphere Event Broker, WebSphere Business Integration Event Broker, or WebSphere Business Integration Message Broker in native mode. This is the default value if TRANSPORT is set to DIRECT or DIRECTHTTP.	The version of the broker being used
ccdtURL	String	• **null** • A uniform resource locator (URL)	A URL that identifies the name and location of the file containing the client channel definition table and specifies how the file can be accessed
CCSID	String	• **819** • A coded character set identifier supported by the Java virtual machine (JVM)	The coded character set identifier for a connection
channel	String	• **SYSTEM.DEF.SVRCONN** • The name of an MQI channel	The name of the MQI channel to use
cleanupInterval[1]	int	• **3 600 000** • A positive integer	The interval, in milliseconds, between background runs of the publish/subscribe cleanup utility
cleanupLevel[1]	String	• **SAFE** • NONE • STRONG • FORCE • NONDUR	The cleanup level for a broker based subscription store
clientID	String	• **null** • A client identifier	The client identifier for a connection
cloneSupport	String	• **DISABLED** - Only one instance of a durable topic subscriber can run at a time. • ENABLED - Two or more instances of the same durable topic subscriber can run simultaneously, but each instance must run in a separate Java virtual machine (JVM).	Whether two or more instances of the same durable topic subscriber can run simultaneously
failIfQuiesce	boolean	• **true** • false	Whether calls to certain methods fail if the queue manager is in a quiescing state
headerCompression	String	• **NONE** • SYSTEM - RLE message header compression is performed.	A list of the techniques that can be used for compressing header data on a connection

Table 12. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
hostName	String	• **localhost** • A host name • An IP address	The host name or IP address of the system on which the queue manager resides
localAddress	String	• **null** • A string in the format: `[`*`host_name`*`][(`*`low_port`*`[,`*`high_port`*`])]` where *host_name* is a host name or IP address, *low_port* and *high_port* are TCP port numbers, and brackets denote an optional component	For a connection to a queue manager, this property specifies either or both of the following: • The local network interface to be used • The local port, or range of local ports, to be used
messageCompression	String	• **NONE** • A list of one or more of the following values separated by blank characters: RLE ZLIBFAST ZLIBHIGH	A list of the techniques that can be used for compressing message data on a connection
messageSelection[1]	String	• **CLIENT** • BROKER	Determines whether message selection is done by WebSphere MQ classes for JMS or by the broker. Message selection by the broker is not supported when brokerVersion has the value 1.
password	String	• **null** • A password	The default password to use when creating a connection to the queue manager
pollingInterval[1]	int	• **5000** • Any positive integer	If each message listener within a session has no suitable message on its queue, this is the maximum interval, in milliseconds, that elapses before each message listener tries again to get a message from its queue. If it frequently happens that no suitable message is available for any of the message listeners in a session, consider increasing the value of this property. This property is relevant only if TRANSPORT has the value BIND or CLIENT.
port	int	• **1414** • A TCP port number	The port on which the queue manager listens

Table 12. Properties of a ConnectionFactory object (continued)

Name of property	**Type**	**Valid values (default value in bold)**	**Description**
providerVersion	string	• **unspecified** • A string in one of the following formats – V.R.M.F – V.R.M – V.R – V where V, R, M and F are integer values greater than or equal to zero.	The version, release, modification level and fix pack of the queue manager to which the application intends to connect.
pubAckInterval[1]	int	• **25** • A positive integer	The number of messages published by a publisher before WebSphere MQ classes for JMS requests an acknowledgement from the broker.
queueManager	String	• "" **(empty string)** • A queue manager name	The name of the queue manager to connect to
receiveExit	String	• **null** • A string comprising one or more items separated by commas, where each item is the fully qualified name of a class that implements the WebSphere MQ classes for Java interface, MQReceiveExit	Identifies a channel receive exit program, or a sequence of receive exit programs to be run in succession
receiveExitInit	String	• **null** • A string comprising one or more items of user data separated by commas	The user data that is passed to channel receive exit programs when they are called

Table 12. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
rescanInterval[1]	int	• **5000** • Any positive integer	When a message consumer in the point-to-point domain uses a message selector to select which messages it wants to receive, WebSphere MQ classes for JMS searches the WebSphere MQ queue for suitable messages in the sequence determined by the *MsgDeliverySequence* attribute of the queue. When WebSphere MQ classes for JMS finds a suitable message and delivers it to the consumer, WebSphere MQ classes for JMS resumes the search for the next suitable message from its current position in the queue. WebSphere MQ classes for JMS continues to search the queue in this way until it reaches the end of the queue, or until the interval of time in milliseconds, as determined by the value of this property, has expired. In each case, WebSphere MQ classes for JMS returns to the beginning of the queue to continue its search, and a new time interval commences.
securityExit	String	• **null** • The fully qualified name of a class that implements the WebSphere MQ classes for Java interface, MQSecurityExit	Identifies a channel security exit program
securityExitInit	String	• **null** • A string of user data	The user data that is passed to a channel security exit program when it is called
sendCheckCount	int	• **0** • Any positive integer	The number of send calls to allow between checking for asynchronous put errors, within a single non-transacted JMS session
sendExit	String	• **null** • A string comprising one or more items separated by commas, where each item is the fully qualified name of a class that implements the WebSphere MQ classes for Java interface, MQSendExit	Identifies a channel send exit program, or a sequence of send exit programs to be run in succession
sendExitInit	String	• **null** • A string comprising one or more items of user data separated by commas	The user data that is passed to channel send exit programs when they are called

Table 12. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
shareConvAllowed	boolean	• **false** - A client connection cannot share its socket. • true - A client connection can share its socket.	Whether a client connection can share its socket with other top-level JMS connections from the same process to the same queue manager, if the channel definitions match
sparseSubscriptions[1]	boolean	• **false** - Subscriptions receive frequent matching messages. • true - Subscriptions receive infrequent matching messages. This value requires that the subscription queue can be opened for browse.	Controls the message retrieval policy of a TopicSubscriber object
sslCertStores	String	• **null** • A string of one or more LDAP URLs separated by blanks. Each LDAP URL has the format: `ldap://host_name[:port]` where *host_name* is a host name or IP address, *port* is a TCP port number, and brackets denote an optional component.	The Lightweight Directory Access Protocol (LDAP) servers that hold certificate revocation lists (CRLs) for use on an SSL connection
sslCipherSuite	String	• **null** • The name of a CipherSuite	The CipherSuite to use for an SSL connection
sslFipsRequired[2]	boolean	• **false** • true	Whether an SSL connection must use a CipherSuite that is supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS)
sslPeerName	String	• **null** • A template for distinguished names	For an SSL connection, a template that is used to check the distinguished name in the digital certificate provided by the queue manager
sslResetCount	int	• **0** • An integer in the range 0 to 999 999 999	The total number bytes sent and received by an SSL connection before the secret keys used by SSL are renegotiated
sslSocketFactory	String	A string representing the fully-qualified class name of a class providing an implementation of the javax.net.ssl.SSLSocketFactory interface, optionally including an argument to be passed to the constructor method, enclosed in parentheses.	Any connections established in the scope of the administered destination object use sockets obtained from this implementation of the SSLSocketFactory interface.

Table 12. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
statusRefreshInterval[1]	int	• **60000** • Any positive integer	The interval, in milliseconds, between refreshes of the long running transaction that detects when a subscriber loses its connection to the queue manager. This property is relevant only if SUBSTORE has the value QUEUE.
subscriptionStore[1]	String	• **BROKER** • MIGRATE • QUEUE	Determines where WebSphere MQ classes for JMS stores persistent data about active subscriptions
targetClientMatching	boolean	• **true** • false	Whether a reply message, sent to the queue identified by the JMSReplyTo header field of an incoming message, has an MQRFH2 header only if the incoming message has an MQRFH2 header
temporaryModel	String	• **SYSTEM.DEFAULT.MODEL.QUEUE** • Any string	The name of the model queue from which JMS temporary queues are created
tempQPrefix	String	• "" **(empty string)** • A prefix that can be used to form the name of a WebSphere MQ dynamic queue. The rules for forming the prefix are the same as those for forming the contents of the *DynamicQName* field in a WebSphere MQ object descriptor, structure MQOD, but the last non blank character must be an asterisk (*). If the value of the property is the empty string, WebSphere MQ classes for JMS uses the value AMQ.* when creating a dynamic queue.	The prefix that is used to form the name of a WebSphere MQ dynamic queue.
tempTopicPrefix	String	Any non-null string consisting only of valid characters for a WebSphere MQ topic string	When creating temporary topics, JMS will generate a topic string of the form "TEMP/ *TEMPTOPICPREFIX*/ *unique_id*", or if this property is left with the default value, just "TEMP/*unique_id*". Specifying a non-empty TEMPTOPICPREFIX allows specific model queues to be defined for creating the managed queues for subscribers to temporary topics created under this connection.

Table 12. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
transportType	String	• **CLIENT** • BINDINGS • BINDINGS_THEN_CLIENT	Whether a connection to a queue manager uses client mode or bindings mode.If the value BINDINGS_THEN_CLIENT is specified, the resource adapter first tries to make a connection in bindings mode, and if this fails then tries to make a client mode connection.
username	String	• **null** • A user name	The default user name to use when creating a connection to a queue manager
wildcardFormat	int	• CHAR- Recognizes character wildcards only, as used in broker version 1 • TOPIC - Recognizes topic level wildcards only, as used in broker version 2	Which version of wildcard syntax is to be used

Notes:

1. This property can be used with Version 7.0 of WebSphere MQ classes for JMS but has no effect for an application connected to a Version 7.0 queue manager unless the providerVersion property is set to a version number less than 7.
2. For important information about using the sslFipsRequired property, see "Limitations of the WebSphere MQ resource adapter" on page 42.

The following example shows a typical set of properties of a ConnectionFactory object:

```
channel:        SYSTEM.DEF.SVRCONN
hostName:       192.168.0.42
port:           1414
queueManager:   ExampleQM
transportType:  CLIENT
```

Table 13 lists the properties that are common to a Queue object and a Topic object.

Table 13. Properties that are common to a Queue object and a Topic object

Name of property	Type	Valid values (default value in bold)	Description
CCSID	String	• **1208** • A coded character set identifier supported by the Java virtual machine (JVM)	The coded character set identifier for the destination

Table 13. Properties that are common to a Queue object and a Topic object (continued)

Name of property	Type	Valid values (default value in bold)	Description
encoding	String	• **NATIVE** • A string of three characters: – The first character specifies the representation of binary integers: - *N* denotes normal encoding. - *R* denotes reverse encoding. – The second character specifies the representation of packed decimal integers: - *N* denotes normal encoding. - *R* denotes reverse encoding. – The third character specifies the representation of floating point numbers: - *N* denotes standard IEEE encoding. - *R* denotes reverse IEEE encoding. - *3* denotes zSeries encoding. NATIVE is equivalent to the string NNN.	The representation of binary integers, packed decimal integers, and floating point numbers for the destination.
expiry	String	• **APP** - The expiry time of a message is determined by the message producer. • UNLIM - A message never expires. • 0 - A message never expires. • A positive integer representing the expiry time of a message in milliseconds.	The expiry time of a message sent to the destination
failIfQuiesce	String	• **true** • false	Whether an attempt to access the destination fails if the queue manager is in a quiescing state
persistence	String	• **APP** - The persistence of a message is determined by the message producer. • QDEF - The persistence of a message is determined by the `DefPersistence` attribute of the WebSphere MQ queue. • PERS - A message is persistent. • NON - A message is nonpersistent. • HIGH - The persistence of a message is determined by the `NonPersistentMessageClass` attribute of the WebSphere MQ queue according to the explanation in "JMS persistent messages" on page 140.	The persistence of a message sent to the destination
priority	String	• **APP** - The priority of a message is determined by the message producer. • QDEF - The priority of a message is determined by the `DefPriority` attribute of the WebSphere MQ queue. • An integer in the range 0, lowest priority, to 9, highest priority.	The priority of a message sent to the destination
readAheadAllowed	int	• **DESTINATION** - Determine whether read ahead is allowed by referring to the queue or topic definition. • DISABLED - Read ahead is not allowed. • ENABLED - Read ahead is allowed. • QUEUE - Determine whether read ahead is allowed by referring to the queue definition. • TOPIC - Determine whether read ahead is allowed by referring to the topic definition.	Whether message consumers and queue browsers are allowed to use read ahead to get nonpersistent messages from the destination into an internal buffer before receiving them

Table 13. Properties that are common to a Queue object and a Topic object (continued)

Name of property	Type	Valid values (default value in bold)	Description
targetClient	String	• **JMS** - The target of a message is a JMS application. • MQ - The target of a message is a non-JMS WebSphere MQ application.	Whether the target of a message sent to the destination is a JMS application. A message whose target is a JMS application contains an MQRFH2 header.

Table 14 lists the properties that are specific to a Queue object.

Table 14. Properties that are specific to a Queue object

Name of property	Type	Valid values (default value in bold)	Description
baseQueueManagerName	String	• **"" (empty string)** • A queue manager name	The name of the queue manager that owns the underlying WebSphere MQ queue
baseQueueName	String	• **"" (empty string)** • A queue name	The name of the underlying WebSphere MQ queue

Table 15 lists the properties that are specific to a Topic object.

Table 15. Properties that are specific to a Topic object

Name of property	Type	Valid values (default value in bold)	Description
baseTopicName	String	• **"" (empty string)** • A topic name	The name of the underlying topic
brokerCCDurSubQueue[1]	String	• **SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a connection consumer receives durable subscription messages
brokerDurSubQueue[1]	String	• **SYSTEM.JMS.D.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a durable topic subscriber receives messages
brokerPubQueue[1]	String	• **Not set** • A queue name	The name of the queue where published messages are sent (the stream queue). The value of this property overrides the value of the brokerPubQueue property of the ConnectionFactory object. However, if you do not set the value of this property, the value of the brokerPubQueue property of the ConnectionFactory object is used instead.
brokerPubQueueManager[1]	String	• **"" (empty string)** • A queue manager name	The name of the queue manager that owns the queue where messages published on the topic are sent

Table 15. Properties that are specific to a Topic object (continued)

Name of property	Type	Valid values (default value in bold)	Description
brokerVersion[1]	String	• **Not set** • 1 • 2	The version of the broker being used. The value of this property overrides the value of the brokerVersion property of the ConnectionFactory object. However, if you do not set the value of this property, the value of the brokerVersion property of the ConnectionFactory object is used instead.
Note: 1. This property can be used with Version 7.0 of WebSphere MQ classes for JMS but has no effect for an application connected to a Version 7.0 queue manager unless the providerVersion property of the ConnectionFactory object is set to a version number less than 7.			

The following example shows a set of properties of a Queue object:

```
expiry:               UNLIM
persistence:          QDEF
baseQueueManagerName: ExampleQM
baseQueueName:        SYSTEM.DEFAULT.LOCAL.QUEUE
```

The following example shows a set of properties of a Topic object:

```
expiry:               UNLIM
persistence:          NON
baseTopicName:        myTestTopic
```

Limitations of the WebSphere MQ resource adapter

When you use the WebSphere MQ resource adapter, some features of WebSphere MQ are unavailable or limited.

The WebSphere MQ resource adapter has the following limitations:

- The WebSphere MQ resource adapter is supported on all WebSphere MQ platforms, except z/OS.
- The WebSphere MQ resource adapter does not support real-time connections to a broker. It supports only connections to a WebSphere MQ queue manager in client or bindings mode.
- The WebSphere MQ resource adapter does not support channel exit programs that are written in languages other than Java.
- While an application server is running, the value of the sslFipsRequired property must be true for all JCA resources or false for all JCA resources. This is a requirement even if the JCA resources are not used concurrently. If the sslFipsRequired property has different values for different JCA resources, WebSphere MQ issues the reason code MQRC_UNSUPPORTED_CIPHER_SUITE, even if an SSL connection is not being used.
- You cannot specify more than one key store for an application server. If connections are made to more than one queue manager, all the connections must use the same key store.
- If you use a client channel definition table (CCDT) with more than one suitable client connection channel definition, in the event of a failure the resource adapter might select a different channel definition and therefore a different

queue manager from the CCDT, which would cause problems for transaction recovery. The resource adapter does not take any action to prevent such a configuration from being used, and it is your responsibility to avoid configurations that may cause problems for transaction recovery.

Using WebSphere MQ classes for JMS

This topic describes how to run WebSphere MQ classes for JMS applications. It tells you how to configure WebSphere MQ, and how to run the installation verification test programs. It describes the scripts provided with WebSphere MQ classes for JMS and the support for OSGi, and provides guidance on solving problems.

Post installation setup for WebSphere MQ classes for JMS applications

This topic tells you what authorities WebSphere MQ classes for JMS applications need in order to access the resources of a queue manager. It also introduces connection modes and describes how to configure a queue manager so that applications can connect in client mode.

Remember to check the WebSphere MQ readme file. It might contain information that supersedes the information in this topic.

Objects that require authorization for non-privileged users

Non-privileged users need authorization granted to access the queues used by JMS. Every JMS application needs authorization to the queue manager with which it words.

For details about access control in WebSphere MQ, see the chapter about WebSphere MQ security in the *WebSphere MQ System Administration Guide*.

A WebSphere MQ classes for JMS application needs connect and inq authority to the queue manager. You can set appropriate authorizations using the setmqaut control command, for example:

```
setmqaut -m QM1 -t qmgr -g jmsappsgroup +connect +inq
```

For the point-to-point domain, the following authorities are required:

- Queues that are used by MessageProducer objects need put authority.
- Queues that are used by MessageConsumer and QueueBrowser objects need get, inq, and browse authorities.
- The QueueSession.createTemporaryQueue() method needs access to the model queue specified by the TEMPMODEL property of the QueueConnectionFactory object. By default this model queue is SYSTEM.DEFAULT.MODEL.QUEUE.

For the publish/subscribe domain, the following queues are used if WebSphere MQ classes for JMS is in WebSphere MQ messaging provider migration mode:

- SYSTEM.JMS.ADMIN.QUEUE
- SYSTEM.JMS.REPORT.QUEUE
- SYSTEM.JMS.MODEL.QUEUE
- SYSTEM.JMS.PS.STATUS.QUEUE
- SYSTEM.JMS.ND.SUBSCRIBER.QUEUE

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

Additionally, if WebSphere MQ classes for JMS is in this mode, any application that publishes messages needs access to the stream queue specified by the TopicConnectionFactory or Topic object. By default, this queue is SYSTEM.BROKER.DEFAULT.STREAM.

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

Connection modes for WebSphere MQ classes for JMS

A WebSphere MQ classes for JMS application can connect to a queue manager in either client or bindings mode. In client mode, WebSphere MQ classes for JMS connects to the queue manager over TCP/IP. In bindings mode, WebSphere MQ classes for JMS connects directly to the queue manager using the Java Native Interface (JNI).

An application running in WebSphere Application Server on z/OS can connect to a queue manager in either bindings or client mode, but an application running in any other environment on z/OS can connect to a queue manager only in bindings mode. An application running on any other platform can connect to a queue manager in either bindings or client mode.

The following sections describe each of the connection modes in more detail.

Client mode

To connect to a queue manager in client mode, a WebSphere MQ classes for JMS application can run on the same system on which the queue manager is running, or on a different system. In each case, WebSphere MQ classes for JMS connects to the queue manager over TCP/IP.

Bindings mode

To connect to a queue manager in bindings mode, a WebSphere MQ classes for JMS application must run on the same system on which the queue manager is running. WebSphere MQ classes for JMS connects directly to the queue manager using the Java Native Interface (JNI). In some environments, connecting in bindings mode can provide better performance than connecting in client mode.

Configuring your queue manager so that WebSphere MQ classes for JMS applications can connect in client mode

To configure your queue manager so that WebSphere MQ classes for JMS applications can connect in client mode, you must create a server connection channel definition and start a listener.

On z/OS, the Client Attachment feature must be installed.

Creating a server connection channel definition

On all platforms, you can use the MQSC command DEFINE CHANNEL to create a server connection channel definition. See the following example:

```
DEFINE CHANNEL(JAVA.CHANNEL) CHLTYPE(SVRCONN) TRPTYPE(TCP)
```

On i5/OS, you can use the CL command CRTMQMCHL instead, as in the following example:

```
CRTMQMCHL CHLNAME(JAVA.CHANNEL) CHLTYPE(*SVRCN)
                                TRPTYPE(*TCP)
                                MQMNAME(QMGRNAME)
```

In this command, *QMGRNAME* is the name of your queue manager.

You can also create a server connection channel definition using WebSphere MQ Explorer, which runs on Linux and Windows, or the operations and control panels on z/OS.

The name of the channel (JAVA.CHANNEL in the previous examples) must be the same as the channel name specified by the CHANNEL property of the connection factory that your application uses to connect to the queue manager. The default value of the CHANNEL property is SYSTEM.DEF.SVRCONN.

Starting a listener

You must start a listener for your queue manager if one is not already started.

On all platforms, you can use the MQSC command START LISTENER to start a listener but, except on z/OS, you must first create a listener object by using the MQSC command DEFINE LISTENER. See the following example:

```
DEFINE LISTENER(LISTENER.TCP) TRPTYPE(TCP) PORT(1414)
START LISTENER(LISTENER.TCP)
```

On z/OS, you use only the START LISTENER command, as in the following example, but note that the channel initiator address space must be started before you can start a listener:

```
START LISTENER TRPTYPE(TCP) PORT(1414)
```

On i5/OS, you can also use the CL command STRMQMLSR to start a listener, as in the following example:

```
STRMQMLSR PORT(1414) MQMNAME(QMGRNAME)
```

In this command, *QMGRNAME* is the name of your queue manager.

On UNIX systems and Windows, you can also use the control command **runmqlsr** to start a listener, as in the following example:

```
runmqlsr -t tcp -p 1414 -m QMgrName
```

In this command, *QMgrName* is the name of your queue manager.

You can also start a listener using WebSphere MQ Explorer, which runs on Linux and Windows, or the operations and control panels on z/OS.

The number of the port on which the listener is listening must be the same as the port number specified by the PORT property of the connection factory that your application uses to connect to the queue manager. The default value of the PORT property is 1414.

The point-to-point installation verification test for WebSphere MQ classes for JMS

A point-to-point installation verification test (IVT) program is supplied with WebSphere MQ classes for JMS. The program connects to a queue manager in either bindings or client mode, sends a message to the queue called SYSTEM.DEFAULT.LOCAL.QUEUE, and then receives the message from the queue. The program can create and configure all the objects that it requires dynamically at run time, or it can use JNDI to retrieve administered objects from a directory service.

Run the installation verification test without using JNDI first because the test is self contained and does not require the use of a directory service. For a description of administered objects, see "Object types" on page 172.

The point-to-point installation verification test without using JNDI

In this test, the IVT program creates and configures all the objects that it requires dynamically at run time and does not use JNDI.

A script is provided to run the IVT program. The script is called IVTRun on UNIX systems and IVTRun.bat on Windows, and is in the `bin` subdirectory of the WebSphere MQ classes for JMS installation directory.

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

```
IVTRun -nojndi [-m qmgr] [-v providerVersion] [-t]
```

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

```
IVTRun -nojndi -client -m qmgr -host hostname [-port port] [-channel channel]
      [-v providerVersion] [-ccsid ccsid] [-t]
```

The parameters on the commands have the following meanings:

-m *qmgr*
: The name of the queue manager to which the IVT program connects. If you run the test in bindings mode and omit this parameter, the IVT program connects to the default queue manager.

-host *hostname*
: The host name or IP address of the system on which the queue manager is running.

-port *port*
: The number of the port on which the listener of the queue manager is listening. The default value is `1414`.

-channel *channel*
: The name of the MQI channel that the IVT program uses to connect to the queue manager. The default value is `SYSTEM.DEF.SVRCONN`.

-v *providerVersion*
: The release level of the queue manager to which the IVT program expects to connect.

: This parameter is used to set the PROVIDERVERSION property of an MQQueueConnectionFactory object and has the same valid values as those of the PROVIDERVERSION property. For more information about this parameter therefore, including its valid values, see the description of the PROVIDERVERSION property in “Properties of WebSphere MQ classes for JMS objects” on page 176.

The default value is `unspecified`.

-ccsid *ccsid*
: The identifier (CCSID) of the coded character set, or code page, to be used by the connection. The default value is 819.

-t
: Tracing is switched on. By default, tracing is switched off.

A successful test produces output similar to the following sample output:

```
5724-H72, 5655-R36, 5724-L26, 5655-L82 (c) Copyright IBM Corp. 2008. All Rights Reserved.
Websphere MQ classes for Java(tm) Message Service 7.0
Installation Verification Test

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

Got message
  JMSMessage class: jms_text
  JMSType:          null
  JMSDeliveryMode:  2
  JMSExpiration:    0
  JMSPriority:      4
  JMSMessageID:     ID:414d5120514d5f6d627720202020202001edb14620005e03
  JMSTimestamp:     1187170264000
  JMSCorrelationID: null
  JMSDestination:   queue:///SYSTEM.DEFAULT.LOCAL.QUEUE
  JMSReplyTo:       null
  JMSRedelivered:   false
    JMSXUserID: mwhite
    JMS_IBM_Encoding: 273
    JMS_IBM_PutApplType: 28
    JMSXAppID: WebSphere MQ Client for Java
    JMSXDeliveryCount: 1
    JMS_IBM_PutDate: 20070815
    JMS_IBM_PutTime: 09310400
    JMS_IBM_Format: MQSTR
    JMS_IBM_MsgType: 8
A simple text message from the MQJMSIVT
Reply string equals original string
Closing QueueReceiver
Closing QueueSender
Closing Session
Closing Connection
IVT completed OK
IVT finished
```

The point-to-point installation verification test using JNDI

In this test, the IVT program uses JNDI to retrieve administered objects from a directory service.

Before you can run the test, you must configure a directory service that is based on a Lightweight Directory Access Protocol (LDAP) server or the local file system. You must also configure the WebSphere MQ JMS administration tool so that it can use the directory service to store administered objects. For more information about these prerequisites, see "Prerequisites for WebSphere MQ classes for JMS" on page 7. For information about how to configure the WebSphere MQ JMS administration tool, see "Configuration" on page 168.

The IVT program must be able to use JNDI to retrieve an MQQueueConnectionFactory object and an MQQueue object from the directory service. A script is provided to create these administered objects for you. The script is called IVTSetup on UNIX systems and IVTSetup.bat on Windows, and is in the `bin` subdirectory of the WebSphere MQ classes for JMS installation directory. To run the script, enter the following command:

```
IVTSetup
```

The script invokes the WebSphere MQ JMS administration tool to create the administered objects.

The MQQueueConnectionFactory object is bound with the name ivtQCF and is created with the default values for all its properties, which means that the IVT program runs in bindings mode and connects to the default queue manager. If you want the IVT program to run in client mode, or connect to a queue manager other than the default queue manager, you must use the WebSphere MQ JMS administration tool or WebSphere MQ Explorer to change the appropriate properties of the MQQueueConnectionFactory object. For information about how to use the WebSphere MQ JMS administration tool, see "Using the WebSphere MQ JMS administration tool" on page 167. For information about how to use WebSphere MQ Explorer, see the help provided with WebSphere MQ Explorer.

The MQQueue object is bound with the name ivtQ and is created with the default values for all its properties, except for the QUEUE property, which has the value SYSTEM.DEFAULT.LOCAL.QUEUE.

When you have created the administered objects, you can run the IVT program. To run the test using JNDI, enter the following command:

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

The parameters on the command have the following meanings:

-url *"providerURL"*
The uniform resource locator (URL) of the directory service. The URL can have one of the following formats:
- `ldap://hostname/contextName`, for a directory service based on an LDAP server
- `file:/directoryPath`, for a directory service based on the local file system

Note that you must enclose the URL in quotation marks (").

-icf *initCtxFact*
The class name of the initial context factory, which must be one of the following values:
- `com.sun.jndi.ldap.LdapCtxFactory`, for a directory service based on an LDAP server. This is the default value.
- `com.sun.jndi.fscontext.RefFSContextFactory`, for a directory service based on the local file system.

-t Tracing is switched on. By default, tracing is switched off.

A successful test produces output similar to that for a successful test without using JNDI. The main difference is that the output indicates that the test is using JNDI to retrieve an MQQueueConnectionFactory object and an MQQueue object.

Although not strictly necessary, it is good practice to tidy up after the test by deleting the administered objects created by the IVTSetup script. A script is provided for this purpose. The script is called IVTTidy on UNIX systems and IVTTidy.bat on Windows, and is in the `bin` subdirectory of the WebSphere MQ classes for JMS installation directory.

Problem determination for the point-to-point installation verification test

The installation verification test might fail for the following reasons:

- If the IVT program writes a message indicating that it cannot find a class, check that your class path is set correctly, as described in "Environment variables used by WebSphere MQ classes for JMS" on page 9.
- The test might fail with the following message:

  ```
  Failed to connect to queue manager 'qmgr' with connection mode 'connMode'
  and host name 'hostname'
  ```

  and an associated reason code of 2059. The variables in the message have the following meanings:

  qmgr The name of the queue manager to which the IVT program is trying to connect. This message insert is blank if the IVT program is trying to connect to the default queue manager in bindings mode.

  connMode
  The connection mode, which is either `Bindings` or `Client`.

  hostname
  The host name or IP address of the system on which the queue manager is running.

  This message means that the queue manager to which the IVT program is trying to connect is not available. Check that the queue manager is running and, if the IVT program is trying to connect to the default queue manager, make sure that the queue manager is defined as the default queue manager for your system.
- The test might fail with the following message:

  ```
  Failed to open MQ queue 'SYSTEM.DEFAULT.LOCAL.QUEUE'
  ```

  This message means that the queue SYSTEM.DEFAULT.LOCAL.QUEUE does not exist on the queue manager to which the IVT program is connected. Alternatively, if the queue does exist, the IVT program cannot open the queue because it is not enabled for putting and getting messages. Check that the queue exists and that it is enabled for putting and getting messages.
- The test might fail with the following message:

  ```
  Unable to bind to object
  ```

  This message means that there is a connection to the LDAP server, but that the LDAP server is not correctly configured. Either the LDAP server is not configured for storing Java objects, or the permissions on the objects or the suffix are not correct. For more help in this situation, see the documentation for your LDAP server.

The publish/subscribe installation verification test for WebSphere MQ classes for JMS

A publish/subscribe installation verification test (IVT) program is supplied with WebSphere MQ classes for JMS. The program connects to a queue manager in either bindings or client mode, subscribes to a topic, publishes a message on the topic, and then receives the message that it has just published. The program can create and configure all the objects that it requires dynamically at run time, or it can use JNDI to retrieve administered objects from a directory service.

Run the installation verification test without using JNDI first because the test is self contained and does not require the use of a directory service. For a description of administered objects, see "Object types" on page 172.

The publish/subscribe installation verification test without using JNDI

In this test, the IVT program creates and configures all the objects that it requires dynamically at run time and does not use JNDI.

A script is provided to run the IVT program. The script is called PSIVTRun on UNIX systems and PSIVTRun.bat on Windows, and is in the `bin` subdirectory of the WebSphere MQ classes for JMS installation directory.

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

```
PSIVTRun -nojndi [-m qmgr] [-bqm brokerQmgr] [-v providerVersion] [-t]
```

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

```
PSIVTRun -nojndi -client -m qmgr -host hostname [-port port] [-channel channel]
        [-bqm brokerQmgr] [-v providerVersion] [-ccsid ccsid] [-t]
```

The parameters on the commands have the following meanings:

-m *qmgr*
: The name of the queue manager to which the IVT program connects. If you run the test in bindings mode and omit this parameter, the IVT program connects to the default queue manager.

-host *hostname*
: The host name or IP address of the system on which the queue manager is running.

-port *port*
: The number of the port on which the listener of the queue manager is listening. The default value is `1414`.

-channel *channel*
: The name of the MQI channel that the IVT program uses to connect to the queue manager. The default value is `SYSTEM.DEF.SVRCONN`.

-bqm *brokerQmgr*
: The name of the queue manager on which the broker is running. The default value is the name of the queue manager to which the IVT program connects.

  This parameter is relevant only if the -v parameter specifies a queue manager version number less than 7 and you are using WebSphere Event Broker or WebSphere Message Broker as the publish/subscribe broker.

-v *providerVersion*
: The release level of the queue manager to which the IVT program expects to connect.

  This parameter is used to set the PROVIDERVERSION property of an MQTopicConnectionFactory object and has the same valid values as those of the PROVIDERVERSION property. For more information about this parameter therefore, including its valid values, see the description of the PROVIDERVERSION property in "Properties of WebSphere MQ classes for JMS objects" on page 176.

  The default value is `unspecified`.

-ccsid *ccsid*
: The identifier (CCSID) of the coded character set, or code page, to be used by the connection. The default value is 819.

-t
: Tracing is switched on. By default, tracing is switched off.

A successful test produces output similar to the following sample output:

```
5724-H72, 5655-R36, 5724-L26, 5655-L82 (c) Copyright IBM Corp. 2008. All Rights Reserved.
Websphere MQ classes for Java(tm) Message Service 7.0
Publish/Subscribe Installation Verification Test

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

Got message:
  JMSMessage class: jms_text
  JMSType:          null
  JMSDeliveryMode:  2
  JMSExpiration:    0
  JMSPriority:      4
  JMSMessageID:     ID:414d5120514d5f6d627720202020202001edb14620006706
  JMSTimestamp:     1187182520203
  JMSCorrelationID: ID:414d5120514d5f6d627720202020202001edb14620006704
  JMSDestination:   topic://MQJMS/PSIVT/Information
  JMSReplyTo:       null
  JMSRedelivered:   false
    JMSXUserID: mwhite
    JMS_IBM_Encoding: 273
    JMS_IBM_PutApplType: 26
    JMSXAppID: QM_mbw
    JMSXDeliveryCount: 1
    JMS_IBM_PutDate: 20070815
    JMS_IBM_ConnectionID: 414D5143514D5F6D627720202020202001EDB14620006601
    JMS_IBM_PutTime: 12552020
    JMS_IBM_Format: MQSTR
    JMS_IBM_MsgType: 8
A simple text message from the MQJMSPSIVT program
Reply string equals original string
Closing TopicSubscriber
Closing TopicPublisher
Closing Session
Closing Connection
PSIVT finished
```

The publish/subscribe installation verification test using JNDI

In this test, the IVT program uses JNDI to retrieve administered objects from a directory service.

Before you can run the test, you must configure a directory service that is based on a Lightweight Directory Access Protocol (LDAP) server or the local file system. You must also configure the WebSphere MQ JMS administration tool so that it can use the directory service to store administered objects. For more information about these prerequisites, see “Prerequisites for WebSphere MQ classes for JMS” on page 7. For information about how to configure the WebSphere MQ JMS administration tool, see “Configuration” on page 168.

The IVT program must be able to use JNDI to retrieve an MQTopicConnectionFactory object and an MQTopic object from the directory service. A script is provided to create these administered objects for you. The script is called IVTSetup on UNIX systems and IVTSetup.bat on Windows, and is in the `bin` subdirectory of the WebSphere MQ classes for JMS installation directory. To run the script, enter the following command:

```
IVTSetup
```

The script invokes the WebSphere MQ JMS administration tool to create the administered objects.

The MQTopicConnectionFactory object is bound with the name ivtTCF and is created with the default values for all its properties, which means that the IVT program runs in bindings mode, connects to the default queue manager, and uses the embedded publish/subscribe function. If you want the IVT program to run in client mode, connect to a queue manager other than the default queue manager, or use WebSphere Event Broker or WebSphere Message Broker instead of the embedded publish/subscribe function, you must use the WebSphere MQ JMS administration tool or WebSphere MQ Explorer to change the appropriate properties of the MQTopicConnectionFactory object. For information about how to use the WebSphere MQ JMS administration tool, see “Using the WebSphere MQ JMS administration tool” on page 167. For information about how to use WebSphere MQ Explorer, see the help provided with WebSphere MQ Explorer.

The MQTopic object is bound with the name ivtT and is created with the default values for all its properties, except for the TOPIC property, which has the value MQJMS/PSIVT/Information.

When you have created the administered objects, you can run the IVT program. To run the test using JNDI, enter the following command:

```
PSIVTRun -url "providerURL" [-icf initCtxFact] [-t]
```

The parameters on the command have the following meanings:

-url *"providerURL"*
: The uniform resource locator (URL) of the directory service. The URL can have one of the following formats:
  - `ldap://hostname/contextName`, for a directory service based on an LDAP server
  - `file:/directoryPath`, for a directory service based on the local file system

  Note that you must enclose the URL in quotation marks (").

-icf *initCtxFact*
: The class name of the initial context factory, which must be one of the following values:
  - `com.sun.jndi.ldap.LdapCtxFactory`, for a directory service based on an LDAP server. This is the default value.
  - `com.sun.jndi.fscontext.RefFSContextFactory`, for a directory service based on the local file system.

-t
: Tracing is switched on. By default, tracing is switched off.

A successful test produces output similar to that for a successful test without using JNDI. The main difference is that the output indicates that the test is using JNDI to retrieve an MQTopicConnectionFactory object and an MQTopic object.

Although not strictly necessary, it is good practice to tidy up after the test by deleting the administered objects created by the IVTSetup script. A script is provided for this purpose. The script is called IVTTidy on UNIX systems and IVTTidy.bat on Windows, and is in the `bin` subdirectory of the WebSphere MQ classes for JMS installation directory.

Problem determination for the publish/subscribe installation verification test

The installation verification test might fail for the following reasons:

- If the IVT program writes a message indicating that it cannot find a class, check that your class path is set correctly, as described in “Environment variables used by WebSphere MQ classes for JMS” on page 9.
- The test might fail with the following message:

  ```
  Failed to connect to queue manager 'qmgr' with
  connection mode 'connMode' and host name 'hostname'
  ```

  and an associated reason code of 2059. The variables in the message have the following meanings:

  qmgr
  : The name of the queue manager to which the IVT program is trying to connect. This message insert is blank if the IVT program is trying to connect to the default queue manager in bindings mode.

  connMode
  : The connection mode, which is either `Bindings` or `Client`.

  hostname
  : The host name or IP address of the system on which the queue manager is running.

  This message means that the queue manager to which the IVT program is trying to connect is not available. Check that the queue manager is running and, if the IVT program is trying to connect to the default queue manager, make sure that the queue manager is defined as the default queue manager for your system.
- The test might fail with the following message:

  ```
  Unable to bind to object
  ```

  This message means that there is a connection to the LDAP server, but that the LDAP server is not correctly configured. Either the LDAP server is not configured for storing Java objects, or the permissions on the objects or the suffix are not correct. For more help in this situation, see the documentation for your LDAP server.

The installation verification test program for the WebSphere MQ resource adapter

The installation verification test (IVT) program is supplied as an enterprise archive (EAR) file called wmq.jmsra.ivt.ear. This file is installed with WebSphere MQ classes for JMS in the same directory as the WebSphere MQ resource adapter RAR file, wmq.jmsra.rar. For information about where these files are installed, see "Installation of the WebSphere MQ resource adapter" on page 16.

You must deploy the IVT program on your application server. The IVT program runs as a servlet and tests that a message can be sent to, and received from, a WebSphere MQ classes for JMS Queue or Topic object. Optionally, you can use the IVT program to verify that the WebSphere MQ resource adapter has been correctly configured to support distributed transactions.

Before you can run the IVT program, you must define the properties of a ConnectionFactory object and a Queue or Topic object as JCA resources, and ensure that your application server creates JMS objects from these definitions and binds them into a JNDI namespace. You can choose the properties of the objects, but the following set of properties is a simple example:

ConnectionFactory object

```
channel:              SYSTEM.DEF.SVRCONN
hostName:             192.168.0.42
port:                 1414
queueManager:         ExampleQM
transportType:        CLIENT
```

Queue object

```
baseQueueManagerName: ExampleQM
baseQueueName:        SYSTEM.DEFAULT.LOCAL.QUEUE
```

By default, the IVT program expects a ConnectionFactory object to be bound in the JNDI namespace with the name IVTCF and a Queue object to be bound with the name IVTQueue. You can use different names, and you can use a Topic object instead of a Queue object as a destination. But if you do, you must enter the names of the objects on the initial page of the IVT program.

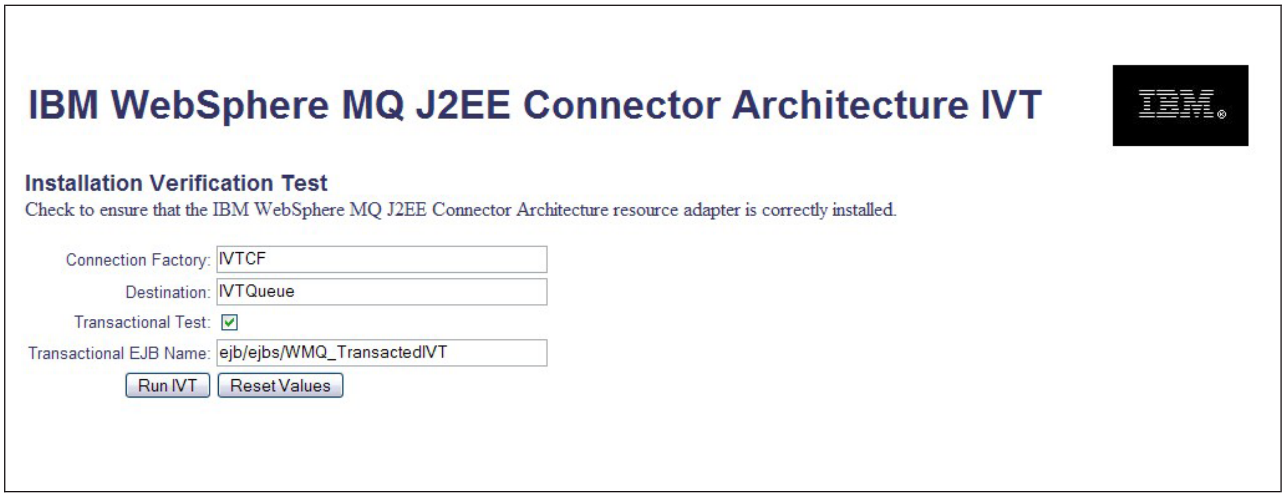
After you have deployed the IVT program, and the application server has created the JMS objects and bound them into the JNDI namespace, you can start the IVT program by entering a URL in the following format into your Web browser:

`http://`*`app_server_host`*`:`*`port`*`/WMQ_IVT/`

where *app_server_host* is the IP address or host name of the system on which your application server is running, and *port* is the number of the TCP port on which the application server is listening. Here is an example:

```
http://localhost:9080/WMQ_IVT/
```

Figure 1 on page 55 shows the initial page of the IVT program.


IBM WebSphere MQ J2EE Connector Architecture IVT

Installation Verification Test

Check to ensure that the IBM WebSphere MQ J2EE Connector Architecture resource adapter is correctly installed.

Connection Factory: IVTCF

Destination: IVTQueue

Transactional Test: ✓

Transactional EJB Name: ejb/ejbs/WMQ_TransactedIVT

Figure 1. The initial page of the IVT program

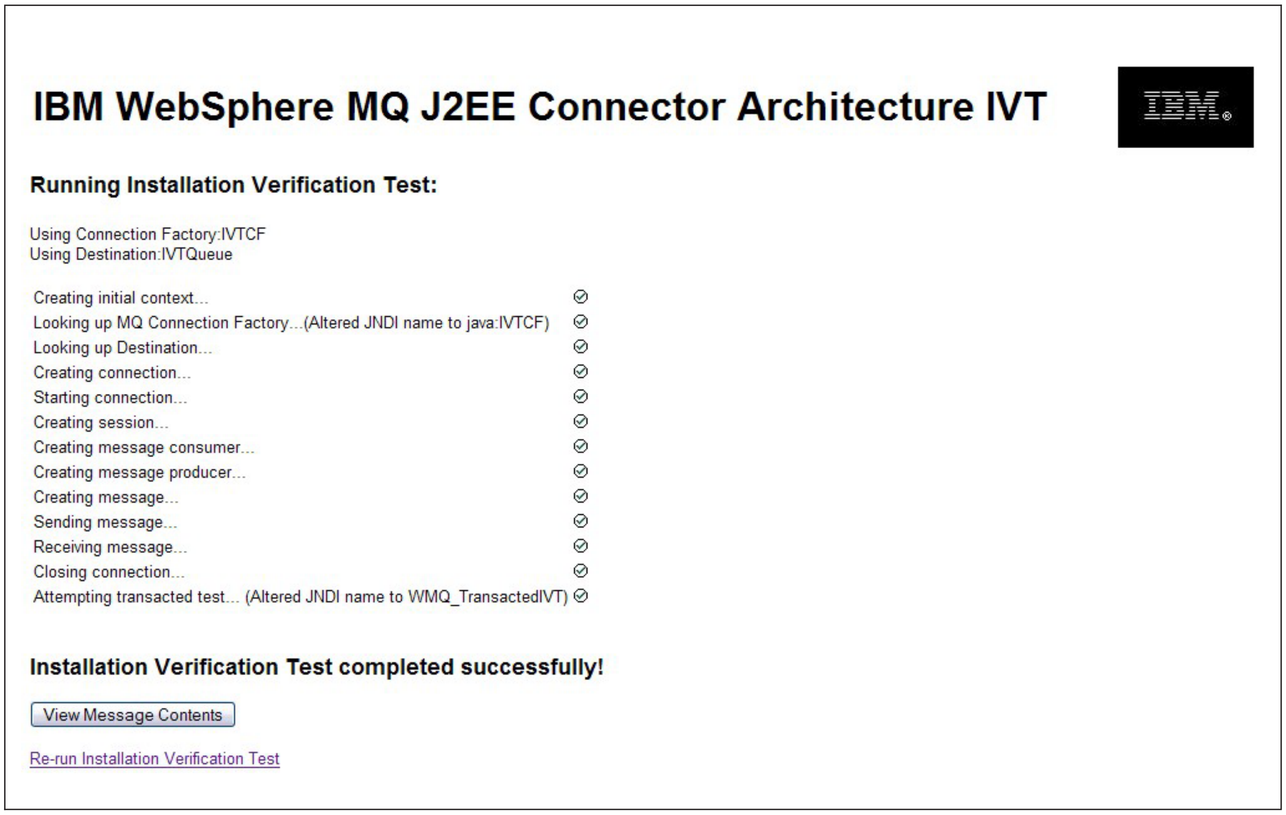
On the initial page, the **Connection Factory** field already contains the name `IVTCF` and the **Destination** field already contains the name `IVTQueue`. If you want to run the IVT program using a ConnnectionFactory object and a Queue or Topic object that are bound in the JNDI namespace with different names, you must enter the JNDI names of the objects into these fields, replacing the existing contents.

If you want to verify that the WebSphere MQ resource adapter can support distributed transactions, select the **Transactional Test** check box.

The **Transactional EJB Name** field specifies the JNDI name of the enterprise Java bean (EJB) to be used for the transactional test. By default, the IVT program expects the EJB to be bound with the name ejb/ejbs/WMQ_TransactedIVT, and this name is the initial value of the field. However, application servers use different naming conventions, and this name might not match the JNDI name used by your application server. The IVT program attempts to use some common variations of the default name but, if none of these variations are valid, the IVT program fails with a javax.naming.NameNotFoundException exception. If this happens, you must set this field to the JNDI name used by your application server, replacing the existing contents of the field. To help you work out the JNDI name used by your application server, the file ejb-jar.xml contains the following definition for the EJB:

```
<display-name>WMQ_TransactedIVT</display-name>
<enterprise-beans>
  <session id="WMQ_TransactedIVT">
    <ejb-name>WMQ_TransactedIVT</ejb-name>
    <home>ejbs.WMQ_TransactedIVTHome</home>
    <remote>ejbs.WMQ_TransactedIVT</remote>
    <ejb-class>ejbs.WMQ_TransactedIVTBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
  </session>
</enterprise-beans>
```

To run the test, click **Run IVT**. Figure 2 on page 56 shows the page that is displayed if the IVT is successful.


IBM WebSphere MQ J2EE Connector Architecture IVT

Running Installation Verification Test:

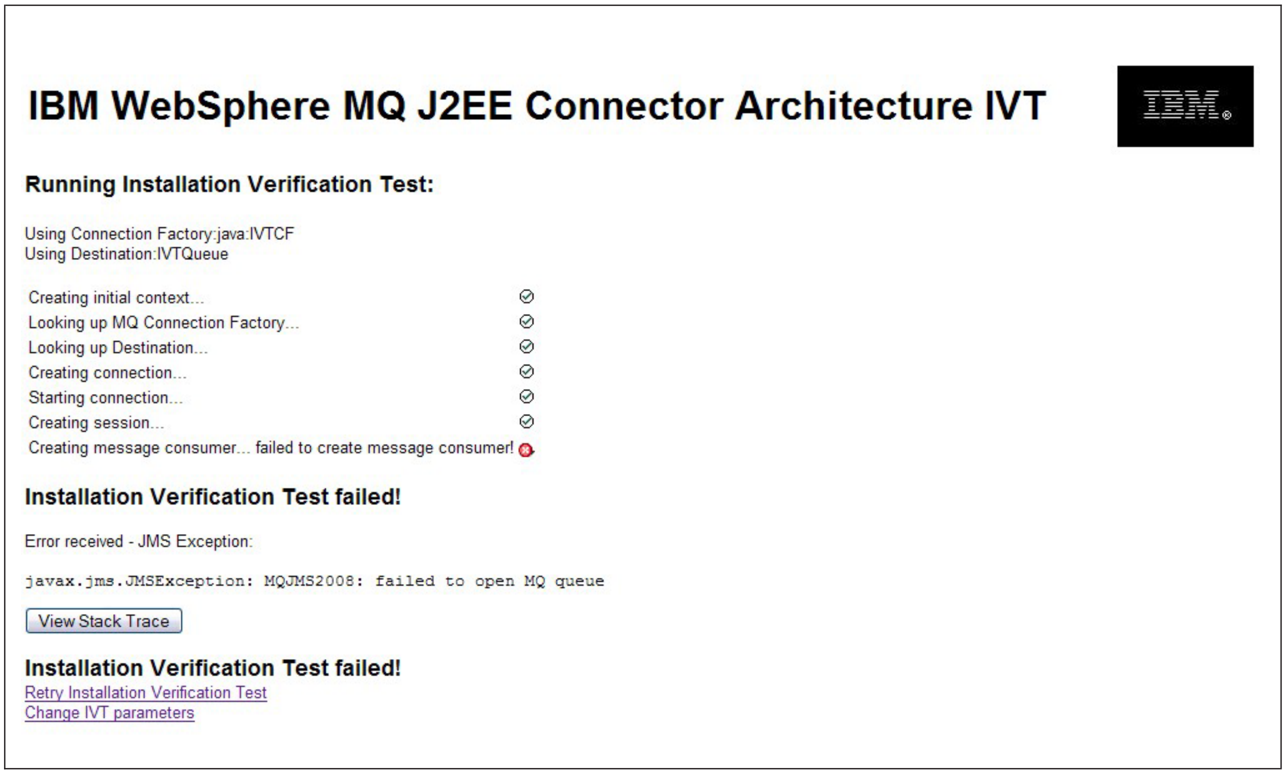
Using Connection Factory:IVTCF
Using Destination:IVTQueue

Creating initial context...	✓
Looking up MQ Connection Factory...(Altered JNDI name to java:IVTCF)	✓
Looking up Destination...	✓
Creating connection...	✓
Starting connection...	✓
Creating session...	✓
Creating message consumer...	✓
Creating message producer...	✓
Creating message...	✓
Sending message...	✓
Receiving message...	✓
Closing connection...	✓
Attempting transacted test... (Altered JNDI name to WMQ_TransactedIVT)	✓

Installation Verification Test completed successfully!

Figure 2. Page showing the results of a successful IVT

If the IVT fails, a page similar to that shown in Figure 3 on page 57 is displayed. To obtain further information about the cause of the failure, click **View Stack Trace**.



IBM WebSphere MQ J2EE Connector Architecture IVT

Running Installation Verification Test:

Using Connection Factory:java:IVTCF
Using Destination:IVTQueue

Creating initial context...	✓
Looking up MQ Connection Factory...	✓
Looking up Destination...	✓
Creating connection...	✓
Starting connection...	✓
Creating session...	✓
Creating message consumer... failed to create message consumer!	✗

Installation Verification Test failed!

Error received - JMS Exception:

```
javax.jms.JMSException: MQJMS2008: failed to open MQ queue
```

Installation Verification Test failed!

Retry Installation Verification Test
Change IVT parameters

Figure 3. Page showing the results of an IVT that failed

Scripts provided with WebSphere MQ classes for JMS

A number of scripts are provided to assist with common tasks that need to be performed when using WebSphere MQ classes for JMS.

Table 16 lists all the scripts and their uses. The scripts are in the `bin` subdirectory of the WebSphere MQ classes for JMS installation directory.

Table 16. Scripts provided with WebSphere MQ classes for JMS

Utility	Use
Cleanup[1]	This script is maintained for compatibility with previous releases but performs no function.
DefaultConfiguration	Runs the default configuration application on platforms other than Windows.
formatLog[1]	This script is maintained for compatibility with previous releases but performs no function.
IVTRun[1] IVTSetup[1] IVTTidy[1]	Used in the point-to-point installation verification test, as described in "The point-to-point installation verification test for WebSphere MQ classes for JMS" on page 46.
JMSAdmin[1]	Runs the WebSphere MQ JMS administration tool, as described in "Invoking the administration tool" on page 167.
JMSAdmin.config	The configuration file for the WebSphere MQ JMS administration tool, as described in "Configuration" on page 168.

Table 16. Scripts provided with WebSphere MQ classes for JMS (continued)

Utility	Use
PSIVTRun[1]	Runs the publish/subscribe installation verification test program, as described in "The publish/subscribe installation verification test for WebSphere MQ classes for JMS" on page 50.
PSReportDump.class	This class is maintained for compatibility with previous releases, but performs no function.
setjmsenv	Sets the environment variables for running a WebSphere MQ classes for JMS application in a 32-bit Java virtual machine (JVM) on a UNIX system, as described in "Environment variables used by WebSphere MQ classes for JMS" on page 9.
setjmsenv64	Sets the environment variables for running a WebSphere MQ classes for JMS application in a 64-bit JVM on a UNIX system, as described in "Environment variables used by WebSphere MQ classes for JMS" on page 9.
Note: 1. On Windows, the file name has the extension .bat .	

Support for OSGi

The Open Services Gateway Initiative (OSGi) provides a framework that supports the deployment of applications as bundles. Eight OSGi bundles are supplied as part of WebSphere MQ classes for JMS.

OSGi provides a general purpose, secure, and managed Java framework, which supports the deployment of applications that come in the form of bundles. OSGi-compliant devices can download and install bundles, and remove them when they are no longer required. The framework manages the installation and update of bundles in a dynamic and scalable fashion.

WebSphere MQ classes for JMS includes the following OSGi bundles. The bundles are in the java/lib/OSGi subdirectory of your WebSphere MQ installation, or the Java\lib\OSGI folder on Windows.

com.ibm.msg.client.osgi.jms_1.0.0.0.jar
The common layer of code in WebSphere MQ classes for JMS. For information about the layered architecture of WebSphere MQ classes for JMS, see "A layered architecture" on page 69.

com.ibm.msg.client.osgi.jms.prereq_1.0.0.0.jar
The prerequisite Java archive (JAR) files for the common layer.

com.ibm.msg.client.osgi.commonservices.j2se_1.0.0.0.jar
Common services for Java Platform, Standard Edition (Java SE) applications.

com.ibm.msg.client.osgi.nls_1.0.0.0.jar
Messages for the common layer.

com.ibm.msg.client.osgi.wmq_7.0.0.0.jar
The WebSphere MQ messaging provider in WebSphere MQ classes for JMS. For information about the layered architecture of WebSphere MQ classes for JMS, see "A layered architecture" on page 69.

com.ibm.msg.client.osgi.wmq.prereq_7.0.0.0.jar
: The prerequisite JAR files for the WebSphere MQ messaging provider.

com.ibm.msg.client.osgi.wmq.nls_7.0.0.0.jar
: Messages for the WebSphere MQ messaging provider.

com.ibm.mq.osgi.directip_7.0.0.0.jar
: The JAR files to allow the WebSphere MQ messaging provider to create a real-time connection to a broker.

These bundles have been written to the OSGi Release 4 specification. They do not work in an OSGi Release 3 environment.

You must set your system path or library path correctly so that the OSGi runtime environment can find any required DLL files or shared libraries.

If you use the OSGi bundles for WebSphere MQ classes for JMS, temporary topics do not work. In addition, channel exit classes written in Java are not supported because of an inherent problem in loading classes in a multiple class loader environment such as OSGi. A user bundle can be aware of the WebSphere MQ classes for JMS bundles, but the WebSphere MQ classes for JMS bundles are not aware of any user bundle. As a result, the class loader used in a WebSphere MQ classes for JMS bundle cannot load a channel exit class that is in a user bundle.

For more information about OSGi, see the OSGi Alliance Web site at http://www.osgi.org.

Solving problems

If a program does not complete successfully, run one of the installation verification programs, as described in "The point-to-point installation verification test for WebSphere MQ classes for JMS" on page 46 and "The publish/subscribe installation verification test for WebSphere MQ classes for JMS" on page 50, and follow the advice given in the diagnostic messages.

Tracing programs

The WebSphere MQ classes for JMS trace facility is provided to help IBM staff to diagnose customer problems. Various properties are provided to control its behavior.

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

Except where otherwise stated, all the properties described in the following information are set in the WebSphere MQ classes for JMS configuration file. For information about this file, see "The WebSphere MQ classes for JMS configuration file" on page 12.

If you are asked to provide trace output, turn tracing on by setting the property com.ibm.msg.client.commonservices.trace.status to `ON`. To turn tracing off, set the property com.ibm.msg.client.commonservices.trace.status to `OFF`.

Configure the trace output using the following properties:

com.ibm.msg.client.commonservices.trace.outputName
: The directory and file name to which trace output will be sent.

Defaults to a file named mqjms_*PID*.trc in the current working directory where *PID* is the current process ID. If a process ID is not available, a random number is generated and prefixed with the letter *f*. To include the process ID in a file name you specify, use the string %PID%. Any directory referenced must already exist, and you must have write permission for this directory. If you do not have write permission, the trace output is written to System.err.

com.ibm.msg.client.commonservices.trace.include
A list of packages and classes that will be traced, or the special values ALL or NONE.

Separate package or class names with a semicolon (;). Defaults to ALL and traces all packages and classes in WebSphere MQ classes for JMS.

Note that you can include a package but then exclude subpackages of that package. For example, if you include package ″a.b″ and exclude package ″a.b.x″, the trace includes everything in ″a.b.y″, ″a.b.z″ and so on, but not ″a.b.x″ or ″a.b.x.1″.

com.ibm.msg.client.commonservices.trace.exclude
A list of packages and classes that will not be traced, or the special values ALL or NONE.

Separate package or class names with a semicolon (;). Defaults to NONE and therefore excludes no packages and classes in WebSphere MQ classes for JMS from being traced.

Note that you can exclude a package but then include subpackages of that package. For example, if you exclude package ″a.b″ and include package ″a.b.x″, the trace includes everything in ″a.b.x″ and ″a.b.x.1″ and so on, but not ″a.b.y″ or ″a.b.z″.

Any package or class that is specified, at the same level, as both included and excluded is included.

com.ibm.msg.client.commonservices.trace.maxBytes
The maximum number of bytes that will be traced from any byte arrays.

If set to a positive integer, when a byte array is formatted to be written out to the trace file, it will be truncated at this number of bytes rather than the whole array being output. This allows for the amount of data being written to the trace file to be reduced, especially in cases where large byte arrays are being traced out, and therefore helps to reduce both the size of the resulting trace file and the performance impact of tracing the application.

A value of 0 for this property means that none of the contents of any byte arrays will be sent to the trace file.

The default value is -1, which indicates that the amount of data to be output is unlimited.

com.ibm.msg.client.commonservices.trace.limit
The maximum number of bytes to be written to a trace output file.

This property works in conjunction with the com.ibm.msg.client.commonservices.trace.count property. When the number of bytes of trace output that is approximately equal to the specified maximum have been written to a trace output file, the file is closed and a new trace output file is started.

A value of 0 means that a trace output file has zero length. The default value is -1, which means that the amount of data to be written to a trace output file is unlimited.

com.ibm.msg.client.commonservices.trace.count
The number of trace output files to cycle through.

When the current trace output file reaches the maximum size as specified by the com.ibm.msg.client.commonservices.trace.limit property, the file is closed and further trace output is written to the next trace output file in sequence. Each trace output file is distinguished by a numeric suffix appended to the file name. The current or most recent trace output file is mqjms.trc.0, the next most recent trace output file is mqjms.trc.1, and so on.

The default value is 1, which means that, when the current trace output file reaches the maximum size, the file is closed and deleted, and a new trace output file with the same name is started. Therefore, only one trace output file exists at a time.

com.ibm.msg.client.commonservices.trace.parameter
Controls whether method parameters and return values are included in the trace.

Defaults to TRUE. If set to FALSE then only method signatures are traced.

com.ibm.msg.client.commonservices.trace.startup
There is a an initialization phase of WebSphere MQ classes for JMS during which resources are allocated, which includes the initialization of the main trace facility.

If set to TRUE, startup trace is used, that is, trace information is produced immediately and therefore includes the setup of all components, including the trace facility itself. This information can be used to help diagnose configuration problems. Startup trace information is always written to System.err.

Defaults to FALSE.

Because this property is checked before initialization is complete, the property can be specified only on the command line as a Java system property, not within the WebSphere MQ classes for JMS configuration file.

com.ibm.msg.client.commonservices.trace.compress
Whether trace output is compressed.

The default is FALSE.

If set to TRUE, the trace output is compressed. The default file name will have a .trz extension. If compression is set to FALSE, the default value, the file will have a .trc extension to indicate it is uncompressed. However if the file name for the trace output has been specified in com.ibm.msg.client.commonservices.trace.outputName that name is used instead; no suffix will be applied to the file.

Compressed trace output is smaller than uncompressed and, because there is less I/O to be performed, it can be written out faster than uncompressed trace, meaning tracing will have less impact on the performance of WebSphere MQ classes for JMS.

If the com.ibm.msg.client.commonservices.trace.limit or com.ibm.msg.client.commonservices.trace.count properties are also set, this results in multiple compressed trace files being created in place of multiple flat files.

The nature of the trace compression algorithm means that if WebSphere MQ classes for JMS ends in an uncontrolled manner, the file might not be correctly completed and closed. For this reason, trace compression should be used only in cases where WebSphere MQ classes for JMS can close down in a controlled manner (that is, when the problems being investigated do not cause the JVM itself to stop unexpectedly). Do not use trace compression when diagnosing problems that can result in System.Halt() shutdowns or abnormal, uncontrolled JVM terminations.

com.ibm.msg.client.commonservices.trace.level
Specifies a filtering level for the trace. The defined trace levels are as follows:

TRACE_NONE	0
TRACE_EXCEPTION	1
TRACE_WARNING	3
TRACE_INFO	6
TRACE_ENTRYEXIT	8
TRACE_DATA	9
TRACE_ALL	Integer.MAX_VALUE

Each trace level includes all lower levels. For example, if trace level is set at TRACE_INFO, then any trace point with a defined level of TRACE_EXCEPTION, TRACE_WARNING, or TRACE_INFO appears in trace. All other trace points will be excluded.

To dynamically enable or disable trace from within an application, or to change the trace level, use the methods of the com.ibm.msg.client.services.Trace class.

setOn()
Turns the trace facility on.

setOff()
Turns the trace facility off.

setStatus(boolean traceOn)
Turns the trace facility on or off, depending on the value of *traceOn*.

isOn() Checks whether the trace facility is on.

setTraceLevel(int newTraceLevel)
Sets the tracing detail level.

getTraceLevel()
Returns the tracing detail level.

If a severe or unrecoverable error occurs, first-failure support technology (FFST™) information is recorded in a file with a name of the format JMSCC*xxxx*.FDC, where *xxxx* is an incrementing, four-digit count used to differentiate subsequent .FDC files. .FDC files are always written to a directory called FDC. If trace is active and the com.ibm.msg.client.commonservices.trace.outputName property is set, the FDC directory is created as a subdirectory of the directory to which the trace file is being written. If trace is not active or com.ibm.msg.client.commonservices.trace.outputName is not set, the FDC directory is created as a subdirectory of the current working directory.

The JSE common services uses java.util.logging as its trace and logging infrastructure. The root object of this infrastructure is the 'LogManager'. The log manager has a reset method, which closes all handlers and sets the log level to null - in effect turning off all the trace. If your application or application server calls java.util.logging.LogManager.getLogManager().reset(); it will close all trace, which might prevent you from diagnosing any problems. To avoid this, create a LogManager class with an overridden reset() class that does nothing, as in the following example:

```
package com.ibm.javaut.tests;

import java.util.logging.LogManager;

public class JmsLogManager extends LogManager {

 // final shutdown hook to ensure that the trace is finally shutdown
 // and that the lock file is cleaned-up
 public class ShutdownHook extends Thread{
  public void run(){
   doReset();
  }
 }

 public JmsLogManager(){
  // add shutdown hook to ensure final cleanup
  Runtime.getRuntime().addShutdownHook(new ShutdownHook());
 }

 public void reset() throws SecurityException {
  // does nothing

 }

 public void doReset(){
  super.reset();
 }

}
```

The shutdown hook is necessary to ensure that trace is properly shutdown on JVM termination. To use this log manager instead of the default one, add a system property to the JVM startup, as follows

```
java  -Djava.util.logging.manager=com.mycompany.logging.LogManager ...
```

Note: When trace is activated, it creates a file named mqjms.trc.lck. If you use a version of Java earlier than Java 5, this file is not removed when trace ends. This is caused by a defect in the Java class libraries. You can delete this file manually after the trace file has been closed.

Tracing using MQJMS_TRACE_LEVEL:

To maintain backwards compatibility, the trace parameters used by versions of WebSphere MQ classes for JMS earlier than Version 7.0 are still supported. However, these should be considered deprecated for any new application.

You can enable trace by setting the Java property MQJMS_TRACE_LEVEL to one of the following values:

on In versions earlier than Version 7.0, this value traces WebSphere MQ classes for JMS calls only

base In versions earlier than Version 7.0, this value traces both WebSphere MQ classes for JMS calls and the underlying WebSphere MQ classes for Java calls

In Version 7.0, both these values behave as if the com.ibm.msg.client.commonservices.trace.status property is set to ON.

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

Setting MQJMS_TRACE_DIR to /somepath/tracedir is equivalent to setting the com.ibm.msg.client.commonservices.trace.outputName property to /somepath/tracedir/mqjms_%PID%.trc.

Logging

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

You can redirect log output to a file by setting the property com.ibm.msg.client.commonservices.log.outputName. If the value of the property identifies a directory, log output is written to a file called mqjms.log in that directory. If the value of the property identifies a specific file, log output is written to that file.

You can set this property in the WebSphere MQ classes for JMS configuration file or as a system property on the **java** command. In the following example, the property is set as a system property and identifies a specific file:

```
java -Djava.library.path=library_path
     -Dcom.ibm.msg.client.commonservices.log.outputName=/mydir/mylog.txt
      MyAppClass
```

In the command, *library_path* is the path to the directory containing the WebSphere MQ classes for JMS libraries (see "The Java Native Interface (JNI) libraries required by WebSphere MQ classes for JMS applications" on page 11).

Problem determination for the WebSphere MQ resource adapter

When using the WebSphere MQ resource adapter, most errors cause exceptions to be thrown, and these exceptions are reported to the user in a manner that depends on the application server. The resource adapter makes extensive use of linked exceptions to report problems. Typically, the first exception in a chain is a high level description of the error, and subsequent exceptions in the chain provide the more detailed information that is required to diagnose the problem.

For example, if the IVT program fails to obtain a connection to a WebSphere MQ queue manager, the following exception might be thrown:

```
javax.jms.JMSException: MQJCA0001: An exception occurred in the JMS layer.
                                   See the linked exception for details.
```

Linked to this exception is a second exception:

```
javax.jms.JMSException: MQJMS2005: failed to create an MQQueueManager for
                                   'localhost:ExampleQM'
```

This exception is thrown by WebSphere MQ classes for JMS and has a further linked exception:

```
com.ibm.mq.MQException: MQJE001: An MQException occurred: Completion Code 2,
                                 Reason 2059
```

This final exception indicates the source of the problem. Reason code 2059 is MQRC_Q_MGR_NOT_AVAILABLE, which indicates that the queue manager specified in the definition of the ConnectionFactory object might not have been started.

If the information provided by exceptions is not sufficient to diagnose a problem, you might need to request a diagnostic trace. For information about how to enable diagnostic tracing, see "Configuration of the WebSphere MQ resource adapter" on page 17.

Configuration problems commonly occur in the following areas:

- "Problems in deploying the resource adapter"
- "Problems in deploying MDBs"
- "Problems in creating connections for outbound communication" on page 66

Problems in deploying the resource adapter

Failures in deploying the resource adapter are generally caused by not configuring JCA resources correctly. For example, a property of the ResourceAdapter object might not be specified correctly, or the deployment plan required by the application server might not be written correctly. Failures might also occur when the application server attempts to create objects from the definitions of JCA resources and bind the objects into the JNDI namespace, but certain properties are not specified correctly or the format of a resource definition is incorrect.

The resource adapter can also fail to deploy because it loaded incorrect versions of JCA or WebSphere MQ classes for JMS classes from JAR files in the class path. This kind of failure can commonly occur on a system where WebSphere MQ is already installed. On such a system, the application server might find existing copies of the WebSphere MQ classes for JMS JAR files and load classes from them in preference to the classes supplied in the WebSphere MQ resource adapter RAR file. If the extended transactional client JAR file, com.ibm.mqetclient.jar, cannot be loaded when the resource adapter is deployed, a warning is written to the diagnostic trace, if enabled, but this does not cause deployment to fail.

Problems in deploying MDBs

Failures might occur when the application server attempts to start message delivery to an MDB. This kind of failure is typically caused by an error in the definition of the associated ActivationSpec object, or because the resources referenced in the definition are not available. For example, the queue manager might not be running, or a specified queue might not exist.

An ActivationSpec object attempts to validate its properties when the MDB is deployed, and deployment fails if the ActivationSpec object has any properties that are mutually exclusive or does not have all the required properties. However, not all problems associated with the properties of the ActivationSpec object can be detected at this time.

Deployment might also fail if an MDB is transacted and the connection is in client mode, but distributed transactions are not available because the extended transactional client JAR file, com.ibm.mqetclient.jar, is not in the class path.

Failures to start message delivery are reported to the user in a manner that depends on the application server. Typically, these failures are reported in the logs and diagnostic trace of the application server. If enabled, the diagnostic trace of the WebSphere MQ resource adapter also records these failures.

Problems in creating connections for outbound communication

Failures in outbound communication commonly occur when an application attempts to look up and use a ConnectionFactory object in a JNDI namespace. A JNDI exception is thrown if the ConnectionFactory object cannot be found in the namespace. A ConnectionFactory object might not be found for the following reasons:

- The application specified an incorrect name for the ConnectionFactory object.
- The application server was not able to create the ConnectionFactory object and bind it into the namespace. In this case, the startup logs of the application server usually contain information about the failure.

If the application successfully retrieves the ConnectionFactory object from the JNDI namespace, an exception might still be thrown when the application calls the ConnectionFactory.createConnection() method. An exception in this context indicates that it is not possible to create a connection to a WebSphere MQ queue manager. Here are some common reasons why an exception might be thrown:

- The queue manager is not available, or cannot be found using the properties of the ConnectionFactory object. For example, the queue manager is not running, or the specified host name, IP address, or port number of the queue manager is incorrect.
- The user is not authorized to connect to the queue manager. For a client connection, if the createConnection() call does not specify a user name, and the application server supplies no user identity information, the JVM process ID is passed to the queue manager as the user name. For the connection to succeed, this process ID must be a valid user name in the system on which the queue manager is running.
- The application is a transacted EJB and therefore the connection must be transacted, but distributed transactions are not available because the extended transactional client JAR file, com.ibm.mqetclient.jar, is not in the class path.
- The ConnectionFactory object has a property called ccdtURL and a property called channel. These properties are mutually exclusive.
- On an SSL connection, the SSL related properties, or the SSL related attributes in the server connection channel definition, have not been specified correctly.
- The sslFipsRequired property has different values for different JCA resources. For more information about this limitation, see “Limitations of the WebSphere MQ resource adapter” on page 42.

Introduction to WebSphere MQ classes for JMS for programmers

This topic introduces WebSphere MQ classes for JMS from the point of view of the programmer. It also summarizes the main enhancements that are contained in the latest release of WebSphere MQ classes for JMS.

Introduction to WebSphere MQ classes for JMS

WebSphere MQ classes for JMS is the JMS provider that is supplied with WebSphere MQ. As well as implementing the interfaces defined in the javax.jms package, WebSphere MQ classes for JMS provides two sets of extensions to the JMS API. Both Java Platform, Standard Edition (Java SE) and Java Platform, Enterprise Edition (Java EE) applications can use WebSphere MQ classes for JMS.

The JMS specification defines a set of interfaces that applications can use to perform messaging operations. The latest version of the specification is Version 1.1. The javax.jms package specifies the details of the JMS interfaces, and a JMS provider implements these interfaces for a specific messaging product. WebSphere MQ classes for JMS is a JMS provider that implements the JMS interfaces for WebSphere MQ.

The flow of logic within a JMS application starts with ConnectionFactory and Destination objects. The application uses a ConnectionFactory object to create a Connection object, which represents the application's active connection to a messaging server. The application uses the Connection object to create a Session object, which is a single threaded context for producing and consuming messages. The application can then use the Session object and a Destination object to create a MessageProducer object, which the application uses to send messages to the specified destination. The destination is either a queue or a topic in the messaging system and is encapsulated by the Destination object. The application can also use the Session object and a Destination object to create a MessageConsumer object, which the application uses to receive messages that have been sent to the specified destination.

The JMS specification expects ConnectionFactory and Destination objects to be administered objects. An administrator creates and maintains administered objects in a central repository, and a JMS application retrieves these objects using the Java Naming and Directory Interface (JNDI). The repository of administered objects can range from a simple file to a Lightweight Directory Access Protocol (LDAP) directory.

WebSphere MQ classes for JMS supports the use of administered objects. An application can use all the features of WebSphere MQ that are exposed through WebSphere MQ classes for JMS without having any WebSphere MQ specific information hard coded into the application itself. This arrangement provides the application with a degree of independence from the underlying WebSphere MQ configuration. To achieve this, the application can use JNDI to retrieve connection factories and destinations that are stored as administered objects, and use only the interfaces defined in the javax.jms package to perform messaging operations. An administrator can use the WebSphere MQ JMS administration tool or WebSphere MQ Explorer to create and maintain administered objects in a central repository. An application server, however, typically provides its own repository for administered objects and its own tools for creating and maintaining the objects. A Java EE application can therefore use JNDI to retrieve administered objects either from the applications server's own repository or from a central repository.

WebSphere MQ classes for JMS also provides extensions to the JMS API. Previous releases of WebSphere MQ classes for JMS contain extensions that are implemented in MQConnectionFactory, MQQueue, and MQTopic objects. These objects have properties and methods that are specific to WebSphere MQ. The objects can be administered objects, or an application can create the objects dynamically at run time. This release of WebSphere MQ classes for JMS maintains these extensions, and you can continue to use, without change, any applications that use these extensions. These extensions are known as the *WebSphere MQ JMS extensions*. Note that, in this set of documentation, objects that are created dynamically by an application at run time are *not* considered to be administered objects.

In addition to the WebSphere MQ JMS extensions, this release of WebSphere MQ classes for JMS provides a more generic set of extensions to the JMS API. These extensions are known as the *IBM JMS extensions*, and have the following broad objectives:

- To provide a greater level of consistency across IBM JMS providers
- To make it easier to write a bridge application between two IBM messaging systems
- To make it easier to port an application from one IBM JMS provider to another

The main focus of these extensions concerns creating and configuring connection factories and destinations dynamically at run time, but the extensions also provide function that is not directly related to messaging, such as function for problem determination. The IBM JMS extensions are also implemented in Lotus Expeditor micro broker.

Both Java SE and Java EE applications can use WebSphere MQ classes for JMS. On the Java EE platform, WebSphere MQ classes for JMS supports two types of communication between a component of an application and a WebSphere MQ queue manager:

Outbound communication
Using the JMS API directly, an application component creates a connection to a queue manager, and then sends and receives messages.

For example, the application component can be an application client, a servlet, a JavaServer page (JSP), an enterprise Java bean (EJB), or a message driven bean (MDB). In this type of communication, the application server container provides only low level functions in support of messaging operations, such as connection pooling and thread management.

Inbound communication
A message arriving at a destination is delivered to an MDB, which then processes the message.

Java EE applications use MDBs to process messages asynchronously. An MDB acts as a JMS message listener and is implemented by an onMessage() method, which defines how a message is processed. An MDB is deployed in the EJB container of an application server. The precise way in which an MDB is configured depends on which application server you are using, but the configuration information must specify which queue manager to connect to, how to connect to the queue manager, which destination to monitor for messages, and the transactional behavior of the MDB. This information is then used by the EJB container. When a message satisfying the MDB's selection criteria arrives at the specified destination, the EJB container uses WebSphere MQ classes for JMS to retrieve the message from the queue manager, and then delivers the message to the MDB by calling its onMessage() method.

What is new in WebSphere MQ Version 7.0?

WebSphere MQ classes for JMS, as supplied in WebSphere MQ Version 7.0, contains a number of enhancements compared to previous releases. Some of these enhancements are as a result of changes to the implementation of WebSphere MQ classes for JMS, and some are as a result of WebSphere MQ classes for JMS exploiting changes to the underlying WebSphere MQ function.

The following sections summarize the key enhancements.

A layered architecture

In previous releases of WebSphere MQ, the implementation of WebSphere MQ classes for JMS has been entirely specific to WebSphere MQ. Other IBM products that provide messaging systems have also included JMS providers, but these JMS providers have very little or nothing in common with the implementation of WebSphere MQ classes for JMS.

In WebSphere MQ V7.0, WebSphere MQ classes for JMS now has a layered architecture. The top layer of code is a common layer that can be used by any IBM JMS provider. Lotus Expeditor micro broker is such JMS provider, and uses this common layer. When an application calls a JMS method, any processing of the call that is not specific to a messaging system is performed by the common layer, which also provides a consistent response to the call. Any processing of the call that is specific to a messaging system is delegated to a lower layer. Figure 4 shows the layered architecture.

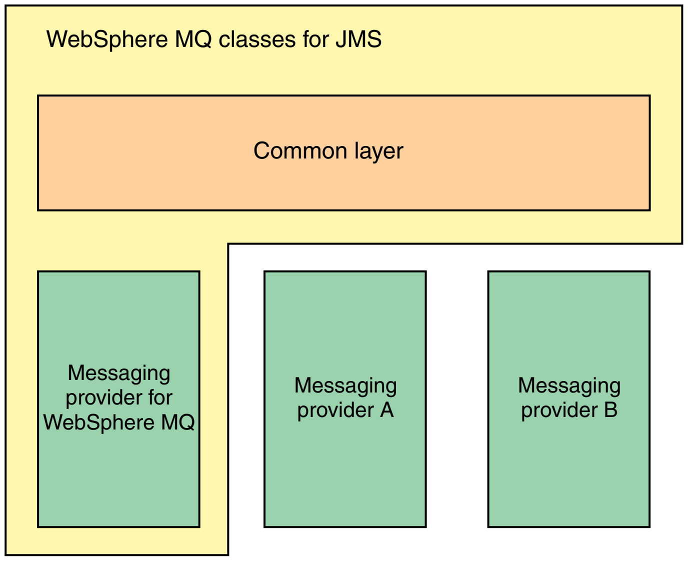

WebSphere MQ classes for JMS
Common layer
Messaging provider for WebSphere MQ
Messaging provider A
Messaging provider B

Figure 4. The layered architecture for IBM JMS providers

Moving to a layered architecture has following objectives:

- To improve the consistency of behavior of the various IBM JMS providers
- To make it easier to write an bridge application between two IBM messaging systems
- To make it easier to port an application from one IBM JMS provider to another

This implementation of WebSphere MQ classes for JMS also introduces a new set of extensions to the JMS API. These extensions are known as the *IBM JMS extensions*. The main focus of these extensions concerns creating and configuring connection factories and destinations dynamically at run time. The IBM JMS extensions are also implemented in Lotus Expeditor micro broker.

An application using the IBM JMS extensions starts by creating a JmsFactoryFactory object, specifying as a parameter a constant that identifies the chosen messaging system. The application uses the JmsFactoryFactory object to create connection factories and destinations that have the correct specialized classes for the chosen messaging system.

The application can then configure the connection factories and destinations by setting their properties. The IBM JMS extensions provide a set of methods to set properties. These methods are independent of any messaging system. Each data type has its own set method, and each property is identified by a name, which is defined as a static final member of the WMQConstants class. When an application calls one of these methods, one of the parameters on the call is the name of the property, and the other parameter is the value of the property.

For example, if WebSphere MQ is the messaging system, one of the properties of a connection factory is the name of the queue manager to connect to. Using the IBM JMS extensions, an application sets the name of the queue manager to JUPITER by calling the following method:

```
JmsConnectionFactory myCF;
...
myCF.setStringProperty(WMQConstants.WMQ_QUEUE_MANAGER, "JUPITER");
```

By contrast, an application can perform the same function by calling the following method:

```
MQConnectionFactory myCF;
...
myCF.setQueueManager("JUPITER");
```

This method is a WebSphere MQ JMS extension and is specific to WebSphere MQ as the messaging system. The use of this method therefore makes the application potentially less easy to port to another IBM JMS provider.

The relationship between WebSphere MQ classes for JMS and WebSphere MQ classes for Java

In previous releases of WebSphere MQ, WebSphere MQ classes for JMS was implemented almost entirely as a layer of code on top of WebSphere MQ classes for Java. This arrangement has caused some confusion among application developers because setting fields or calling methods in the MQEnvironment class can cause unwanted and unexpected effects on the runtime behavior of code that is written using WebSphere MQ classes for JMS. In addition, the implementation of WebSphere MQ classes for JMS had some constraints in areas where the JMS API is not a natural fit on top of WebSphere MQ classes for Java, and these constraints have led to some issues regarding runtime performance.

In WebSphere MQ V7.0, the implementation of WebSphere MQ classes for JMS is no longer dependent on WebSphere MQ classes for Java. WebSphere MQ classes for Java and WebSphere MQ classes for JMS are now peers that use a common Java interface to the MQI. This arrangement allows more scope for optimizing performance, and means that setting fields or calling methods in the MQEnvironment class has no effect on the runtime behavior of code that is written using WebSphere MQ classes for JMS. Figure 5 on page 71 shows the relationship between WebSphere MQ classes for JMS and WebSphere MQ classes for Java in previous releases of WebSphere MQ and in WebSphere MQ V7.0.

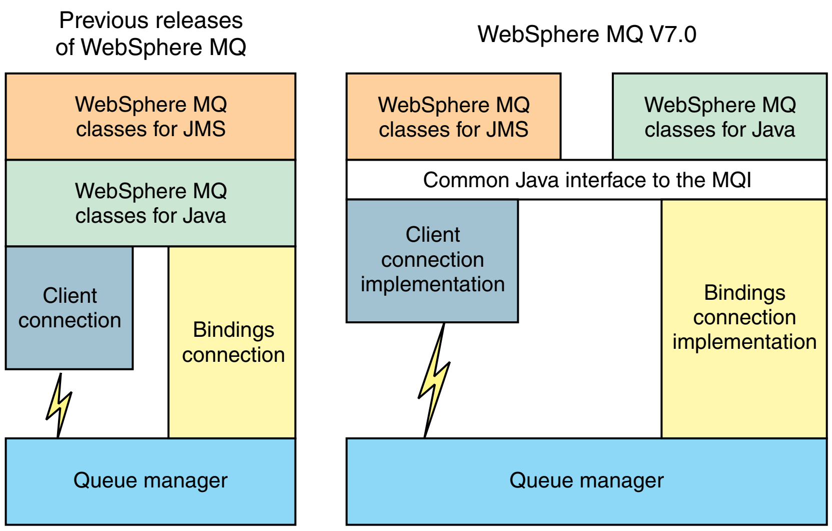

Previous releases of WebSphere MQ
WebSphere MQ V7.0
WebSphere MQ classes for JMS
WebSphere MQ classes for Java
Client connection
Bindings connection
Queue manager
WebSphere MQ classes for JMS
WebSphere MQ classes for Java
Common Java interface to the MQI
Client connection implementation
Bindings connection implementation
Queue manager

Figure 5. The relationship between WebSphere MQ classes for JMS and WebSphere MQ classes for Java

In order to maintain compatibility with earlier releases, channel exit classes written in Java can still use the WebSphere MQ classes for Java interfaces, even if the channel exit classes are called from WebSphere MQ classes for JMS. However, using the WebSphere MQ classes for Java interfaces means that your applications are still dependent on the WebSphere MQ classes for Java JAR file, com.ibm.mq.jar. If you do not want com.ibm.mq.jar in your class path, you can use the new set of interfaces in the com.ibm.mq.exits package instead.

You can now create and configure JMS administered objects with the WebSphere MQ Explorer.

Publish/subscribe messaging

WebSphere MQ V7.0 contains embedded publish/subscribe function. This function replaces WebSphere MQ Publish/Subscribe, which was supplied with WebSphere MQ V6.0.

WebSphere MQ classes for JMS applications can use the embedded publish/subscribe function, and can use it instead of using WebSphere Event Broker or WebSphere Message Broker for publish/subscribe messaging with WebSphere MQ as the transport. Configuring WebSphere MQ classes for JMS to use the new function is simpler than configuring WebSphere MQ classes for JMS to use WebSphere MQ Publish/Subscribe, WebSphere Event Broker, or WebSphere Message Broker. Administrators and application developers no longer need to manage publication queues, subscriber queues, subscription stores, and subscriber cleanup. In addition, ConnectionFactory and Topic objects have a smaller number of properties.

The embedded publish/subscribe function also provides some additional features such as retained publications and a choice of two wildcard schemes for specifying a range of topics to which an application wishes to subscribe.

An application can still use a real-time connection to a broker of WebSphere Event Broker or WebSphere Message Broker for publish/subscribe messaging. This support is unchanged.

Applications using WebSphere MQ Publish/Subscribe can use the embedded publish/subscribe function without change when the queue manager to which they are connected is upgraded. Properties that are set by an application, but are not required by the embedded publish/subscribe function, are ignored.

WebSphere MQ messaging provider

The WebSphere MQ messaging provider has two modes of operation:

- *WebSphere MQ messaging provider normal mode*
- *WebSphere MQ messaging provider migration mode*

The WebSphere MQ messaging provider normal mode uses all the features of the WebSphere MQ Version 7.0 queue managers to implement JMS. This mode is used only to connect to a WebSphere MQ queue manager and can connect to WebSphere MQ Version 7.0 queue managers in either client or bindings mode. This mode is optimized to use the new WebSphere MQ Version 7.0 function.

The WebSphere MQ messaging provider migration mode is based on WebSphere MQ Version 6.0 function and uses only features that were available in the WebSphere MQ Version 6.0 queue manager to implement JMS. You can connect to a WebSphere MQ Version 7.0 queue manager using WebSphere MQ messaging provider migration mode but you cannot use any of the Version 7.0 optimizations. This mode allows connections to either of the following queue manager versions:

1. WebSphere MQ Version 7.0 queue manager in bindings or client mode, but this mode uses only those features that were available to a WebSphere MQ Version 6.0 queue manager
2. WebSphere MQ Version 6.0 or earlier queue manager in client mode

If you want to connect to WebSphere Event Broker or WebSphere Message Broker using either WebSphere MQ Enterprise Transport, use the WebSphere MQ messaging provider migration mode. If you use WebSphere MQ Real-Time Transport, the WebSphere MQ messaging provider migration mode is automatically selected, because you have explicitly selected properties in the connection factory object. Connection to WebSphere Event Broker or WebSphere Message Broker using the WebSphere MQ Enterprise Transport follows the general rules for mode selection described in Rules for selecting the WebSphere MQ messaging provider mode .

Asynchronous message consumption

WebSphere MQ V7.0 supports asynchronous message consumption. An application can register a callback function for a destination. When a suitable message is sent to the destination, WebSphere MQ calls the function and passes the message as a parameter. The function then processes the message asynchronously. In previous releases of WebSphere MQ, this feature was available only when using WebSphere MQ classes for JMS.

WebSphere MQ classes for JMS has been changed to exploit this new feature in WebSphere MQ V7.0. The implementation of JMS message listeners is now a more natural fit with WebSphere MQ, and WebSphere MQ classes for JMS no longer has to poll a destination to check whether a suitable message has been sent to the destination. The performance of JMS message listeners is improved as a result, particularly when an application uses multiple message listeners in a session to monitor multiple destinations. Message throughput is increased, and the time taken to deliver a message to a message listener after it has arrived at a destination is reduced.

Message driven beans (MDBs) have similar performance improvements. In addition, because of another enhancement to WebSphere MQ function, multiple MDBs that are consuming messages from the same destination now experience reduced contention on the messages.

Message selection

As a result, message throughput is increased for applications that consume messages using message selection. The performance improvement is greater for an application that connects in client mode because only those messages that satisfy the selection criteria are transported over the network, and WebSphere MQ classes for JMS sees only those messages that it delivers to the application.

Sharing a communications connection

In previous releases of WebSphere MQ, if a WebSphere MQ client application connected to a queue manager more than once using the same MQI channel, each instance of the MQI channel required a separate TCP connection. In WebSphere MQ V7.0, each connection to the queue manager using the same MQI channel can share a single TCP connection. This arrangement means that fewer network resources are required and the total time taken to create multiple connections to the queue manager is reduced, particularly when using SSL because the SSL handshake takes place only once at the start of the TCP connection.

WebSphere MQ classes for JMS exploits this enhancement. For an application that connects to a queue manager in client mode, WebSphere MQ classes for JMS might create more than one connection to a queue manager using the MQI channel whose name is specified as a property of the ConnectionFactory object. Each of these connections to the queue manager can now share a single TCP connection.

Read ahead on client connections

If an application uses a client connection to consume nonpersistent messages from a destination, the destination can be configured so that WebSphere MQ classes for JMS uses a buffer to store the messages of interest before delivering them to the application. This optimization is called *read ahead* and can be used by applications that consume messages synchronously by calling the receive() method, and by message listeners and MDBs, which consume messages asynchronously. Read ahead is particularly effective for destinations with a large number of messages that need to be consumed rapidly.

Read ahead does not apply to persistent messages because, if persistent messages were read into a buffer, the queue manager would no longer be able to recover the messages following a failure. However, an application that consumes messages from a destination with a mixture of persistent and nonpersistent messages can still use read ahead. The order of the messages is preserved, but the runtime benefits of read ahead apply only to the nonpersistent messages.

When deciding whether to use read ahead, consider the following points:

- If an application is consuming messages from a destination that is configured for read ahead, and the application ends for any reason, any nonpersistent messages that are currently stored in the buffer are discarded.

- If all the following conditions are true, messages sent to a queue in a session might not be received in the order in which they were sent:
  - An application uses two message consumers in the same session to consume the messages from the queue.
  - Each message consumer uses a different Destination object for the queue.
  - Any or both of the Destination objects are configured for read ahead.

Sending messages

When an application sends messages to a destination, the destination can be configured so that, when the application calls send(), WebSphere MQ classes for JMS forwards the message to the queue manager and returns control back to the application without determining whether the queue manager has received the message safely. WebSphere MQ classes for JMS can work in this way only for nonpersistent messages and for persistent messages sent in a transacted session.

For persistent messages sent in a transacted session, the application ultimately determines whether the queue manager has received the messages safely when it calls commit(). For any messages sent in a session that is not transacted, the SENDCHECKCOUNT property of the ConnectionFactory object specifies how many messages are to be sent before WebSphere MQ classes for JMS checks that the queue manager has received the messages safely.

This optimization is of most benefit to an application that connects to a queue manager in client mode and needs to send a sequence of messages in rapid succession, but does not require immediate feedback from the queue manager for each message sent.

Channel exits

When called from WebSphere MQ classes for JMS, channel exit programs written in C or C++ now behave in the same way as when they are called from a Websphere MQ client. The performance of channel exit classes written in Java has been improved, and you can now write channel exit classes using a new set of interfaces in the com.ibm.mq.exits package instead of using the interfaces in WebSphere MQ classes for Java.

Message properties

A JMS message consists of a set of header fields, a set of properties, and a body that contains the application data. As a minimum, a WebSphere MQ message consists of a message descriptor and the application data.

When a WebSphere MQ classes for JMS application sends a JMS message, WebSphere MQ classes for JMS maps the JMS message into a WebSphere MQ message. Some of the JMS header fields and properties are mapped into fields in the message descriptor, and some are mapped into fields in an additional WebSphere MQ header called an MQRFH2 header. When a WebSphere MQ classes for JMS application receives a JMS message, WebSphere MQ classes for JMS performs the reverse mapping.

An application that is using the MQI to receive messages from a WebSphere MQ classes for JMS application must therefore be able to handle an MQRFH2 header. If the application cannot handle an MQRFH2 header, the TARGCLIENT property of the Destination object can be set to tell WebSphere MQ classes for JMS not to include an MQRFH2 header in the WebSphere MQ messages. However, by excluding the MQRFH2 header, the information held in some of the JMS header fields and properties is lost.

Similarly, an application that is using the MQI to send messages to a WebSphere MQ classes for JMS application must include an MQRFH2 header in each message. If an MQRFH2 header is not included, WebSphere MQ classes for JMS can set only those JMS header fields and properties that can be derived from the fields in a message descriptor.

WebSphere MQ V7.0 provides some additional support for applications that use the MQI to receive messages from, and send messages to, WebSphere MQ classes for JMS applications.

When an application calls MQGET to receive a message from a WebSphere MQ classes for JMS application, the application can choose to receive the message in one of the following ways:

1. The message is delivered with a message descriptor, an MQRFH2 header that contains data derived from JMS header fields and properties, and the application data.
2. The message is delivered with a message descriptor, the application data, and a set of message properties.

In option 2, each message property represents a JMS header field or property that was originally mapped by WebSphere MQ classes for JMS into a field in an MQRFH2 header. After the MQGET call, the application can use the MQINQMP call to get the values of the message properties. Using option 2 instead of option 1 to receive a message simplifies the application logic in the following ways:

- The application does not have to parse the variable portion of the MQRFH2 header, which contains the JMS header field and property data encoded in an XML-like format.
- The application does not have to convert the character data in the variable portion of the MQRFH2 header.

Correspondingly, before an application calls MQPUT to send a message to a WebSphere MQ classes for JMS application, the application can use the MQSETMP call to set the values of message properties instead of constructing an MQRFH2 header.

Serviceability

WebSphere MQ classes for JMS contains a number of improvements related to serviceability:

- Tracing.

  WebSphere MQ classes for JMS contains a class that an application can use to control tracing. An application can start and stop tracing, specify the required level of detail in a trace, and customize trace output in various ways..
- Logging.

  WebSphere MQ classes for JMS maintains a log file, which contains messages about errors that you need to correct. The messages are written in plain text. WebSphere MQ classes for JMS contains a class that an application can use to specify the location of the log file and its maximum size.
- First Failure Support Technology™ (FFST).

If a serious failure occurs, WebSphere MQ classes for JMS generates an FFST report in an FDC file. The FFST report contains information that IBM Service can use to diagnose the problem more quickly.

- Version information.

  WebSphere MQ classes for JMS contains a class that an application can use to query the version of WebSphere MQ classes for JMS.
- Exception messages.

  Exception messages have been enhanced to provide more information about the causes of errors and the actions required to correct errors.
- Application servers.

  The integration of the serviceability features of WebSphere MQ classes for JMS with those of WebSphere Application Server has been improved.

MQC is replaced by MQConstants

A new package, com.ibm.mq.constants, is supplied with WebSphere MQ Version 7.0. This package contains the class MQConstants, which implements a number of interfaces. MQConstants contains definitions of all the constants that were in the MQC interface and a number of new constants. The interfaces in this package closely follow the names of the constants header files used in Websphere MQ.

For example, the interface CMQC contains a constant MQOO_INPUT_SHARED; this corresponds to the header file cmqc.h and the constant MQOO_INPUT_SHARED.

com.ibm.mq.constants can be used with both WebSphere MQ classes for Java and WebSphere MQ classes for JMS.

MQC is still present, and has the constants it previously had; however, for any new applications, you should use the com.ibm.mq.constants package.

Writing WebSphere MQ classes for JMS applications

After a brief introduction to the JMS model, this topic provides detailed guidance on how to write WebSphere MQ classes for JMS applications.

The JMS model

The JMS model defines a set of interfaces that Java applications can use to perform messaging operations. WebSphere MQ classes for JMS, as a JMS provider, defines how JMS objects are related to WebSphere MQ concepts. The JMS specification expects certain JMS objects to be administered objects.

The JMS specification and the javax.jms package define a set of interfaces that Java applications can use to perform messaging operations. The following list summarizes the main JMS interfaces:

Destination
A destination is where an application sends messages, or it is a source from which an application receives messages, or both.

ConnectionFactory
A ConnectionFactory object encapsulates a set of configuration properties for a connection. An application uses a connection factory to create a connection.

Connection
A Connection object encapsulates an application's active connection to a messaging server. An application uses a connection to create sessions.

Session
A session is a single threaded context for sending and receiving messages. An application uses a session to create messages, message producers, and message consumers. A session is either transacted or not transacted.

Message
A Message object encapsulates a message that an application sends or receives.

MessageProducer
An application uses a message producer to send messages to a destination.

MessageConsumer
An application uses a message consumer to receive messages sent to a destination.

Figure 6 shows these objects and their relationships.

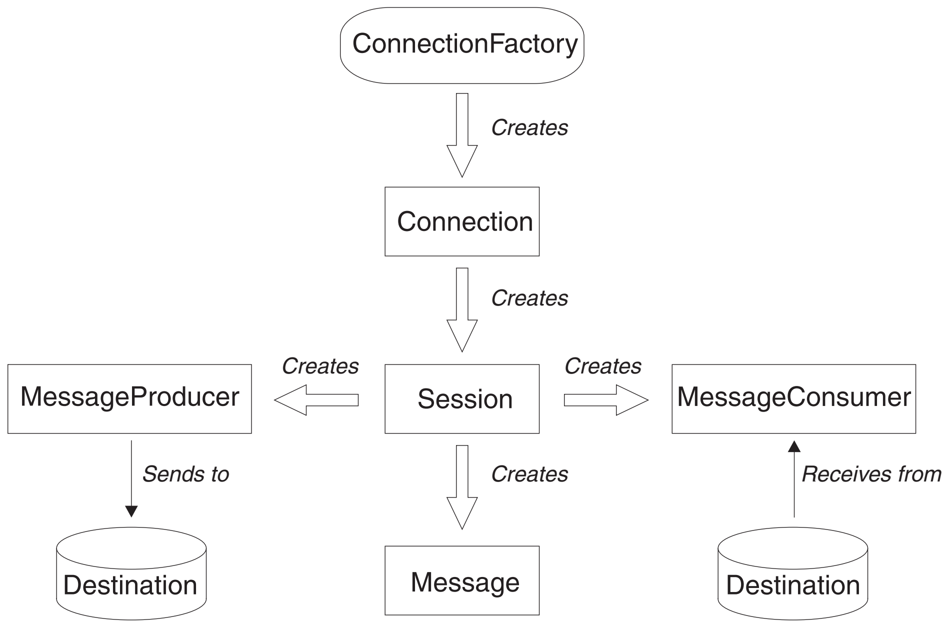


Figure 6. JMS objects and their relationships

A Destination, ConnectionFactory, or Connection object can be used concurrently by different threads of a multithreaded application, but a Session, MessageProducer, or MessageConsumer object cannot be used concurrently by different threads. The simplest way of ensuring that a Session, MessageProducer, or MessageConsumer object is not used concurrently is to create a separate Session object for each thread.

JMS support two styles of messaging:
- Point-to-point messaging
- Publish/subscribe messaging

These styles of messaging are also referred to as *messaging domains*, and you can combine both styles of messaging in an application. In the point-to-point domain, a destination is a queue and, in the publish/subscribe domain, a destination is a topic.

With versions of JMS before JMS 1.1, programming for the point-to-point domain uses one set of interfaces and methods, and programming for the publish/subscribe domain uses another set. The two sets are similar, but separate. With JMS 1.1, you can use a common set of interfaces and methods that support both messaging domains. The common interfaces provide a domain independent view of each messaging domain. Table 17 lists the JMS domain independent interfaces and their corresponding domain specific interfaces.

Table 17. The JMS domain independent and domain specific interfaces

Domain independent interfaces	**Domain specific interfaces for the point-to-point domain**	**Domain specific interfaces for the publish/subscribe domain**
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver QueueBrowser	TopicSubscriber

JMS 1.1 retains all the domain specific interfaces, and so existing applications can still use these interfaces. For new applications, however, consider using the domain independent interfaces.

In WebSphere MQ classes for JMS, JMS objects are related to WebSphere MQ concepts in the following ways:

- A Connection object has properties that are derived from the properties of the connection factory that was used to create the connection. These properties control how an application connects to a queue manager. Examples of these properties are the name of the queue manager and, for an application that connects to the queue manager in client mode, the host name or IP address of the system on which the queue manager is running.
- A Session object encapsulates a WebSphere MQ connection handle, which therefore defines the transactional scope of the session.
- A MessageProducer object and a MessageConsumer object each encapsulates a WebSphere MQ object handle.

When using WebSphere MQ classes for JMS, all the normal rules of WebSphere MQ apply. Note, in particular, that an application can send a message to a remote queue but it can receive a message only from a queue that is owned by the queue manager to which the application is connected.

The JMS specification expects ConnectionFactory and Destination objects to be administered objects. An administrator creates and maintains administered objects in a central repository, and a JMS application retrieves these objects using the Java Naming and Directory Interface (JNDI).

In WebSphere MQ classes for JMS, the implementation of the Destination interface is an abstract superclass of Queue and Topic, and so an instance of Destination is either a Queue object or a Topic object. The domain independent interfaces treat a queue or a topic as a destination. The messaging domain for a MessageProducer or MessageConsumer object is determined by whether the destination is a queue or a topic.

In WebSphere MQ classes for JMS therefore, objects of the following types can be administered objects:

- ConnectionFactory
- QueueConnectionFactory
- TopicConnectionFactory
- Queue
- Topic
- XAConnectionFactory
- XAQueueConnectionFactory
- XATopicConnectionFactory

JMS messages

JMS messages are composed of the following parts:

Header
All messages support the same set of header fields. Header fields contain values that are used by both clients and providers to identify and route messages.

Properties
Each message contains a built-in facility to support application-defined property values. Properties provide an efficient mechanism to filter application-defined messages.

Body JMS defines several types of message body that cover the majority of messaging styles currently in use.

JMS defines five types of message body:

Stream
A stream of Java primitive values. It is filled and read sequentially.

Map A set of name-value pairs, where names are strings and values are Java primitive types. The entries can be accessed sequentially or randomly by name. The order of the entries is undefined.

Text A message containing a java.lang.String.

Object
A message that contains a serializable Java object

Bytes A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.

The JMSCorrelationID header field is used to link one message with another. It typically links a reply message with its requesting message. JMSCorrelationID can hold a provider-specific message ID, an application-specific String, or a provider-native byte[] value.

Message selectors

A message contains a built-in facility to support application-defined property values. In effect, this provides a mechanism to add application-specific header fields to a message. Properties allow an application, using message selectors, to have a JMS provider select or filter messages on its behalf, using application-specific criteria. Application-defined properties must obey the following rules:

- Property names must obey the rules for a message selector identifier.
- Property values can be boolean, byte, short, int, long, float, double, and String.
- The JMSX and JMS_ name prefixes are reserved.

Property values are set before sending a message. When a client receives a message, the message properties are read-only. If a client attempts to set properties at this point, a MessageNotWriteableException is thrown. If clearProperties is called, the properties can now be both read from, and written to.

A property value might duplicate a value in a message's body. JMS does not define a policy for what should or should not be made into a property. However, application developers must be aware that JMS providers probably handle data in a message's body more efficiently than data in a message's properties. For best performance, applications must use message properties only when they need to customize a message's header. The primary reason for doing this is to support customized message selection.

A JMS message selector allows a client to specify the messages that it is interested in by using the message header. Only messages whose headers match the selector are delivered.

Message selectors cannot refer to message body values.

A message selector matches a message when the selector evaluates to true when the message's header field and property values are substituted for their corresponding identifiers in the selector.

A message selector is a String, whose syntax is based on a subset of the SQL92 conditional expression syntax. The order in which a message selector is evaluated is from left to right within a precedence level. You can use parentheses to change this order. Predefined selector literals and operator names are written here in upper case; however, they are not case-sensitive.

A selector can contain:

- Literals
  - A string literal is enclosed in single quotes. A doubled single quote represents a single quote. Examples are 'literal' and 'literal''s'. Like Java string literals, these use the Unicode character encoding.
  - An exact numeric literal is a numeric value without a decimal point, such as 57, -957, and +62. Numbers in the range of Java long are supported.
  - An approximate numeric literal is a numeric value in scientific notation, such as 7E3 or -57.9E2, or a numeric value with a decimal, such as 7., -95.7, or +6.2. Numbers in the range of Java double are supported.
  - The boolean literals TRUE and FALSE.
- Identifiers:

- An identifier is an unlimited length sequence of Java letters and Java digits, the first of which must be a Java letter. A letter is any character for which the method Character.isJavaLetter returns true. This includes _ and $. A letter or digit is any character for which the method Character.isJavaLetterOrDigit returns true.
  - Identifiers cannot be the names NULL, TRUE, or FALSE.
  - Identifiers cannot be NOT, AND, OR, BETWEEN, LIKE, IN, or IS.
  - Identifiers are either header field references or property references.
  - Identifiers are case-sensitive.
  - Message header field references are restricted to:
    - JMSDeliveryMode
    - JMSPriority
    - JMSMessageID
    - JMSTimestamp
    - JMSCorrelationID
    - JMSType

    JMSMessageID, JMSTimestamp, JMSCorrelationID, and JMSType values can be null, and if so, are treated as a NULL value.
  - Any name beginning with JMSX is a JMS-defined property name.
  - Any name beginning with JMS_ is a provider-specific property name.
  - Any name that does not begin with JMS is an application-specific property name. If there is a reference to a property that does not exist in a message, its value is NULL. If it does exist, its value is the corresponding property value.
- White space is the same as it is defined for Java: space, horizontal tab, form feed, and line terminator.
- Expressions:
  - A selector is a conditional expression. A selector that evaluates to true matches; a selector that evaluates to false or unknown does not match.
  - Arithmetic expressions are composed of themselves, arithmetic operations, identifiers (whose value is treated as a numeric literal), and numeric literals.
  - Conditional expressions are composed of themselves, comparison operations, and logical operations.
- Standard bracketing (), to set the order in which expressions are evaluated, is supported.
- Logical operators in precedence order: NOT, AND, OR.
- Comparison operators: =, >, >=, <, <=, <> (not equal).
  - Only values of the same type can be compared. One exception is that it is valid to compare exact numeric values and approximate numeric values. (The type conversion required is defined by the rules of Java numeric promotion.) If there is an attempt to compare different types, the selector is always false.
  - String and boolean comparison is restricted to = and <>. Two strings are equal only if they contain the same sequence of characters.
- Arithmetic operators in precedence order:
  - +, - unary.
  - *, /, multiplication, and division.
  - +, -, addition, and subtraction.
  - Arithmetic operations on a NULL value are not supported. If they are attempted, the complete selector is always false.

- Arithmetic operations must use Java numeric promotion.

- arithmetic-expr1 [NOT] BETWEEN arithmetic-expr2 and arithmetic-expr3 comparison operator:
  - Age BETWEEN 15 and 19 is equivalent to age >= 15 AND age <= 19.
  - Age NOT BETWEEN 15 and 19 is equivalent to age < 15 OR age > 19.
  - If any of the expressions of a BETWEEN operation are NULL, the value of the operation is false. If any of the expressions of a NOT BETWEEN operation are NULL, the value of the operation is true.
- identifier [NOT] IN (string-literal1, string-literal2,...) comparison operator where identifier has a String or NULL value.
  - Country IN (’UK’, ’US’, ’France’) is true for ’UK’ and false for ’Peru’. It is equivalent to the expression (Country = ’UK’) OR (Country = ’US’) OR (Country = ’France’).
  - Country NOT IN (’UK’, ’US’, ’France’) is false for ’UK’ and true for ’Peru’. It is equivalent to the expression NOT ((Country = ’UK’) OR (Country = ’US’) OR (Country = ’France’)).
  - If the identifier of an IN or NOT IN operation is NULL, the value of the operation is unknown.
- identifier [NOT] LIKE pattern-value [ESCAPE escape-character] comparison operator, where identifier has a string value. pattern-value is a string literal, where _ stands for any single character and % stands for any sequence of characters (including the empty sequence). All other characters stand for themselves. The optional escape-character is a single character string literal, whose character is used to escape the special meaning of the _ and % in pattern-value.
  - phone LIKE ’12%3’ is true for 123 and 12993 and false for 1234.
  - word LIKE ’l_se’ is true for lose and false for loose.
  - underscored LIKE ’_%’ ESCAPE ’\’ is true for _foo and false for bar.
  - phone NOT LIKE ’12%3’ is false for 123 and 12993 and true for 1234.
  - If the identifier of a LIKE or NOT LIKE operation is NULL, the value of the operation is unknown.
- identifier IS NULL comparison operator tests for a null header field value, or a missing property value.
  - prop_name IS NULL.
- identifier IS NOT NULL comparison operator tests for the existence of a non-null header field value or a property value.
  - prop_name IS NOT NULL.

The following message selector selects messages with a message type of car, color of blue, and weight greater than 2500 lbs:

```
"JMSType = 'car' AND color = 'blue' AND weight > 2500"
```

As noted above, property values can be NULL. The evaluation of selector expressions that contain NULL values is defined by SQL 92 NULL semantics. The following is a brief description of these semantics:

- SQL treats a NULL value as unknown.
- Comparison or arithmetic with an unknown value always yields an unknown value.
- The IS NULL and IS NOT NULL operators convert an unknown value into the respective TRUE and FALSE values.

Although SQL supports fixed decimal comparison and arithmetic, JMS message selectors do not. This is why exact numeric literals are restricted to those without a decimal. It is also why there are numerics with a decimal as an alternate representation for an approximate numeric value.

SQL comments are not supported.

Mapping JMS messages onto WebSphere MQ messages

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

This section does not apply if an application uses a real-time connection to a broker. When an application uses a real-time connection, all communication is performed directly over TCP/IP; no WebSphere MQ queues or messages are involved.

WebSphere MQ messages are composed of three components:

- The WebSphere MQ Message Descriptor (MQMD)
- A WebSphere MQ MQRFH2 header
- The message body.

The MQRFH2 is optional, and its inclusion in an outgoing message is governed by a flag in the JMS Destination class. You can set this flag using the WebSphere MQ JMS administration tool. Because the MQRFH2 carries JMS-specific information, always include it in the message when the sender knows that the receiving destination is a JMS application. Normally, omit the MQRFH2 when sending a message directly to a non-JMS application. This is because such an application does not expect an MQRFH2 in its WebSphere MQ message.

If an incoming message does not have an MQRFH2 header, the Queue or Topic object derived from the JMSReplyTo header field of the message, by default, has this flag set so that a reply message sent to the queue or topic also does not have an MQRFH2 header. You can switch off this behavior of including an MQRFH2 header in a reply message only if the original message has an MQRFH2 header by setting the TARGCLIENTMATCHING property of the connection factory to NO.

Figure 7 on page 84 shows how the structure of a JMS message is transformed to a WebSphere MQ message and back again:

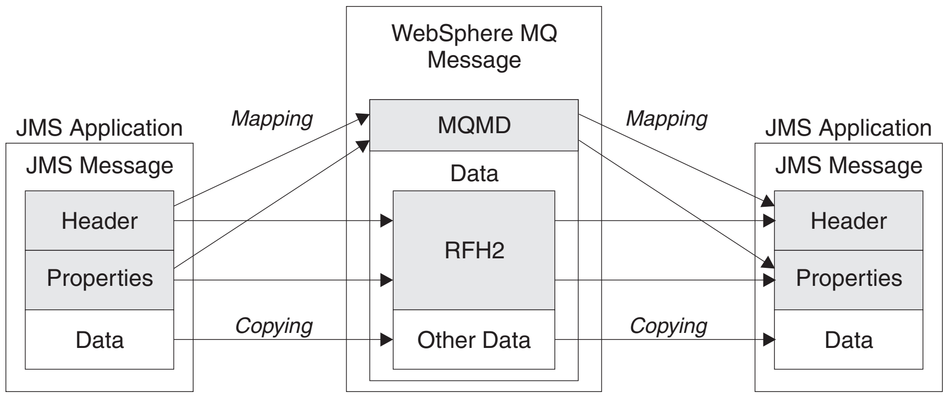


Figure 7. How messages are transformed between JMS and WebSphere MQ using the MQRFH2 header

The structures are transformed in two ways:

Mapping
: Where the MQMD includes a field that is equivalent to the JMS field, the JMS field is mapped onto the MQMD field. Additional MQMD fields are exposed as JMS properties, because a JMS application might need to get or set these fields when communicating with a non-JMS application.

Copying
: Where there is no MQMD equivalent, a JMS header field or property is passed, possibly transformed, as a field inside the MQRFH2.

The MQRFH2 header:

This section describes the MQRFH Version 2 header, which carries JMS-specific data that is associated with the message content. The MQRFH2 Version 2 is an extensible header, and can also carry additional information that is not directly associated with JMS. However, this section covers only its use by JMS.

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

Fixed portion

The fixed portion is modelled on the *standard* WebSphere MQ header pattern and consists of the following fields:

StrucId (MQCHAR4)

Structure identifier.

Must be MQRFH_STRUC_ID (value: "RFH ") (initial value).

MQRFH_STRUC_ID_ARRAY (value: "R","F","H"," ") is also defined in the usual way.

Version (MQLONG)

Structure version number.

Must be MQRFH_VERSION_2 (value: 2) (initial value).

StrucLength (MQLONG)

Total length of MQRFH2, including the NameValueData fields.

The value set into StrucLength must be a multiple of 4 (the data in the NameValueData fields can be padded with space characters to achieve this).

Encoding (MQLONG)

Data encoding.

Encoding of any numeric data in the portion of the message following the MQRFH2 (the next header, or the message data following this header).

CodedCharSetId (MQLONG)

Coded character set identifier.

Representation of any character data in the portion of the message following the MQRFH2 (the next header, or the message data following this header).

Format (MQCHAR8)

Format name.

Format name for the portion of the message following the MQRFH2.

Flags (MQLONG)

Flags.

MQRFH_NO_FLAGS =0. No flags set.

NameValueCCSID (MQLONG)

The coded character set identifier (CCSID) for the NameValueData character strings contained in this header. The NameValueData can be coded in a character set that differs from the other character strings that are contained in the header (StrucID and Format).

If the NameValueCCSID is a 2-byte Unicode CCSID (1200, 13488, or 17584), the byte order of the Unicode is the same as the byte ordering of the numeric fields in the MQRFH2. (For example, Version, StrucLength, and NameValueCCSID itself.)

The NameValueCCSID takes values from the following table:

Table 18. Possible values for NameValueCCSID field

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

Variable portion

The variable portion follows the fixed portion. The variable portion contains a variable number of MQRFH2 folders. Each folder contains a variable number of elements or properties. Folders group together related properties. The MQRFH2 headers created by JMS can contain up to three folders:

The <mcd> folder

This contains properties that describe the *shape* or *format* of the message. For example, the `Msd` property identifies the message as being Text, Bytes, Stream, Map, Object, or null. This folder is always present in a JMS MQRFH2.

The <jms> folder
This is used to transport JMS header fields, and JMSX properties that cannot be fully expressed in the MQMD. This folder is always present in a JMS MQRFH2.

The <usr> folder
This is used to transport any application-defined properties associated with the message. This folder is present only if the application has set some application-defined properties.

The <mqext> folder
This is used to transport IBM defined properties that are used only by WebSphere Application Server. This folder is present only if the application has set at least one of these properties.

Table 19 shows a full list of property names.

Table 19. MQRFH2 folders and properties used by JMS

JMS field name	Java type	MQRFH2 folder name	Property name	Type/values
JMSDestination	Destination	jms	Dst	string
JMSExpiration	long	jms	Exp	i8
JMSPriority	int	jms	Pri	i4
JMSDeliveryMode	int	jms	Dlv	i4
JMSCorrelationID	String	jms	Cid	string
JMSReplyTo	Destination	jms	Rto	string
JMSTimestamp	long	jms	Tms	i8
JMSType	String	mcd	Type, Set, Fmt	string
JMSXGroupID	String	jms	Gid	string
JMSXGroupSeq	int	jms	Seq	i4
xxx (user defined)	Any	usr	xxx	any
		mcd	Msd	jms_none jms_text jms_bytes jms_map jms_stream jms_object

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

NameValueLength (MQLONG)
Length in bytes of the NameValueData string that immediately follows this length field (it does not include its own length). The value set into NameValueLength is always a multiple of 4 (the NameValueData field is padded with space characters to achieve this).

NameValueData (MQCHARn)
A single character string, whose length in bytes is given by the preceding NameValueLength field. It contains a folder holding a sequence of properties. Each property is a name/type/value triplet, contained within an XML element whose name is the folder name, as follows:

```
<foldername> triplet1 triplet2 .....   tripletn </foldername>
```

The closing `</foldername>` tag can be followed by spaces as padding characters. Each triplet is encoded using an XML-like syntax:

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

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

`name` is the name of the property; see Table 19 on page 86.

`datatype`
must match, after folding, one of the data types listed in Table 20.

`value` is a string representation of the value to be conveyed, using the definitions in Table 20.

A null value is encoded using the following syntax:

```
<name dt='datatype' xsi:nil='true'></name>
```

Do not use `xsi:nil='false'`.

Table 20. Property data types

Data type	Definition
string	Any sequence of characters excluding `<` and `&`
boolean	The character `0` or `1` (`0` = false, `1` = true)
bin.hex	Hexadecimal digits representing octets
i1	A number, expressed using digits `0..9`, with optional sign (no fractions or exponent). Must lie in the range -128 to 127 inclusive
i2	A number, expressed using digits `0..9`, with optional sign (no fractions or exponent). Must lie in the range -32768 to 32767 inclusive
i4	A number, expressed using digits `0..9`, with optional sign (no fractions or exponent). Must lie in the range -2147483648 to 2147483647 inclusive
i8	A number, expressed using digits `0..9`, with optional sign (no fractions or exponent). Must lie in the range -9223372036854775808 to 92233720368547750807 inclusive
int	A number, expressed using digits `0..9`, with optional sign (no fractions or exponent). Must lie in the same range as i8. This can be used in place of one of the i* types if the sender does not want to associate a particular precision with the property
r4	Floating point number, magnitude <= 3.40282347E+38, >= 1.175E-37 expressed using digits `0..9`, optional sign, optional fractional digits, optional exponent
r8	Floating point number, magnitude <= 1.7976931348623E+308, >= 2.225E-307 expressed using digits `0..9`, optional sign, optional fractional digits, optional exponent

A string value can contain spaces. You must use the following escape sequences in a string value:

- `&amp;` for the `&` character
- `&lt;` for the `<` character

You can use the following escape sequences, but they are not required:

- `&gt;` for the `>` character
- `&apos;` for the `'` character
- `&quot;` for the `"` character

JMS fields and properties with corresponding MQMD fields:

Table 21 lists the JMS header fields and Table 22 lists the JMS properties that are mapped directly to MQMD fields. Table 23 lists the provider specific properties and the MQMD fields that they are mapped to.

Table 21. JMS header fields mapping to MQMD fields

JMS header field	Java type	MQMD field	C type
JMSDeliveryMode	int	Persistence	MQLONG
JMSExpiration	long	Expiry	MQLONG
JMSPriority	int	Priority	MQLONG
JMSMessageID	String	MsgID	MQBYTE24
JMSTimestamp	long	PutDate PutTime	MQCHAR8 MQCHAR8
JMSCorrelationID	String	CorrelId	MQBYTE24
Note: 1. JMS_IBM_Character_Set property value is a String value that contains the Java character set equivalent for the numeric CodedCharacterSetId value. MQMD field CodedCharacterSetId is a numeric value that contains the equivalent of the Java character set string specified by the JMS_IBM_Character_Set property.			

Table 22. JMS properties mapping to MQMD fields

JMS property	Java type	MQMD field	C type
JMSXUserID	String	UserIdentifier	MQCHAR12
JMSXAppID	String	PutApplName	MQCHAR28
JMSXDeliveryCount	int	BackoutCount	MQLONG
JMSXGroupID	String	GroupId	MQBYTE24
JMSXGroupSeq	int	MsgSeqNumber	MQLONG

Table 23. JMS provider specific properties mapping to MQMD fields

JMS provider specific property	Java type	MQMD field	C type
JMS_IBM_Report_Exception	int	Report	MQLONG
JMS_IBM_Report_Expiration	int	Report	MQLONG
JMS_IBM_Report_COA	int	Report	MQLONG
JMS_IBM_Report_COD	int	Report	MQLONG
JMS_IBM_Report_PAN	int	Report	MQLONG
JMS_IBM_Report_NAN	int	Report	MQLONG

Table 23. JMS provider specific properties mapping to MQMD fields (continued)

JMS provider specific property	Java type	MQMD field	C type
JMS_IBM_Report_Pass_Msg_ID	int	Report	MQLONG
JMS_IBM_Report_Pass_Correl_ID	int	Report	MQLONG
JMS_IBM_Report_Discard_Msg	int	Report	MQLONG
JMS_IBM_MsgType	int	MsgType	MQLONG
JMS_IBM_Feedback	int	Feedback	MQLONG
JMS_IBM_Format	String	Format[1]	MQCHAR8
JMS_IBM_PutApplType	int	PutApplType	MQLONG
JMS_IBM_Encoding	int	Encoding	MQLONG
JMS_IBM_Character_Set	String	CodedCharacterSetId	MQLONG
JMS_IBM_PutDate	String	PutDate	MQCHAR8
JMS_IBM_PutTime	String	PutTime	MQCHAR8
JMS_IBM_Last_Msg_In_Group	boolean	MsgFlags	MQLONG

Note:

1. JMS_IBM_Format represents the format of the message body. This can be defined by the application setting the JMS_IBM_Format property of the message (note that there is an 8 character limit), or can default to the WebSphere MQ format of the message body appropriate to the JMS message type. JMS_IBM_Format maps to the MQMD Format field only if the message contains no RFH or RFH2 sections. In a typical message, it will map to the Format field of the RFH2 immediately preceding the message body.

Mapping JMS fields onto WebSphere MQ fields (outgoing messages):

Table 24 shows how the JMS header fields are mapped into MQMD/RFH2 fields at send() or publish() time. Table 25 on page 90 shows how JMS properties and Table 26 on page 90 shows how JMS provider specific properties are mapped to MQMD fields at send() or publish() time,

For fields marked Set by Message Object, the value transmitted is the value held in the JMS message immediately before the send() or publish() operation. The value in the JMS message is left unchanged by the operation.

For fields marked Set by Send Method, a value is assigned when the send() or publish() is performed (any value held in the JMS message is ignored). The value in the JMS message is updated to show the value used.

Fields marked as Receive-only are not transmitted and are left unchanged in the message by send() or publish().

Table 24. Outgoing message field mapping

JMS header field name	MQMD field used for transmission	Header	Set by
JMSDestination		MQRFH2	Send Method
JMSDeliveryMode	Persistence	MQRFH2	Send Method
JMSExpiration	Expiry	MQRFH2	Send Method
JMSPriority	Priority	MQRFH2	Send Method
JMSMessageID	MsgID		Send Method

Table 24. Outgoing message field mapping (continued)

JMS header field name	MQMD field used for transmission	Header	Set by
JMSTimestamp	PutDate/PutTime		Send Method
JMSCorrelationID	CorrelId	MQRFH2	Message Object
JMSReplyTo	ReplyToQ/ReplyToQMgr	MQRFH2	Message Object
JMSType		MQRFH2	Message Object
JMSRedelivered			Receive-only

Note:

1. MQMD field CodedCharacterSetId is a numeric value that contains the equivalent of the Java character set string specified by the JMS_IBM_Character_Set property.

Table 25. Outgoing message JMS property mapping

JMS property name	MQMD field used for transmission	Header	Set by
JMSXUserID	UserIdentifier		Send Method
JMSXAppID	PutApplName		Send Method
JMSXDeliveryCount			Receive-only
JMSXGroupID	GroupId	MQRFH2	Message Object
JMSXGroupSeq	MsgSeqNumber	MQRFH2	Message Object

Table 26. Outgoing message JMS provider specific property mapping

JMS provider specific property name	MQMD field used for transmission	Header	Set by
JMS_IBM_Report_Exception	Report		Message Object
JMS_IBM_Report_Expiration	Report		Message Object
JMS_IBM_Report_COA/COD	Report		Message Object
JMS_IBM_Report_NAN/PAN	Report		Message Object
JMS_IBM_Report_Pass_Msg_ID	Report		Message Object
JMS_IBM_Report_Pass_Correl_ID	Report		Message Object
JMS_IBM_Report_Discard_Msg	Report		Message Object
JMS_IBM_MsgType	MsgType		Message Object
JMS_IBM_Feedback	Feedback		Message Object
JMS_IBM_Format	Format		Message Object
JMS_IBM_PutApplType	PutApplType		Send Method
JMS_IBM_Encoding	Encoding		Message Object
JMS_IBM_Character_Set	CodedCharacterSetId		Message Object
JMS_IBM_PutDate	PutDate		Send Method
JMS_IBM_PutTime	PutTime		Send Method
JMS_IBM_Last_Msg_In_Group	MsgFlags		Message Object

Mapping JMS header fields at send() or publish():

These notes relate to the mapping of JMS fields at send() or publish().

JMSDestination to MQRFH2

This is stored as a string that serializes the salient characteristics of the destination object, so that a receiving JMS can reconstitute an equivalent destination object. The MQRFH2 field is encoded as URI (see "Uniform resource identifiers (URIs)" on page 116 for details of the URI notation).

JMSReplyTo to MQMD.ReplyToQ, ReplyToQMgr, MQRFH2

The queue and queue manager name are copied to the MQMD.ReplyToQ and ReplyToQMgr fields respectively. The destination extension information (other useful details that are kept in the destination object) is copied into the MQRFH2 field. The MQRFH2 field is encoded as a URI (see "Uniform resource identifiers (URIs)" on page 116 for details of the URI notation).

JMSDeliveryMode to MQMD.Persistence

The JMSDeliveryMode value is set by the send() or publish() Method or MessageProducer, unless the Destination Object overrides it. The JMSDeliveryMode value is mapped to the MQMD.Persistence field as follows:

- JMS value PERSISTENT is equivalent to MQPER_PERSISTENT
- JMS value NON_PERSISTENT is equivalent to MQPER_NOT_PERSISTENT

If the MQQueue persistence property is not set to JMSC.MQJMS_PER_QDEF, the delivery mode value is also encoded in the MQRFH2.

JMSExpiration to/from MQMD.Expiry, MQRFH2

JMSExpiration stores the time to expire (the sum of the current time and the time to live), whereas MQMD stores the time to live. Also, JMSExpiration is in milliseconds, but MQMD.Expiry is in tenths of a second.

- If the send() method sets an unlimited time to live, MQMD.Expiry is set to MQEI_UNLIMITED, and no JMSExpiration is encoded in the MQRFH2.
- If the send() method sets a time to live that is less than 214748364.7 seconds (about 7 years), the time to live is stored in MQMD.Expiry, and the expiration time (in milliseconds), is encoded as an i8 value in the MQRFH2.
- If the send() method sets a time to live greater than 214748364.7 seconds, MQMD.Expiry is set to MQEI_UNLIMITED. The true expiration time in milliseconds is encoded as an i8 value in the MQRFH2.

JMSPriority to MQMD.Priority

Directly map JMSPriority value (0-9) onto MQMD priority value (0-9). If JMSPriority is set to a non-default value, the priority level is also encoded in the MQRFH2.

JMSMessageID from MQMD.MessageID

All messages sent from JMS have unique message identifiers assigned by WebSphere MQ. The value assigned is returned in the MQMD.MessageId field after the MQPUT call, and is passed back to the application in the JMSMessageID field. The WebSphere MQ messageId is a 24-byte binary value, whereas the JMSMessageID is a string. The JMSMessageID is composed of the binary messageId value converted to a sequence of 48 hexadecimal characters, prefixed with the characters ID:. JMS provides a hint that can be set to disable the production of message identifiers. This hint is ignored, and a unique identifier is assigned in all cases. Any value that is set into the JMSMessageId field before a send() is overwritten.

JMSTimestamp to MQRFH2
: During a send, the JMSTimestamp field is set according to the JVM's clock. This value is set into the MQRFH2. Any value that is set into the JMSTimestamp field before a send() is overwritten. See also the JMS_IBM_PutDate and JMS_IBM_PutTime properties.

JMSType to MQRFH2
: This string is set into the MQRFH2 mcd.Type field. If it is in URI format, it can also affect mcd.Set and mcd.Fmt fields. See also "Using a real-time connection to a broker of WebSphere Event Broker or WebSphere Message Broker" on page 156.

JMSCorrelationID to MQMD.CorrelId, MQRFH2
: The JMSCorrelationID can hold one of the following:

  A provider specific message ID
  : This is a message identifier from a message previously sent or received, and so should be a string of 48 hexadecimal digits that are prefixed with ID:. The prefix is removed, the remaining characters are converted into binary, and then they are set into the MQMD.CorrelId field. No CorrelId value is encoded in the MQRFH2.

  A provider-native byte[] value
  : The value is copied into the MQMD.CorrelId field - padded with nulls, or truncated to 24 bytes if necessary. No CorrelId value is encoded in the MQRFH2.

  An application-specific string
  : The value is copied into the MQRFH2. The first 24 bytes of the string, in UTF8 format, are written into the MQMD.CorrelID.

Mapping JMS property fields:

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

JMSXUserID from MQMD UserIdentifier
: JMSXUserID is set on return from send call.

JMSXAppID from MQMD PutApplName
: JSMXAppID is set on return from send call.

JMSXGroupID to MQRFH2 (point-to-point)
: For point-to-point messages, the JMSXGroupID is copied into the MQMD GroupID field. If the JMSXGroupID starts with the prefix ID:, it is converted into binary. Otherwise, it is encoded as a UTF8 string. The value is padded or truncated if necessary to a length of 24 bytes. The MQMF_MSG_IN_GROUP flag is set.

JMSXGroupID to MQRFH2 (publish/subscribe)
: For publish/subscribe messages, the JMSXGroupID is copied into the MQRFH2 as a string.

JMSXGroupSeq MQMD MsgSeqNumber (point-to-point)
: For point-to-point messages, the JMSXGroupSeq is copied into the MQMD MsgSeqNumber field. The MQMF_MSG_IN_GROUP flag is set.

JMSXGroupSeq MQMD MsgSeqNumber (publish/subscribe)
For publish/subscribe messages, the JMSXGroupSeq is copied into the MQRFH2 as an i4.

Mapping JMS provider-specific fields:

The following notes refer to the mapping of JMS Provider specific fields into WebSphere MQ messages:

JMS_IBM_Report_<name> to MQMD Report
A JMS application can set the MQMD Report options, using the following JMS_IBM_Report_XXX properties. The single MQMD is mapped to several JMS_IBM_Report_XXX properties. The application must set the value of these properties to the standard WebSphere MQ MQRO_ constants (included in com.ibm.mq.MQC). So, for example, to request COD with full Data, the application must set JMS_IBM_Report_COD to the value MQC.MQRO_COD_WITH_FULL_DATA.

JMS_IBM_Report_Exception

MQRO_EXCEPTION or
MQRO_EXCEPTION_WITH_DATA or
MQRO_EXCEPTION_WITH_FULL_DATA

JMS_IBM_Report_Expiration

MQRO_EXPIRATION or
MQRO_EXPIRATION_WITH_DATA or
MQRO_EXPIRATION_WITH_FULL_DATA

JMS_IBM_Report_COA

MQRO_COA or
MQRO_COA_WITH_DATA or
MQRO_COA_WITH_FULL_DATA

JMS_IBM_Report_COD

MQRO_COD or
MQRO_COD_WITH_DATA or
MQRO_COD_WITH_FULL_DATA

JMS_IBM_Report_PAN
MQRO_PAN

JMS_IBM_Report_NAN
MQRO_NAN

JMS_IBM_Report_Pass_Msg_ID
MQRO_PASS_MSG_ID

JMS_IBM_Report_Pass_Correl_ID
MQRO_PASS_CORREL_ID

JMS_IBM_Report_Discard_Msg
MQRO_DISCARD_MSG

JMS_IBM_MsgType to MQMD MsgType
Value maps directly onto MQMD MsgType. If the application has not set an explicit value of JMS_IBM_MsgType, a default value is used. This default value is determined as follows:

- If JMSReplyTo is set to a WebSphere MQ queue destination, MSGType is set to the value MQMT_REQUEST
- If JMSReplyTo is not set, or is set to anything other than a WebSphere MQ queue destination, MsgType is set to the value MQMT_DATAGRAM

JMS_IBM_Feedback to MQMD Feedback
Value maps directly onto MQMD Feedback.

JMS_IBM_Format to MQMD Format
Value maps directly onto MQMD Format.

JMS_IBM_Encoding to MQMD Encoding
If set, this property overrides the numeric encoding of the Destination Queue or Topic.

JMS_IBM_Character_Set to MQMD CodedCharacterSetId
If set, this property overrides the coded character set property of the Destination Queue or Topic.

JMS_IBM_PutDate from MQMD PutDate
The value of this property is set, during send, directly from the PutDate field in the MQMD. Any value that is set into the JMS_IBM_PutDate property before a send is overwritten. This field is a String of eight characters, in the WebSphere MQ Date format of YYYYMMDD. This property can be used in conjunction with the JMS_IBM_PutTime property to determine the time the message was put according to the queue manager.

JMS_IBM_PutTime from MQMD PutTime
The value of this property is set, during send, directly from the PutTime field in the MQMD. Any value that is set into the JMS_IBM_PutTime property before a send is overwritten. This field is a String of eight characters, in the WebSphere MQ Time format of HHMMSSTH. This property can be used in conjunction with the JMS_IBM_PutDate property to determine the time the message was put according to the queue manager.

JMS_IBM_Last_Msg_In_Group to MQMD MsgFlags
For point-to-point messaging, this boolean value maps to the MQMF_LAST_MSG_IN_GROUP flag in the MQMD MsgFlags field. It is normally used in conjunction with the JMSXGroupID and JMSXGroupSeq properties to indicate to a legacy WebSphere MQ application that this is the last message in a group. This property is ignored for publish/subscribe messaging.

Mapping WebSphere MQ fields onto JMS fields (incoming messages):

Table 27 shows how JMS header fields and Table 28 on page 95 shows how JMS property fields are mapped into MQMD/MQRFH2 fields at send() or publish() time. Table 29 on page 95 shows how JMS provider specific properties are mapped.

Table 27. Incoming message JMS header field mapping

JMS header field name	MQMD field retrieved from	MQRFH2 field retrieved from
JMSDestination		jms.Dst
JMSDeliveryMode	Persistence[1]	jms.Dlv[1]
JMSExpiration		jms.Exp

Table 27. Incoming message JMS header field mapping (continued)

JMS header field name	MQMD field retrieved from	MQRFH2 field retrieved from
JMSPriority	Priority	
JMSMessageID	MsgID	
JMSTimestamp	PutDate[1] PutTime[1]	jms.Tms[1]
JMSCorrelationID	CorrelId[1]	jms.Cid[1]
JMSReplyTo	ReplyToQ[1] ReplyToQMgr[1]	jms.Rto[1]
JMSType		mcd.Type, mcd.Set, mcd.Fmt
JMSRedelivered	BackoutCount	

Note:

1. For properties that can have values retrieved from the MQRFH2 or the MQMD, if both are available, the setting in the MQRFH2 is used.
2. JMS_IBM_Character_Set property value is a String value that contains the Java character set equivalent for the numeric CodedCharacterSetId value.

Table 28. Incoming message property mapping

JMS property name	MQMD field retrieved from	MQRFH2 field retrieved from
JMSXUserID	UserIdentifier	
JMSXAppID	PutApplName	
JMSXDeliveryCount	BackoutCount	
JMSXGroupID	GroupId[1]	jms.Gid[1]
JMSXGroupSeq	MsgSeqNumber[1]	jms.Seq[1]

Note:

1. For properties that can have values retrieved from the MQRFH2 or the MQMD, if both are available, the setting in the MQRFH2 is used.

Table 29. Incoming message provider specific JMS property mapping

JMS property name	MQMD field retrieved from	MQRFH2 field retrieved from
JMS_IBM_Report_Exception	Report	
JMS_IBM_Report_Expiration	Report	
JMS_IBM_Report_COA	Report	
JMS_IBM_Report_COD	Report	
JMS_IBM_Report_PAN	Report	
JMS_IBM_Report_NAN	Report	
JMS_IBM_Report_ Pass_Msg_ID	Report	
JMS_IBM_Report_Pass_Correl_ID	Report	
JMS_IBM_Report_Discard_Msg	Report	
JMS_IBM_MsgType	MsgType	

Table 29. Incoming message provider specific JMS property mapping (continued)

JMS property name	MQMD field retrieved from	MQRFH2 field retrieved from
JMS_IBM_Feedback	Feedback	
JMS_IBM_Format	Format	
JMS_IBM_PutApplType	PutApplType	
JMS_IBM_Encoding [1]	Encoding	
JMS_IBM_Character_Set [1]	CodedCharacterSetId	
JMS_IBM_PutDate	PutDate	
JMS_IBM_PutTime	PutTime	
JMS_IBM_Last_Msg_In_Group	MsgFlags	
1. Only set if the incoming message is a Bytes Message.		

Exchanging messages between a JMS application and a traditional WebSphere MQ application:

This section describes what happens when a JMS application exchanges messages with a traditional WebSphere MQ application that has no knowledge of the MQRFH2 header. Figure 8 on page 97 shows the mapping.

The administrator indicates that the JMS application is communicating with such an application by setting the TARGCLIENT property of the destination to MQ. This indicates that no MQRFH2 header is to be produced. Note that, if this is not done, the receiving application must be able to handle the MQRFH2 header.

The mapping from JMS to MQMD targeted at a traditional WebSphere MQ application is the same as mapping from JMS to MQMD targeted at a JMS application. If WebSphere MQ classes for JMS receives a WebSphere MQ message with the MQMD *Format* field set to other than MQFMT_RFH2, data is being received from a non-JMS application. If the format is MQFMT_STRING, the message is received as a JMS text message. Otherwise, it is received as a JMS bytes message. Because there is no MQRFH2, only those JMS properties that are transmitted in the MQMD can be restored.

If WebSphere MQ classes for JMS receives a message that does not have an MQRFH2 header, the TARGCLIENT property of the Queue or Topic object derived from the JMSReplyTo header field of the message is set to MQ by default. This means that a reply message sent to the queue or topic also does not have an MQRFH2 header. You can switch off this behavior of including an MQRFH2 header in a reply message only if the original message has an MQRFH2 header by setting the TARGCLIENTMATCHING property of the connection factory to NO.

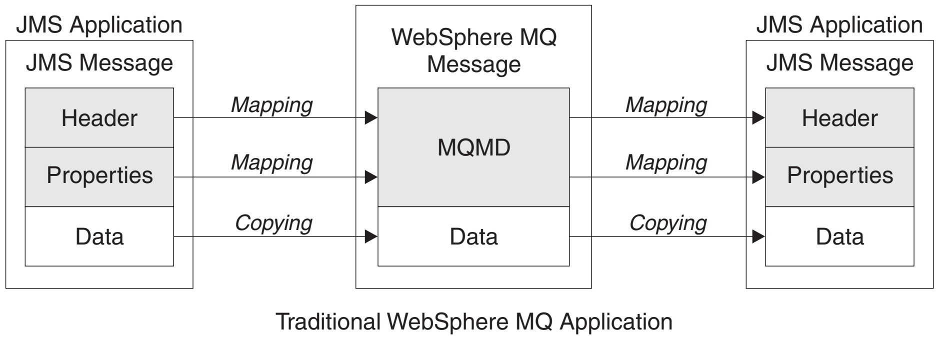


Figure 8. How JMS messages are transformed to WebSphere MQ messages with no MQRFH2 header

Message body:

This topic discusses the encoding of the message body itself. The encoding depends on the type of JMS message.

ObjectMessage
: An ObjectMessage is an object serialized by the Java Runtime in the normal way.

TextMessage
: A TextMessage is an encoded string. For an outgoing message, the string is encoded in the character set given by the destination object. This defaults to UTF8 encoding (the UTF8 encoding starts with the first character of the message; there is no length field at the start). It is, however, possible to specify any other character set supported by WebSphere MQ classes for JMS. Such character sets are used mainly when you send a message to a non-JMS application.

  If the character set is a double-byte set (including UTF16), the destination object's integer encoding specification determines the order of the bytes.

  An incoming message is interpreted using the character set and encoding that are specified in the message itself. These specifications are in the last WebSphere MQ header (or MQMD if there are no headers). For JMS messages, the last header is usually the MQRFH2.

BytesMessage
: A BytesMessage is, by default, a sequence of bytes as defined by the JMS 1.0.2 specification and associated Java documentation.

  For an outgoing message that was assembled by the application itself, the destination object's encoding property can be used to override the encodings of integer and floating point fields contained in the message. For example, you can request that floating point values are stored in S/390® rather than IEEE format).

  An incoming message is interpreted using the numeric encoding specified in the message itself. This specification is in the rightmost WebSphere MQ header (or MQMD if there are no headers). For JMS messages, the rightmost header is usually the MQRFH2.

  If a BytesMessage is received, and is re-sent without modification, its body is transmitted byte for byte, as it was received. The destination object's encoding property has no effect on the body. The only string-like entity that can be sent explicitly in a BytesMessage is a UTF8 string. This is encoded in Java UTF8 format, and starts with a 2-byte length field. The destination object's character set property has no effect on the encoding of an outgoing BytesMessage. The character set value in an incoming WebSphere MQ message has no effect on the interpretation of that message as a JMS BytesMessage.

Non-Java applications are unlikely to recognize the Java UTF8 encoding. Therefore, for a JMS application to send a BytesMessage that contains text data, the application itself must convert its strings to byte arrays, and write these byte arrays into the BytesMessage.

MapMessage

A MapMessage is a string containing XML name/type/value triplets encoded as:

```
<map>
  <elt name="elementname1" dt="datatype1">value1</elt>
  <elt name="elementname2" dt="datatype2">value2</elt>
  ...
</map>
```

where `datatype` is one of the data types listed in Table 20 on page 87. The default data type is `string`, and so the attribute `dt="string"` is omitted for string elements.

The character set used to encode or interpret the XML string that forms the body of a map message is determined according to the rules that apply to a text message.

Versions of WebSphere MQ classes for JMS earlier than Version 5.3 encoded the body of a map message in the following format:

```
<map>
  <elementname1 dt="datatype1">value1</elementname1>
  <elementname2 dt="datatype2">value2</elementname2>
  ...
</map>
```

Version 5.3 and later versions of WebSphere MQ classes for JMS can interpret either format, but versions of WebSphere MQ classes for JMS earlier than Version 5.3 cannot interpret the current format.

If an application needs to send map messages to another application that is using a version of WebSphere MQ classes for JMS earlier than Version 5.3, the sending application must call the connection factory method `setMapNameStyle(JMSC.MAP_NAME_STYLE_COMPATIBLE)` to specify that the map messages are sent in the previous format. By default, all map messages are sent in the current format.

StreamMessage

A StreamMessage is like a map message, but without element names:

```
<stream>
  <elt dt="datatype1">value1</elt>
  <elt dt="datatype2">value2</elt>
  ...
</stream>
```

where `datatype` is one of the data types listed in Table 20 on page 87. The default data type is `string`, and so the attribute `dt="string"` is omitted for string elements.

The character set used to encode or interpret the XML string that makes up the StreamMessage body is determined following the rules that apply to a TextMessage.

The MQRFH2.format field is set as follows:

MQFMT_NONE
for ObjectMessage, BytesMessage, or messages with no body.

MQFMT_STRING
for TextMessage, StreamMessage, or MapMessage.

Creating and configuring connection factories and destinations in a WebSphere MQ classes for JMS application

A WebSphere MQ classes for JMS application can create connection factories and destinations by retrieving them as administered objects from a Java Naming and Directory Interface (JNDI) namespace, by using the IBM JMS extensions, or by using the WebSphere MQ JMS extensions. An application can also use the IBM JMS extensions or WebSphere MQ JMS extensions to set the properties of connection factories and destinations.

Connection factories and destinations are starting points in the flow of logic of a JMS application. An application uses a ConnectionFactory object to create a connection to a messaging server, and uses a Queue or Topic object as a target to send messages to or a source from which to receive messages. An application therefore needs to create at least one connection factory and one or more destinations. Having created a connection factory or destination, the application might then need to configure the object by setting one or more of its properties.

In summary, an application can create and configure connection factories and destinations in the following ways:

Using JNDI to retrieve administered objects
An administrator can use the WebSphere MQ JMS administration tool or WebSphere MQ Explorer to create and configure connection factories and destinations as administered objects in a JNDI namespace. An application can then retrieve the administered objects from the JNDI namespace. Having retrieved an administered object, the application can, if required, set or change one or more of its properties by using either the IBM JMS extensions or the WebSphere MQ JMS extensions.

Using the IBM JMS extensions
An application can use the IBM JMS extensions to create connection factories and destinations dynamically at run time. The application first creates a JmsFactoryFactory object, and then uses methods of this object to create connection factories and destinations. Having created a connection factory or destination, the application can use methods inherited from the JmsPropertyContext interface to set its properties. Alternatively, the application can use a uniform resource identifier (URI) to specify one or more properties of a destination when it creates the destination.

Using the WebSphere MQ JMS extensions
An application can also use the WebSphere MQ JMS extensions to create connection factories and destinations dynamically at run time. The application uses the supplied constructors to create connection factories and destinations. Having created a connection factory or destination, the application can use methods of the object to set its properties.

Alternatively, the application can use a URI to specify one or more properties of a destination when it creates the destination.

Using JNDI to retrieve administered objects in a JMS application

To retrieve administered objects from a Java Naming and Directory Interface (JNDI) namespace, a JMS application must create an initial context and then use the lookup() method to retrieve the objects.

Before an application can retrieve administered objects from a JNDI namespace, an administrator must first create the administered objects. The administrator can use the WebSphere MQ JMS administration tool or WebSphere MQ Explorer to create and maintain administered objects in a JNDI namespace. For information about how to use the WebSphere MQ JMS administration tool, see "Using the WebSphere MQ JMS administration tool" on page 167. For information about how to use WebSphere MQ Explorer, see the help provided with WebSphere MQ Explorer. An application server, however, typically provides its own repository for administered objects and its own tools for creating and maintaining the objects.

To retrieve administered objects from a JNDI namespace, an application must first create an initial context, as shown in the following example:

```
import javax.jms.*;
import javax.naming.*;
import javax.naming.directory.*;
.
.
.
String url = "ldap://server.company.com/o=company_us,c=us";
String icf = "com.sun.jndi.ldap.LdapCtxFactory";
.
java.util.Hashtable environment = new java.util.Hashtable();
environment.put(Context.PROVIDER_URL, url);
environment.put(Context.INITIAL_CONTEXT_FACTORY, icf);
Context ctx = new InitialDirContext(environment);
```

In this code, the String variables `url` and `icf` have the following meanings:

url The uniform resource locator (URL) of the directory service. The URL can have one of the following formats:

- `ldap://`*`hostname/contextName`*, for a directory service based on an LDAP server
- `file:/`*`directoryPath`*, for a directory service based on the local file system

icf The class name of the initial context factory, which can be one of the following values:

- `com.sun.jndi.ldap.LdapCtxFactory`, for a directory service based on an LDAP server
- `com.sun.jndi.fscontext.RefFSContextFactory`, for a directory service based on the local file system

Note that some combinations of a JNDI package and a Lightweight Directory Access Protocol (LDAP) service provider can cause LDAP error 84 to occur. To resolve this problem, insert the following line of code before the call to InitialDirContext():

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

After an initial context is obtained, the application can retrieve administered objects from the JNDI namespace by using the lookup() method, as shown in the following example:

```
ConnectionFactory factory;
Queue queue;
Topic topic;
.
.
.
factory = (ConnectionFactory)ctx.lookup("cn=myCF");
queue = (Queue)ctx.lookup("cn=myQ");
topic = (Topic)ctx.lookup("cn=myT");
```

This code retrieves the following objects from an LDAP based namespace:

- A ConnectionFactory object bound with the name myCF
- A Queue object bound with the name myQ
- A Topic object bound with the name myT

For more information about using JNDI, see the JNDI documentation provided by Sun Microsystems, Inc..

Using the IBM JMS extensions

WebSphere MQ classes for JMS contains a set of extensions to the JMS API called the IBM JMS extensions. An application can use these extensions to create connection factories and destinations dynamically at run time, and to set the properties of WebSphere MQ classes for JMS objects. The extensions can be used with any messaging provider.

The IBM JMS extensions are a set of interfaces and classes in the following packages:

- com.ibm.msg.client.jms
- com.ibm.msg.client.services

These extensions provide the following function:

- A factory based mechanism for creating connection factories and destinations dynamically at run time, instead of retrieving them as administered objects from a Java Naming and Directory Interface (JNDI) namespace
- A set of methods for setting the properties of WebSphere MQ classes for JMS objects
- A set of exception classes with methods for obtaining detailed information about a problem
- A set of methods for controlling tracing
- A set of methods for obtaining version information about WebSphere MQ classes for JMS

With regard to creating connection factories and destinations dynamically at run time, and setting and getting their properties, the IBM JMS extensions provide an alternative set of interfaces to the WebSphere MQ JMS extensions. However, whereas the WebSphere MQ JMS extensions are specific to the WebSphere MQ messaging provider, the IBM JMS extensions are not specific to WebSphere MQ and can be used with any messaging provider within the layered architecture described in “A layered architecture” on page 69.

The interface com.ibm.msg.client.wmq.WMQConstants contains the definitions of constants, which an application can use when setting the properties of WebSphere

MQ classes for JMS objects using the IBM JMS extensions. The interface contains constants for the WebSphere MQ messaging provider and JMS constants that are independent of any messaging provider.

The examples of code that follow assume that the following import statements have been run:

```
import com.ibm.msg.client.jms.*;
import com.ibm.msg.client.services.*;
import com.ibm.msg.client.wmq.WMQConstants;
```

Creating connection factories and destinations

Before an application can create connection factories and destinations using the IBM JMS extensions, it must first create a JmsFactoryFactory object. To create a JmsFactoryFactory object, the application calls the getInstance() method of the JmsFactoryFactory class, as shown in the following example:

```
JmsFactoryFactory ff = JmsFactoryFactory.getInstance(JmsConstants.WMQ_PROVIDER);
```

The parameter on the getInstance() call is a constant that identifies the WebSphere MQ messaging provider as the chosen messaging provider. The application can then use the JmsFactoryFactory object to create connection factories and destinations.

To create a connection factory, the application calls the createConnectionFactory() method of the JmsFactoryFactory object, as shown in the following example:

```
JmsConnectionFactory factory = ff.createConnectionFactory();
```

This statement creates a JmsConnectionFactory object with the default values for all its properties, which means that the application connects to the default queue manager in bindings mode. If you want an application to connect in client mode, or connect to a queue manager other than the default queue manager, the application must set the appropriate properties of the JmsConnectionFactory object before creating the connection. For information about how to do this, see “Setting the properties of WebSphere MQ classes for JMS objects” on page 103.

The JmsFactoryFactory class also contains methods to create connection factories of the following types:

- JmsQueueConnectionFactory
- JmsTopicConnectionFactory
- JmsXAConnectionFactory
- JmsXAQueueConnectionFactory
- JmsXATopicConnectionFactory

To create a Queue object, the application calls the createQueue() method of the JmsFactoryFactory object, as shown in the following example:

```
JmsQueue q1 = ff.createQueue("Q1");
```

This statement creates an JmsQueue object with the default values for all its properties. The object represents a WebSphere MQ queue called Q1 that belongs to the local queue manager. This queue can be a local queue, an alias queue, or a remote queue definition.

The createQueue() method can also accept a queue uniform resource identifier (URI) as a parameter. A queue URI is a string that specifies the name of a

WebSphere MQ queue and, optionally, the name of the queue manager that owns the queue, and one or more properties of the JmsQueue object. The following statement contains an example of a queue URI:

```
JmsQueue q2 = ff.createQueue("queue://QM2/Q2?persistence=2&priority=5");
```

The JmsQueue object created by this statement represents a WebSphere MQ queue called Q2 that is owned by queue manager QM2, and all messages sent to this destination are persistent and have a priority of 5. For more information about queue URIs, see “Uniform resource identifiers (URIs)” on page 116. For an alternative way of setting the properties of a JmsQueue object, see “Setting the properties of WebSphere MQ classes for JMS objects.”

To create a Topic object, an application can use the createTopic() method of the JmsFactoryFactory object, as shown in the following example:

```
JmsTopic t1 = ff.createTopic("Sport/Football/Results");
```

This statement creates a JmsTopic object with the default values for all its properties. The object represents a topic called Sport/Football/Results.

The createTopic() method can also accept a topic URI as a parameter. A topic URI is a string that specifies the name of a topic and, optionally, one or more properties of the JmsTopic object. The following statements contain an example of a topic URI:

```
String s1 = "topic://Sport/Tennis/Results?persistence=1&priority=0";
JmsTopic t2 = ff.createTopic(s1);
```

The JmsTopic object created by these statements represents a topic called Sport/Tennis/Results, and all messages sent to this destination are nonpersistent and have a priority of 0. For more information about topic URIs, see “Uniform resource identifiers (URIs)” on page 116. For an alternative way of setting the properties of a JmsTopic object, see “Setting the properties of WebSphere MQ classes for JMS objects.”

After an application has created a connection factory or destination, that object can be used only with the selected messaging provider.

Setting the properties of WebSphere MQ classes for JMS objects

To set the properties of WebSphere MQ classes for JMS objects using the IBM JMS extensions, an application uses the methods of the com.ibm.msg.client.JmsPropertyContext interface.

For each Java data type, the JmsPropertyContext interface contains a method to set the value of a property with that data type, and a method to get to get the value of a property with that data type. For example, an application calls the setIntProperty() method to set a property with an integer value, and calls the getIntProperty() method to get a property with an integer value.

Instances of classes in the com.ibm.mq.jms package also inherit the methods of the JmsPropertyContext interface. An application can therefore use these methods to set the properties of MQConnectionFactory, MQQueue, and MQTopic objects.

When an application creates a WebSphere MQ classes for JMS object, any properties with default values are set automatically. When an application sets a property, the new value replaces any previous value the property had. After a property has been set, it cannot be deleted, but its value can be changed.

If an application attempts to set a property to a value that is not valid value for the property, WebSphere MQ classes for JMS throws a JMSException exception. If an application attempts to get a property that has not been set, the behavior is as described in the JMS specification. WebSphere MQ classes for JMS throws a NumberFormatException exception for primitive data types and returns null for referenced data types.

In addition to the predefined properties of a WebSphere MQ classes for JMS object, an application can set its own properties. These application defined properties are ignored by WebSphere MQ classes for JMS.

For more information about the properties of WebSphere MQ classes for JMS objects, see "Properties of WebSphere MQ classes for JMS objects" on page 176.

The following code is an example of how to set properties using the IBM JMS extensions. The code sets five properties of a connection factory.

```
factory.setIntProperty(WMQConstants.WMQ_CONNECTION_MODE,
                       WMQConstants.WMQ_CM_CLIENT);
factory.setStringProperty(WMQConstants.WMQ_QUEUE_MANAGER, "QM1");
factory.setStringProperty(WMQConstants.WMQ_HOST_NAME, "HOST1");
factory.setIntProperty(WMQConstants.WMQ_PORT, 1415);
factory.setStringProperty(WMQConstants.WMQ_CHANNEL, "QM1.SVR");
```

The effect of setting these properties is that the application connects to queue manager QM1 in client mode using an MQI channel called QM1.SVR. The queue manager is running on a system with host name HOST1, and the listener for the queue manager is listening in port number 1415.

The JmsPropertyContext interface also contains the setObjectProperty() method, which an application can use to set properties. The second parameter of the method is an object that encapsulates the value of the property. For example, the following code creates an Integer object that encapsulates the integer 1415, and then calls setObjectProperty() to set the PORT property of a connection factory to the value 1415:

```
Integer port = new Integer(1415);
factory.setObjectProperty(WMQConstants.WMQ_PORT, port);
```

This code is therefore equivalent to the following statement:

```
factory.setIntProperty(WMQConstants.WMQ_PORT, 1415);
```

Conversely, the getObjectProperty() method returns an object that encapsulates the value of a property.

Implicit conversion of a property value from one data type to another

When an application uses a method of the JmsPropertyContext interface to set or get the property of a WebSphere MQ classes for JMS object, the value of the property can be implicitly converted from one data type to another.

For example, the following statement sets the PRIORITY property of the JmsQueue object q1:

```
q1.setStringProperty(WMQConstants.WMQ_PRIORITY, "5");
```

The PRIORITY property has an integer value, and so the setStringProperty() call implicitly converts the string "5" (the source value) to the integer 5 (the target value), which then becomes the value of the PRIORITY property.

Conversely, the following statement gets the PRIORITY property of the JmsQueue object q1:

```
String s1 = q1.getStringProperty(WMQConstants.WMQ_PRIORITY);
```

The integer 5 (the source value), which is the value of the PRIORITY property, is implicitly converted to the string “5” (the target value) by the getStringProperty() call.

The conversions supported by WebSphere MQ classes for JMS are shown in Table 30.

Table 30. Supported conversions from one data type to another

Source data type	**Supported target data types**
boolean	String
byte	int, long, short, String
char	String
double	String
float	double, String
int	long, String
long	String
short	int, long, String
String	boolean, byte, double, float, int, long, short

The general rules governing the supported conversions are as follows:
- Numeric values can be converted from one data type to another provided no data is lost during the conversion. For example, a value with data type `int` can be converted into a value with data type `long`, but cannot be converted into a value with data type `short`.
- A value of any data type can be converted into a string.
- A string can be converted to a value of any other data type (except `char`) provided the string is in the correct format for the conversion. If an application attempts to convert a string that is not in the correct format, WebSphere MQ classes for JMS throws a NumberFormatException exception.
- If an application attempts a conversion that is not supported, WebSphere MQ classes for JMS throws a MessageFormatException exception.

The specific rules for converting a value from one data type to another are as follows:
- When converting a boolean value to a string, the value `true` is converted to the string “true”, and the value `false` is converted to the string “false”.
- When converting a string to a boolean value, the string “true” (not case sensitive) is converted to `true`, and the string “false” (not case sensitive) is converted to `false`. Any other string is converted to `false`.
- When converting a string to a value with data type `byte`, `int`, `long`, or `short`, the string must have the following format:

  [*blanks*][*sign*]*digits*

  The meanings of the components of the string are as follows:

  blanks Optional leading blank characters.

sign An optional plus sign (+) or minus sign (-).

digits A contiguous sequence of digits (0-9). At least one digit must be present.

After the sequence of digits, the string can contain other characters that are not digits, but the conversion stops as soon as the first of these characters is reached. The string is assumed to represent a decimal integer.

If the string is not in the correct format, WebSphere MQ classes for JMS throws a NumberFormatException exception.

- When converting a string to a value with data type `double` or `float`, the string must have the following format:

  [*blanks*][*sign*]*digits*[*e_char*[*e_sign*]*e_digits*]

  The meanings of the components of the string are as follows:

  blanks Optional leading blank characters.

  sign An optional plus sign (+) or minus sign (-).

  digits A contiguous sequence of digits (0-9). At least one digit must be present.

  e_char An exponent character, which is either *E* or *e*.

  e_sign An optional plus sign (+) or minus sign (-) for the exponent.

  e_digits
  A contiguous sequence of digits (0-9) for the exponent. At least one digit must be present if the string contains an exponent character.

  After the sequence of digits, or the optional characters representing an exponent, the string can contain other characters that are not digits, but the conversion stops as soon as the first of these characters is reached. The string is assumed to represent a decimal floating point number with an exponent that is a power of 10.

  If the string is not in the correct format, WebSphere MQ classes for JMS throws a NumberFormatException exception.
- When converting a numeric value (including a value with data type `byte`) to a string, the value is converted to the string representation of the value as a decimal number, not the string containing the ASCII character for that value. For example, the integer 65 is converted to the string "65", not the string "A".

Setting more than one property in a single call

The JmsPropertyContext interface also contains the setBatchProperties() method, which an application can use to set more than one property in a single call. The parameter of the method is a Map object that encapsulates a set of property name-value pairs.

For example, the following code uses the setBatchProperties() method to set the same five properties of a connection factory as shown in "Setting the properties of WebSphere MQ classes for JMS objects" on page 103. The code creates an instance of the HashMap class, which implements the Map interface.

```
HashMap batchProperties = new HashMap();
batchProperties.put(WMQConstants.WMQ_CONNECTION_MODE,
                    new Integer(WMQConstants.WMQ_CM_CLIENT));
batchProperties.put(WMQConstants.WMQ_QUEUE_MANAGER, "QM1");
batchProperties.put(WMQConstants.WMQ_WMQ_HOST_NAME, "HOST1");
batchProperties.put(WMQConstants.WMQ_PORT, "1414");
batchProperties.put(WMQConstants.WMQ_CHANNEL, "QM1.SVR");
factory.setBatchProperties(batchProperties);
```

Note that the second parameter of the Map.put() method must be an object. Therefore a property value with a primitive data type must be encapsulated within an object or represented by a string, as shown in the example.

The setBatchProperties() method validates each property. If the setBatchProperties() method cannot set a property because, for example, its value is not valid, none of the specified properties are set.

Property names and values

If an application uses the methods of the JmsPropertyContext interface to set and get the properties of WebSphere MQ classes for JMS objects, the application can specify the names and values of properties in any of the following ways. Each of the accompanying examples shows how to set the PRIORITY property of the JmsQueue object q1 so that a message sent to the queue has the priority specified on the send() call.

Using the property names and values that are defined as constants in the com.ibm.msg.client.wmq.WMQConstants interface

The following statement is an example of how to specify the names and values of properties in this way:

```
q1.setIntProperty(WMQConstants.WMQ_PRIORITY, WMQConstants.WMQ_PRI_APP);
```

Using the property names and values that can be used in queue and topic uniform resource identifiers (URIs)

The following statement is an example of how to specify the names and values of properties in this way:

```
q1.setIntProperty("priority", -2);
```

Only the names and values of properties of destinations can be specified in this way.

Using the property names and values that are recognized by the WebSphere MQ JMS administration tool

The following statement is an example of how to specify the names and values of properties in this way:

```
q1.setStringProperty("PRIORITY", "APP");
```

The short form of the property name is also acceptable, as shown in the following statement:

```
q1.setStringProperty("PRI", "APP");
```

When an application gets a property, the value returned depends on the way in which the application specifies the name of the property. For example, if an application specifies the constant WMQConstants.WMQ_PRIORITY as the property name, the value returned is the integer -2:

```
int n1 = getIntProperty(WMQConstants.WMQ_PRIORITY);
```

The same value is returned if the application specifies the string ″priority″ as the property name:

```
int n2 = getIntProperty("priority");
```

However, if the application specifies the string ″PRIORITY″ or ″PRI″ as the property name, the value returned is the string ″APP″:

```
String s1 = getStringProperty("PRI");
```

Internally, WebSphere MQ classes for JMS stores property names and values as the literal values defined in the com.ibm.msg.client.wmq.WMQConstants interface. This is the defined canonical format for property names and values. As a general rule, if an application sets properties using one of the other two ways of specifying property names and values, WebSphere MQ classes for JMS has to convert the names and values from the specified input format into the canonical format. Similarly, if an application gets properties using one of the other two ways of specifying property names and values, WebSphere MQ classes for JMS must convert the names from the specified input format into the canonical format, and convert the values from the canonical format into the required output format. Having to perform these conversions might have implications for performance.

Property names and values returned by exceptions, in trace files, or in the WebSphere MQ classes for JMS log are always in the canonical format.

Using the Map interface

The JmsPropertyContext interface extends the java.util.Map interface. An application can therefore use the methods of the Map interface to access the properties of a WebSphere MQ classes for JMS object.

For example, the following code prints out the names and values of all the properties of a connection factory. The code uses only the methods of the Map interface to get the names and values of the properties.

```
// Get the names of all the properties
Set propNames = factory.keySet();

// Loop round all the property names and get the property values
Iterator iterator = propNames.iterator();
while (iterator.hasNext()){
    String pName = (String)iterator.next();
    System.out.println(pName+"="+factory.get(pName));
}
```

Using the methods of the Map interface does not bypass any property validations or conversions.

Using the WebSphere MQ JMS extensions

WebSphere MQ classes for JMS contains a set of extensions to the JMS API called the WebSphere MQ JMS extensions. An application can use these extensions to create connection factories and destinations dynamically at run time, and to set the properties of connection factories and destinations.

WebSphere MQ classes for JMS contains a set of classes in the packages com.ibm.jms and com.ibm.mq.jms. These classes implement the JMS interfaces and contain the WebSphere MQ JMS extensions. The examples of code that follow assume that these packages have been imported by the following statements:

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

An application can use the WebSphere MQ JMS extensions to perform the following functions:

- Create connection factories and destinations dynamically at run time, instead of retrieving them as administered objects from a Java Naming and Directory Interface (JNDI) namespace
- Set the properties of connection factories and destinations

Creating connection factories

To create a connection factory, an application can use the MQConnectionFactory constructor, as shown in the following example:

```
MQConnectionFactory factory = new MQConnectionFactory();
```

This statement creates an MQConnectionFactory object with the default values for all its properties, which means that the application connects to the default queue manager in bindings mode. If you want an application to connect in client mode, or connect to a queue manager other than the default queue manager, the application must set the appropriate properties of the MQConnectionFactory object before creating the connection. For information about how to do this, see "Setting the properties of connection factories."

An application can create connection factories of the following types in a similar way:

- MQQueueConnectionFactory
- MQTopicConnectionFactory
- MQXAConnectionFactory
- MQXAQueueConnectionFactory
- MQXATopicConnectionFactory

Setting the properties of connection factories

An application can set the properties of a connection factory by calling the appropriate methods of the connection factory. The connection factory can either be an administered object or an object created dynamically at run time.

Consider the following code, for example:

```
MQConnectionFactory factory = new MQConnectionFactory();
.
factory.setTransportType(JMSC.MQJMS_TP_CLIENT_MQ_TCPIP);
factory.setQueueManager("QM1");
factory.setHostName("HOST1");
factory.setPort(1415);
factory.setChannel("QM1.SVR");
```

This code creates an MQConnectionFactory object and then sets five properties of the object. The effect of setting these properties is that the application connects to queue manager QM1 in client mode using an MQI channel called QM1.SVR. The queue manager is running on a system with host name HOST1, and the listener for the queue manager is listening in port number 1415.

For a real-time connection to a broker, an application can use the following code:

```
MQConnectionFactory factory = new MQConnectionFactory();
.
factory.setTransportType(JMSC.MQJMS_TP_DIRECT_TCPIP);
factory.setHostName("HOST2");
factory.setPort(1507);
```

This code assumes that the broker is running on a system with host name HOST2 and listening on port number 1507.

An application that uses a real-time connection to a broker can use only the publish/subscribe style of messaging. It cannot use the point-to-point style of messaging.

Only certain combinations of properties of a connection factory are valid. For information about which combinations are valid, see "Property dependencies" on page 209.

For more information about the properties of a connection factory, and the methods used to set its properties, see "Properties of WebSphere MQ classes for JMS objects" on page 176.

Creating destinations

To create a Queue object, an application can use the MQQueue constructor, as shown in the following example:

```
MQQueue q1 = new MQQueue("Q1");
```

This statement creates an MQQueue object with the default values for all its properties. The object represents a WebSphere MQ queue called Q1 that belongs to the local queue manager. This queue can be a local queue, an alias queue, or a remote queue definition.

An alternative form of the MQQueue constructor has two parameters, as shown in the following example:

```
MQQueue q2 = new MQQueue("QM2", "Q2");
```

The MQQueue object created by this statement represents a WebSphere MQ queue called Q2 that is owned by queue manager QM2. The queue manager identified in this way can be the local queue manager or a remote queue manager. If it is a remote queue manager, WebSphere MQ must be configured so that, when the application sends a message to this destination, Websphere MQ can route the message from the local queue manager to the remote queue manager.

The MQQueue constructor can also accept a queue uniform resource identifier (URI) as a single parameter. A queue URI is a string that specifies the name of a WebSphere MQ queue and, optionally, the name of the queue manager that owns the queue, and one or more properties of the MQQueue object. The following statement contains an example of a queue URI:

```
MQQueue q3 = new MQQueue("queue://QM3/Q3?persistence=2&priority=5");
```

The MQQueue object created by this statement represents a WebSphere MQ queue called Q3 that is owned by queue manager QM3, and all messages sent to this destination are persistent and have a priority of 5. For more information about queue URIs, see "Uniform resource identifiers (URIs)" on page 116. For an alternative way of setting the properties of an MQQueue object, see "Setting the properties of destinations" on page 111.

To create a Topic object, an application can use the MQTopic constructor, as shown in the following example:

```
MQTopic t1 = new MQTopic("Sport/Football/Results");
```

This statement creates an MQTopic object with the default values for all its properties. The object represents a topic called Sport/Football/Results.

The MQTopic constructor can also accept a topic URI as a parameter. A topic URI is a string that specifies the name of a topic and, optionally, one or more properties of the MQTopic object. The following statement contains an example of a topic URI:

```
MQTopic t2 = new MQTopic("topic://Sport/Tennis/Results?persistence=1&priority=0");
```

The MQTopic object created by this statement represents a topic called Sport/Tennis/Results, and all messages sent to this destination are nonpersistent and have a priority of 0. For more information about topic URIs, see "Uniform resource identifiers (URIs)" on page 116. For an alternative way of setting the properties of an MQTopic object, see "Setting the properties of destinations."

Setting the properties of destinations

An application can set the properties of a destination by calling the appropriate methods of the destination. The destination can either be an administered object or an object created dynamically at run time.

Consider the following code, for example:

```
MQQueue q1 = new MQQueue("Q1");
.
q1.setPersistence(JMSC.MQJMS_PER_PER);
q1.setPriority(5);
```

This code creates an MQQueue object and then sets two properties of the object. The effect of setting these properties is that all messages sent to the destination are persistent and have a priority of 5.

An application can set the properties of MQTopic object in a similar way, as shown in the following example:

```
MQTopic t1 = new MQTopic("Sport/Football/Results");
.
t1.setPersistence(JMSC.MQJMS_PER_NON);
t1.setPriority(0);
```

This code creates an MQTopic object and then sets two properties of the object. The effect of setting these properties is that all messages sent to the destination are nonpersistent and have a priority of 0.

For more information about the properties of a destination, and the methods used to set its properties, see "Properties of WebSphere MQ classes for JMS objects" on page 176.

Building a connection in a JMS application

To build a connection, a JMS application uses a ConnectionFactory object to create a Connection object and then starts the connection.

To create a Connection object, an application uses the createConnection() method of a ConnectionFactory object, as shown in the following example:

```
ConnectionFactory factory;
Connection connection;
.
.
.
connection = factory.createConnection();
```

The QueueConnectionFactory interface and the TopicConnectionFactory interface each inherits the createConnection() method from the ConnectionFactory interface. You can therefore use the createConnection() method to create a domain specific object, as shown in the following example:

```
QueueConnectionFactory qcf;
Connection connection;
.
.
.
connection = qcf.createConnection();
```

This fragment of code creates a QueueConnection object. An application can now perform a domain independent operation on this object, or an operation that is applicable only to the point-to-point domain. However, if the application attempts to perform an operation that is applicable only to the publish/subscribe domain, an IllegalStateException exception is thrown with the following message:

```
JMSMQ1112: Operation for a domain specific object was not valid.
           Operation createProducer() is not valid for type com.ibm.mq.jms.MQTopic
```

This is because the connection was created from a domain specific connection factory.

Note: Note that the application process ID is used as the default user identity to be passed to the queue manager. If the application is running in client transport mode then this process ID must exist, with the relevant authorizations, on the server. If you want a different identity to be used, then use the createConnection(username, password) method.

The JMS specification states that a connection is created in the `stopped` state. Until a connection starts, a message consumer that is associated with the connection cannot receive any messages. To start a connection, an application uses the start() method of a Connection object, as shown in the following example:

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

Creating a session in a JMS application

To create a session, a JMS application uses the createSession() method of a Connection object.

The createSession() method has two parameters:

1. A parameter that specifies whether the session is transacted or not transacted
2. A parameter that specifies the acknowledgement mode for the session

For example, the following code creates a session that is not transacted and has an acknowledgement mode of AUTO_ACKNOWLEDGE:

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

A Session object, and any MessageProducer or MessageConsumer object created from it, cannot be used concurrently by different threads of a multithreaded application. The simplest way of ensuring that these objects are not used concurrently is to create a separate Session object for each thread.

Transacted sessions

JMS applications can run local transactions by first creating a transacted session. An application can commit or roll back a transaction.

JMS applications can run local transactions. A local transaction is a transaction that involves changes only to the resources of the queue manager to which the application is connected.To run local transactions, an application must first create a transacted session by calling the createSession() method of a Connection object, specifying as a parameter that the session is transacted. Subsequently, all messages sent and received within the session are grouped into a sequence of transactions. A transaction ends when the application commits or rolls back the messages it has sent and received since the transaction began.

To commit a transaction, an application calls the commit() method of the Session object. When a transaction is committed, all messages sent within the transaction become available for delivery to other applications, and all messages received within the transaction are acknowledged so that the messaging server does not attempt to deliver them to the application again. In the point-to-point domain, the messaging server also removes the received messages from their queues.

To roll back a transaction, an application calls the rollback() method of the Session object. When a transaction is rolled back, all messages sent within the transaction are discarded by the messaging server, and all messages received within the transaction become available for delivery again. In the point-to-point domain, the messages that were received are put back on their queues and become visible to other applications again.

A new transaction starts automatically when an application creates a transacted session or calls the commit() or rollback() method. Therefore, a transacted session always has an active transaction.

When an application closes a transacted session, an implicit rollback occurs. When an application closes a connection, an implicit rollback occurs for all the connection's transacted sessions.

A transaction is wholly contained within a transacted session. A transaction cannot span sessions. This means that it is not possible for an application to send and receive messages in two or more transacted sessions and then commit or roll back all these actions as a single transaction.

Acknowledgement modes

Every session that is not transacted has an acknowledgement mode that determines how messages received by the application are acknowledged. Three acknowledgement modes are available, and the choice of acknowledgement mode affects the design of the application.

If a session is not transacted, the way that messages received by the application are acknowledged is determined by the acknowledgement mode of the session. The three acknowledgement modes are described in the following paragraphs:

AUTO_ACKNOWLEDGE

The session automatically acknowledges each message received by the application.

If messages are delivered synchronously to the application, the session acknowledges receipt of a message every time a Receive call completes successfully. If messages are delivered asynchronously, the session acknowledges receipt of a message every time a call to the onMessage() method of a message listener completes successfully.

If the application receives a message successfully, but a failure prevents acknowledgement from occurring, the message becomes available for delivery again. The application must therefore be able to handle a message that is re-delivered.

DUPS_OK_ACKNOWLEDGE

The session acknowledges the messages received by the application at times it selects.

Using this acknowledgement mode reduces the amount of work the session must do, but a failure that prevents message acknowledgement might result in more than one message becoming available for delivery again. The application must therefore be able to handle messages that are re-delivered.

Restriction: In AUTO_ACKNOWLEDGE and DUPS_OK_ACKNOWLEDGE modes, JMS does not support an application throwing an unhandled exception in a message listener. This means that messages are always acknowledged when the message listener returns, regardless of whether it was processed successfully (provided any failures are non-fatal and do not prevent the application from continuing). If you require finer control of message acknowledgement, use the CLIENT_ACKNOWLEDGE or transacted modes, which give the application full control of the acknowledgement functions.

CLIENT_ACKNOWLEDGE

The application acknowledges the messages it receives by calling the Acknowledge method of the Message class.

The application can acknowledge the receipt of each message individually, or it can receive a batch of messages and call the Acknowledge method only for the last message it receives. When the Acknowledge method is called all messages received since the last time the method was called are acknowledged.

In conjunction with any of these acknowledgement modes, an application can stop and restart the delivery of messages in a session by calling the Recover method of the Session class. Messages whose receipt was previously unacknowledged are re-delivered. However, they might not be delivered in the same sequence in which they were previously delivered. In the meantime, higher priority messages might have arrived, and some of the original messages might have expired. In the point-to-point domain, some of the original messages might have been consumed by another application.

An application can determine whether a message is being re-delivered by examining the contents of the JMSRedelivered header field of the message. The application does this by calling the Get JMSRedelivered method of the Message class.

Creating destinations in a JMS application

Instead of retrieving destinations as administered objects from a Java Naming and Directory Interface (JNDI) namespace, a JMS application can use a session to create destinations dynamically at run time. An application can use a uniform resource identifier (URI) to identify a WebSphere MQ queue or a topic and, optionally, to specify one or more properties of a Queue or Topic object.

Using a session to create Queue objects

To create a Queue object, an application can use the createQueue() method of a Session object, as shown in the following example:

```
Session session;
.
Queue q1 = session.createQueue("Q1");
```

This code creates a Queue object with the default values for all its properties. The object represents a WebSphere MQ queue called Q1 that belongs to the local queue manager. This queue can be a local queue, an alias queue, or a remote queue definition.

The createQueue() method also accepts a queue URI as a parameter. A queue URI is a string that specifies the name of a WebSphere MQ queue and, optionally, the name of the queue manager that owns the queue and one or more properties of the Queue object. The following statement contains an example of a queue URI:

```
Queue q2 = session.createQueue("queue://QM2/Q2?persistence=2&priority=5");
```

The Queue object created by this statement represents a WebSphere MQ queue called Q2 that is owned by a queue manager called QM2, and all messages sent to this destination are persistent and have a priority of 5. The queue manager identified in this way can be the local queue manager or a remote queue manager. If it is a remote queue manager, WebSphere MQ must be configured so that, when the application sends a message to this destination, Websphere MQ can route the message from the local queue manager to queue manager QM2. For more information about URIs, see “Uniform resource identifiers (URIs)” on page 116.

Note that the parameter on the createQueue() method contains provider specific information. Therefore, using the createQueue() method to create a Queue object, instead of retrieving a Queue object as an administered object from a JNDI namespace, might make your application less portable.

An application can create a TemporaryQueue object by using the createTemporaryQueue() method of a Session object, as shown in the following example:

```
TemporaryQueue q3 = session.createTemporaryQueue();
```

Although a session is used to create a temporary queue, the scope of a temporary queue is the connection that was used to create the session. Any of the connection’s sessions can create message producers and message consumers for the temporary queue. The temporary queue remains until the connection ends or the application explicitly deletes the temporary queue by using the TemporaryQueue.delete() method, whichever is the sooner.

When an application creates a temporary queue, WebSphere MQ classes for JMS creates a dynamic queue in the queue manager to which the application is connected. The TEMPMODEL property of the connection factory specifies the name of the model queue that is used to create the dynamic queue, and the TEMPQPREFIX property of the connection factory specifies the prefix that is used to form the name of the dynamic queue.

Using a session to create Topic objects

To create a Topic object, an application can use the createTopic() method of a Session object, as shown in the following example:

```
Session session;
.
Topic t1 = session.createTopic("Sport/Football/Results");
```

This code creates an Topic object with the default values for all its properties. The object represents a topic called Sport/Football/Results.

The createTopic() method also accepts a topic URI as a parameter. A topic URI is a string that specifies the name of a topic and, optionally, one or more properties of the Topic object. The following code contains an example of a topic URI:

```
String uri = "topic://Sport/Tennis/Results?persistence=1&priority=0";
Topic t2 = session.createTopic(uri);
```

The Topic object created by this code represents a topic called Sport/Tennis/Results, and all messages sent to this destination are nonpersistent and have a priority of 0. For more information about topic URIs, see "Uniform resource identifiers (URIs)."

Note that the parameter on the createTopic() method contains provider specific information. Therefore, using the createTopic() method to create a Topic object, instead of retrieving a Topic object as an administered object from a JNDI namespace, might make your application less portable.

An application can create a TemporaryTopic object by using the createTemporaryTopic() method of a Session object, as shown in the following example:

```
TemporaryTopic t3 = session.createTemporaryTopic();
```

Although a session is used to create a temporary topic, the scope of a temporary topic is the connection that was used to create the session. Any of the connection's sessions can create message producers and message consumers for the temporary topic. The temporary topic remains until the connection ends or the application explicitly deletes the temporary topic by using the TemporaryTopic.delete() method, whichever is the sooner.

When an application creates a temporary topic, WebSphere MQ classes for JMS creates a topic whose name commences with the characters TEMP/*tempTopicPrefix*, where *tempTopicPrefix* is the value of the TEMPTOPICPREFIX property of the connection factory.

Uniform resource identifiers (URIs)

A queue URI is a string that specifies the name of a WebSphere MQ queue and, optionally, the name of the queue manager that owns the queue and one or more properties of the Queue object created by the application. A topic URI is a string that specifies the name of a topic and, optionally, one or more properties of the Topic object created by the application.

A queue URI has the following format:

```
queue://[qMgrName]/qName[?propertyName1=propertyValue1
                         &propertyName2=propertyValue2
                         &...]
```

A topic URI has the following format:

```
topic://topicName[?propertyName1=propertyValue1
                  &propertyName2=propertyValue2
                  &...]
```

The variables in these formats have the following meanings:

qMgrName
: The name of the queue manager that owns the queue identified by the URI.

  The queue manager can the local queue manager or a remote queue manager. If it is a remote queue manager, WebSphere MQ must be configured so that, when an application sends a message to the queue, Websphere MQ can route the message from the local queue manager to the remote queue manager.

  If no name is specified, the local queue manager is assumed.

qName
: The name of the WebSphere MQ queue.

  The queue can be a local queue, an alias queue, or a remote queue definition.

  For the rules for creating queue names, see the section ″Rules for naming WebSphere MQ objects″ in *WebSphere MQ Intercommunication*.

topicName
: The name of the topic.

  For the rules for creating topic names, see the section ″Rules for naming WebSphere MQ objects″ in *WebSphere MQ Intercommunication*. Avoid the use of the wildcard characters +, #, *, and ? in topic names. Topic names containing these characters can cause unexpected results when you subscribe to them. See the section ″Using topic strings″ in *WebSphere MQ Application Programming Reference*.

propertyName1, *propertyName2*, **...**
: The names of the properties of the Queue or Topic object created by the application. Table 31 lists the valid property names that can be used in a URI.

  If no properties are specified, the Queue or Topic object has the default values for all its properties.

propertyValue1, *propertyValue2*, **...**
: The values of the properties of the Queue or Topic object created by the application. Table 31 lists the valid property values that can be used in a URI.

Brackets ([]) denotes an optional component, and the ellipsis (...) means that the list of property name-value pairs, if present, can contain one or more name-value pairs.

Table 31 lists the valid property names and valid values that can be used in queue and topic URIs. Although the WebSphere MQ JMS administration tool uses symbolic constants for the values of properties, URIs cannot contain symbolic constants.

Table 31. Property names and valid values for use in queue and topic URIs

Property name	Description	Valid values
CCSID	How the character data in the body of a message is represented when WebSphere MQ classes for JMS forwards the message to the destination	• Any coded character set identifier supported by WebSphere MQ.

Table 31. Property names and valid values for use in queue and topic URIs (continued)

Property name	Description	Valid values
encoding	How the numerical data in the body of a message is represented when WebSphere MQ classes for JMS forwards the message to the destination	• Any valid value for the *Encoding* field in a WebSphere MQ message descriptor.
expiry	The time to live for messages sent to the destination	• -2 - As specified on the send() call or, if not specified on the send() call, the default time to live of the message producer. • 0 - A message sent to the destination never expires. • A positive integer specifying the time to live in milliseconds.
multicast	The multicast setting for a topic when using a real-time connection to a broker	The following list contains the valid values. Associated with each value is the corresponding value of the MULTICAST property as used in the WebSphere MQ JMS administration tool. For a description of the MULTICAST property and its valid values, see "Properties of WebSphere MQ classes for JMS objects" on page 176. • -1 - ASCF • 0 - DISABLED • 3 - NOTR • 5 - RELIABLE • 7 - ENABLED
persistence	The persistence of messages sent to the destination	• -2 - As specified on the send() call or, if not specified on the send() call, the default persistence of the message producer. • -1 - As specified by the *DefPersistence* attribute of the WebSphere MQ queue or topic. • 1 - Nonpersistent. • 2 - Persistent. • 3 - Equivalent to the value HIGH for the PERSISTENCE property as used in the WebSphere MQ JMS administration tool. For an explanation of this value, see "JMS persistent messages" on page 140.

Table 31. Property names and valid values for use in queue and topic URIs (continued)

Property name	Description	Valid values
priority	The priority of messages sent to the destination	• -2 - As specified on the send() call or, if not specified on the send() call, the default priority of the message producer. • -1 - As specified by the *DefPriority* attribute of the WebSphere MQ queue or topic. • An integer in the range 0-9 specifying the priority of messages sent to the destination.
targetClient	Whether messages sent to the destination contain an MQRFH2 header	• 0 - Messages contain an MQRFH2 header. • 1 - Messages do not contain an MQRFH2 header.

For example, the following URI identifies a WebSphere MQ queue called Q1 that is owned by the local queue manager. A Queue object created using this URI has the default values for all its properties.

```
queue:///Q1
```

The following URI identifies a WebSphere MQ queue called Q2 that is owned by a queue manager called QM2. All messages sent to this destination have a priority of 6. The remaining properties of the Queue object created using this URI have their default values.

```
queue://QM2/Q2?priority=6
```

The following URI identifies a topic called Sport/Athletics/Results. All messages sent to this destination are nonpersistent and have a priority of 0. The remaining properties of the Topic object created using this URI have their default values.

```
topic://Sport/Athletics/Results?persistence=1&priority=0
```

Sending messages in a JMS application

Before a JMS application can send messages to a destination, it must first create a MessageProducer object for the destination. To send a message to the destination, the application creates a Message object and then calls the send() method of the MessageProducer object.

An application uses a MessageProducer object to send messages. An application normally creates a MessageProducer object for a specific destination, which can be a queue or a topic, so that all messages sent using the message producer are sent to the same destination. Therefore, before an application can create a MessageProducer object, it must first create a Queue or Topic object. For information about how to create a Queue or Topic object, see the following topics:

- “Using JNDI to retrieve administered objects in a JMS application” on page 100
- “Using the IBM JMS extensions” on page 101
- “Using the WebSphere MQ JMS extensions” on page 108
- “Creating destinations in a JMS application” on page 114

To create a MessageProducer object, an application uses the createProducer() method of a Session object, as shown in the following example:

```
MessageProducer producer = session.createProducer(destination);
```

The parameter `destination` is a Queue or Topic object that the application has created previously.

Before an application can send a message, it must create a Message object. The body of a message contains the application data, and JMS defines five types of message body:

- Bytes
- Map
- Object
- Stream
- Text

Each type of message body has its own JMS interface, which is a sub-interface of the Message interface, and a method in the Session interface for creating a message with that type of body. For example, the interface for a text message is called TextMessage, and an application uses the createTextMessage() method of a Session object to create a text message, as shown in the following statement:

```
TextMessage outMessage = session.createTextMessage(outString);
```

For more information about messages and message bodies, see "JMS messages" on page 79.

To send a message, an application uses the send() method of a MessageProducer object, as shown in the following example:

```
producer.send(outMessage);
```

An application can use the send() method to send messages in either messaging domain. The nature of the destination determines which messaging domain is used. However, TopicPublisher, the sub-interface of MessageProducer that is specific to the publish/subscribe domain, also has a publish() method, which can be used instead of the send() method. The two methods are functionally the same.

An application can create a MessageProducer object with no specified destination. In this case, the application must specify the destination when calling the send() method.

If an application sends a message within a transaction, the message is not delivered to its destination until the transaction is committed. This means that an application cannot send a message and receive a reply to the message within the same transaction.

A destination can be configured so that when an application sends messages to it, WebSphere MQ classes for JMS forwards the message and returns control back to the application without determining whether the queue manager has received the message safely. This is sometimes referred to as *asynchronous put*. For more information, see "Putting messages asynchronously in Websphere MQ classes for JMS" on page 154.

Receiving messages in a JMS application

An application uses a message consumer to receive messages. A durable topic subscriber is a message consumer that receives all messages sent to a destination, including those sent while the consumer is inactive. An application can select which messages it wants to receive by using a message selector, and can receive messages asynchronously by using a message listener.

An application uses a MessageConsumer object to receive messages. An application creates a MessageConsumer object for a specific destination, which can be a queue or a topic, so that all messages received using the message consumer are received from the same destination. Therefore, before an application can create a MessageConsumer object, it must first create a Queue or Topic object. For information about how to create a Queue or Topic object, see the following topics:

- "Using JNDI to retrieve administered objects in a JMS application" on page 100
- "Using the IBM JMS extensions" on page 101
- "Using the WebSphere MQ JMS extensions" on page 108
- "Creating destinations in a JMS application" on page 114

To create a MessageConsumer object, an application uses the createConsumer() method of a Session object, as shown in the following example:

```
MessageConsumer consumer = session.createConsumer(destination);
```

The parameter `destination` is a Queue or Topic object that the application has created previously.

The application then uses the receive() method of the MessageConsumer object to receive a message from the destination, as shown in the following example:

```
Message inMessage = consumer.receive(1000);
```

The parameter on the receive() call specifies how long in milliseconds the method waits for a suitable message to arrive if no message is available immediately. If you omit this parameter, the call blocks indefinitely until a suitable message arrives. If you do not want the application to wait for a message, use the receiveNoWait() method instead.

The receive() method returns a message of a specific type. For example, when an application receives a text message, the object returned by the receive() call is a TextMessage object.

However, the declared type of object returned by a receive() call is a Message object. Therefore, in order to extract the data from the body of a message that has just been received, the application must cast from the Message class to the more specific subclass, such as TextMessage. If the type of the message is not known, the application can use the `instanceof` operator to determine the type. It is always good practice for an application to determine the type of a message before casting so that errors can be handled gracefully.

The following code uses the `instanceof` operator and shows how to extract the data from the body of a text message:

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

If an application sends a message within a transaction, the message is not delivered to its destination until the transaction is committed. This means that an application cannot send a message and receive a reply to the message within the same transaction.

If a message consumer receives messages from a destination that is configured for read ahead, any nonpersistent messages that are in the read ahead buffer when the application ends are discarded.

In the publish/subscribe domain, JMS identifies two types of message consumer, nondurable topic subscriber and durable topic subscriber, which are described in the following two sections.

Nondurable topic subscribers

A nondurable topic subscriber receives only those messages that are published while the subscriber is active. A nondurable subscription starts when an application creates a nondurable topic subscriber and ends when the application closes the subscriber, or when the subscriber falls out of scope. As an extension in WebSphere MQ classes for JMS, a nondurable topic subscriber also receives retained publications, but not when using a real-time connection to a broker.

To create a nondurable topic subscriber, an application can use the domain independent createConsumer() method, specifying a Topic object as the destination. Alternatively, an application can use the domain specific createSubscriber() method, as shown in the following example:

```
TopicSubscriber subscriber = session.createSubscriber(topic);
```

The parameter `topic` is a Topic object that the application has created previously.

Durable topic subscribers

Restriction: An application cannot create durable topic subscribers when using a real-time connection to a broker.

A durable topic subscriber receives all messages that are published during the life of a durable subscription. These messages include all those that are published while the subscriber is not active. As an extension in WebSphere MQ classes for JMS, a durable topic subscriber also receives retained publications.

To create a durable topic subscriber, an application uses the createDurableSubscriber() method of a Session object, as shown in the following example:

```
TopicSubscriber subscriber = session.createDurableSubscriber(topic, "D_SUB_000001");
```

On the createDurableSubscriber() call, the first parameter is a Topic object that the application has created previously, and the second parameter is a name that is used to identify the durable subscription.

The session used to create a durable topic subscriber must have an associated client identifier. The client identifier associated with a session is the same as the client identifier for the connection that is used to create the session. The client identifier can be specified by setting the CLIENTID property of the ConnectionFactory object. Alternatively, an application can specify the client identifier by calling the setClientID() method of the Connection object.

The name that is used to identify a durable subscription must be unique only within the client identifier, and therefore the client identifier forms part of the full, unique identifier of a durable subscription. To continue using a durable subscription that was created previously, an application must create a durable topic subscriber using a session with the same client identifier as that associated with the durable subscription, and using the same subscription name.

A durable subscription starts when an application creates a durable topic subscriber using a client identifier and subscription name for which no durable subscription currently exists. However, a durable subscription does not end when the application closes the durable topic subscriber. To end a durable subscription, an application must call the unsubscribe() method of a Session object that has the same client identifier as that associated with the durable subscription. The parameter on the unsubscribe() call is the subscription name, as shown in the following example:

```
session.unsubscribe("D_SUB_000001");
```

The scope of a durable subscription is a queue manager. If a durable subscription exists on one queue manager, and an application connected to another queue manager creates a durable subscription with the same client identifier and subscription name, the two durable subscriptions are completely independent.

Message selectors

An application can specify that only those messages that satisfy certain criteria are returned by successive receive() calls. When creating a MessageConsumer object, the application can specify a Structured Query Language (SQL) expression that determines which messages are retrieved. This SQL expression is called a *message selector*. The message selector can contain the names of JMS message header fields and message properties. For information about how to construct a message selector, see “Message selectors” on page 80.

The following example shows how an application can select messages based on a user defined property called myProp:

```
MessageConsumer consumer;
.
consumer = session.createConsumer(destination, "myProp = 'blue'");
```

The JMS specification does not allow an application to change the message selector of a message consumer. After an application creates a message consumer with a message selector, the message selector remains for the life of that consumer. If an application requires more than one message selector, the application must create a message consumer for each message selector.

Suppressing local publications

An application can create a message consumer that ignores publications published on the consumer’s own connection. The application does this by setting the third parameter on a createConsumer() call to `true`, as shown in the following example:

```
MessageConsumer consumer = session.createConsumer(topic, null, true);
```

On a createDurableSubscriber() call, the application does this by setting the fourth parameter to `true`, as shown in the following example

```
String selector = "company = 'IBM'";
TopicSubscriber subscriber = session.createDurableSubscriber(topic, "D_SUB_000001",
                                                             selector, true);
```

Asynchronous delivery of messages

An application can receive messages asynchronously by registering a message listener with a message consumer. The message listener has a method called onMessage, which is called asynchronously when a suitable message is available and whose purpose is to process the message. The following code illustrates the mechanism:

```
import javax.jms.*;

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

    // The code to process the message
    .
    .
    .
  }
}
.
.
.
// Main program (possibly in another class)
.
// Creating the message listener
MyClass listener = new MyClass();

// Registering the message listener with a message consumer
consumer.setMessageListener(listener);

// The main program now continues with other processing
```

An application can use a session either for receiving messages synchronously using receive() calls, or for receiving messages asynchronously using message listeners, but not for both. If an application needs to receive messages synchronously and asynchronously, it must create separate sessions.

Poison messages

A poison message is one which cannot be processed by a receiving MDB application. If a poison message is encountered, the JMS MessageConsumer object can requeue it according to two queue properties, BOQUEUE, and BOTHRESH.

Sometimes, a badly-formatted message arrives on a queue. In this context, badly-formatted means that the receiving application cannot process the message correctly. Such a message can cause the receiving application to fail and to back out this badly-formatted message. The message can then be repeatedly delivered to the input queue and repeatedly backed out by the application. These messages are known as *poison messages*. The JMS MessageConsumer object detects poison messages and reroutes them to an alternative destination.

The WebSphere MQ queue manager keeps a record of the number of times that each message has been backed out. When this number reaches a configurable threshold value, the message consumer requeues the message to a named backout queue. If this re-queuing fails for any reason, the message is removed from the input queue and either requeued to the dead-letter queue, or discarded. See "Removing messages from the queue in ASF" on page 163 for more details.

JMS ConnectionConsumer objects handle poison messages in the same way and using the same queue properties. If multiple connection consumers are monitoring the same queue, it is possible that the poison message may be delivered to an application more times than the threshold value before the requeue occurs. This behavior is due to the way individual connection consumers monitor queues and requeue poison messages.

The threshold value and the name of the back out queue are attributes of a WebSphere MQ queue. The names of the attributes are BackoutThreshold and BackoutRequeueQName. The queue they apply to is as follows:

- For point-to-point messaging, this is the underlying local queue. This is important when message consumers and connection consumers use queue aliases.
- For publish/subscribe messaging in WebSphere MQ messaging provider normal mode, it is the model queue from which the Topic's managed queue is created.
- For publish/subscribe messaging in WebSphere MQ messaging provider migration mode, it is the CCSUB queue defined on the TopicConnectionFactory object, or the CCDSUB queue defined on the Topic object.

To set the BackoutThreshold and BackoutRequeueQName attributes, issue the following MQSC command:

```
ALTER QLOCAL(your.queue.name) BOTHRESH(threshold value) BOQUEUE(your.backout.queue.name)
```

For publish/subscribe messaging, if your system creates a dynamic queue for each subscription, these attribute values are obtained from the WebSphere MQ classes for JMS model queue, SYSTEM.JMS.MODEL.QUEUE. To alter these settings, use:

```
ALTER QMODEL(SYSTEM.JMS.MODEL.QUEUE) BOTHRESH(threshold value) BOQUEUE(your.backout.queue.name)
```

If the backout threshold value is zero, poison message handling is disabled, and poison messages remain on the input queue. Otherwise, when the backout count reaches the threshold value, the message is sent to the named backout queue. If the backout count reaches the threshold value, but the message cannot go to the backout queue, the message is sent to the dead-letter queue or it is discarded. This situation occurs if the backout queue is not defined, or if the MessageConsumer object cannot send the message to the backout queue. See "Removing messages from the queue in ASF" on page 163 for further details.

Closing down a WebSphere MQ classes for JMS application

It is important for a WebSphere MQ classes for JMS application to close certain JMS objects explicitly before stopping. Finalizers might not be called, so do not rely on them to free resources. Do not allow an application to terminate with compressed trace active.

Garbage collection alone cannot release all WebSphere MQ classes for JMS and WebSphere MQ resources in a timely manner, especially if an application creates many short lived JMS objects at the session level or lower. It is therefore important for an application to close a Connection, Session, MessageConsumer, or MessageProducer object when it is no longer required.

Do not use finalizers in an application to close JMS objects. Because finalizers might not be called, resources might not be freed. When a Connection is closed it closes all the Sessions that were created from it. Similarly, the MessageConsumers and MessageProducers created from a Session are closed when the Session is closed. However, consider closing Sessions, MessageConsumers, and MessageProducers explicitly to ensure resources are freed in a timely manner.

If trace compression is activated, System.Halt() shutdowns and abnormal, uncontrolled JVM terminations are likely to result in a corrupt trace file. Where possible, turn off the trace facility when you have collected the trace information you need. If you are tracing an application up to an abnormal end, use uncompressed trace output.

Handling errors in WebSphere MQ classes for JMS

This topic describes the exceptions that can be thrown by JMS API calls or delivered to an exception handler, and how an application can obtain information from exceptions. It also tells you about the WebSphere MQ classes for JMS error log and the first failure support technology (FFST) information that is generated if a serious internal error occurs.

Exceptions in WebSphere MQ classes for JMS

A WebSphere MQ classes for JMS application must be able to handle exceptions that are thrown by a JMS API calls or delivered to an exception handler. Compared to previous versions, most exception messages and error codes have changed in Version 7 of WebSphere MQ classes for JMS. If your application parses or tests exception messages and error codes, it probably needs to be modified in order to use Version 7.

WebSphere MQ classes for JMS reports runtime problems by throwing exceptions. JMSException is the root class for exceptions thrown by JMS methods, and catching JMSException exceptions provides a generic way of handling all JMS related exceptions.

Every JMSException exception encapsulates the following information:

- A provider specific exception message, which an application obtains by calling the Throwable.getMessage() method.
- A provider specific error code, which an application obtains by calling the JMSException.getErrorCode() method.
- A linked exception. An exception thrown by a JMS API call is often the result of a lower level problem, which is reported by another exception linked to this exception. An application obtains a linked exception by calling the JMSException.getLinkedException() or the Throwable.getCause() method.

Most exceptions thrown by WebSphere MQ classes for JMS are instances of subclasses of JMSException. These subclasses implement the com.ibm.msg.client.jms.JmsExceptionDetail interface, which provides the following additional information:

- An explanation of the exception message, which an application obtains by calling the JmsExceptionDetail.getExplanation() method.
- A recommended user response to the exception, which an application obtains by calling the JmsExceptionDetail.getUserAction() method.
- The keys for the message inserts in the exception message. An application obtains an iterator for all the keys by calling the JmsExceptionDetail.getKeys() method.
- The message inserts in the exception message. For example, a message insert might be the name of the queue that caused the exception, and it might be

useful for an application to be able to access that name. An application obtains the message insert corresponding to a specified key by calling the JmsExceptionDetail.getValue() method.

All the methods in the JmsExceptionDetail interface might return null if no details are available.

For example, if an application tries to create a message producer for a WebSphere MQ queue that does not exist, an exception is thrown with the following information:

```
Message : JMSWMQ2008: Failed to open MQ queue 'Q_test'.
Class : class com.ibm.msg.client.jms.DetailedInvalidDestinationException
Error Code : JMSWMQ2008
Explanation : JMS attempted to perform an MQOPEN, but WebSphere MQ reported an
              error.
User Action : Use the linked exception to determine the cause of this error. Check
              that the specified queue and queue manager are defined correctly.
```

The exception thrown, com.ibm.msg.client.jms.DetailedInvalidDestinationException, is a subclass of javax.jms.InvalidDestinationException and implements the com.ibm.msg.client.jms.JmsExceptionDetail interface.

Linked exceptions

A linked exception provides further information about a runtime problem. Therefore, for each JMSException exception that is thrown, an application should check the linked exception. The linked exception itself might have another linked exception, and so the linked exceptions form a chain leading back to the original underlying problem. A linked exception is implemented by using the chained exception mechanism of the java.lang.Throwable class, and an application obtains a linked exception by calling the Throwable.getCause() method. For a JMSException exception, the getLinkedException() method actually delegates to the Throwable.getCause() method.

For example, if an application specifies an incorrect port number when connecting to a queue manager, the exceptions form the following chain:

```
com.ibm.msg.client.jms.DetailIllegalStateException
   |
   +--->com.ibm.mq.MQException
           |
           +--->com.ibm.mq.jmqi.JmqiException
                   |
                   +--->java.net.ConnectionException
```

Typically, each exception in a chain is thrown from a different layer in the code. For example, the exceptions in the preceding chain are thrown by the following layers:

- The first exception, an instance of a subclass of JMSException, is thrown by the common layer in WebSphere MQ classes for JMS.
- The next exception, an instance of com.ibm.mq.MQException, is thrown by the WebSphere MQ messaging provider.
- The next exception, an instance of com.ibm.mq.jmqi.JmqiException, is thrown by the common Java interface to the MQI.
- The final exception, an instance of java.net.ConnectionException, is thrown by the Java class library.

For more information about the layered architecture of WebSphere MQ classes for JMS, see "A layered architecture" on page 69.

Using code similar to the following code, an application can iterate through this chain to extract all the appropriate information:

```
import com.ibm.msg.client.jms.JmsExceptionDetail;
import com.ibm.mq.MQException;
import com.ibm.mq.jmqi.JmqiException;
import javax.jms.JMSException;
.
.
.
catch (JMSException je) {
    System.err.println("Caught JMSException");

    // Check for linked exceptions in JMSException
    Throwable t = je;
    while (t != null) {
        // Write out the message that is applicable to all exceptions
        System.err.println("Exception Msg: " + t.getMessage());
        // Write out the exception stack trace
        t.printStackTrace(System.err);

        // Add on specific information depending on the type of exception
        if (t instanceof JMSException) {
            JMSException je1 = (JMSException) t;
            System.err.println("JMS Error code: " + je1.getErrorCode());

            if (t instanceof JmsExceptionDetail){
                JmsExceptionDetail jed = (JmsExceptionDetail)je1;
                System.err.println("JMS Explanation: " + jed.getExplanation());
                System.err.println("JMS Explanation: " + jed.getUserAction());
            }
        } else if (t instanceof MQException) {
            MQException mqe = (MQException) t;
            System.err.println("WMQ Completion code: " + mqe.getCompCode());
            System.err.println("WMQ Reason code: " + mqe.getReason());
        } else if (t instanceof JmqiException){
            JmqiException jmqie = (JmqiException)t;
            System.err.println("WMQ Log Message: " + jmqie.getWmqLogMessage());
            System.err.println("WMQ Explanation: " + jmqie.getWmqMsgExplanation());
            System.err.println("WMQ Msg Summary: " + jmqie.getWmqMsgSummary());
            System.err.println("WMQ Msg User Response: "
                               + jmqie.getWmqMsgUserResponse());
            System.err.println("WMQ Msg Severity: " + jmqie.getWmqMsgSeverity());
        }

        // Get the next cause
        t = t.getCause();
    }
}
```

Note that an application should always check the type of each exception in a chain because the type of exception can vary and exceptions of different types encapsulate different information.

Obtaining WebSphere MQ specific information about a problem

Instances of com.ibm.mq.MQException and com.ibm.mq.jmqi.JmqiException encapsulate WebSphere MQ specific information about a problem.

An MQException exception encapsulates the following information:

- A completion code, which an application obtains by calling the getCompCode() method
- A reason code, which an application obtains by calling the getReason() method

A JmqiException exception also encapsulates a completion code and a reason code. Additionally, however, a JmqiException exception encapsulates the information in an AMQ*nnnn* or CSQ*nnnn* message, if one is associated with the exception. By calling the appropriate methods of the exception, an application can obtain the various components of this message, such as the severity, explanation, and user response.

For examples of how to use of the methods mentioned in this section, see the sample code in "Linked exceptions" on page 127.

Upgrading from previous versions of WebSphere MQ classes for JMS

Compared to previous versions of WebSphere MQ classes for JMS, most error codes and exception messages have changed in Version 7. The reason for these changes is that WebSphere MQ classes for JMS now has a layered architecture and exceptions are thrown from different layers in the code.

For example, if an application tries to connect to a queue manager that does not exist, a previous version of WebSphere MQ classes for JMS threw a JMSException exception with the following information:

```
MQJMS2005: Failed to create MQQueueManager for 'localhost:QM_test'.
```

This exception contained a linked MQException exception with the following information:

```
MQJE001: Completion Code 2, Reason 2058
```

By comparison in the same circumstances, Version 7 of WebSphere MQ classes for JMS throws a JMSException exception with the following information:

```
Message : JMSWMQ0018: Failed to connect to queue manager 'QM_test' with
          connection mode 'Client' and host name 'localhost'.
Class : class com.ibm.msg.client.jms.DetailedJMSException
Error Code : JMSWMQ0018
Explanation : null
User Action : Check the queue manager is started and if running in client mode,
              check there is a listener running. Please see the linked exception
              for more information.
```

This exception contains a linked MQException exception with the following information:

```
Message : JMSCMQ0001: WebSphere MQ call failed with compcode '2' ('MQCC_FAILED')
          reason '2058' ('MQRC_Q_MGR_NAME_ERROR').
Class : class com.ibm.mq.MQException
Completion Code : 2
Reason Code : 2058
```

If your application parses or tests exception messages returned by the Throwable.getMessage() method, or error codes returned by the JMSException.getErrorCode() method, your application probably needs to be modified in order to use Version 7 of WebSphere MQ classes for JMS.

Exception listeners

An application can register an exception listener with a Connection object. Subsequently, if a problem occurs that makes the connection unusable, WebSphere MQ classes for JMS delivers an exception to the exception listener by calling its onException() method. The application then has the opportunity to reestablish the connection.

By default, WebSphere MQ classes for JMS also delivers an exception to the exception listener if a problem occurs while trying to deliver a message asynchronously. However, this behavior is more than that required by the JMS specification. To be consistent with the JMS specification, you must ensure that the ASYNCEXCEPTION property of the connection factory is set to ASYNC_EXCEPTIONS_CONNECTIONBROKEN. Then, if a problem occurs while trying to deliver a message asynchronously, WebSphere MQ classes for JMS writes an exception to its log and does not deliver an exception to the exception listener.

For any other type of problem, a JMSException exception is thrown by the current JMS API call.

If an application does not register an exception listener with a Connection object, any exceptions that would have been delivered to the exception listener are written to the WebSphere MQ classes for JMS log.

Logging errors in WebSphere MQ classes for JMS

Information about runtime problems that might require corrective action by the user is written to the WebSphere MQ classes for JMS log.

For example, if an application attempts to set a property of a connection factory, but the name of the property is not recognized, WebSphere MQ classes for JMS writes information about the problem to its log.

By default, the file containing the log is called mqjms.log and is in the current working directory. However, you can change the name and location of the log file by setting the com.ibm.msg.client.commonservices.log.outputName property in the WebSphere MQ classes for JMS configuration file. For information about the WebSphere MQ classes for JMS configuration file, see "The WebSphere MQ classes for JMS configuration file" on page 12.

First failure support technology (FFST) in WebSphere MQ classes for JMS

If a serious internal error occurs within WebSphere MQ classes for JMS, first failure support technology (FFST) information is generated.

The FFST information is written to a file is called JMSC*nnnn*.FDC, where *nnnn* is a four digit number. This file is in a directory called FDC, which is a subdirectory of the directory to which trace output is written. By default, trace output is written to the current working directory, but you can redirect trace output to a different directory by setting the com.ibm.msg.client.commonservices.trace.outputName property in the WebSphere MQ classes for JMS configuration file. For information about the WebSphere MQ classes for JMS configuration file, see "The WebSphere MQ classes for JMS configuration file" on page 12.

If tracing is enabled when FFST information is generated, the FFST information is also be written to the trace file.

Accessing WebSphere MQ features from a WebSphere MQ classes for JMS application

WebSphere MQ classes for JMS provides facilities to exploit a number of features of WebSphere MQ.

Attention: These features are outside the JMS specification or, in certain cases, violate the JMS specification. If you use them, your application is unlikely to be compatible with other JMS providers. Those features which do not comply with the JMS specification are labelled with an Attention notice.

Reading and writing the message descriptor from a WebSphere MQ classes for JMS application

You control the ability to access the message descriptor (MQMD) by setting properties on a Destination and a Message.

The function described in this topic is not available for publish/subscribe messaging when you are either connecting to a WebSphere MQ V6 queue manager, or when the PROVIDERVERSION is ″6″.

Some WebSphere MQ applications require specific values to be set in the MQMD of messages sent to them. WebSphere MQ classes for JMS provides message attributes that allow JMS applications to set MQMD fields and so enable JMS applications to ″drive″ WebSphere MQ applications.

You must set the Destination object property WMQ_MQMD_WRITE_ENABLED to true for the setting of MQMD properties to have any effect. You can then use the property setting methods of the message (for example setStringProperty) to assign values to the MQMD fields. All MQMD fields are exposed except StrucId and Version; BackoutCount can be read but not written to.

This example results in a message being put to a queue or topic with MQMD.UserIdentifier set to “JoeBloggs”.

```
// Create a ConnectionFactory, connection, session, producer, message
// ...

// Create a destination
// ...

// Enable MQMD write
dest.setBooleanProperty(WMQConstants.WMQ_MQMD_WRITE_ENABLED, true);

// Optionally, set a message context if applicable for this MD field
dest.setIntProperty(WMQConstants.WMQ_MQMD_MESSAGE_CONTEXT,
  WMQConstants.WMQ_MDCTX_SET_IDENTITY_CONTEXT);

// On the message, set property to provide custom UserId
msg.setStringProperty("JMS_IBM_MQMD_UserIdentifier", "JoeBloggs");

// Send the message
// ...
```

It is necessary to set WMQ_MQMD_MESSAGE_CONTEXT before setting JMS_IBM_MQMD_UserIdentifier. For more information about the use of WMQ_MQMD_MESSAGE_CONTEXT, see “Message object properties” on page 134.

Similarly, you can extract the contents of the MQMD fields by setting WMQ_MQMD_READ_ENABLED to true before receiving a message and then using the get methods of the message, such as getStringProperty. Any properties received are read-only.

This example results in the *value* field holding the value of the MQMD.ApplIdentityData field of a message got from a queue or a topic.

```
// Create a ConnectionFactory, connection, session, consumer
// ...

// Create a destination
// ...

// Enable MQMD read
dest.setBooleanProperty(WMQConstants.WMQ_MQMD_READ_ENABLED, true);

// Receive a message
// ...

// Get desired MQMD field value using a property
String value = rcvMsg.getStringProperty("JMS_IBM_MQMD_ApplIdentityData");
```

Destination object properties:

Two properties of the Destination object control access to the MQMD from JMS, and a third controls message context.

Table 32. Property names and descriptions

Property	Short form	Description
WMQ_MQMD_WRITE_ENABLED	MDW	Whether a JMS application can set the values of MQMD fields
WMQ_MQMD_READ_ENABLED	MDR	Whether a JMS application can extract the values of MQMD fields
WMQ_MQMD_MESSAGE_ CONTEXT	MDCTX	What level of message context is to be set by the JMS application. The application must be running with appropriate context authority for this property to take effect

Table 33. Property names, values, and set methods

Property	Valid values in administration tool (defaults in bold)	Valid values in programs	Set method
WMQ_MQMD_ WRITE_ENABLED	• **NO** All JMS_IBM_MQMD* properties are ignored and their values are not copied into the underlying MQMD structure. • YES JMS_IBM_MQMD* properties are processed. Their values are copied into the underlying MQMD structure.	• **False** • True	setMQMDWriteEnabled

Table 33. Property names, values, and set methods (continued)

Property	Valid values in administration tool (defaults in bold)	Valid values in programs	Set method
WMQ_MQMD_ READ_ENABLED	• **NO** When sending messages, the JMS_IBM_MQMD* properties on a sent message are not updated to reflect the updated field values in the MQMD. When receiving messages, none of the JMS_IBM_MQMD* properties are available on a received message, even if the sender had set some or all of them. • YES When sending messages, all of the JMS_IBM_MQMD* properties on a sent message are updated to reflect the updated field values in the MQMD, including those that the sender did not set explcitly. When receiving messages, all of the JMS_IBM_MQMD* properties are available on a received message, including those that the sender did not set explicitly.	• **False** • True	setMQMDReadEnabled

Table 33. Property names, values, and set methods (continued)

Property	Valid values in administration tool (defaults in bold)	Valid values in programs	Set method
WMQ_MQMD_MESSAGE _CONTEXT	• **DEFAULT** The MQOPEN API call and the MQPMO structure will specify no explicit message context options. • SET_IDENTITY_ CONTEXT The MQOPEN API call specifies the message context option MQOO_ SET_IDENTITY_ CONTEXT and the MQPMO structure specifies MQPMO_SET_ IDENTITY_CONTEXT. • SET_ALL_CONTEXT The MQOPEN API call specifies the message context option MQOO_SET_ALL_ CONTEXT and the MQPMO structure specifies MQPMO_ SET_ALL_CONTEXT.	• **WMQ_MDCTX_ DEFAULT** • WMQ_MDCTX_SET_ IDENTITY_CONTEXT • WMQ_MDCTX_SET_ ALL_CONTEXT	setMQMDMessageContext

Message object properties:

Message object properties prefixed JMS_IBM_MQMD allow you to set or read the corresponding MQMD field.

Sending messages

All MQMD fields except StrucId and Version are represented. These properties refer only to the MQMD fields; where a property occurs both in the MQMD and in the MQRFH2 header, the version in the MQRFH2 is not set or extracted.

Any of these properties can be set, except JMS_IBM_MQMD_BackoutCount. Any value set for JMS_IBM_MQMD_BackoutCount is ignored.

If a property has a maximum length and you supply a value that is too long, the value is truncated.

For certain properties, you must also set the WMQ_MQMD_MESSAGE_CONTEXT property on the Destination object. The application must be running with appropriate context authority for this property to take effect. If you do not set WMQ_MQMD_MESSAGE_CONTEXT to an appropriate value, the property value is ignored. If you set WMQ_MQMD_MESSAGE_CONTEXT to an appropriate value but you do not have sufficient context authority for the queue manager, a JMSException is issued. Properties requiring specific values of WMQ_MQMD_MESSAGE_CONTEXT are as follows.

The following properties require WMQ_MQMD_MESSAGE_CONTEXT to be set to WMQ_MDCTX_SET_IDENTITY_CONTEXT or WMQ_MDCTX_SET_ALL_CONTEXT:

- JMS_IBM_MQMD_UserIdentifier
- JMS_IBM_MQMD_AccountingToken
- JMS_IBM_MQMD_ApplIdentityData

The following properties require WMQ_MQMD_MESSAGE_CONTEXT to be set to WMQ_MDCTX_SET_ALL_CONTEXT :

- JMS_IBM_MQMD_PutApplType
- JMS_IBM_MQMD_PutApplName
- JMS_IBM_MQMD_PutDate
- JMS_IBM_MQMD_PutTime
- JMS_IBM_MQMD_ApplOriginData

Receiving messages

All these properties are available on a received message if WMQ_MQMD_READ_ENABLED property is set to true, irrespective of the actual properties the producing application has set. An application cannot modify the properties of a received message unless all properties are cleared first, according to the JMS specification. The received message can be forwarded without modifying the properties.

Attention: If your application receives a message from a destination with WMQ_MQMD_READ_ENABLED property set to true, and forwards it to a destination with WMQ_MQMD_WRITE_ENABLED set to true, this results in all the MQMD field values of the received message being copied into the forwarded message.

Table of properties

This table lists the properties of the Message object representing the MQMD fields. See the links for full descriptions of the fields and their allowable values.

Table 34. Property names, descriptions, and types

Property	Description	Java Type	Link to full description
JMS_IBM_MQMD_Report	Options for report messages	Integer	Report
JMS_IBM_MQMD_MsgType	Message type	Integer	MsgType
JMS_IBM_MQMD_Expiry	Message lifetime	Integer	Expiry
JMS_IBM_MQMD_Feedback	Feedback or reason code	Integer	Feedback
JMS_IBM_MQMD_Encoding	Numeric encoding of message data	Integer	Encoding
JMS_IBM_MQMD_CodedCharSetId	Character set identifier of message data	Integer	CodedCharSetId
JMS_IBM_MQMD_Format	Format name of message data	String	Format
JMS_IBM_MQMD_Priority [1]	Message priority	Integer	Priority
JMS_IBM_MQMD_Persistence	Message persistence	Integer	Persistence
JMS_IBM_MQMD_MsgId [2]	Message identifier	Object (byte[]) [4]	MsgId

Table 34. Property names, descriptions, and types (continued)

Property	Description	Java Type	Link to full description
JMS_IBM_MQMD_CorrelId [3]	Correlation identifier	Object (byte[]) [4]	CorrelId
JMS_IBM_MQMD_BackoutCount	Backout counter	Integer	BackoutCount
JMS_IBM_MQMD_ReplyToQ	Name of reply queue	String	ReplyToQ
JMS_IBM_MQMD_ReplyToQMgr	Name of reply queue manager	String	ReplyToQMgr
JMS_IBM_MQMD_UserIdentifier	User identifier	String	UserIdentifier
JMS_IBM_MQMD_AccountingToken	Accounting token	Object (byte[]) [4]	AccountingToken
JMS_IBM_MQMD_ApplIdentityData	Application data relating to identity	String	ApplIdentityData
JMS_IBM_MQMD_PutApplType	Type of application that put the message	Integer	PutApplType
JMS_IBM_MQMD_PutApplName	Name of application that put the message	String	PutApplName
JMS_IBM_MQMD_PutDate	Date when message was put	String	PutDate
JMS_IBM_MQMD_PutTime	Time when message was put	String	PutTime
JMS_IBM_MQMD_ApplOriginData	Application data relating to origin	String	ApplOriginData
JMS_IBM_MQMD_GroupId	Group identifier	Object (byte[]) [4]	GroupId
JMS_IBM_MQMD_MsgSeqNumber	Sequence number of logical message within group	Integer	MsgSeqNumber
JMS_IBM_MQMD_Offset	Offset of data in physical message from start of logical message	Integer	Offset
JMS_IBM_MQMD_MsgFlags	Message flags	Integer	MsgFlags
JMS_IBM_MQMD_OriginalLength	Length of original message	Integer	OriginalLength

1. **Attention:** If you assign a value to JMS_IBM_MQMD_Priority that is not within the range 0-9, this violates the JMS specification.
2. **Attention:** The JMS specification states that the message ID must be set by the JMS provider and that it must either be unique or null. If you assign a value to JMS_IBM_MQMD_MsgId, this value is copied to the JMSMessageID. Thus it is not set by the JMS provider and might not be unique: this violates the JMS specification.
3. **Attention:** If you assign a value to JMS_IBM_MQMD_CorrelId that starts with the string 'ID:', this violates the JMS specification.
4. **Attention:** The use of byte array properties on a message violates the JMS specification.

Accessing WebSphere MQ Message data from a WebSphere MQ classes for JMS application

You can access the complete WebSphere MQ message data including the MQRFH2 header (if present) and any other WebSphere MQ headers (if present) within a WebSphere MQ classes for JMS application as the body of a JMSBytesMessage.

The function described in this topic is not available for publish/subscribe messaging when you are either connecting to a WebSphere MQ V6 queue manager, or when the PROVIDERVERSION is ″6″.

Use the WMQ_MESSAGE_BODY property of a Destination object to indicate whether the JMS application accesses the whole of a WebSphere MQ message (including the MQRFH2 header, if present) as the body of a JMSBytesMessage.

Sending a message

When sending messages, WMQ_MESSAGE_BODY takes precedence over WMQ_TARGET_CLIENT.

If WMQ_MESSAGE_BODY is set to WMQ_MESSAGE_BODY_JMS, WebSphere MQ classes for JMS automatically generates an MQRFH2 header based on the settings of the JMS Message properties and header fields.

If WMQ_MESSAGE_BODY is set to WMQ_MESSAGE_BODY_MQ, no additional header is added to the message body

If WMQ_MESSAGE_BODY is set to WMQ_MESSAGE_BODY_UNSPECIFIED, the default value, WebSphere MQ classes for JMS does or does not generate and include an MQRFH2 header, depending on the value of WMQ_TARGET_CLIENT.

When you send a JMSBytesMessage, you can use the following properties to override the default format settings for the JMS message body when the WebSphere MQ message is constructed:

- JMS_IBM_Format or JMS_IBM_MQMD_Format: This property specifies the format of the WebSphere MQ header or application payload that starts the JMS message body if there is no preceding Websphere MQ header.
- JMS_IBM_Character_Set or JMS_IBM_MQMD_CodedCharSetId: This property specifies the CCSID of the WebSphere MQ header or application payload that starts the JMS message body if there is no preceding Websphere MQ header.
- JMS_IBM_Encoding or JMS_IBM_MQMD_Encoding: This property specifies the encoding of the WebSphere MQ header or application payload that starts the JMS message body if there is no preceding Websphere MQ header.

If both are specified, the JMS_IBM_MQMD properties override the corresponding JMS_IBM_ properties.

This example results in a message being put to a queue or topic, with its body containing the application payload without an automatically generated MQRFH2 header being added.

```
// Create a ConnectionFactory, connection, session, producer, message
  // ...

  // Create a destination
  // ...

  // Set message body to be MQ
  dest.setIntProperty(WMQConstants.WMQ_MESSAGE_BODY, WMQConstants.WMQ_MESSAGE_
  BODY_MQ);

  // Send the message
```

Receiving a message

If WMQ_MESSAGE_BODY is set to WMQ_MESSAGE_BODY_JMS, the inbound JMS message type and body are determined by the contents of the MQRFH2 header (if present) or the MQMD (if there is no MQRFH2) in the received Websphere MQ message.

If WMQ_MESSAGE_BODY is set to WMQ_MESSAGE_BODY_MQ, the inbound JMS message type is JMSBytesMessage. The JMS message body is the message data returned by the underlying MQGET API call and the length of message body is the length returned by the MQGET call. The character set and encoding of the data in the message body is determined by the CodedCharSetId and Encoding fields of the MQMD. The format of the data in the message body is determined by the Format field of the MQMD

If WMQ_MESSAGE_BODY is set to WMQ_MESSAGE_BODY_UNSPECIFIED, the default value, WebSphere MQ classes for JMS sets it to WMQ_MESSAGE_BODY_JMS.

When you receive a JMSBytesMessage, you can decode it by reference to the following properties:

- JMS_IBM_Format or JMS_IBM_MQMD_Format: This property specifies the format of the WebSphere MQ header or application payload that starts the JMS message body if there is no preceding Websphere MQ header.
- JMS_IBM_Character_Set or JMS_IBM_MQMD_CodedCharSetId: This property specifies the CCSID of the WebSphere MQ header or application payload that starts the JMS message body if there is no preceding Websphere MQ header.
- JMS_IBM_Encoding or JMS_IBM_MQMD_Encoding: This property specifies the encoding of the WebSphere MQ header or application payload that starts the JMS message body if there is no preceding Websphere MQ header.

This example results in a received message that is a JMSBytesMessage, irrespective of the content of the received message and of the format field of the received MQMD.

```
// Create a ConnectionFactory, connection, session, consumer
  // ...

  // Create a destination
  // ...

  // Set message body to be MQ
  dest.setIntProperty(WMQConstants.WMQ_MESSAGE_BODY, WMQConstants.WMQ_MESSAGE_
  BODY_MQ);

  // Receive the message
```

Destination property WMQ_MESSAGE_BODY:

WMQ_MESSAGE_BODY determines whether a JMS application processes the MQRFH2 of a WebSphere MQ message as part of the message payload (that is, as part of the JMS message body).

Table 35. Property names and descriptions

Property	Short form	Description
WMQ_MESSAGE_BODY	MBODY	Whether a JMS application processes the MQRFH2 of a WebSphere MQ message as part of the message payload (that is, as part of the JMS message body).

Table 36. Property names, values, and set methods

Property	Valid values in administration tool (defaults in bold)	Valid values in programs	Set method
WMQ_MESSAGE_BODY	• **UNSPECIFIED** When sending, WebSphere MQ classes for JMS does or does not generate and include an MQRFH2 header, depending on the value of WMQ_ TARGET_CLIENT. When receiving, acts as value JMS. • JMS When sending, WebSphere MQ classes for JMS automatically generates an MQRFH2 header and includes it in the WebSphere MQ message. When receiving, WebSphere MQ classes for JMS set the JMS message properties according to values in the MQRFH2 (if present); it does not present the MQRFH2 as part of the JMS message body. • MQ When sending, WebSphere MQ classes for JMS does not generate an MQRFH2. When receiving, WebSphere MQ classes for JMS presents the MQRFH2 and any other headers as part of the JMS message body.	• **WMQ_MESSAGE_ BODY_UNSPECIFIED** • WMQ_MESSAGE_ BODY_JMS • WMQ_MESSAGE_ BODY_MQ	setMessageBodyStyle

JMS persistent messages

A WebSphere MQ queue has an attribute called *NonPersistentMessageClass*. The value of this attribute determines whether nonpersistent messages on the queue are discarded when the queue manager restarts.

You can set the attribute for a local queue by using the WebSphere MQ Script (MQSC) command, DEFINE QLOCAL, with either of the following parameters:

NPMCLASS(NORMAL)
Nonpersistent messages on the queue are discarded when the queue manager restarts. This is the default value.

NPMCLASS(HIGH)
Nonpersistent messages on the queue are not discarded when the queue manager restarts following a quiesced or immediate shutdown. Nonpersistent messages might be discarded, however, following a preemptive shutdown or a failure.

This topic describes how WebSphere MQ classes for JMS applications can use this queue attribute to provide better performance for JMS persistent messages.

The PERSISTENCE property of a Queue or Topic object can have the value HIGH. You can use the WebSphere MQ JMS administration tool to set this value, or an application can call the Destination.setPersistence() method passing the value JMSC.MQJMS_PER_NPHIGH as a parameter.

If an application sends a JMS persistent message or a JMS nonpersistent message to a destination whose PERSISTENCE property has the value HIGH, and the underlying WebSphere MQ queue is set to NPMCLASS(HIGH), the message is put on the queue as a WebSphere MQ nonpersistent message. If the PERSISTENCE property of the destination does not have the value HIGH, or if the underlying queue is set to NPMCLASS(NORMAL), a JMS persistent message is put on the queue as a WebSphere MQ persistent message, and a JMS nonpersistent message is put on the queue as a WebSphere MQ nonpersistent message.

If a JMS persistent message is put on a queue as a WebSphere MQ nonpersistent message, and you want to ensure that the message is not discarded following a quiesced or immediate shutdown of a queue manager, all queues through which the message might be routed must be set to NPMCLASS(HIGH). In the publish/subscribe domain, these queues include subscriber queues. As an aid to enforcing this configuration, WebSphere MQ classes for JMS throws an InvalidDestinationException if an application tries to create a message consumer for a destination whose PERSISTENCE property has the value HIGH and the underlying WebSphere MQ queue is set to NPMCLASS(NORMAL).

Setting the PERSISTENCE property of a destination to HIGH has no effect on how a message is received from that destination. A message sent as a JMS persistent message is received as a JMS persistent message, and a message sent as a JMS nonpersistent message is received as a JMS nonpersistent message.

When an application sends the first message to a destination whose PERSISTENCE property has the value HIGH, or when an application creates the first message consumer for a destination whose PERSISTENCE property has the value HIGH, WebSphere MQ classes for JMS issues an MQINQ call to determine whether NPMCLASS(HIGH) is set on the underlying WebSphere MQ queue. The application must therefore have the authority to inquire on the queue. In addition, WebSphere MQ classes for JMS preserves the result of the MQINQ call until the destination is deleted, and does not issue more MQINQ calls. Therefore, if you change the NPMCLASS setting on the underlying queue while the application is still using the destination, WebSphere MQ classes for JMS does not notice the new setting.

By allowing JMS persistent messages to be put on WebSphere MQ queues as WebSphere MQ nonpersistent messages, you are gaining performance at the expense of some reliability. If you require maximum reliability for JMS persistent messages, do not send the messages to a destination whose PERSISTENCE property has the value HIGH.

Using Secure Sockets Layer (SSL) with WebSphere MQ classes for JMS

WebSphere MQ classes for JMS applications can use SSL encryption. To do this they require a JSSE provider.

WebSphere MQ classes for JMS connections using TRANSPORT(CLIENT) support Secure Sockets Layer (SSL) encryption. SSL provides communication encryption, authentication, and message integrity. It is typically used to secure communications between any two peers on the Internet or within an intranet.

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

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

If your WebSphere MQ classes for JMS application uses a client channel definition table to connect to a queue manager, see "Using a client channel definition table" on page 150.

SSL administrative properties for WebSphere MQ classes for JMS:

This section introduces the SSL administrative properties.

SSLCIPHERSUITE object property:

Set SSLCIPHERSUITE to enable SSL encryption on a ConnectionFactory object.

To enable SSL encryption on a ConnectionFactory object, use JMSAdmin to set the SSLCIPHERSUITE property to a CipherSuite supported by your JSSE provider. This must match the CipherSpec set on the target channel. However, CipherSuites are distinct from CipherSpecs and therefore have different names. "SSL CipherSpecs and CipherSuites" on page 145 contains a table mapping the CipherSpecs supported by WebSphere MQ to their equivalent CipherSuites as known to JSSE. For more information about CipherSpecs and CipherSuites with WebSphere MQ, see *WebSphere MQ Security*.

For example, to set up a ConnectionFactory object that can be used to create a connection over an SSL enabled MQI channel with a CipherSpec of RC4_MD5_EXPORT, issue the following command to JMSAdmin:

```
ALTER CF(my.cf) SSLCIPHERSUITE(SSL_RSA_EXPORT_WITH_RC4_40_MD5)
```

This can also be set from an application, using the setSSLCipherSuite() method on an MQConnectionFactory object.

For convenience, if a CipherSpec is specified on the SSLCIPHERSUITE property, JMSAdmin attempts to map the CipherSpec to an appropriate CipherSuite and issues a warning. This attempt to map is not made if the property is specified by an application.

SSLFIPSREQUIRED object property:

If you require a connection to use a CipherSuite that is supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS), set the SSLFIPSREQUIRED property of the connection factory to YES.

The default value of this property is NO, which means that a connection can use any CipherSuite that is supported by WebSphere MQ.

If an application uses more than one connection, the value of SSLFIPSREQUIRED that is used when the application creates the first connection determines the value that is used when the application creates any subsequent connection. This means that the value of the SSLFIPSREQUIRED property of the connection factory that is used to create a subsequent connection is ignored. You must restart the application if you want to use a different value of SSLFIPSREQUIRED.

An application can set this property by calling the setSSLFipsRequired() method of a ConnectionFactory object. The property is ignored if no CipherSuite is set.

SSLPEERNAME object property:

Use SSLPEERNAME to specify a distinguished name pattern, to ensure that your JMS application connects to the correct queue manager.

A JMS application can ensure that it connects to the correct queue manager by specifying a distinguished name (DN) pattern. The connection succeeds only if the queue manager presents a DN that matches the pattern. For more details of the format of this pattern, see the related topics.

The DN is set using the SSLPEERNAME property of a ConnectionFactory object. For example, the following JMSAdmin command sets a ConnectionFactory object to expect the queue manager to identify itself with a Common Name beginning with the characters `QMGR.`, and with at least two Organizational Unit names, the first of which must be IBM and the second WEBSPHERE:

```
ALTER CF(my.cf) SSLPEERNAME(CN=QMGR.*, OU=IBM, OU=WEBSPHERE)
```

Checking is not case sensitive and semicolons can be used in place of commas. SSLPEERNAME can also be set from an application using the setSSLPeerName() method on an MQConnectionFactory object. If this property is not set, no checking is performed on the Distinguished Name supplied by the queue manager. This property is ignored if no CipherSuite is set.

SSLCERTSTORES object property:

Use SSLCERTSTORES to specify a list of LDAP servers to use for certificate revocation list (CRL) checking.

It is common to use a certificate revocation list (CRL) to identify certificates that are no longer trusted. CRLs are typically hosted on LDAP servers. JMS allows an LDAP server to be specified for CRL checking under Java 2 v1.4 or later. The following JMSAdmin example directs JMS to use a CRL hosted on an LDAP server named crl1.ibm.com:

```
ALTER CF(my.cf) SSLCRL(ldap://crl1.ibm.com)
```

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

If your LDAP server is not running on the default port of 389, you can specify the port by appending a colon (:) and the port number to the host name. If the certificate presented by the queue manager is present in the CRL hosted on crl1.ibm.com, the connection is not completed. To avoid a single point of failure, JMS allows multiple LDAP servers to be supplied by supplying a list of LDAP servers delimited by the space character. Here is an example:

```
ALTER CF(my.cf) SSLCRL(ldap://crl1.ibm.com ldap://crl2.ibm.com)
```

When multiple LDAP servers are specified, JMS tries each one in turn until it finds a server with which it can successfully verify the queue manager's certificate. Each server must contain identical information.

A string in this format can be supplied by an application on the MQConnectionFactory.setSSLCertStores() method. Alternatively, the application can create one or more java.security.cert.CertStore objects, place these in a suitable Collection object, and supply this Collection object to the setSSLCertStores() method. In this way, the application can customize CRL checking. See your JSSE documentation for details on constructing and using CertStore objects.

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

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

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

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

If your application uses setSSLCertStores() to set a Collection of CertStore objects, the MQConnectionFactory can no longer be bound into a JNDI namespace. Attempting to do so causes an exception. If the sslCertStores property is not set, no revocation checking is performed on the certificate provided by the queue manager. This property is ignored if no CipherSuite is set.

SSLRESETCOUNT object property:

This property represents the total number of bytes sent and received by a connection before the secret key that is used for encryption is renegotiated.

The number of bytes sent is the number before encryption, and the number of bytes received is the number after decryption. The number of bytes also includes control information sent and received by WebSphere MQ classes for JMS.

For example, to configure a ConnectionFactory object that can be used to create a connection over an SSL enabled MQI channel whose secret key is renegotiated after 4 MB of data have flowed, issue the following command to JMSAdmin:

```
ALTER CF(my.cf) SSLRESETCOUNT(4194304)
```

An application can set this property by calling the setSSLResetCount() method of a ConnectionFactory object.

If the value of this property is zero, which is the default value, the secret key is never renegotiated. The property is ignored if no CipherSuite is set.

In some environments, you must not set the reset count to a value other than zero. If you do set the reset count to a value other than zero, a client connection fails when it attempts to renegotiate the secret key. These environments are:
- an HP or Sun V1.4.2 JDK
- any V1.4.2 JDK when using FIPS mode
- any V5.0 or later JDK

For more information about the secret key that is used for encryption on an SSL enabled channel, see *WebSphere MQ Security*.

SSLSocketFactory object property:

To customize other aspects of the SSL connection for an application, create an SSLSocketFactory and configure JMS to use it.

You might want to customize other aspects of the SSL connection for an application. For example, you might want to initialize cryptographic hardware or change the keystore and truststore in use. To do this, the application must first create a javax.net.ssl.SSLSocketFactory object that is customized accordingly. See your JSSE documentation for information about how to do this, because the customizable features vary from provider to provider. After a suitable SSLSocketFactory object is obtained, use the MQConnectionFactory.setSSLSocketFactory() method to configure JMS to use the customized SSLSocketFactory object.

If your application uses the setSSLSocketFactory() method to set a customized SSLSocketFactory object, the MQConnectionFactory object can no longer be bound into a JNDI namespace. Attempting to do so causes an exception. If this property is not set, the default SSLSocketFactory object is used. See your JSSE documentation for details of the behavior of the default SSLSocketFactory object. This property is ignored if no CipherSuite is set.

Important: Do not assume that the use of the SSL properties ensures security when a ConnectionFactory object is retrieved from a JNDI namespace that is not itself secure. Specifically, the standard LDAP implementation of JNDI is not secure. An attacker can imitate the LDAP server, misleading a JMS application into connecting to the wrong server without noticing. With suitable security arrangements in place, other implementations of JNDI (such as the fscontext implementation) are secure.

Making changes to the JSSE keystore or truststore:

If you make changes to the keystore or truststore, you must take certain actions for the changes to be picked up.

If you change the contents of the JSSE keystore or truststore, or change the location of the keystore or truststore file, WebSphere MQ classes for JMS applications that are running at the time do not automatically pick up the changes. For the changes to take effect, the following actions must be performed:

- The applications must close all their connections, and destroy any unused connections in connection pools.
- If your JSSE provider caches information from the keystore and truststore, this information must be refreshed.

After these actions have been performed, the applications can then re-create their connections.

Depending on how you design your applications, and on the function provided by your JSSE provider, it might be possible to perform these actions without stopping and restarting your applications. However, stopping and restarting the applications might be the simplest solution.

SSL CipherSpecs and CipherSuites:

CipherSpecs supported by WebSphere MQ and their equivalent CipherSuites.

Table 37 on page 146 lists the CipherSpecs supported by WebSphere MQ and their equivalent CipherSuites. The table also indicates whether a WebSphere MQ classes for JMS application can connect to a queue manager if a CipherSpec is specified at the server end of the MQI channel and the equivalent CipherSuite is specified at the client end. For each combination of CipherSpec and CipherSuite, whether the application can connect to the queue manager depends on the value of the SSLFIPSREQUIRED property of the ConnectionFactory object.

At the server end of an MQI channel, the name of a CipherSpec can be specified as the value of the SSLCIPH parameter on a DEFINE CHANNEL CHLTYPE(SVRCONN) command. At the client end of an MQI channel, the name of a CipherSuite can be specified in the following ways:

- An application can call the setSSLCipherSuite() method of a ConnectionFactory object.
- Using the WebSphere MQ JMS administration tool, you can set the SSLCIPHERSUITE property of a ConnectionFactory object.

Table 37. CipherSpecs supported by WebSphere MQ and their equivalent CipherSuites

CipherSpec	Equivalent CipherSuite	Connection possible if SFIPS[1] is set to NO?	Connection possible if SFIPS[1] is set to YES?
NULL_MD5	SSL_RSA_WITH_NULL_MD5	Yes	No
NULL_SHA	SSL_RSA_WITH_NULL_SHA	Yes	No
RC4_MD5_EXPORT	SSL_RSA_EXPORT_WITH_RC4_40_MD5	Yes	No
RC4_MD5_US	SSL_RSA_WITH_RC4_128_MD5	Yes	No
RC4_SHA_US	SSL_RSA_WITH_RC4_128_SHA	Yes	No
RC2_MD5_EXPORT	SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	Yes	No
DES_SHA_EXPORT	SSL_RSA_WITH_DES_CBC_SHA	Yes	No
RC4_56_SHA_EXPORT1024	SSL_RSA_EXPORT1024_WITH_RC4_56_SHA	No	No
DES_SHA_EXPORT1024	SSL_RSA_EXPORT1024_WITH_DES_CBC_SHA	No	No
TRIPLE_DES_SHA_US	SSL_RSA_WITH_3DES_EDE_CBC_SHA	Yes	No
TLS_RSA_WITH_AES_128_CBC_SHA	SSL_RSA_WITH_AES_128_CBC_SHA	No	Yes
TLS_RSA_WITH_AES_256_CBC_SHA	SSL_RSA_WITH_AES_256_CBC_SHA	No	Yes
AES_SHA_US[2]			
TLS_RSA_WITH_DES_CBC_SHA	SSL_RSA_WITH_DES_CBC_SHA	No	No[3]
TLS_RSA_WITH_3DES_EDE_CBC_SHA	SSL_RSA_WITH_3DES_EDE_CBC_SHA	No	Yes
FIPS_WITH_DES_CBC_SHA	SSL_RSA_FIPS_WITH_DES_CBC_SHA	Yes	No[4]
FIPS_WITH_3DES_EDE_CBC_SHA	SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA	Yes	No

Notes:

1. When using the WebSphere MQ JMS administration tool, SFIPS is the short name of the ConnectionFactory property SSLFIPSREQUIRED.
2. This CipherSpec has no equivalent CipherSuite.
3. This CipherSpec was FIPS 140-2 certified prior to 19th May 2007.
4. This CipherSpec was FIPS 140-2 certified prior to 19th May 2007. The name FIPS_WITH_DES_CBC_SHA is historical and reflects the fact that this CipherSpec was previously FIPS-compliant.

Using channel exits with WebSphere MQ classes for JMS

This topic describes how to write channel exits in Java, how to assign channel exits, and how to pass user data to channel exits when they are called. The topic also describes how to use channel exits written in C and C++, and how to assign a sequence of send and receive exits that are run in succession.

Note that only an application that connects to a queue manager in client mode can use channel exits. An application cannot use channel exits if it connects in bindings mode.

Writing channel exits in Java for WebSphere MQ classes for JMS:

You create channel exits by defining Java classes that implement specified interfaces.

Three interfaces are defined in the com.ibm.mq.exits package:

- WMQSendExit, for a send exit
- WMQReceiveExit, for a receive exit
- WMQSecurityExit, for a security exit

The following sample code defines a class that implements all three interfaces:

```
public class MyMQExits implements WMQSendExit, WMQReceiveExit, WMQSecurityExit {
    // Default constructor
  public MyMQExits(){
  }
    // This method implements the send exit interface
  public ByteBuffer channelSendExit(MQCXP channelExitParms,
                                    MQCD channelDefinition,
                                    ByteBuffer agentBuffer)
  {
    // Fill in the body of the send exit here
  }
    // This method implements the receive exit interface
  public ByteBuffer channelReceiveExit(MQCXP channelExitParms,
                                       MQCD channelDefinition,
                                       ByteBuffer agentBuffer)
  {
    // Fill in the body of the receive exit here
  }
    // This method implements the security exit interface
  public ByteBuffer channelSecurityExit(MQCXP channelExitParms,
                                        MQCD channelDefinition,
                                        ByteBuffer agentBuffer)
  {
    // Fill in the body of the security exit here
  }
}
```

Each exit receives as parameters an MQCXP object and an MQCD object. These objects represent the MQCXP and MQCD structures defined in the procedural interface.

When a send exit is called, the agentBuffer parameter contains the data that is about to be sent to the server queue manager. A length parameter is not required because the expression agentBuffer.limit() provides the length of the data. The send exit returns as its value the data to be sent to the server queue manager. However, if the send exit is not the last send exit in a sequence of send exits, the data returned is passed instead to the next send exit in the sequence. A send exit can return a modified version of the data that it receives in the agentBuffer parameter, or it can return the data unchanged. The simplest possible exit body is therefore:

```
{ return agentBuffer; }
```

When a receive exit is called, the agentBuffer parameter contains the data that has been received from the server queue manager. The receive exit returns as its value the data to be passed to the application by WebSphere MQ classes for JMS. However, if the receive exit is not the last receive exit in a sequence of receive exits, the data returned is passed instead to the next receive exit in the sequence.

When a security exit is called, the agentBuffer parameter contains the data that has been received in a security flow from the security exit at the server end of the connection. The security exit returns as its value the data to be sent in a security flow to the server security exit.

Channel exits are called with a buffer that has a backing array. For best performance, the exit should return a buffer with a backing array.

Up to 32 characters of user data can be passed to a channel exit when it is called. The exit accesses the user data by calling the getExitData() method of the MQCXP object. Although the exit can change the user data by calling the setExitData() method, the user data is refreshed every time the exit is called. Any changes made to the user data are therefore lost. However, the exit can pass data from one call to the next by using the exit user area of the MQCXP object. The exit accesses the exit user area by reference by calling the getExitUserArea() method.

Every exit class must have a constructor. The constructor can be either the default constructor, as shown in the previous example, or a constructor with a string parameter. The constructor is called to create an instance of the exit class for each exit defined in the class. Therefore, in the previous example, an instance of the MyMQExits class is created for the send exit, another instance is created for the receive exit, and a third instance is created for the security exit. When a constructor with a string parameter is called, the parameter contains the same user data that is passed to the channel exit for which the instance is being created. If an exit class has both a default constructor and a single parameter constructor, the single parameter constructor takes precedence.

Do not close the connection from within a channel exit.

When data is sent to the server end of a connection, SSL encryption is performed *after* any channel exits are called. Similarly, when data is received from the server end of a connection, SSL decryption is performed *before* any channel exits are called.

In versions of WebSphere MQ classes for JMS earlier than Version 7.0, channel exits were implemented using the interfaces MQSendExit, MQReceiveExit, and MQSecurityExit. You can still use these interfaces, but the new interfaces are preferred for improved function and performance.

Assigning channel exits for WebSphere MQ classes for JMS:

A WebSphere MQ classes for JMS application can use channel security, send, and receive exits on the MQI channel that starts when the application connects to a queue manager. The application can use exits written in Java, C, or C++. The application can also use a sequence of send or receive exits that are run in succession.

The SENDEXIT property of an MQConnectionFactory object specifies a send exit, or a sequence of send exits, used by a connection. The value of the property is a string that comprises one or more items separated by commas. Each item identifies a send exit in one of the following ways:

- The name of a class that implements the WMQSendExit interface (for a send exit written in Java)
- A string in the format *libraryName*(*entryPointName*) (for a send exit written in C or C++)

You can set the SENDEXIT property by using the WebSphere MQ JMS administration tool or WebSphere MQ Explorer. Alternatively, an application can set the property by calling the setSendExit() method.

In a similar way, the RECEXIT property of an MQConnectionFactory object specifies the receive exit, or sequence of receive exits, used by a connection, and the SECEXIT property specifies the security exit used by a connection. You can set these properties by using the WebSphere MQ JMS administration tool or

WebSphere MQ Explorer. Alternatively, an application can set the properties by calling the setReceiveExit() and setSecurityExit() methods.

Channel exits are loaded by their own class loader. To find a channel exit, the class loader searches the following locations in the specified order. The class loader loads the first occurrence that it finds.

1. The class path specified by the property com.ibm.mq.cfg.ClientExitPath.JavaExitsClasspath. You can set this property in the WebSphere MQ client configuration file, in the WebSphere MQ classes for JMS configuration file, or as a system property on the **java** command. The value of this property is treated like a normal Java class path.
2. The class path specified by the property com.ibm.mq.exitClasspath. You can set this property in the WebSphere MQ classes for JMS configuration file or as a system property on the **java** command. The value of this property is treated like a normal Java class path.

   Note that this property is now deprecated.
3. The WebSphere MQ exits directory, as shown in Table 38. The class loader first searches the directory for class files that are not packaged in Java archive (JAR) files. If the channel exit is not found, the class loader then searches the JAR files in the directory.

Table 38. The WebSphere MQ exits directory

Platform	Directory
AIX, HP-UX, Linux, and Solaris	/var/mqm/exits (32-bit channel exits) /var/mqm/exits64 (64-bit channel exits)
Windows	*install_data_dir*\exits
Note: *install_data_dir* is the directory that you chose for the WebSphere MQ data files during installation. The default directory is C:\Program Files\IBM\WebSphere MQ.	

The parent of the class loader is the class loader that is used to load WebSphere MQ classes for JMS. It is therefore possible for the parent class loader to load a channel exit if it cannot be found in any of the preceding locations. However, in an environment such as an application server, you are not likely to be able to influence the choice of the parent class loader.

Your application must have the correct security permission to load a channel exit class.

The MQSendExit, MQReceiveExit, and MQSecurityExit interfaces supplied with versions of WebSphere MQ earlier than Version 7.0 are still supported. If you use channel exits that implement these interfaces, com.ibm.mq.jar must be present in the class path.

For information about how to write channel exits in C or C++, see *WebSphere MQ Intercommunication*. You must store channel exit programs written in C or C++ in the directory shown in Table 38.

If your application uses a client channel definition table to connect to a queue manager, see "Using a client channel definition table" on page 150.

Specifying the user data to be passed to channel exits when using WebSphere MQ classes for JMS:

Up to 32 characters of user data can be passed to a channel exit when it is called.

The SENDEXITINIT property of an MQConnectionFactory object specifies the user data that is passed to each send exit when it is called. The value of the property is a string that comprises one or more items of user data separated by commas. The position of each item of user data within the string determines which send exit, in a sequence of send exits, the user data is passed to. For example, the first item of user data in the string is passed to the first send exit in a sequence of send exits.

You can set the SENDEXITINIT property by using the WebSphere MQ JMS administration tool or WebSphere MQ Explorer. Alternatively, an application can set the property by calling the setSendExitInit() method.

In a similar way, the RECEXITINIT property of a ConnectionFactory object specifies the user data that is passed to each receive exit, and the SECEXITINIT property specifies the user data passed to a security exit. You can set these properties by using the WebSphere MQ JMS administration tool or WebSphere MQ Explorer. Alternatively, an application can set the properties by calling the setReceiveExitInit() and setSecurityExitInit() methods.

Note the following rules when specifying user data that is passed to channel exits:
- If the number of items of user data in a string is more than the number of exits in a sequence, the excess items of user data are ignored.
- If the number of items of user data in a string is less than the number of exits in a sequence, each unspecified item of user data is set to an empty string. Two commas in succession within a string, or a comma at the beginning of a string, also denotes an unspecified item of user data.

If an application uses a client channel definition table to connect to a queue manager, any user data specified in a client connection channel definition is passed to channel exits when they are called. For more information about using a client channel definition table, see "Using a client channel definition table."

Using a client channel definition table

As an alternative to creating a client connection channel definition by setting certain properties of a ConnectionFactory object, a WebSphere MQ classes for JMS application can use client connection channel definitions that are stored in a client channel definition table. These definitions are created by WebSphere MQ Script (MQSC) commands or WebSphere MQ Programmable Command Format (PCF) commands. When the application creates a Connection object, WebSphere MQ classes for JMS searches the client channel definition table for a suitable client connection channel definition, and uses the channel definition to start an MQI channel. For more information about client channel definition tables and how to construct one, see *WebSphere MQ Clients*.

To use a client channel definition table, the CCDTURL property of a ConnectionFactory object must be set to a URL object. The URL object encapsulates a uniform resource locator (URL) that identifies the name and location of the file containing the client channel definition table and specifies how the file can be accessed. You can set the CCDTURL property by using the WebSphere MQ JMS administration tool, or an application can set the property by creating a URL object and calling the setCCDTURL() method of the ConnectionFactory object.

For example, if the file ccdt1.tab contains a client channel definition table and is stored on the same system on which the application is running, the application can set the CCDTURL property in the following way:

```
java.net.URL chanTab1 = new URL("file:///home/admdata/ccdt1.tab");
factory.setCCDTURL(chanTab1);
```

As another example, suppose the file ccdt2.tab contains a client channel definition table and is stored on a system that is different to the one on which the application is running. If the file can be accessed using the FTP protocol, the application can set the CCDTURL property in the following way:

```
java.net.URL chanTab2 = new URL("ftp://ftp.server/admdata/ccdt2.tab");
factory.setCCDTURL(chanTab2);
```

In addition to setting the CCDTURL property of the ConnectionFactory object, the QMANAGER property of the same object must be set to one of the following values:

- The name of a queue manager
- An asterisk (*) followed by the name of a queue manager group
- An asterisk (*)
- An empty string, or a string containing all blank characters

These are the same values that can be used for the *QMgrName* parameter on an MQCONN call issued by a client application that is using Message Queue Interface (MQI). For more information about the meaning of these values therefore, see the *WebSphere MQ Application Programming Reference* and *WebSphere MQ Clients*. You can set the QMANAGER property by using the WebSphere MQ JMS administration tool or WebSphere MQ Explorer. Alternatively, an application can set the property by calling the setQueueManager() method of the ConnectionFactory object.

If an application then creates a Connection object from the ConnectionFactory object, WebSphere MQ classes for JMS accesses the client channel definition table identified by the CCDTURL property, uses the QMANAGER property to search the table for a suitable client connection channel definition, and then uses the channel definition to start an MQI channel to a queue manager. The way that WebSphere MQ classes for JMS uses the QMANAGER property to search the client channel definition table is also as described in the *WebSphere MQ Application Programming Reference* and *WebSphere MQ Clients*.

Note that the CCDTURL and CHANNEL properties of a ConnectionFactory object cannot both be set when the application calls the createConnection() method. If both properties are set, the method throws an exception. The CCDTURL or CHANNEL property is considered to be set if its value is anything other than null, an empty string, or a string containing all blank characters.

When WebSphere MQ classes for JMS finds a suitable client connection channel definition in the client channel definition table, it uses only the information extracted from the table to start an MQI channel. Any channel related properties of the ConnectionFactory object are ignored.

In particular, note the following points if you are using Secure Sockets Layer (SSL):

- An MQI channel uses SSL only if the channel definition extracted from the client channel definition table specifies the name of a CipherSpec supported by WebSphere MQ classes for JMS.
- A client channel definition table also contains information about the location of Lightweight Directory Access Protocol (LDAP) servers that hold certificate revocation lists (CRLs). WebSphere MQ classes for JMS uses only this information to access LDAP servers that hold CRLs.

For more information about using SSL with a client channel definition table, see *WebSphere MQ Clients*.

Note also the following points if you are using channel exits:

- An MQI channel uses only the channel exits and associated user data specified by the channel definition extracted from the client channel definition table.
- A channel definition extracted from a client channel definition table can specify channel exits that are written in Java. This means, for example, that the SCYEXIT parameter on the DEFINE CHANNEL command to create a client connection channel definition can specify the name of a class that implements the WMQSecurityExit interface. Similarly, the SENDEXIT parameter can specify the name of a class that implements the WMQSendExit interface, and the RCVEXIT parameter can specify the name of a class that implements the WMQReceiveExit interface. For more information about how to write a channel exit in Java, see "Using channel exits with WebSphere MQ classes for JMS" on page 146.

  The use of channel exits written in a language other than Java is also supported. For information about how to specify the SCYEXIT, SENDEXIT, and RCVEXIT parameters on the DEFINE CHANNEL command for channel exits written in another language, see the *WebSphere MQ Script (MQSC) Command Reference*.

Sharing a TCP/IP connection

Multiple instances of an MQI channel can be made to share a single TCP/IP connection

If a channel is defined with the SHARECNV parameter set to a value greater than 1, then that number of conversations can share a channel instance. To enable a connection factory to exploit this function, use the setShareConvAllowed method of the ConnectionFactory object. If more than one suitable channel is defined in a client channel definition table (CCDT), the AFFINITY and CLNTWGHT channel attributes influence which channel definition is used. For more information about the SHARECNV parameter, see the description of the DEFINE CHANNEL command in *WebSphere MQ Script (MQSC) Command Reference*. For more information about AFFINITY and CLNTWGHT, see the descripitons of channel attributes in *WebSphere MQ Intercommunication*.

Specifying a range of ports for client connections

Use the LOCALADDRESS property to specify a range of ports that your application can bind to.

When a WebSphere MQ classes for JMS application attempts to connect to a WebSphere MQ queue manager in client mode, a firewall might allow only those connections that originate from specified ports or a range of ports. In this situation, you can use the LOCALADDRESS property of a ConnectionFactory, QueueConnectionFactory, or TopicConnectionFactory object to specify a port, or a range of ports, that the application can bind to.

You can set the LOCALADDRESS property by using the WebSphere MQ JMS administration tool, or by calling the setLocalAddress() method in a JMS application. Here is an example of setting the property from within an application:

```
mqConnectionFactory.setLocalAddress("9.20.0.1(2000,3000)");
```

When the application connects to a queue manager subsequently, the application binds to a local IP address and port number in the range 9.20.0.1(2000) to 9.20.0.1(3000).

In a system with more than one network interface, you can also use the LOCALADDRESS property to specify which network interface must be used for a connection.

For a real-time connection to a broker, the LOCALADDRESS property is relevant only when multicast is used. In this case, you can use the property to specify which local network interface must be used for a connection, but the value of the property must not contain a port number, or a range of port numbers.

Connection errors might occur if you restrict the range of ports. If an error occurs, a JMSException is thrown with an embedded MQException that contains the WebSphere MQ reason code MQRC_Q_MGR_NOT_AVAILABLE and the following message:

```
Socket connection attempt refused due to LOCAL_ADDRESS_PROPERTY restrictions
```

An error might occur if all the ports in the specified range are in use, or if the specified IP address, host name, or port number is not valid (a negative port number, for example).

Because WebSphere MQ classes for JMS might create connections other than those required by an application, always consider specifying a range of ports. In general, every session created by an application requires one port and WebSphere MQ classes for JMS might require three or four additional ports. If a connection error does occur, increase the range of ports.

Connection pooling, which is used by default in WebSphere MQ classes for JMS, might have an effect on the speed at which ports can be reused. As a result, a connection error might occur while ports are being freed.

Channel compression

Compressing the data that flows on a WebSphere MQ channel can improve the performance of the channel and reduce network traffic. Using function supplied with WebSphere MQ, you can compress the data that flows on message channels and MQI channels and, on either type of channel, you can compress header data and message data independently of each other. By default, no data is compressed on a channel. For a full description of channel compression, including how it is implemented in WebSphere MQ, see WebSphere MQ Intercommunication, for message channels, and WebSphere MQ Clients, for MQI channels.

A WebSphere MQ classes for JMS application specifies the techniques that can be used for compressing header or message data on a connection by creating a java.util.Collection object. Each compression technique is an Integer object in the collection, and the order in which the application adds the compression techniques to the collection is the order in which the compression techniques are negotiated with the queue manager when the application creates the connection. The application can then pass the collection to a ConnectionFactory object by calling the setHdrCompList() method, for header data, or the setMsgCompList() method, for message data. When the application is ready, it can create the connection.

The following code fragments illustrate the approach just described. The first code fragment shows you how to implement header data compression:

```
Collection headerComp = new Vector();
headerComp.add(new Integer(JMSC.MQJMS_COMPHDR_SYSTEM));
.
.
.
```

```
((MQConnectionFactory) cf).setHdrCompList(headerComp);
.
.
.
connection = cf.createConnection();
```

The second code fragment shows you how to implement message data compression:

```
Collection msgComp = new Vector();
msgComp.add(new Integer(JMSC.MQJMS_COMPMSG_RLE));
msgComp.add(new Integer(JMSC.MQJMS_COMPMSG_ZLIB_HIGH));
.
.
.
((MQConnectionFactory) cf).setMsgCompList(msgComp);
.
.
.
connection = cf.createConnection();
```

In the second example, the compression techniques are negotiated in the order RLE, then ZLIB_HIGH, when the connection is created. The compression technique that is selected cannot be changed during the lifetime of the Connection object. Note that, to use compression on a connection, the setHdrCompList() and the setMsgCompList() methods must be called before creating the Connection object.

Putting messages asynchronously in Websphere MQ classes for JMS

Normally, when an application sends messages to a destination, the application has to wait for the queue manager to confirm that it has processed the request. You can improve messaging performance in some circumstances by choosing instead to put messages asynchronously. When an application puts a message asynchronously, the queue manager does not return the success or failure of each call, but you can instead check for errors periodically.

To configure a destination to return control back to the application without determining whether the queue manager has received the message safely, set the PUTASYNCALLOWED property of the Queue or Topic object and the DefPutResponse attribute of the underlying WebSphere MQ queue or topic. WebSphere MQ classes for JMS can work in this way only for nonpersistent messages and for persistent messages sent in a transacted session.

For messages sent in a transacted session, the application ultimately determines whether the queue manager has received the messages safely when it calls commit(). If an application sends persistent messages within a transacted session, and one or more of the messages are not received safely, the transaction fails to commit and produces an exception. However, if an application sends nonpersistent messages within a transacted session, and one or more of the messages are not received safely, the transaction commits successfully. The application does not receive any feedback that the nonpersistent messages did not arrive safely.

For nonpersistent messages sent in a session that is not transacted, the SENDCHECKCOUNT property of the ConnectionFactory object specifies how many messages are to be sent before WebSphere MQ classes for JMS checks that the queue manager has received the messages safely. If a check discovers that one or more messages were not received safely, and the application has registered an exception listener with the connection, Websphere MQ classes for JMS calls the onException() method of the exception listener to pass a JMS exception to the application. The JMS exception has an error code of JMSWMQ0028 and the following reason:

```
At least one asynchronous put message failed or gave a warning.
```

The JMS Exception also has a linked exception that provides more details. The default value of the SENDCHECKCOUNT property is 0, which means that no such checks are made.

This optimization is of most benefit to an application that connects to a queue manager in client mode and needs to send a sequence of messages in rapid succession, but does not require immediate feedback from the queue manager for each message sent. However, an application can still use this optimization even if it connects to a queue manager in bindings mode, but the expected performance benefit is not as great.

Using read ahead with WebSphere MQ classes for JMS

A WebSphere MQ classes for JMS client can be configured to use *read ahead*. Read ahead allows messages to be sent to a client before an application requests them.

Using read ahead can improve performance when browsing messages or consuming non persistent messages from a client application. This performance improvement is available to both MQI and JMS applications. Client applications using MQGET or asynchronous consume will benefit from the performance improvements when browsing messages or consuming non-persistent messages. For general information about the read ahead facility, see the topic on improving performance of non-persistent messages in *WebSphere MQ Application Programming Guide*.

In WebSphere MQ classes for JMS, you use the READAHEADALLOWED property of a Queue or Topic object to determine whether message consumers and queue browsers are allowed to use read ahead on that object, and the READAHEADCLOSEPOLICY to determine what happens to messages in the internal read ahead buffer when the message consumer is closed.

Retained publications in WebSphere MQ classes for JMS

A WebSphere MQ classes for JMS client can be configured to use retained publications.

A publisher can specify that a copy of a publication should be retained so that it can be sent to future subscribers who register an interest in the topic. This is done in WebSphere MQ classes for JMS by setting the property name JMS_IBM_RETAIN to the value RETAIN_PUBLICATION on an MQMessage object before putting the message.

A new subscriber to a topic with a retained publication receives a copy of that retained publication, unless it chooses not to do so. In WebSphere MQclasses for JMS, this applies to durable topic subscribers, and to non-durable topic subscribers that are not using a real-time connection to a broker. If you do not want a subscriber to receive retained publications, set the option CMQC.MQSO_NEW_PUBLICATIONS_ONLY when you create the MQTopic object.

A subscriber can also choose to receive publications only when it requests them; it issues a request to be sent a topic's retained publication when appropriate and does not receive any non-retained publications. In WebSphere MQ classes for JMS, you set the option CMQC.MQSO_PUBLICATIONS_ON_REQUEST on the MQTopic object and request an update publication for a topic using the requestPublicationUpdate() method of the MQSubscription class.

XA support in WebSphere MQ classes for JMS

JMS supports XA-compliant transactions in bindings mode. In client mode, XA is available only if you use the Extended Transactional Client or within an application server environment.

If you require XA functionality in an application server environment, you must configure your application appropriately. Refer to your application server's own documentation for information about how to configure applications to use distributed transactions.

Using a real-time connection to a broker of WebSphere Event Broker or WebSphere Message Broker

A WebSphere MQ classes for JMS application can use a real-time connection to a broker of WebSphere Event Broker or WebSphere Message Broker for publish/subscribe messaging. Both the broker and WebSphere MQ classes for JMS must be configured to enable a real-time connection.

When an application uses a real-time connection to a broker of WebSphere Event Broker or WebSphere Message Broker, the application and the broker exchange messages using WebSphere MQ Real-Time Transport. Depending on the configuration, messages can also be delivered to the application using WebSphere MQ Multicast Transport.

For information about how an application can connect to a WebSphere MQ queue manager and use WebSphere MQ Enterprise Transport to exchange messages with a broker of WebSphere Event Broker or WebSphere Message Broker, see the documentation for previous releases of WebSphere MQ classes for JMS. Note that, in order to use WebSphere MQ Enterprise Transport, an application must connect to a queue manager using a connection factory running in WebSphere MQ messaging provider migration mode.

Configuring a broker of WebSphere Event Broker or WebSphere Message Broker for a real-time connection

For a WebSphere MQ classes for JMS application to use a real-time connection to a broker of WebSphere Event Broker or WebSphere Message Broker, you must configure the broker by creating and deploying a message flow to read messages from the TCP/IP port on which the broker is listening and publish the messages. Depending on your requirements, you might need to configure the broker in additional ways.

To configure the broker, you must create and deploy one of the following message flows:

- A message flow that contains a Real-timeOptimizedFlow message processing node
- A message flow that contains a Real-timeInput message processing node and a Publication message processing node

You must configure the Real-timeOptimizedFlow or Real-timeInput node to listen on the TCP/IP port used for real-time connections. By default, the port number for real-time connections is 1506.

You must also configure the broker if you have any of the following requirements:

- If you want the application to connect to the broker using Secure Sockets Layer (SSL) authentication
- If you want the application to connect to the broker using HTTP tunnelling
- If you want messages to be delivered to a message consumer using multicast

For information about how to configure a broker, see the *WebSphere Event Broker Information Center* or *WebSphere Message Broker Information Center*.

Configuring WebSphere MQ classes for JMS for a real-time connection to a broker of WebSphere Event Broker or WebSphere Message Broker

For a WebSphere MQ classes for JMS application to use a real-time connection to a broker of WebSphere Event Broker or WebSphere Message Broker, WebSphere MQ classes for JMS must be configured by setting certain properties of the connection factory. Depending on your requirements, WebSphere MQ classes for JMS might need to be configured in additional ways.

To configure WebSphere MQ classes for JMS, the following properties of the connection factory must be set:

- The TRANSPORT property must be set to DIRECT.

  However, for an application to connect using HTTP tunnelling, the TRANSPORT property must be set to DIRECTHTTP instead. See "Using HTTP tunnelling" on page 158.
- The HOSTNAME property must be set to the host name or IP address of the system on which the broker is running.
- The PORT property must be set to the number of the port on which the broker is listening for real-time connections.

An application can set these properties dynamically at run time by using the IBM JMS extensions or the WebSphere MQ JMS extensions. Alternatively, if the connection factory is an administered object, an administrator can set these properties by using the WebSphere MQ JMS administration tool or WebSphere MQ Explorer.

For information about properties, and the methods used by applications to set their values, see "Properties of WebSphere MQ classes for JMS objects" on page 176. For information about how to use the WebSphere MQ JMS administration tool, see "Using the WebSphere MQ JMS administration tool" on page 167. For information about how to use WebSphere MQ Explorer, see the help provided with WebSphere MQ Explorer.

If you have any of the following requirements, WebSphere MQ classes for JMS requires additional configuration:

- If you want an application to connect to the broker using Secure Sockets Layer (SSL) authentication
- If you want an application to connect to the broker using HTTP tunnelling
- If you want an application to connect to the broker through a proxy server
- If you want messages to be delivered to a message consumer using multicast

The following sections describe how to configure WebSphere MQ classes for JMS for each of these requirements.

Using Secure Sockets Layer (SSL) authentication

SSL authentication can be used on a real-time connection to a broker. Only authentication is supported for this type of connection. You cannot use SSL to encrypt and decrypt the message data that flows between the application and the broker or to detect tampering of the data.

Note the difference between this situation and that when an application connects to a queue manager in client mode. In the latter case, you can use the WebSphere MQ SSL support to encrypt and decrypt the message data that flows between the application and the queue manager and to detect tampering of the data, as well as to provide authentication.

If you want to protect message data on a real-time connection to a broker, you can use the function provided by the broker instead. You can assign a quality of protection (QoP) value to each topic whose messages you want to protect. You can therefore select a different level of message protection for each topic. For more information about the message protection provided by a broker, see the *WebSphere Event Broker Information Center* or *WebSphere Message Broker Information Center*.

To use SSL authentication on a real-time connection to a broker, the DIRECTAUTH property of the connection factory must be set to CERTIFICATE.

If you want to use SSL for mutual authentication, the Authentication Protocol Type property of the broker must specify the option R for symmetric SSL. If you want to use SSL only for authenticating the broker, the Authentication Protocol Type property of the broker must specify the option S for asymmetric SSL. But, in this case, the application must connect to the broker by calling createConnection() with a user ID and password as parameters, as in the following example:

```
factory.createConnection("user1", "user1pw");
```

The broker then uses the user ID and password, instead of SSL, to authenticate the application. For more information about how to configure the broker for SSL authentication, see the *WebSphere Event Broker Information Center* or *WebSphere Message Broker Information Center*.

Notes:

1. The value of the DIRECTAUTH property determines whether SSL authentication is used on a real-time connection to a broker, not the value of the SSLCIPHERSUITE property.
2. When SSL authentication is used on a real-time connection to a broker, the SSLPEERNAME and SSLCRL properties are used to perform the same checks as those performed when an application connects to a queue manager in client mode.
3. WebSphere MQ classes for JMS can use the same Java Secure Socket Extension (JSSE) keystore and truststore configuration to provide the SSL support in either of the following situations:
   - When an application uses a real-time connection to a broker
   - When an application connects to a queue manager in client mode

Using HTTP tunnelling

A WebSphere MQ classes for JMS application can connect to a broker using HTTP tunnelling, which means that the application connects to the broker using the HTTP protocol as though connecting to a Web site.

To use HTTP tunnelling on a real-time connection to a broker, the TRANSPORT property of the connection factory must be set to DIRECTHTTP.

HTTP tunnelling cannot be used in conjunction with SSL authentication, connecting through a proxy server, or delivering messages using multicast. The supported version of the HTTP protocol is 1.0. HTTP version 1.1 is not supported.

Connecting through a proxy server

A WebSphere MQ classes for JMS application can use a real-time connection to a broker by connecting through a proxy server. WebSphere MQ classes for JMS connects directly to the proxy server and uses the Internet protocol defined in RFC 2817 to ask the proxy server to forward the connection request to the broker.

To connect to a broker through a proxy server, the following properties of the connection factory must be set:

- The PROXYHOSTNAME property must be set to the host name or IP address of the system on which the proxy server is running.
- The PROXYPORT property must be set to the number of the port on which the proxy server is listening.

If the PROXYHOSTNAME property is not set, or is set to the empty string, WebSphere MQ classes for JMS attempts to connect directly to the broker using only the HOSTNAME and PORT properties, and does not attempt to connect through a proxy server.

Delivering messages using multicast

Using a real-time connection to a broker, messages can be delivered to a message consumer using multicast.

To enable multicast, the MULTICAST property of the Topic object must be set to the required multicast option. Alternatively, if the MULTICAST property of the Topic object is set to ASCF, the MULTICAST property of the connection factory must be set to the required multicast option.

WebSphere MQ classes for JMS supports both the Packet Transfer Layer (PTL) and the Pragmatic General Multicast (PGM) multicast protocols, and includes support for both implementations of the PGM protocol, PGM/IP and PGM UDP encapsulated. However, PGM/IP support is available only on the following platforms:

- AIX (32-bit only)
- HP-UX PA-RISC (32-bit only)
- Linux (x86 platform)
- Linux (zSeries platform, 32-bit only)
- Solaris SPARC (32-bit only)
- Windows (32-bit only)
- z/OS

WebSphere MQ classes for JMS Application Server Facilities

This topic describes how WebSphere MQ classes for JMS implements the ConnectionConsumer class and advanced functionality in the Session class. It also summarizes the function of a server session pool.

WebSphere MQ classes for JMS supports the Application Server Facilities (ASF) that are specified in the *Java Message Service Specification, Version 1.1* (see Sun's Java Web site at http://java.sun.com). This specification identifies three roles within this programming model:

- **The JMS provider** supplies ConnectionConsumer and advanced Session functionality.
- **The application server** supplies ServerSessionPool and ServerSession functionality.
- **The client application** uses the functionality that the JMS provider and application server supply.

The information in this topic does not apply if an application uses a real-time connection to a broker.

ConnectionConsumer

The JMS specification enables an application server to integrate closely with a JMS implementation by using the `ConnectionConsumer` interface. This feature provides concurrent processing of messages. Typically, an application server creates a pool of threads, and the JMS implementation makes messages available to these threads. A JMS-aware application server (such as WebSphere Application Server) can use this feature to provide high-level messaging functionality, such as message driven beans.

Normal applications do not use the ConnectionConsumer, but expert JMS clients might use it. For such clients, the ConnectionConsumer provides a high-performance method to deliver messages concurrently to a pool of threads. When a message arrives on a queue or a topic, JMS selects a thread from the pool and delivers a batch of messages to it. To do this, JMS runs an associated MessageListener's `onMessage()` method.

You can achieve the same effect by constructing multiple Session and MessageConsumer objects, each with a registered MessageListener. However, the ConnectionConsumer provides better performance, less use of resources, and greater flexibility. In particular, fewer Session objects are required.

Planning an application

This topic tells you how to plan an application including:

- "General principles for point-to-point messaging"
- "General principles for publish/subscribe messaging" on page 161
- "Handling poison messages in ASF" on page 162
- "Removing messages from the queue in ASF" on page 163

General principles for point-to-point messaging

When an application creates a ConnectionConsumer from a QueueConnection object, it specifies a JMS queue object and a selector string. The ConnectionConsumer then begins to provide messages to sessions in the associated ServerSessionPool. Messages arrive on the queue, and if they match the selector, they are delivered to sessions in the associated ServerSessionPool.

In WebSphere MQ terms, the queue object refers to either a QLOCAL or a QALIAS on the local queue manager. If it is a QALIAS, that QALIAS must refer to a

QLOCAL. The fully-resolved WebSphere MQ QLOCAL is known as the *underlying QLOCAL*. A ConnectionConsumer is said to be *active* if it is not closed and its parent QueueConnection is started.

It is possible for multiple ConnectionConsumers, each with different selectors, to run against the same underlying QLOCAL. To maintain performance, unwanted messages must not accumulate on the queue. Unwanted messages are those for which no active ConnectionConsumer has a matching selector. You can set the QueueConnectionFactory so that these unwanted messages are removed from the queue (for details, see “Removing messages from the queue in ASF” on page 163). You can set this behavior in one of two ways:

- Use the JMS administration tool to set the QueueConnectionFactory to MRET(NO).
- In your program, use:
  ```
  MQQueueConnectionFactory.setMessageRetention(JMSC.MQJMS_MRET_NO)
  ```

If you do not change this setting, the default is to retain such unwanted messages on the queue.

When you set up the WebSphere MQ queue manager, consider the following points:

- The underlying QLOCAL must be enabled for shared input. To do this, use the following MQSC command:
  ```
  ALTER QLOCAL(your.qlocal.name) SHARE GET(ENABLED)
  ```
- Your queue manager must have an enabled dead-letter queue. If a ConnectionConsumer experiences a problem when it puts a message on the dead-letter queue, message delivery from the underlying QLOCAL stops. To define a dead-letter queue, use:
  ```
  ALTER QMGR DEADQ(your.dead.letter.queue.name)
  ```
- The user that runs the ConnectionConsumer must have authority to perform MQOPEN with MQOO_SAVE_ALL_CONTEXT and MQOO_PASS_ALL_CONTEXT. For details, see the WebSphere MQ documentation for your specific platform.
- If unwanted messages are left on the queue, they degrade the system performance. Therefore, plan your message selectors so that between them, the ConnectionConsumers will remove all messages from the queue.

For details about MQSC commands, see the WebSphere MQ Script (MQSC) Command Reference.

General principles for publish/subscribe messaging

| ConnectionConsumers receive messages for a specified Topic. A
| ConnectionConsumer can be durable or non-durable. You must specify which
| queue or queues the ConnectionConsumer uses.

When an application creates a ConnectionConsumer from a TopicConnection object, it specifies a Topic object and a selector string. The ConnectionConsumer
| then begins to receive messages that match the selector on that Topic, including
| any retained publications for the topic subscribed to.

Alternatively, an application can create a durable ConnectionConsumer that is associated with a specific name. This ConnectionConsumer receives messages that have been published on the Topic since the durable ConnectionConsumer was last
| active. It receives all such messages that match the selector on the Topic. However,

if the ConnectionConsumer is using read-ahead, it can lose nonpersistent messages that are in the client buffer when it closes.

If WebSphere MQ classes for JMS is in WebSphere MQ messaging provider migration mode, a separate queue is used for non-durable ConnectionConsumer subscriptions. The CCSUB configurable option on the TopicConnectionFactory specifies the queue to use. Normally, the CCSUB specifies a single queue for use by all ConnectionConsumers that use the same TopicConnectionFactory. However, it is possible to make each ConnectionConsumer generate a temporary queue by specifying a queue name prefix followed by an asterisk (*).

If WebSphere MQ classes for JMS is in WebSphere MQ messaging provider migration mode, the CCDSUB property of the Topic specifies the queue to use for durable subscriptions. Again, this can be a queue that already exists or a queue name prefix followed by an asterisk (*). If you specify a queue that already exists, all durable ConnectionConsumers that subscribe to the Topic use this queue. If you specify a queue name prefix followed by an asterisk (*), a queue is generated the first time that a durable ConnectionConsumer is created with a given name. This queue is reused later when a durable ConnectionConsumer is created with the same name.

When you set up the WebSphere MQ queue manager, consider the following points:

- Your queue manager must have an enabled dead-letter queue. If a ConnectionConsumer experiences a problem when it puts a message on the dead-letter queue, message delivery from the underlying QLOCAL stops. To define a dead-letter queue, use:

  ```
  ALTER QMGR DEADQ(your.dead.letter.queue.name)
  ```

- The user that runs the ConnectionConsumer must have authority to perform MQOPEN with MQOO_SAVE_ALL_CONTEXT and MQOO_PASS_ALL_CONTEXT. For details, see the WebSphere MQ documentation for your platform.
- You can optimize performance for an individual ConnectionConsumer by creating a separate, dedicated, queue for it. This is at the cost of extra resource usage.

Handling poison messages in ASF

Within the Application Server Facilities, poison message handling is handled slightly differently to elsewhere in WebSphere MQ classes for JMS.

For information about poison message handling in WebSphere MQ classes for JMS, see “Poison messages” on page 124. When you use Application Server Facilities (ASF), the ConnectionConsumer, rather than the MessageConsumer, processes poison messages. The ConnectionConsumer requeues messages according to the BackoutThreshold and BackoutRequeueQName properties of the queue.

When an application uses ConnectionConsumers, the circumstances in which a message is backed out depend on the session that the application server provides:

- When the session is non-transacted, with AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE, a message is backed out only after a system error, or if the application terminates unexpectedly.
- When the session is non-transacted with CLIENT_ACKNOWLEDGE, unacknowledged messages can be backed out by the application server calling Session.recover().

Typically, the client implementation of MessageListener or the application server calls Message.acknowledge(). Message.acknowledge() acknowledges all messages delivered on the session so far.

- When the session is transacted, unacknowledged messages can be backed out by the application server calling Session.rollback().
- If the application server supplies an XASession, messages are committed or backed out depending on a distributed transaction. The application server takes responsibility for completing the transaction.

| The embedded JMS provider in WebSphere Application Server, Version 5.0 and
| Version 5.1 handles poison messages in a different way to that just described for
| WebSphere MQ classes for JMS. For information about how the embedded JMS
| provider handles poison messages, see the relevant WebSphere Application Server
| information center.

Removing messages from the queue in ASF

When an application uses ConnectionConsumers, JMS might need to remove messages from the queue in a number of situations.

These situations are as follows:

Badly formatted message
A message might arrive that JMS cannot parse.

Poison message
A message might reach the backout threshold, but the ConnectionConsumer fails to requeue it on the backout queue.

No interested ConnectionConsumer
For point-to-point messaging, when the QueueConnectionFactory is set so that it does not retain unwanted messages, a message arrives that is unwanted by any of the ConnectionConsumers.

In these situations, the ConnectionConsumer attempts to remove the message from the queue. The disposition options in the report field of the message's MQMD set the exact behavior. These options are:

MQRO_DEAD_LETTER_Q
The message is requeued to the queue manager's dead-letter queue. This is the default.

MQRO_DISCARD_MSG
The message is discarded.

The ConnectionConsumer also generates a report message, and this also depends on the report field of the message's MQMD. This message is sent to the message's ReplyToQ on the ReplyToQmgr. If there is an error while the report message is being sent, the message is sent to the dead-letter queue instead. The exception report options in the report field of the message's MQMD set details of the report message. These options are:

MQRO_EXCEPTION
A report message is generated that contains the MQMD of the original message. It does not contain any message body data.

MQRO_EXCEPTION_WITH_DATA
A report message is generated that contains the MQMD, any MQ headers, and 100 bytes of body data.

MQRO_EXCEPTION_WITH_FULL_DATA
A report message is generated that contains all data from the original message.

default
No report message is generated.

When report messages are generated, the following options are honored:
- MQRO_NEW_MSG_ID
- MQRO_PASS_MSG_ID
- MQRO_COPY_MSG_ID_TO_CORREL_ID
- MQRO_PASS_CORREL_ID

If a poison message cannot be requeued, perhaps because the dead-letter queue is full or authorization is wrongly specified, what happens depends on the persistence of the message. If the message is nonpersistent, the message is discarded and no report message is generated. If the message is persistent, delivery of messages to all connection consumers listening on that destination stops. Such connection consumers must be closed and the problem resolved before they can be recreated and message delivery restarted.

It is important to define a dead-letter queue, and to check it regularly to ensure that no problems occur. Particularly, ensure that the dead-letter queue does not reach its maximum depth, and that its maximum message size is large enough for all messages.

When a message is requeued to the dead-letter queue, it is preceded by a WebSphere MQ dead-letter header (MQDLH). See the WebSphere MQ Application Programming Reference for details about the format of the MQDLH. You can identify messages that a ConnectionConsumer has placed on the dead-letter queue, or report messages that a ConnectionConsumer has generated, by the following fields:
- PutApplType is `MQAT_JAVA (0x1C)`
- PutApplName is "`MQ JMS ConnectionConsumer`"

These fields are in the MQDLH of messages on the dead-letter queue, and the MQMD of report messages. The feedback field of the MQMD, and the Reason field of the MQDLH, contain a code describing the error. For details about these codes, see "Error handling." Other fields are as described in the *WebSphere MQ Application Programming Reference*.

Error handling

This section covers various aspects of error handling, including "Recovering from error conditions" and "Reason and feedback codes" on page 165.

Recovering from error conditions

If a ConnectionConsumer experiences a serious error, message delivery to all ConnectionConsumers with an interest in the same QLOCAL stops. Typically, this occurs if the ConnectionConsumer cannot requeue a message to the dead-letter queue, or it experiences an error when reading messages from the QLOCAL.

When this occurs, any ExceptionListener that is registered with the affected Connection is notified.

You can use these to identify the cause of the problem. In some cases, the system administrator must intervene to resolve the problem.

There are two ways in which an application can recover from these error conditions:

- Call `close()` on all affected ConnectionConsumers. The application can create new ConnectionConsumers only after all affected ConnectionConsumers are closed and any system problems are resolved.
- Call `stop()` on all affected Connections. Once all Connections are stopped and any system problems are resolved, the application should be able to `start()` all Connections successfully.

Reason and feedback codes

To determine the cause of an error, you can use:

- The feedback code in any report messages
- The reason code in the MQDLH of any messages in the dead-letter queue

ConnectionConsumers generate the following reason codes.

MQRC_BACKOUT_THRESHOLD_REACHED (0x93A; 2362)

Cause The message has reached the Backout Threshold defined on the QLOCAL, but no Backout Queue is defined.

On platforms where you cannot define the Backout Queue, the message has reached the JMS-defined backout threshold of 20.

Action

If this is not wanted, define the Backout Queue for the relevant QLOCAL. Also look for the cause of the multiple backouts.

MQRC_MSG_NOT_MATCHED (0x93B; 2363)

Cause In point-to-point messaging, there is a message that does not match any of the selectors for the ConnectionConsumers monitoring the queue. To maintain performance, the message is requeued to the dead-letter queue.

Action

To avoid this situation, ensure that ConnectionConsumers using the queue provide a set of selectors that deal with all messages, or set the QueueConnectionFactory to retain messages.

Alternatively, investigate the source of the message.

MQRC_JMS_FORMAT_ERROR (0x93C; 2364)

Cause JMS cannot interpret the message on the queue.

Action

Investigate the origin of the message. JMS usually delivers messages of an unexpected format as a BytesMessage or TextMessage. Occasionally, this fails if the message is very badly formatted.

Other codes that appear in these fields are caused by a failed attempt to requeue the message to a Backout Queue. In this situation, the code describes the reason that the requeue failed. To diagnose the cause of these errors, refer to the WebSphere MQ Application Programming Reference.

If the report message cannot be put on the ReplyToQ, it is put on the dead-letter queue. In this situation, the feedback field of the MQMD is filled in as described above. The reason field in the MQDLH explains why the report message could not be placed on the ReplyToQ.

The function of a server session pool

| This topic summarizes the function of a server session pool.

Figure 9 summarizes the principles of ServerSessionPool and ServerSession functionality.

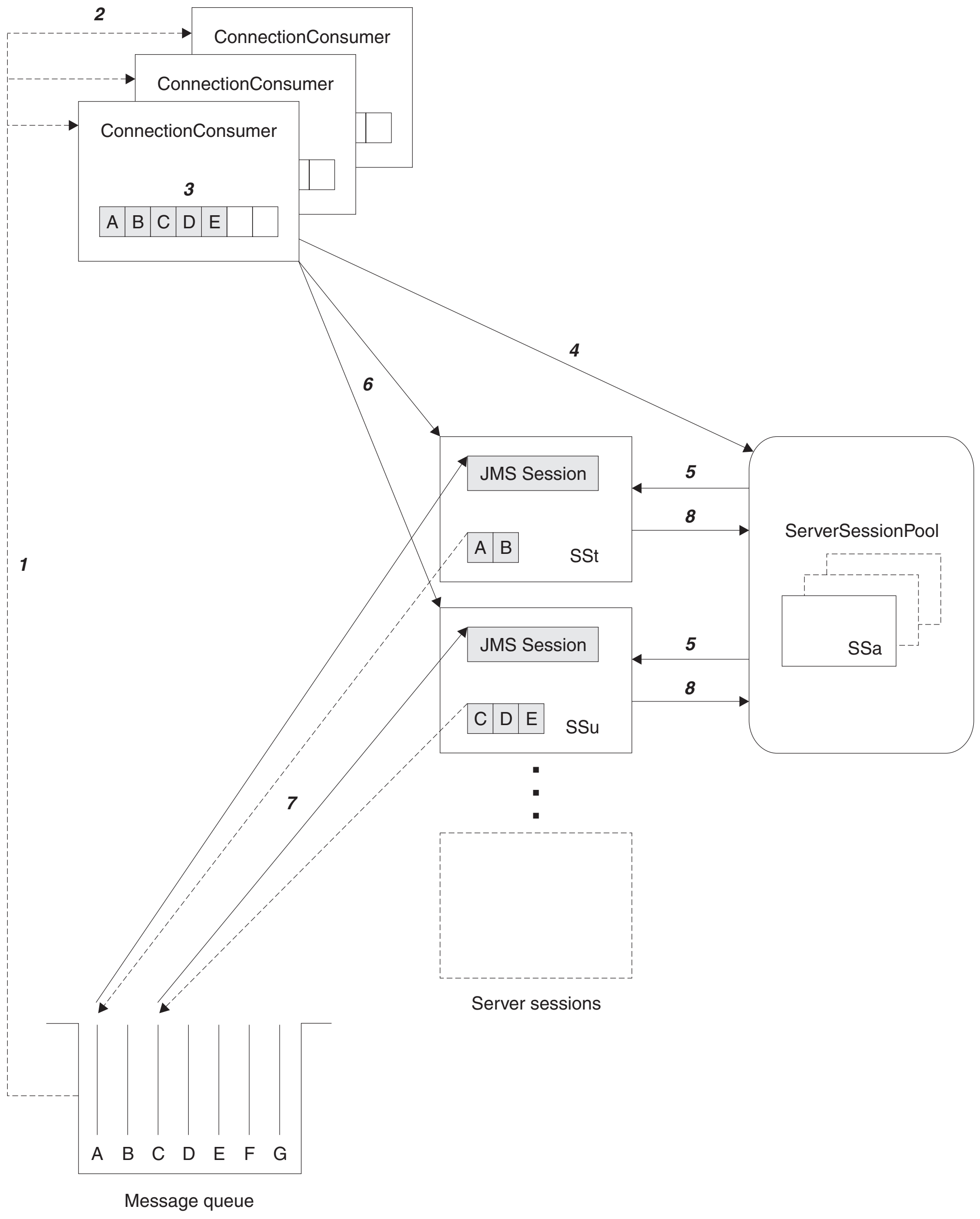


Figure 9. ServerSessionPool and ServerSession functionality

1. The ConnectionConsumers get message references from the queue.

2. Each ConnectionConsumer selects specific message references.
3. The ConnectionConsumer buffer holds the selected message references.
4. The ConnectionConsumer requests one or more ServerSessions from the ServerSessionPool.
5. ServerSessions are allocated from the ServerSessionPool.
6. The ConnectionConsumer assigns message references to the ServerSessions and starts the ServerSession threads running.
7. Each ServerSession retrieves its referenced messages from the queue. It passes them to the `onMessage` method from the MessageListener that is associated with the JMS Session.
8. After it completes its processing, the ServerSession is returned to the pool.

An application server normally supplies ServerSessionPool and ServerSession functionality.

Using the WebSphere MQ JMS administration tool

The administration tool enables administrators to define the properties of eight types of WebSphere MQ classes for JMS object and to store them within a JNDI namespace. Applications can then use JNDI to retrieve these administered objects from the namespace.

The WebSphere MQ classes JMS objects that you can administer by using the tool are:
- MQConnectionFactory
- MQQueueConnectionFactory
- MQTopicConnectionFactory
- MQQueue
- MQTopic
- MQXAConnectionFactory
- MQXAQueueConnectionFactory
- MQXATopicConnectionFactory

For details about these objects, refer to "Administering JMS objects" on page 172.

The tool also allows administrators to manipulate directory namespace subcontexts within the JNDI. See "Manipulating subcontexts" on page 171.

You can also create and configure JMS administered objects with the WebSphere MQ Explorer.

Invoking the administration tool

The administration tool has a command line interface. You can use this interactively, or use it to start a batch process.

The interactive mode provides a command prompt where you can enter administration commands. In the batch mode, the command to start the tool includes the name of a file that contains an administration command script.

Interactive mode

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

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

where:

-t Enables trace (default is trace off)

-v Produces verbose output (default is terse output)

-cfg config_filename
Names an alternative configuration file. If this parameter is omitted, the default configuration file, JMSAdmin.config, is used. (See “Configuration”)

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

```
InitCtx>
```

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

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

Batch mode

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

```
JMSAdmin <test.scp
```

where *test.scp* is a script file that contains administration commands (see “Administration commands” on page 170). The last command in the file must be the `END` command.

Configuration

The WebSphere JMS Administration tool uses a configuration file to set the values of certain properties. The configuration file is a plain-text file that consists of a set of key-value pairs, separated by the equal sign (=). This is shown in the following example:

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

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

A sample configuration file is supplied with Websphere MQ. The file is called `JMSAdmin.config`, and is found in the `<MQ_JAVA_INSTALL_PATH>/bin` directory. Edit this file to suit the setup of your system.

Configure the administration tool with values for the following properties:

INITIAL_CONTEXT_FACTORY
The service provider that the tool uses. The supported values for this property are as follows:

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

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

You can also use an InitialContextFactory that is not in the list above. See "Using an unlisted InitialContextFactory" for more details.

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

- ldap://hostname/contextname
- file:[drive:]/pathname

The format of the LDAP URL can vary, depending on your LDAP provider. See your LDAP documentation for more information.

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

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

If a valid value is not supplied, the property defaults to none. See "Security" on page 170 for more details about security with the administration tool.

These properties are set in a configuration file. When you invoke the tool, you can specify this configuration by using the `-cfg` command-line parameter, as described in "Invoking the administration tool" on page 167. If you do not specify a configuration file name, the tool attempts to load the default configuration file (`JMSAdmin.config`). It looks for this file first in the current directory, and then in the `<MQ_JAVA_INSTALL_PATH>/bin` directory, where `<MQ_JAVA_INSTALL_PATH>` is the path to your WebSphere MQ classes for JMS installation.

Using an unlisted InitialContextFactory

You can use the administration tool to connect to JNDI contexts other than those listed in "Configuration" on page 168 by using three parameters defined in the JMSAdmin configuration file.

To use a different InitialContextFactory:

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

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

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

Security

You need to understand the effect of the `SECURITY_AUTHENTICATION` property described in "Configuration" on page 168.

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

If security credentials are required, you are prompted for these when the tool initializes. Avoid this by setting the PROVIDER_USERDN and PROVIDER_PASSWORD properties in the JMSAdmin configuration file.

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

The tool does no authentication itself; the task is delegated to the LDAP server. The LDAP server administrator must set up and maintain access privileges to different parts of the directory. See your LDAP documentation for more information. If authentication fails, the tool displays an appropriate error message and terminates.

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

Administration commands

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

```
verb [param]*
```

where `verb` is one of the administration verbs listed in Table 39. All valid commands consist of at least one (and only one) verb, which appears at the beginning of the command in either its standard or short form.

The parameters a verb can take depend on the verb. For example, the `END` verb cannot take any parameters, but the `DEFINE` verb can take any number of parameters. Details of the verbs that take at least one parameter are discussed in later sections of this chapter.

Table 39. Administration verbs

Verb	Short form	Description
ALTER	ALT	Change at least one of the properties of a given administered object
DEFINE	DEF	Create and store an administered object, or create a new subcontext
DISPLAY	DIS	Display the properties of one or more stored administered objects, or the contents of the current context
DELETE	DEL	Remove one or more administered objects from the namespace, or remove an empty subcontext

Table 39. Administration verbs (continued)

Verb	Short form	Description
CHANGE	CHG	Alter the current context, allowing the user to traverse the directory namespace anywhere below the initial context (pending security clearance)
COPY	CP	Make a copy of a stored administered object, storing it under an alternative name
MOVE	MV	Alter the name under which an administered object is stored
END		Close the administration tool

Verb names are not case-sensitive.

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

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

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

Manipulating subcontexts

Use the verbs `CHANGE`, `DEFINE`, `DISPLAY` and `DELETE` to manipulate directory namespace subcontexts. Their use is described in Table 40.

Table 40. Syntax and description of commands used to manipulate subcontexts

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

Administering JMS objects

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

| You can also create and configure JMS administered objects with the WebSphere
| MQ Explorer.

Object types

Table 41 shows the eight types of administered objects. The Keyword column shows the strings that you can substitute for `TYPE` in the commands shown in Table 42 on page 173.

Table 41. The JMS object types that are handled by the administration tool

Object Type	**Keyword**	**Description**
MQConnectionFactory	CF	The WebSphere MQ implementation of the JMS ConnectionFactory interface. This represents a factory object for creating connections in the both the point-to-point and publish/subscribe domains.
MQQueueConnectionFactory	QCF	The WebSphere MQ implementation of the JMS QueueConnectionFactory interface. This represents a factory object for creating connections in the point-to-point domain.
MQTopicConnectionFactory	TCF	The WebSphere MQ implementation of the JMS TopicConnectionFactory interface. This represents a factory object for creating connections in the publish/subscribe domain.
MQQueue	Q	The WebSphere MQ implementation of the JMS Queue interface. This represents a destination for messages in the point-to-point domain.
MQTopic	T	The WebSphere MQ implementation of the JMS Topic interface. This represents a destination for messages in the publish/subscribe domain.
MQXAConnectionFactory[1]	XACF	The WebSphere MQ implementation of the JMS XAConnectionFactory interface. This represents a factory object for creating connections in both the point-to-point and publish/subscribe domains, and where the connections use the XA versions of JMS classes.
MQXAQueueConnectionFactory[1]	XAQCF	The WebSphere MQ implementation of the JMS XAQueueConnectionFactory interface. This represents a factory object for creating connections in the point-to-point domain that use the XA versions of JMS classes.

Table 41. The JMS object types that are handled by the administration tool (continued)

Object Type	Keyword	Description
MQXATopicConnectionFactory[1]	XATCF	The WebSphere MQ implementation of the JMS XATopicConnectionFactory interface. This represents a factory object for creating connections in the publish/subscribe domain that use the XA versions of JMS classes.

Note:
1. These classes are provided for use by vendors of application servers. They are unlikely to be directly useful to application programmers.

Verbs used with JMS objects

You can use the verbs `ALTER`, `DEFINE`, `DISPLAY`, `DELETE`, `COPY`, and `MOVE` to manipulate administered objects in the directory namespace. Table 42 summarizes their use. Substitute *`TYPE`* with the keyword that represents the required administered object, as listed in Table 41 on page 172.

Table 42. Syntax and description of commands used to manipulate administered objects

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

Creating objects

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

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

That is, the `DEFINE` verb, followed by a *`TYPE`*`(name)` administered object reference, followed by zero or more *properties* (see "Properties of WebSphere MQ classes for JMS objects" on page 176).

LDAP naming considerations:

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

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

You can change the default prefix by setting the NAME_PREFIX property in the JMSAdmin configuration file, as described in "Using an unlisted InitialContextFactory" on page 169.

This is shown in the following example.

```
InitCtx> DEFINE Q(testQueue)

InitCtx> DISPLAY CTX

    Contents of InitCtx

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

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

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

You might need to configure your LDAP server to store Java objects. For information to assist with this configuration, see the documentation for your LDAP server.

Sample error conditions

The following are examples of the error conditions that might arise when creating an object:

CipherSpec mapped to CipherSuite

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

Invalid property for object

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

Invalid type for property value

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

Property clash - client/bindings

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

Property clash - Exit initialization

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

Property value outside valid range

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

Unknown property

```
InitCtx/cn=Trash> DEFINE QCF(testQCF) PIZZA(ham and mushroom)
  Unable to create a valid object, please check the parameters supplied
  Unknown property: PIZZA
```

The following are examples of error conditions that might arise on Windows when looking up JNDI administered objects from a JMS application.

1. If you are using the WebSphere JNDI provider, com.ibm.websphere.naming.WsnInitialContextFactory, you must use a forward slash (/) to access administered objects defined in sub-contexts; for example, jms/MyQueueName. If you use a backslash (\), an InvalidNameException is thrown.
2. If you are using the Sun JNDI provider, com.sun.jndi.fscontext.RefFSContextFactory, you must use a backslash (\) to access administered objects defined in sub-contexts; for example, ctx1\\fred. If you use a forward slash (/), a NameNotFoundException is thrown.

Using WebSphere MQ Explorer for JMS configuration

Use the WebSphere MQ Explorer graphical user interface to create JMS objects from WebSphere MQ objects, and WebSphere MQ objects from JMS objects, as well as for administering and monitoring other WebSphere MQ objects.

Before you begin

Before you create and configure JMS administered objects with the WebSphere MQ Explorer, add an initial context to define the root of the JNDI namespace in which the JMS objects are stored in the naming and directory service. For more information, refer to the WebSphere MQ Explorer user assistance for JMS administered objects.

About this task

You can perform the following tasks with the WebSphere MQ Explorer, either contextually from an existing object in the WebSphere MQ Explorer, or from within a create new object wizard. Refer to the WebSphere MQ Explorer help for examples of the WebSphere MQ Explorer user assistance for some typical tasks.

- Create a JMS Connection Factory from any of the following WebSphere MQ objects:
  1. A WebSphere MQ queue manager, whether on your local computer or on a remote system.
  2. A WebSphere MQ channel
  3. A WebSphere MQ listener
- Add a WebSphere MQ queue manager to WebSphere MQ Explorer using a JMS Connection Factory
- Create a JMS queue from a WebSphere MQ queue
- Create a WebSphere MQ queue from a JMS queue
- Create a JMS topic from a WebSphere MQ topic, which can be a WebSphere MQ object or a dynamic topic
- Create a WebSphere MQ topic from a JMS topic

Properties of WebSphere MQ classes for JMS objects

All objects in WebSphere MQ classes for JMS have properties. Different properties apply to different object types. Different properties have different allowable values, and symbolic property values differ between the administration tool and program code.

WebSphere MQ classes for JMS provides facilities to set and query the properties of objects using the WebSphere MQ JMS administration tool, WebSphere MQ Explorer, or in an application. Many of the properties are relevant only to a specific subset of the object types.

Table 43 on page 177 gives a brief description of each property and shows the valid property values for each property used in the administration tool.

Table 44 on page 193 gives a brief description of each property and shows for each property which object types it applies to. The object types are identified using keywords; see “Object types” on page 172 for an explanation of these.

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

```
PROPERTY_NAME(property_value)
```

Table 45 on page 195 lists the name of each property, and the set method that is used to set the value of the property in an application. This table also shows the valid property values for each property and the mapping between symbolic property values used in the tool and their programmable equivalents.

Property names are not case-sensitive, and are restricted to the set of recognized names shown in these tables.

Numbers refer to notes at the end of each table. See also “Property dependencies” on page 209.

Table 43. Property names, descriptions and values

Property	Description	Valid values in administration tool (defaults in bold)
ASYNCEXCEPTION	This property determines whether WebSphere MQ classes for JMS informs an ExceptionListener only when a connection is broken, or when any exception occurs asynchronously to a JMS API call. This applies to all Connections created from this ConnectionFactory that have an ExceptionListener registered.	• **ASYNC_EXCEPTIONS_ALL** Any exception detected asynchronously, outside the scope of a synchronous API call, and all connection broken exceptions are sent to the ExceptionListener. • ASYNC_EXCEPTIONS_CONNECTIONBROKEN Only exceptions indicating a broken connection are sent to the ExceptionListener. Any other exceptions occurring during asynchronous processing are not reported to the ExceptionListener, and hence the application is not informed of these exceptions.
BROKERCCDURSUBQ[1]	The name of the queue from which durable subscription messages are retrieved for a ConnectionConsumer	• **SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE** • Any valid string.
BROKERCCSUBQ[1]	The name of the queue from which non-durable subscription messages are retrieved for a ConnectionConsumer	• **SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE** • Any valid string.
BROKERCONQ[1]	Broker's control queue name	• **SYSTEM.BROKER.CONTROL.QUEUE** • Any string
BROKERDURSUBQ[1]	The name of the queue from which durable subscription messages are retrieved	• **SYSTEM.JMS.D.SUBSCRIBER.QUEUE** • Any valid string.
BROKERPUBQ[1]	The name of the queue where published messages are sent (the stream queue)	• **SYSTEM.BROKER.DEFAULT.STREAM** • Any string
BROKERPUBQMGR[1]	The name of the queue manager that owns the queue where messages published on the topic are sent	• **null** • Any string
BROKERQMGR[1]	The name of the queue manager on which the broker is running	• **null** • Any string
BROKERSUBQ[1]	The name of the queue from which non-durable subscription messages are retrieved	• **SYSTEM.JMS.ND.SUBSCRIBER.QUEUE** • Any valid string.

Table 43. Property names, descriptions and values (continued)

Property	Description	Valid values in administration tool (defaults in bold)
BROKERVER[1]	The version of the broker being used	• **unspecified** - After the broker has been migrated from V6 to V7, set this property so that RFH2 headers are no longer used. After migration this property is no longer relevant. • **V1** - To use a WebSphere MQ Publish/Subscribe broker, or to use a broker of WebSphere MQ Integrator, WebSphere Event Broker, WebSphere Business Integration Event Broker, or WebSphere Business Integration Message Broker in compatibility mode. This is the default value if TRANSPORT is set to BIND or CLIENT. • **V2** - To use a broker of WebSphere MQ Integrator, WebSphere Event Broker, WebSphere Business Integration Event Broker, or WebSphere Business Integration Message Broker in native mode. This is the default value if TRANSPORT is set to DIRECT or DIRECTHTTP.
CCDTURL[2]	A uniform resource locator (URL) that identifies the name and location of the file containing the client channel definition table and specifies how the file can be accessed	• **null** • A uniform resource locator (URL)
CCSID	The coded character set ID to be used for a connection or destination	• **819** - This is the default for a connection factory. • **1208** - This is the default for a destination. • Any positive integer
CHANNEL[2]	The name of the client connection channel being used	• **SYSTEM.DEF.SVRCONN** • Any string
CLEANUP[1]	Cleanup Level for BROKER or MIGRATE Subscription Stores	• **SAFE** - use safe cleanup • ASPROP - use safe, strong, or no cleanup according to a property set on the Java command line • NONE - use no cleanup • STRONG - use strong cleanup
CLEANUPINT[1]	The interval, in milliseconds, between background executions of the publish/subscribe cleanup utility	• **3600000** • Any positive integer
CLIENTID	The client identifier for a connection	• **null** • Any string
CLONESUPP	Whether two or more instances of the same durable topic subscriber can run simultaneously	• **DISABLED** - Only one instance of a durable topic subscriber can run at a time. • ENABLED[3] - Two or more instances of the same durable topic subscriber can run simultaneously, but each instance must run in a separate Java virtual machine (JVM).
COMPHDR	A list of the techniques that can be used for compressing header data on a connection	• **NONE** • SYSTEM - RLE message header compression is performed

Table 43. Property names, descriptions and values (continued)

Property	**Description**	**Valid values in administration tool (defaults in bold)**
COMPMSG	A list of the techniques that can be used for compressing message data on a connection	• **NONE** • A list of one or more of the following values separated by blank characters: RLE ZLIBFAST ZLIBHIGH
CONNOPT	Options that control how the application connects to the queue manager	• **STANDARD** - The nature of the binding between the application and the queue manager depends on the platform on which the queue manager is running and how the queue manager is configured. • SHARED - The application and the local queue manager agent run in separate units of execution but share some resources. • ISOLATED - The application and the local queue manager agent run in separate units of execution and share no resources. • FASTPATH - The application and the local queue manager agent run in the same unit of execution. • SERIALQM - The application requests exclusive use of the connection tag within the scope of the queue manager. • SERIALQSG - The application requests exclusive use of the connection tag within the scope of the queue sharing group to which the queue manager belongs. • RESTRICTQM - The application requests shared use of the connection tag, but there are restrictions on the shared use of the connection tag within the scope of the queue manager. • RESTRICTQSG - The application requests shared use of the connection tag, but there are restrictions on the shared use of the connection tag within the scope of the queue sharing group to which the queue manager belongs.
CONNTAG	A tag that the queue manager associates with the resources updated by the application within a unit of work while the application is connected to the queue manager	• **A byte array of 128 elements, where each element is 0** • Any string. The value is truncated if it is longer than 128 bytes.
DESCRIPTION	A description of the stored object	• **null** • Any string
DIRECTAUTH	Whether SSL authentication is used on a real-time connection to a broker[4]	• **BASIC** - No authentication, username authentication, or password authentication • CERTIFICATE - Public key certificate authentication

Table 43. Property names, descriptions and values (continued)

Property	Description	Valid values in administration tool (defaults in bold)
ENCODING	How numerical data in the body of a message is represented when the message is sent to this destination. The property specifies the representation of binary integers, packed decimal integers, and floating point numbers.	See "The ENCODING property" on page 211
EXPIRY	The period after which messages at a destination expire	• **APP** - Expiry can be defined by the JMS application.[9] • UNLIM - No expiry occurs. • 0 - No expiry occurs • Any positive integer representing expiry in milliseconds.
FAILIFQUIESCE	Whether calls to certain methods fail if the queue manager is in a quiescing state	• **YES** - Calls to certain methods fail if the queue manager is in a quiescing state. If an application detects that the queue manager is quiescing, the application can complete its immediate task and close the connection, allowing the queue manager to stop. • NO - No method call fails because the queue manager is in a quiescing state. If you specify this value, an application cannot detect that the queue manager is quiescing. The application might continue to perform operations against the queue manager, and therefore prevent the queue manager from stopping.
HOSTNAME	For a connection to a queue manager, the host name or IP address of the system on which the queue manager is running or, for a real-time connection to a broker, the host name or IP address of the system on which the broker is running	• **localhost** • Any string

Table 43. Property names, descriptions and values (continued)

Property	**Description**	**Valid values in administration tool (defaults in bold)**
LOCALADDRESS	For a connection to a queue manager, this property specifies either or both of the following: • The local network interface to be used • The local port, or range of local ports, to be used For a real-time connection to a broker, this property is relevant only when multicast is used, and specifies the local network interface to be used.	• "" **(empty string)** • A string in the format [*ip-addr*][(*low-port*[,*high-port*])] Here are some examples: **9.20.4.98** The channel binds to address 9.20.4.98 locally **9.20.4.98(1000)** The channel binds to address 9.20.4.98 locally and uses port 1000 **9.20.4.98(1000,2000)** The channel binds to address 9.20.4.98 locally and uses a port in the range 1000 to 2000 **(1000)** The channel binds to port 1000 locally **(1000,2000)** The channel binds to a port in the range 1000 to 2000 locally You can specify a host name instead of an IP address. For a real-time connection to a broker, this property is relevant only when multicast is used, and the value of the property must not contain a port number, or a range of port numbers. The only valid values of the property in this case are null, an IP address, or a host name.
MAPNAMESTYLE	Allows compatibility style to be used for MapMessage element names.	• **STANDARD** - the standard com.ibm.jms.JMSMapMessage element naming format is to be used. • COMPATIBLE - the older com.ibm.jms.JMSMapMessage element naming format is to be used. This is needed only if map messages are being sent to an application that is using a version of WebSphere MQ classes for JMS earlier than Version 5.3.
MAXBUFFSIZE	The maximum number of received messages that can be stored in an internal message buffer while waiting to be processed by the application. This property applies only when TRANSPORT has the value DIRECT or DIRECTHTTP.	• **1000** • Any positive integer
MSGBATCHSZ[1]	The maximum number of messages to be taken from a queue in one packet when using asynchronous message delivery	• **10** • Any positive integer
MSGRETENTION	Whether the connection consumer keeps undelivered messages on the input queue	• **Yes** - Undelivered messages remain on the input queue • No - Undelivered messages are dealt with according to their disposition options

Table 43. Property names, descriptions and values (continued)

Property	Description	Valid values in administration tool (defaults in bold)
MSGSELECTION[1]	Determines whether message selection is done by the WebSphere MQ classes for JMS or by the broker. If TRANSPORT has the value DIRECT, message selection is always done by the broker and the value of MSGSELECTION is ignored. Message selection by the broker is not supported when BROKERVER has the value V1.	• **CLIENT** - Message selection is done by WebSphere MQ classes for JMS. • BROKER - Message selection is done by the broker.
MULTICAST	To enable multicast on a real-time connection to a broker and, if enabled, to specify the precise way in which multicast is used to deliver messages from the broker to a message consumer. The property has no effect on how a message producer sends messages to a broker.[4]	• **DISABLED** - Messages are not delivered to a message consumer using multicast transport. This is the default value for ConnectionFactory and TopicConnectionFactory objects. • **ASCF** - Messages are delivered to a message consumer according to the multicast setting for the connection factory associated with the message consumer. The multicast setting for the connection factory is noted at the time that the message consumer is created. This value is valid only for Topic objects, and is the default value for Topic objects. • ENABLED - If the topic is configured for multicast in the broker, messages are delivered to a message consumer using multicast transport. A reliable quality of service is used if the topic is configured for reliable multicast. • RELIABLE - If the topic is configured for reliable multicast in the broker, messages are delivered to the message consumer using multicast transport with a reliable quality of service. If the topic is not configured for reliable multicast, you cannot create a message consumer for the topic. • NOTR - If the topic is configured for multicast in the broker, messages are delivered to the message consumer using multicast transport. A reliable quality of service is not used even if the topic is configured for reliable multicast.
OPTIMISTICPUBLICATION[1]	Whether WebSphere MQ classes for JMS returns control immediately to a publisher that has just published a message, or whether it returns control only after it has completed all the processing associated with the call and can report the outcome to the publisher	• **NO** - When a publisher publishes a message, WebSphere MQ classes for JMS does not return control to the publisher until it has completed all the processing associated with the call and can report the outcome to the publisher. • YES - When a publisher publishes a message, WebSphere MQ classes for JMS returns control to the publisher immediately, before it has completed all the processing associated with the call and can report the outcome to the publisher. WebSphere MQ classes for JMS reports the outcome only when the publisher commits the message.

Table 43. Property names, descriptions and values (continued)

Property	**Description**	**Valid values in administration tool (defaults in bold)**
OUTCOMENOTIFICATION[1]	Whether WebSphere MQ classes for JMS returns control immediately to a subscriber that has just acknowledged or committed a message, or whether it returns control only after it has completed all the processing associated with the call and can report the outcome to the subscriber	• **YES** - When a subscriber acknowledges or commits a message, WebSphere MQ classes for JMS does not return control to the subscriber until it has completed all the processing associated with the call and can report the outcome to the subscriber. • NO[5] - When a subscriber acknowledges or commits a message, WebSphere MQ classes for JMS returns control to the subscriber immediately, before it has completed all the processing associated with the call and can report the outcome to the subscriber.
PERSISTENCE	The persistence of messages sent to a destination	• **APP** - Persistence is defined by the JMS application.[9] • QDEF - Persistence takes the value of the queue default. • PERS - Messages are persistent. • NON - Messages are nonpersistent. • HIGH - See "JMS persistent messages" on page 140.
POLLINGINT[1]	If each message listener within a session has no suitable message on its queue, this is the maximum interval, in milliseconds, that elapses before each message listener tries again to get a message from its queue. If it frequently happens that no suitable message is available for any of the message listeners in a session, consider increasing the value of this property. This property is relevant only if TRANSPORT has the value BIND or CLIENT.	• **5000** • Any positive integer
PORT	For a connection to a queue manager, the number of the port on which the queue manager is listening or, for a real-time connection to a broker, the number of the port on which the broker is listening for real-time connections	• **1414** - This is the default value if TRANSPORT is set to CLIENT. • **1506** - This is the default value if TRANSPORT is set to DIRECT or DIRECTHTTP. • Any positive integer
PRIORITY	The priority for messages sent to a destination	• **APP** - Priority is defined by the JMS application.[9] • QDEF - Priority takes the value of the queue default. • Any integer in the range 0-9.

Table 43. Property names, descriptions and values (continued)

Property	**Description**	**Valid values in administration tool (defaults in bold)**
PROCESSDURATION[1]	Whether a subscriber guarantees to process quickly any message it receives before returning control to WebSphere MQ classes for JMS	• **UNKNOWN** - A subscriber can give no guarantee about how quickly it can process any message it receives. • SHORT - A subscriber guarantees to process quickly any message it receives before returning control to WebSphere MQ classes for JMS.

Table 43. Property names, descriptions and values (continued)

Property	Description	Valid values in administration tool (defaults in bold)
PROVIDERVERSION[10]	The WebSphere MQ messaging provider has two modes of operation: WebSphere MQ messaging provider normal mode and WebSphere MQ messaging provider migration mode. The WebSphere MQ messaging provider normal mode uses all the features of the WebSphere MQ Version 7.0 queue managers to implement JMS. This mode is used only to connect to a WebSphere MQ queue manager and can connect to WebSphere MQ Version 7.0 queue managers in either client or bindings mode. This mode is optimized to use the new WebSphere MQ Version 7.0 function. If you are not using WebSphere MQ Real-Time Transport, then the mode of operation used is determined primarily by the PROVIDERVERSION property of the connection factory.	You can set PROVIDERVERSION to three possible values: 7, 6, or unspecified: **7** Uses the WebSphere MQ messaging provider normal mode. If you set PROVIDERVERSION to 7 only the WebSphere MQ messaging provider normal mode of operation is available. If the queue manager that is connected to as a result of the other settings in the connection factory is not a Version 7.0 queue manager, the createConnection() method fails with an exception. The WebSphere MQ messaging provider normal mode uses the sharing conversations feature and the number of conversations that can be shared is controlled by the SHARECNV() property on the server connection channel. If this property is set to 0, you cannot use WebSphere MQ messaging provider normal mode and the createConnection() method fails with an exception. **6** Uses the WebSphere MQ messaging provider migration mode. The WebSphere MQ classes for JMS use the features and algorithms supplied with WebSphere MQ Version 6.0. If you want to connect to WebSphere Event Broker or WebSphere Message Broker using WebSphere MQ Enterprise Transport, you must use this mode. You can connect to a WebSphere MQ Version 7.0 queue manager using this mode, but none of the new features of a Version 7.0 queue manager are used, for example, read ahead or streaming. **unspecified** This is the default value and the actual text is "unspecified". A connection factory that was created with a previous version of WebSphere MQ classes for JMS in JNDI takes this value when the connection factory is used with the new version of WebSphere MQ classes for JMS. The following algorithm is used to determine which mode of operation is used. This algorithm is used when the createConnection() method is called and uses other aspects of the connection factory to determine if WebSphere MQ messaging provider normal mode or WebSphere MQ messaging provider migration mode is required. • Firstly, an attempt to use WebSphere MQ messaging provider normal mode is made. • If the queue manager connected is not WebSphere MQ Version 7.0, the connection is closed and WebSphere MQ

Table 43. Property names, descriptions and values (continued)

Property	Description	Valid values in administration tool (defaults in bold)
PROXYHOSTNAME	The host name or IP address of the system on which the proxy server is running when using a real-time connection to a broker through a proxy server[4]	• **null** • The host name of the proxy server
PROXYPORT	The number of the port on which the proxy server is listening when using a real-time connection to a broker through a proxy server[4]	• **443** • The port number of the proxy server
PUBACKINT[1]	The number of messages published by a publisher before WebSphere MQ classes for JMS requests an acknowledgement from the broker. If you lower the value of this property, WebSphere MQ classes for JMS requests acknowledgements more often, and therefore the performance of the publisher decreases. If you raise the value, WebSphere MQ classes for JMS takes a longer time to throw an exception if the broker fails. This property is relevant only if TRANSPORT has the value BIND or CLIENT.	• **25** • Any positive integer
PUTASYNCALLOWED	Whether message producers are allowed to use asynchronous puts to send messages to this destination	• AS_Q_DEF[6] - Determine whether asynchronous puts are allowed by referring to the queue definition. • AS_TOPIC_DEF[6] - Determine whether asynchronous puts are allowed by referring to the topic definition. • **AS_DEST**[6] - Determine whether asynchronous puts are allowed by referring to the queue or topic definition. • NO - Asynchronous puts are not allowed. • YES - Asynchronous puts are allowed.
QMANAGER	The name of the queue manager to connect to. But, if your application uses a client channel definition table to connect to a queue manager, see "Using a client channel definition table" on page 150.	• **"" (empty string)** • Any string

Table 43. Property names, descriptions and values (continued)

Property	Description	Valid values in administration tool (defaults in bold)
QUEUE	The underlying name of the queue representing this destination	Any string
READAHEADALLOWED	Whether message consumers and queue browsers are allowed to use read ahead to get nonpersistent messages from this destination into an internal buffer before receiving them	• AS_Q_DEF[7] - Determine whether read ahead is allowed by referring to the queue definition. • AS_TOPIC_DEF[7] - Determine whether read ahead is allowed by referring to the topic definition. • **AS_DEST**[7] - Determine whether read ahead is allowed by referring to the queue or topic definition. • NO - Read ahead is not allowed. • YES - Read ahead is allowed.
READAHEADCLOSEPOLICY	For messages being delivered to an asynchronous message listener, what happens to messages in the internal read ahead buffer when the message consumer is closed.	• DELIVER_CURRENT - Only the current message listener invocation completes before returning, potentially leaving messages in the internal read ahead buffer, which are then discarded. • **DELIVER_ALL** - All messages in the internal read ahead buffer are delivered to the application's message listener before returning.
RECEIVEISOLATION[1]	Whether a subscriber might receive messages that have not been committed on the subscriber queue	• **COMMITTED** - A subscriber receives only those messages on the subscriber queue that have been committed. • UNCOMMITTED[8] - A subscriber can receive messages that have not been committed on the subscriber queue.
RECEXIT	Identifies a channel receive exit, or a sequence of receive exits to be run in succession	• **null** • A string comprising one or more items separated by commas, where each item is one of the following: – The name of a class that implements the WMQReceiveExit interface (for a channel receive exit written in Java) – A string in the format *libraryName*(*entryPointName*) (for a channel receive exit not written in Java)
RECEXITINIT	The user data that is passed to channel receive exits when they are called	• **null** • A string comprising one or more items of user data separated by commas.

Table 43. Property names, descriptions and values (continued)

Property	Description	Valid values in administration tool (defaults in bold)
RESCANINT[1]	When a message consumer in the point-to-point domain uses a message selector to select which messages it wants to receive, WebSphere MQ classes for JMS searches the WebSphere MQ queue for suitable messages in the sequence determined by the `MsgDeliverySequence` attribute of the queue. When WebSphere MQ classes for JMS finds a suitable message and delivers it to the consumer, WebSphere MQ classes for JMS resumes the search for the next suitable message from its current position in the queue. WebSphere MQ classes for JMS continues to search the queue in this way until it reaches the end of the queue, or until the interval of time in milliseconds, as determined by the value of this property, has expired. In each case, WebSphere MQ classes for JMS returns to the beginning of the queue to continue its search, and a new time interval commences.	• **5000** • Any positive integer
SECEXIT	Identifies a channel security exit	• **null** • The name of a class that implements the WMQSecurityExit interface (for a channel security exit written in Java) • A string in the format *libraryName*(*entryPointName*) (for a channel security exit not written in Java)
SECEXITINIT	The user data that is passed to a channel security exit when it is called	• **null** • Any string
SENDCHECKCOUNT	The number of send calls to allow between checking for asynchronous put errors, within a single non-transacted JMS session	• **0** • Any positive integer

Table 43. Property names, descriptions and values (continued)

Property	**Description**	**Valid values in administration tool (defaults in bold)**
SENDEXIT	Identifies a channel send exit, or a sequence of send exits to be run in succession	• **null** • A string comprising one or more items separated by commas, where each item is one of the following: – The name of a class that implements the WMQSendExit interface (for a channel send exit written in Java) – A string in the format *libraryName*(*entryPointName*) (for a channel send exit not written in Java)
SENDEXITINIT	The user data that is passed to channel send exits when they are called	• **null** • A string comprising one or more items of user data separated by commas
SHARECONVALLOWED	Whether a client connection can share its socket with other top-level JMS connections from the same process to the same queue manager, if the channel definitions match	• NO • **YES**
SPARSESUBS[1]	Controls the message retrieval policy of a TopicSubscriber object	• **NO** - Subscriptions receive frequent matching messages. • YES - Subscriptions receive infrequent matching messages. This value requires that the subscription queue can be opened for browse.
SSLCIPHERSUITE	The CipherSuite to use for an SSL connection	• **null** • See "SSL properties" on page 211
SSLCRL	CRL servers to check for SSL certificate revocation	• **null** • Space-separated list of LDAP URLs. See "SSL properties" on page 211
SSLFIPSREQUIRED	Whether an SSL connection must use a CipherSuite that is supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS)	• **NO** - An SSL connection can use any CipherSuite that is not supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS). • YES - An SSL connection must use a CipherSuite that is supported by IBMJSSEFIPS.
SSLPEERNAME	For SSL, a *distinguished name* skeleton that must match that provided by the queue manager	• **null** • See "SSL properties" on page 211
SSLRESETCOUNT	For SSL, the total number bytes sent and received by a connection before the secret key that is used for encryption is renegotiated.	• **0** • Zero, or any positive integer less than or equal to 999 999 999. See "SSL properties" on page 211

Table 43. Property names, descriptions and values (continued)

Property	Description	Valid values in administration tool (defaults in bold)
STATREFRESHINT[1]	The interval, in milliseconds, between refreshes of the long running transaction that detects when a subscriber loses its connection to the queue manager. This property is relevant only if SUBSTORE has the value QUEUE.	• **60000** • Any positive integer
SUBSTORE[1]	Where WebSphere MQ classes for JMS stores persistent data relating to active subscriptions	• **BROKER** - Use the broker-based subscription store to hold details of subscriptions • MIGRATE - Transfer subscription information from the queue-based subscription store to the broker-based subscription store • QUEUE - Use the queue-based subscription store to hold details of subscriptions
SYNCPOINTALLGETS	Whether all gets are to be performed under syncpoint	• **No** • Yes
TARGCLIENT	Whether the WebSphere MQ RFH2 format is used to exchange information with target applications	• **JMS** - The target of the message is a JMS application. • MQ - The target of the message is a non-JMS WebSphere MQ application.
TARGCLIENTMATCHING	Whether a reply message, sent to the queue identified by the JMSReplyTo header field of an incoming message, has an MQRFH2 header only if the incoming message has an MQRFH2 header	• **YES** - If an incoming message does not have an MQRFH2 header, the TARGCLIENT property of the Queue object derived from the JMSReplyTo header field of the message is set to MQ. If the message does have an MQRFH2 header, the TARGCLIENT property is set to JMS instead. • NO - The TARGCLIENT property of the Queue object derived from the JMSReplyTo header field of an incoming message is always set to JMS.
TEMPMODEL	The name of the model queue from which JMS temporary queues are created	• **SYSTEM.DEFAULT.MODEL.QUEUE** • Any string
TEMPQPREFIX	The prefix that is used to form the name of a WebSphere MQ dynamic queue	• `""` **(empty string)** - The prefix used is `CSQ.*` on z/OS and `AMQ.*` on all other platforms. • Any string that conforms to the rules for forming the contents of the *DynamicQName* field in a WebSphere MQ object descriptor (structure MQOD), but the last non blank character must be an asterisk.

Table 43. Property names, descriptions and values (continued)

Property	Description	Valid values in administration tool (defaults in bold)
TEMPTOPICPREFIX	When creating temporary topics, JMS will generate a topic string of the form "TEMP/ *TEMPTOPICPREFIX*/ *unique_id*", or if this property is left with the default value, just "TEMP/*unique_id*". Specifying a non-empty TEMPTOPICPREFIX allows specific model queues to be defined for creating the managed queues for subscribers to temporary topics created under this connection.	• "" **(empty string)** • Any non-null string consisting only of valid characters for a WebSphere MQ topic string
TOPIC	The underlying name of the topic representing this destination	Any string
TRANSPORT	The nature of a connection to a queue manager or broker	• **BIND** - For a connection to a queue manager in bindings mode • CLIENT - For a connection to a queue manager in client mode • DIRECT - For a real-time connection to a broker not using HTTP tunnelling • DIRECTHTTP - For a real-time connection to a broker using HTTP tunnelling. Only HTTP 1.0 is supported.
USECONNPOOLING	Whether to use connection pooling	• **Yes** • No
WILDCARDFORMAT	Which version of wildcard syntax is to be used	• CHAR_ONLY - Recognizes character wildcards only, as used in broker version 1 • **TOPIC_ONLY** - Recognizes topic level wildcards only, as used in broker version 2
WMQ_MQMD_MESSAGE_ CONTEXT [11]	What level of message context is to be set by the JMS application. The application must be running with appropriate context authority for this property to take effect	• **DEFAULT** - The MQOPEN API call and the MQPMO structure will specify no explicit message context options • SET_IDENTITY_CONTEXT - The MQOPEN API call specifies the message context option MQOO_SET_IDENTITY_CONTEXT and the MQPMO structure specifies MQPMO_SET_IDENTITY_CONTEXT. • SET_ALL_CONTEXT - The MQOPEN API call specifies the message context option MQOO_SET_ALL_CONTEXT and the MQPMO structure specifies MQPMO_SET_ALL_CONTEXT

Table 43. Property names, descriptions and values (continued)

Property	Description	Valid values in administration tool (defaults in bold)
WMQ_MQMD_READ_ ENABLED [11]	Whether a JMS application can extract the values of MQMD fields	• **NO** - When sending messages, the JMS_IBM_MQMD* properties on a sent message are not updated to reflect the updated field values in the MQMD. When receiving messages, none of the JMS_IBM_MQMD* properties are available on a received message, even if the sender had set some or all of them. • Yes - When sending messages, all of the JMS_IBM_MQMD* properties on a sent message are updated to reflect the updated field values in the MQMD, including those that the sender did not set explcitly. When receiving messages, all of the JMS_IBM_MQMD* properties are available on a received message, including those that the sender did not set explicitly.
WMQ_MQMD_WRITE_ ENABLED [11]	Whether a JMS application can set the values of MQMD fields	• **NO** - All JMS_IBM_MQMD* properties are ignored and their values are not copied into the underlying MQMD structure • YES - JMS_IBM_MQMD* properties are processed. Their values are copied into the underlying MQMD structure •

Note:

1. This property can be used with Version 7.0 of WebSphere MQ classes for JMS but has no effect for an application connected to a Version 7.0 queue manager unless the PROVIDERVERSION property of the connection factory is set to a version number less than 7.
2. The CCDTURL and CHANNEL properties of an object must not both be set at the same time.
3. Running two or more instances of the same durable topic subscriber simultaneously contravenes the *Java Message Service Specification, Version 1.1*.
4. See "Using a real-time connection to a broker of WebSphere Event Broker or WebSphere Message Broker" on page 156.
5. If you specify NO, and a message is rolled back after WebSphere MQ classes for JMS has returned control to the subscriber, the subscriber still retains a copy of the message but is not informed of the rollback. In this situation, a subscriber might receive the same message more than once.
6. The values PUTASYNCALLOWED_AS_Q_DEF, PUTASYNCALLOWED_AS_TOPIC_DEF, and PUTASYNCALLOWED_AS_DEST are synonymous and can be used interchangeably.
7. The values READAHEADALLOWED_AS_Q_DEF, READAHEADALLOWED_AS_TOPIC_DEF, and READAHEADALLOWED_AS_DEST are synonymous and can be used interchangeably.
8. The value UNCOMMITTED has an effect only if PROCESSDURATION has the value SHORT. It has no effect if PROCESSDURATION has the value UNKNOWN. If you specify UNCOMMITTED, ensure that a subscriber acknowledges or commits each message individually.
9. For those properties with a default value of APP (value is specified by the application), if the application does not specify a value, then the default value as defined in the JMS specification is used.
10. For more information about PROVIDERVERSION, see "Rules for selecting the WebSphere MQ messaging provider mode" on page 213 and "When to use PROVIDERVERSION" on page 214.
11. For more information about WMQ_MQMD_MESSAGE_CONTEXT, WMQ_MQMD_READ_ENABLED, and WMQ_MQMD_WRITE_ENABLED, see "Reading and writing the message descriptor from a WebSphere MQ classes for JMS application" on page 131.

Table 44. Property names and applicable object types

Property	Short form	Object type							
		CF	QCF	TCF	Q	T	XACF	XAQCF	XATCF
ASYNCEXCEPTION	AEX	Y	Y	Y			Y	Y	Y
BROKERCCDURSUBQ[1]	CCDSUB					Y			
BROKERCCSUBQ[1]	CCSUB	Y		Y			Y		Y
BROKERCONQ[1]	BCON	Y		Y			Y		Y
BROKERDURSUBQ[1]	BDSUB					Y			
BROKERPUBQ[1]	BPUB	Y		Y		Y	Y		Y
BROKERPUBQMGR[1]	BPQM					Y			
BROKERQMGR[1]	BQM	Y		Y			Y		Y
BROKERSUBQ[1]	BSUB	Y		Y			Y		Y
BROKERVER[1]	BVER	Y[2]		Y[2]		Y	Y		Y
CCDTURL[3]	CCDT	Y	Y	Y			Y	Y	Y
CCSID	CCS	Y	Y	Y	Y	Y	Y	Y	Y
CHANNEL[3]	CHAN	Y	Y	Y			Y	Y	Y
CLEANUP[1]	CL	Y		Y			Y		Y
CLEANUPINT[1]	CLINT	Y		Y			Y		Y
CLIENTID	CID	Y[2]	Y	Y[2]			Y	Y	Y
CLONESUPP	CLS	Y	Y	Y			Y	Y	Y
COMPHDR	HC	Y		Y			Y		Y
COMPMSG	MC	Y	Y	Y			Y	Y	Y
CONNOPT	CNOPT	Y	Y	Y			Y	Y	Y
CONNTAG	CNTAG	Y	Y	Y			Y	Y	Y
DESCRIPTION	DESC	Y[2]	Y	Y[2]	Y	Y	Y	Y	Y
DIRECTAUTH	DAUTH	Y[2]		Y[2]					
ENCODING	ENC				Y	Y			
EXPIRY	EXP				Y	Y			
FAILIFQUIESCE	FIQ	Y	Y	Y	Y	Y	Y	Y	Y
HOSTNAME	HOST	Y[2]	Y	Y[2]			Y	Y	Y
LOCALADDRESS	LA	Y[2]	Y	Y[2]			Y	Y	Y
MAPNAMESTYLE	MNST	Y	Y	Y			Y	Y	Y
MAXBUFFSIZE	MBSZ	Y[2]		Y[2]					
MSGBATCHSZ[1]	MBS	Y	Y	Y			Y	Y	Y
MSGRETENTION	MRET	Y	Y				Y	Y	
MSGSELECTION[1]	MSEL	Y		Y			Y		Y
MULTICAST	MCAST	Y[2]		Y[2]		Y			
OPTIMISTICPUBLICATION[1]	OPTPUB	Y		Y					
OUTCOMENOTIFICATION[1]	NOTIFY	Y		Y					
PERSISTENCE	PER				Y	Y			
POLLINGINT[1]	PINT	Y	Y	Y			Y	Y	Y
PORT	PORT	Y[2]	Y	Y[2]			Y	Y	Y

Table 44. Property names and applicable object types (continued)

Property	Short form	Object type							
		CF	**QCF**	**TCF**	**Q**	**T**	**XACF**	**XAQCF**	**XATCF**
PRIORITY	PRI				Y	Y			
PROCESSDURATION[1]	PROCDUR	Y		Y					
PROVIDERVERSION	PVER	Y	Y	Y			Y	Y	Y
PROXYHOSTNAME	PHOST	Y[2]		Y[2]					
PROXYPORT	PPORT	Y[2]		Y[2]					
PUBACKINT[1]	PAI	Y		Y			Y		Y
PUTASYNCALLOWED	PAALD				Y	Y			
QMANAGER	QMGR	Y	Y	Y	Y		Y	Y	Y
QUEUE	QU				Y				
READAHEADALLOWED	RAALD				Y	Y			
READAHEADCLOSEPOLICY	RACP				Y	Y			
RECEIVEISOLATION[1]	RCVISOL	Y		Y					
RECEXIT	RCX	Y	Y	Y			Y	Y	Y
RECEXITINIT	RCXI	Y	Y	Y			Y	Y	Y
RESCANINT[1]	RINT	Y	Y				Y	Y	
SECEXIT	SCX	Y	Y	Y			Y	Y	Y
SECEXITINIT	SCXI	Y	Y	Y			Y	Y	Y
SENDCHECKCOUNT	SCC	Y	Y	Y			Y	Y	Y
SENDEXIT	SDX	Y	Y	Y			Y	Y	Y
SENDEXITINIT	SDXI	Y	Y	Y			Y	Y	Y
SHARECONVALLOWED	SCALD	Y	Y	Y			Y	Y	Y
SPARSESUBS[1]	SSUBS	Y		Y					
SSLCIPHERSUITE	SCPHS	Y	Y	Y			Y	Y	Y
SSLCRL	SCRL	Y	Y	Y			Y	Y	Y
SSLFIPSREQUIRED	SFIPS	Y	Y	Y			Y	Y	Y
SSLPEERNAME	SPEER	Y	Y	Y			Y	Y	Y
SSLRESETCOUNT	SRC	Y	Y	Y			Y	Y	Y
STATREFRESHINT[1]	SRI	Y		Y			Y		Y
SUBSTORE[1]	SS	Y		Y			Y		Y
SYNCPOINTALLGETS	SPAG	Y	Y	Y			Y	Y	Y
TARGCLIENT	TC				Y	Y			
TARGCLIENTMATCHING	TCM	Y	Y				Y	Y	
TEMPMODEL	TM	Y	Y				Y	Y	
TEMPQPREFIX	TQP	Y	Y				Y	Y	
TEMPTOPICPREFIX	TTP	Y		Y			Y		Y
TOPIC	TOP					Y			
TRANSPORT	TRAN	Y[2]	Y	Y[2]			Y	Y	Y
USECONNPOOLING	UCP	Y	Y	Y			Y	Y	Y
WILDCARDFORMAT	WCFMT	Y		Y			Y		Y

Table 44. Property names and applicable object types (continued)

Property	Short form	Object type							
		CF	QCF	TCF	Q	T	XACF	XAQCF	XATCF
WMQ_MQMD_MESSAGE_ CONTEXT	MDCTX								
WMQ_MQMD_READ_ ENABLED	MDR								
WMQ_MQMD_WRITE_ ENABLED	MDW				Y	Y			

Note:

1. This property can be used with Version 7.0 of WebSphere MQ classes for JMS but has no effect for an application connected to a Version 7.0 queue manager unless the PROVIDERVERSION property of the connection factory is set to a version number less than 7.
2. Only the BROKERVER, CLIENTID, DESCRIPTION, DIRECTAUTH, HOSTNAME, LOCALADDRESS, MAXBUFFSIZE, MULTICAST, PORT, PROXYHOSTNAME, PROXYPORT, and TRANSPORT properties are supported for a ConnectionFactory or TopicConnectionFactory object when using a real-time connection to a broker.
3. The CCDTURL and CHANNEL properties of an object must not both be set at the same time.

Table 45. Property names, set methods and values

Property	Set method	Valid values in administration tool (defaults in bold)	Valid values in programs
ASYNCEXCEPTION	setAsyncExceptions	• **ASYNC_EXCEPTIONS_ALL** Any exception detected asynchronously, outside the scope of a synchronous API call, and all connection broken exceptions are sent to the ExceptionListener. • ASYNC_EXCEPTIONS_ CONNECTIONBROKEN Only exceptions indicating a broken connection are sent to the ExceptionListener. Any other exceptions occurring during asynchronous processing are not reported to the ExceptionListener, and hence the application is not informed of these exceptions.	
BROKERCCDURSUBQ[1]	setBrokerCCDurSubQueue	• **SYSTEM.JMS.D.CC. SUBSCRIBER.QUEUE** • Any valid string.	
BROKERCCSUBQ[1]	setBrokerCCSubQueue	• **SYSTEM.JMS.ND.CC. SUBSCRIBER.QUEUE** • Any valid string.	
BROKERCONQ[1]	setBrokerControlQueue	• **SYSTEM.BROKER. CONTROL.QUEUE** • Any string	
BROKERDURSUBQ[1]	setBrokerDurSubQueue	• **SYSTEM.JMS.D. SUBSCRIBER.QUEUE** • Any valid string.	
BROKERPUBQ[1]	setBrokerPubQueue	• **SYSTEM.BROKER. DEFAULT.STREAM** • Any string	

Table 45. Property names, set methods and values (continued)

Property	Set method	Valid values in administration tool (defaults in bold)	Valid values in programs
BROKERPUBQMGR[1]	setBrokerPubQueueManager	• **null** • Any string	
BROKERQMGR[1]	setBrokerQueueManager	• **null** • Any string	
BROKERSUBQ[1]	setBrokerSubQueue	• **SYSTEM.JMS.ND. SUBSCRIBER.QUEUE** • Any valid string.	
BROKERVER[1]	setBrokerVersion	• **V1** - To use a WebSphere MQ Publish/Subscribe broker, or to use a broker of WebSphere MQ Integrator, WebSphere Event Broker, WebSphere Business Integration Event Broker, or WebSphere Business Integration Message Broker in compatibility mode. This is the default value if TRANSPORT is set to BIND or CLIENT. • **V2** - To use a broker of WebSphere MQ Integrator, WebSphere Event Broker, WebSphere Business Integration Event Broker, or WebSphere Business Integration Message Broker in native mode. This is the default value if TRANSPORT is set to DIRECT or DIRECTHTTP. • **unspecified** - After the broker has been migrated from V6 to V7, set this property so that RFH2 headers are no longer used. After migration this property is no longer relevant.	• JMSC.MQJMS_BROKER_V1 • JMSC.MQJMS_BROKER_V2 • JMSC.MQJMS_BROKER_ UNSPECIFIED
CCDTURL[2]	setCCDTURL	• **null** • A uniform resource locator (URL)	
CCSID	CCSID	• **819** - This is the default for a connection factory. • **1208** - This is the default for a destination. • Any positive integer	
CHANNEL[2]	setChannel	• **SYSTEM.DEF.SVRCONN** • Any string	
CLEANUP[1]	setCleanupLevel	• **SAFE** - use safe cleanup • ASPROP - use safe, strong, or no cleanup according to a property set on the Java command line • NONE - use no cleanup • STRONG - use strong cleanup	• JMSC.MQJMS_CLEANUP_ NONE • JMSC.MQJMS_CLEANUP_ SAFE • JMSC.MQJMS_CLEANUP_ STRONG • JMSC.MQJMS_CLEANUP_ AS_PROPERTY
CLEANUPINT[1]	setCleanupInterval	• **3600000** • Any positive integer	
CLIENTID	setClientId	• **null** • Any string	

Table 45. Property names, set methods and values (continued)

Property	Set method	Valid values in administration tool (defaults in bold)	Valid values in programs
CLONESUPP	setCloneSupport	• **DISABLED** - Only one instance of a durable topic subscriber can run at a time. • ENABLED[3] - Two or more instances of the same durable topic subscriber can run simultaneously, but each instance must run in a separate Java virtual machine (JVM).	• JMSC.MQJMS_CLONE_ DISABLED • JMSC.MQJMS_CLONE_ ENABLED
COMPHDR	setHdrCompList	• **NONE** • SYSTEM - RLE message header compression is performed	• JMSC.MQJMS_COMPHDR_ NONE • JMSC.MQJMS_COMPHDR_ SYSTEM
COMPMSG	setMsgCompList	• **NONE** • A list of one or more of the following values separated by blank characters: RLE ZLIBFAST ZLIBHIGH	• JMSC.MQJMS_COMPMSG_ NONE • JMSC.MQJMS_COMPMSG_ RLE • JMSC.MQJMS_COMPMSG_ ZLIBFAST • JMSC.MQJMS_COMPMSG_ ZLIBHIGH

Table 45. Property names, set methods and values (continued)

Property	Set method	Valid values in administration tool (defaults in bold)	Valid values in programs
CONNOPT[4]	setMQConnectionOptions	• **STANDARD** - The nature of the binding between the application and the queue manager depends on the platform on which the queue manager is running and how the queue manager is configured. • SHARED - The application and the local queue manager agent run in separate units of execution but share some resources. • ISOLATED - The application and the local queue manager agent run in separate units of execution and share no resources. • FASTPATH - The application and the local queue manager agent run in the same unit of execution. • SERIALQM - The application requests exclusive use of the connection tag within the scope of the queue manager. • SERIALQSG - The application requests exclusive use of the connection tag within the scope of the queue sharing group to which the queue manager belongs. • RESTRICTQM - The application requests shared use of the connection tag, but there are restrictions on the shared use of the connection tag within the scope of the queue manager. • RESTRICTQSG - The application requests shared use of the connection tag, but there are restrictions on the shared use of the connection tag within the scope of the queue sharing group to which the queue manager belongs.	• JMSC.MQCNO_ STANDARD_BINDING • JMSC.MQCNO_ SHARED_BINDING • JMSC.MQCNO_ ISOLATED_BINDING • JMSC.MQCNO_ FASTPATH_BINDING • JMSC.MQCNO_ SERIALIZE_CONN_TAG_ Q_MGR • JMSC.MQCNO_ SERIALIZE_CONN_TAG_ QSG • JMSC.MQCNO_ RESTRICT_CONN_TAG_ Q_MGR • JMSC.MQCNO_ RESTRICT_CONN_TAG_ QSG
CONNTAG[5]	setConnTag	• **A byte array of 128 elements, where each element is 0** • Any string. The value is truncated if it is longer than 128 bytes.	
DESCRIPTION	setDescription	• **null** • Any string	
DIRECTAUTH	setDirectAuth	• **BASIC** - No authentication, username authentication, or password authentication • CERTIFICATE - Public key certificate authentication	• JMSC.MQJMS_ DIRECTAUTH_BASIC • JMSC.MQJMS_ DIRECTAUTH_ CERTIFICATE

Table 45. Property names, set methods and values (continued)

Property	**Set method**	**Valid values in administration tool (defaults in bold)**	**Valid values in programs**
ENCODING	setEncoding	See "The ENCODING property" on page 211	
EXPIRY	setExpiry	• **APP** - Expiry can be defined by the JMS application. • UNLIM - No expiry occurs. • 0 - No expiry occurs • Any positive integer representing expiry in milliseconds.	• JMSC.MQJMS_EXP_APP • JMSC.MQJMS_EXP_ UNLIMITED
FAILIFQUIESCE	setFailIfQuiesce	• **YES** - Calls to certain methods fail if the queue manager is in a quiescing state. If an application detects that the queue manager is quiescing, the application can complete its immediate task and close the connection, allowing the queue manager to stop. • NO - No method call fails because the queue manager is in a quiescing state. If you specify this value, an application cannot detect that the queue manager is quiescing. The application might continue to perform operations against the queue manager, and therefore prevent the queue manager from stopping.	• JMSC.MQJMS_FIQ_YES • JMSC.MQJMS_FIQ_NO
HOSTNAME	setHostName	• **localhost** • Any string	

Table 45. Property names, set methods and values (continued)

Property	Set method	Valid values in administration tool (defaults in bold)	Valid values in programs
LOCALADDRESS	setLocalAddress	• "" **(empty string)** • A string in the format [*ip-addr*][(*low-port*[,*high-port*])] Here are some examples: **9.20.4.98** The channel binds to address 9.20.4.98 locally **9.20.4.98(1000)** The channel binds to address 9.20.4.98 locally and uses port 1000 **9.20.4.98(1000,2000)** The channel binds to address 9.20.4.98 locally and uses a port in the range 1000 to 2000 **(1000)** The channel binds to port 1000 locally **(1000,2000)** The channel binds to a port in the range 1000 to 2000 locally You can specify a host name instead of an IP address. For a real-time connection to a broker, this property is relevant only when multicast is used, and the value of the property must not contain a port number, or a range of port numbers. The only valid values of the property in this case are null, an IP address, or a host name.	
MAPNAMESTYLE	setMapNameStyle	• **STANDARD** - the standard com.ibm.jms.JMSMapMessage element naming format is to be used. • COMPATIBLE - the older com.ibm.jms.JMSMapMessage element naming format is to be used. This is needed only if map messages are being sent to an application that is using a version of WebSphere MQ classes for JMS earlier than Version 5.3.	• JMSC.MAP_NAME_ STYLE_STANDARD • JMSC.MAP_NAME_ STYLE_COMPATIBLE
MAXBUFFSIZE	setMaxBufferSize	• **1000** • Any positive integer	
MSGBATCHSZ[1]	setMsgBatchSize	• **10** • Any positive integer	
MSGRETENTION	setMessageRetention	• **Yes** - Undelivered messages remain on the input queue • No - Undelivered messages are dealt with according to their disposition options	• JMSC.MQJMS_MRET_YES • JMSC.MQJMS_MRET_NO

Table 45. Property names, set methods and values (continued)

Property	**Set method**	**Valid values in administration tool (defaults in bold)**	**Valid values in programs**
MSGSELECTION[1]	setMessageSelection	• **CLIENT** - Message selection is done by WebSphere MQ classes for JMS. • BROKER - Message selection is done by the broker.	• JMSC.MQJMS_MSEL_ CLIENT • JMSC.MQJMS_MSEL_ BROKER
MULTICAST	setMulticast	• **DISABLED** - Messages are not delivered to a message consumer using multicast transport. This is the default value for ConnectionFactory and TopicConnectionFactory objects. • **ASCF** - Messages are delivered to a message consumer according to the multicast setting for the connection factory associated with the message consumer. The multicast setting for the connection factory is noted at the time that the message consumer is created. This value is valid only for Topic objects, and is the default value for Topic objects. • ENABLED - If the topic is configured for multicast in the broker, messages are delivered to a message consumer using multicast transport. A reliable quality of service is used if the topic is configured for reliable multicast. • RELIABLE - If the topic is configured for reliable multicast in the broker, messages are delivered to the message consumer using multicast transport with a reliable quality of service. If the topic is not configured for reliable multicast, you cannot create a message consumer for the topic. • NOTR - If the topic is configured for multicast in the broker, messages are delivered to the message consumer using multicast transport. A reliable quality of service is not used even if the topic is configured for reliable multicast.	• JMSC.MQJMS_ MULTICAST_DISABLED • JMSC.MQJMS_ MULTICAST_AS_CF • JMSC.MQJMS_ MULTICAST_ENABLED • JMSC.MQJMS_ MULTICAST_RELIABLE • JMSC.MQJMS_ MULTICAST_NOT_ RELIABLE

Table 45. Property names, set methods and values (continued)

Property	Set method	Valid values in administration tool (defaults in bold)	Valid values in programs
OPTIMISTICPUBLICATION[1]	setOptimisticPublication	• **NO** - When a publisher publishes a message, WebSphere MQ classes for JMS does not return control to the publisher until it has completed all the processing associated with the call and can report the outcome to the publisher. • YES - When a publisher publishes a message, WebSphere MQ classes for JMS returns control to the publisher immediately, before it has completed all the processing associated with the call and can report the outcome to the publisher. WebSphere MQ classes for JMS reports the outcome only when the publisher commits the message.	• false • true
OUTCOMENOTIFICATION[1]	setOutcomeNotification	• **YES** - When a subscriber acknowledges or commits a message, WebSphere MQ classes for JMS does not return control to the subscriber until it has completed all the processing associated with the call and can report the outcome to the subscriber. • NO[6] - When a subscriber acknowledges or commits a message, WebSphere MQ classes for JMS returns control to the subscriber immediately, before it has completed all the processing associated with the call and can report the outcome to the subscriber.	• true • false
PERSISTENCE	setPersistence	• **APP** - Persistence is defined by the JMS application. • QDEF - Persistence takes the value of the queue default. • PERS - Messages are persistent. • NON - Messages are nonpersistent. • HIGH - See "JMS persistent messages" on page 140.	• JMSC.MQJMS_PER_APP • JMSC.MQJMS_PER_QDEF • JMSC.MQJMS_PER_PER • JMSC.MQJMS_PER_NON • JMSC.MQJMS_PER_NPHIGH
POLLINGINT[1]	setPollingInterval	• **5000** • Any positive integer	
PORT	setPort	• **1414** - This is the default value if TRANSPORT is set to CLIENT. • **1506** - This is the default value if TRANSPORT is set to DIRECT or DIRECTHTTP. • Any positive integer	

Table 45. Property names, set methods and values (continued)

Property	Set method	Valid values in administration tool (defaults in bold)	Valid values in programs
PRIORITY	setPriority	• **APP** - Priority is defined by the JMS application. • QDEF - Priority takes the value of the queue default. • Any integer in the range 0-9.	• JMSC.MQJMS_PRI_APP • JMSC.MQJMS_PRI_QDEF
PROCESSDURATION[1]	setProcessDuration	• **UNKNOWN** - A subscriber can give no guarantee about how quickly it can process any message it receives. • SHORT - A subscriber guarantees to process quickly any message it receives before returning control to WebSphere MQ classes for JMS.	• JMSC.MQJMS_ PROCESSING_UNKNOWN • JMSC.MQJMS_ PROCESSING_SHORT
PROVIDERVERSION	setProviderVersion	• **unspecified** • A string in one of the following formats – V.R.M.F – V.R.M – V.R – V where V, R, M and F are integer values greater than or equal to zero. A value of 7 or greater indicates that this is intended as a WebSphere MQ Version 7.0 ConnectionFactory for connections to a WebSphere MQ Version 7.0 queue manager. A value lower than 7 (for example ″6.0.2.0″), indicates that it is intended for use with queue managers earlier than Version 7.0. The default value, unspecified, allows connections to any level of queue manager, determining the applicable properties and functionality available based on the queue manager's capabilities.	
PROXYHOSTNAME	setProxyHostName	• **null** • The host name of the proxy server	
PROXYPORT	setProxyPort	• **443** • The port number of the proxy server	
PUBACKINT[1]	setPubAckInterval	• **25** • Any positive integer	

Table 45. Property names, set methods and values (continued)

Property	Set method	Valid values in administration tool (defaults in bold)	Valid values in programs
PUTASYNCALLOWED	setPutAsyncAllowed	• AS_Q_DEF[7] - Determine whether asynchronous puts are allowed by referring to the queue definition. • AS_TOPIC_DEF[7] - Determine whether asynchronous puts are allowed by referring to the topic definition. • **AS_DEST**[7] - Determine whether asynchronous puts are allowed by referring to the queue or topic definition. • NO - Asynchronous puts are not allowed. • YES - Asynchronous puts are allowed.	• JMSC.MQJMS_PUT_ ASYNC_ALLOWED_ AS_Q_DEF • JMSC.MQJMS_PUT_ ASYNC_ALLOWED_ AS_TOPIC_DEF • JMSC.MQJMS_PUT_ ASYNC_ALLOWED_ AS_DEST • JMSC.MQJMS_PUT_ ASYNC_ALLOWED_ DISABLED • JMSC.MQJMS_PUT_ ASYNC_ALLOWED_ ENABLED
QMANAGER	setQueueManager	• "" **(empty string)** • Any string	
QUEUE		Any string	
READAHEADALLOWED	setReadAheadAllowed	• AS_Q_DEF[8] - Determine whether read ahead is allowed by referring to the queue definition. • AS_TOPIC_DEF[8] - Determine whether read ahead is allowed by referring to the topic definition. • **AS_DEST**[8] - Determine whether read ahead is allowed by referring to the queue or topic definition. • NO - Read ahead is not allowed. • YES - Read ahead is allowed.	• JMSC.MQJMS_READ_ AHEAD_ALLOWED_ AS_Q_DEF • JMSC.MQJMS_READ_ AHEAD_ALLOWED_ AS_TOPIC_DEF • JMSC.MQJMS_READ_ AHEAD_ALLOWED_ AS_DEST • JMSC.MQJMS_READ_ AHEAD_ALLOWED_ DISABLED • JMSC.MQJMS_READ_ AHEAD_ALLOWED_ ENABLED
READAHEADCLOSEPOLICY	setReadAheadClosePolicy	• DELIVER_CURRENT - Only the current message listener invocation completes before returning, potentially leaving messages in the internal read ahead buffer, which are then discarded. • **DELIVER_ALL** - All messages in the internal read ahead buffer are delivered to the application's message listener before returning.	• JMSC.MQJMS_READ_ AHEAD_ DELIVERCURRENT • JMSC.MQJMS_READ_ AHEAD_DELIVERALL
RECEIVEISOLATION[1]	setReceiveIsolation	• **COMMITTED** - A subscriber receives only those messages on the subscriber queue that have been committed. • UNCOMMITTED[9] - A subscriber can receive messages that have not been committed on the subscriber queue.	• JMSC.MQJMS_RCVISOL_ COMMITTED • JMSC.MQJMS_RCVISOL_ UNCOMMITTED

Table 45. Property names, set methods and values (continued)

Property	Set method	Valid values in administration tool (defaults in bold)	Valid values in programs
RECEXIT	setReceiveExit	• **null** • A string comprising one or more items separated by commas, where each item is one of the following: – The name of a class that implements the WMQReceiveExit interface (for a channel receive exit written in Java) – A string in the format *libraryName*(*entryPointName*) (for a channel receive exit not written in Java)	
RECEXITINIT	setReceiveExitInit	• **null** • A string comprising one or more items of user data separated by commas.	
RESCANINT[1]	setRescanInterval	• **5000** • Any positive integer	
SECEXIT	setSecurityExit	• **null** • The name of a class that implements the WMQSecurityExit interface (for a channel security exit written in Java) • A string in the format *libraryName*(*entryPointName*) (for a channel security exit not written in Java)	
SECEXITINIT	setSecurityExitInit	• **null** • Any string	
SENDCHECKCOUNT[1]	setSendCheckCount	• **0** • Any positive integer	
SENDEXIT	setSendExit	• **null** • A string comprising one or more items separated by commas, where each item is one of the following: – The name of a class that implements the WMQSendExit interface (for a channel send exit written in Java) – A string in the format *libraryName*(*entryPointName*) (for a channel send exit not written in Java)	
SENDEXITINIT	setSendExitInit	• **null** • A string comprising one or more items of user data separated by commas	
SHARECONVALLOWED	setShareConvAllowed	• NO • **YES**	• JMSC.MQJMS_SHARE_ CONV_ALLOWED_NO • JMSC.MQJMS_SHARE_ CONV_ALLOWED_YES

Table 45. Property names, set methods and values (continued)

Property	Set method	Valid values in administration tool (defaults in bold)	Valid values in programs
SPARSESUBS[1]	setSparseSubscriptions	• **NO** - Subscriptions receive frequent matching messages. • YES - Subscriptions receive infrequent matching messages. This value requires that the subscription queue can be opened for browse.	• true • false
SSLCIPHERSUITE	setSSLCipherSuite	• **null** • See “SSL properties” on page 211	
SSLCRL	setSSLCertStores	• **null** • Space-separated list of LDAP URLs. See “SSL properties” on page 211	
SSLFIPSREQUIRED	setSSLFipsRequired	• **NO** - An SSL connection can use any CipherSuite that is not supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS). • YES - An SSL connection must use a CipherSuite that is supported by IBMJSSEFIPS.	• false • true
SSLPEERNAME	setSSLPeerName	• **null** • See “SSL properties” on page 211	
SSLRESETCOUNT	setSSLResetCount	• **0** • Zero, or any positive integer less than or equal to 999 999 999. See “SSL properties” on page 211	
STATREFRESHINT[1]	setStatusRefreshInterval	• **60000** • Any positive integer	
SUBSTORE[1]	setSubscriptionStore	• **BROKER** - Use the broker-based subscription store to hold details of subscriptions • MIGRATE - Transfer subscription information from the queue-based subscription store to the broker-based subscription store • QUEUE - Use the queue-based subscription store to hold details of subscriptions	• JMSC.MQJMS_SUBSTORE_ BROKER • h • JMSC.MQJMS_SUBSTORE_ MIGRATE • JMSC.MQJMS_SUBSTORE_ QUEUE
SYNCPOINTALLGETS	setSyncpointAllGets	• **No** • Yes	
TARGCLIENT[10]	setTargetClient	• **JMS** - The target of the message is a JMS application. • MQ - The target of the message is a non-JMS WebSphere MQ application.	• JMSC.MQJMS_CLIENT_ JMS_COMPLIANT • JMSC.MQJMS_CLIENT_ NONJMS_MQ

Table 45. Property names, set methods and values (continued)

Property	Set method	Valid values in administration tool (defaults in bold)	Valid values in programs
TARGCLIENTMATCHING	setTargClientMatching	• **YES** - If an incoming message does not have an MQRFH2 header, the TARGCLIENT property of the Queue object derived from the JMSReplyTo header field of the message is set to MQ. If the message does have an MQRFH2 header, the TARGCLIENT property is set to JMS instead. • NO - The TARGCLIENT property of the Queue object derived from the JMSReplyTo header field of an incoming message is always set to JMS.	• true • false
TEMPMODEL	setTemporaryModel	• **SYSTEM.DEFAULT. MODEL.QUEUE** • Any string	
TEMPQPREFIX	setTempQPrefix	• "" **(empty string)** - The prefix used is `CSQ.*` on z/OS and `AMQ.*` on all other platforms. • Any string that conforms to the rules for forming the contents of the *DynamicQName* field in a WebSphere MQ object descriptor (structure MQOD), but the last non blank character must be an asterisk.	
TEMPTOPICPREFIX	setTempTopicPrefix	• "" **(empty string)** • Any non-null string consisting only of valid characters for a WebSphere MQ topic string	
TOPIC		Any string	
TRANSPORT	setTransportType	• **BIND** - For a connection to a queue manager in bindings mode • CLIENT - For a connection to a queue manager in client mode • DIRECT - For a real-time connection to a broker not using HTTP tunnelling • DIRECTHTTP - For a real-time connection to a broker using HTTP tunnelling. Only HTTP 1.0 is supported.	• JMSC.MQJMS_TP_ BINDINGS_MQ • JMSC.MQJMS_TP_ CLIENT_MQ_TCPIP • JMSC.MQJMS_TP_ DIRECT_TCPIP • JMSC.MQJMS_TP_ DIRECT_HTTP
USECONNPOOLING	setUseConnectionPooling	• **Yes** • No	
WILDCARDFORMAT	setWildCardFormat	• CHAR_ONLY - Recognizes character wildcards only, as used in broker version 1 • **TOPIC_ONLY** - Recognizes topic level wildcards only, as used in broker version 2	• JMSC.MQJMS_ WILDCARD_CHAR_ONLY • JMSC.MQJMS_ WILDCARD_TOPIC_ONLY

Table 45. Property names, set methods and values (continued)

Property	Set method	Valid values in administration tool (defaults in bold)	Valid values in programs
WMQ_MQMD_MESSAGE_ CONTEXT	setMQMDMessageContext	• **DEFAULT** - The MQOPEN API call and the MQPMO structure will specify no explicit message context options • SET_IDENTITY_CONTEXT - The MQOPEN API call specifies the message context option MQOO_SET_IDENTITY_ CONTEXT and the MQPMO structure specifies MQPMO_SET_IDENTITY_ CONTEXT. • SET_ALL_CONTEXT - The MQOPEN API call specifies the message context option MQOO_SET_ALL_CONTEXT and the MQPMO structure specifies MQPMO_SET_ALL _CONTEXT	• **WMQ_MDCTX_DEFAULT** • WMQ_MDCTX_SET_ IDENTITY_CONTEXT • WMQ_MDCTX_SET_ ALL_CONTEXT
WMQ_MQMD_READ_ ENABLED	setMQMDReadEnabled	• **NO** - When sending messages, the JMS_IBM_MQMD* properties on a sent message are not updated to reflect the updated field values in the MQMD. When receiving messages, none of the JMS_IBM_MQMD* properties are available on a received message, even if the sender had set some or all of them. • Yes - When sending messages, all of the JMS_IBM_MQMD* properties on a sent message are updated to reflect the updated field values in the MQMD, including those that the sender did not set explcitly. When receiving messages, all of the JMS_IBM_MQMD* properties are available on a received message, including those that the sender did not set explicitly.	• **False** • True
WMQ_MQMD_WRITE_ ENABLED	setMQMDWriteEnabled	• **NO** - All JMS_IBM_MQMD* properties are ignored and their values are not copied into the underlying MQMD structure • YES - JMS_IBM_MQMD* properties are processed. Their values are copied into the underlying MQMD structure •	• **False** • True

Table 45. Property names, set methods and values (continued)

Property	Set method	Valid values in administration tool (defaults in bold)	Valid values in programs

Notes::

1. This property can be used with Version 7.0 of WebSphere MQ classes for JMS but has no effect for an application connected to a Version 7.0 queue manager unless the PROVIDERVERSION property of the connection factory is set to a version number less than 7.
2. The CCDTURL and CHANNEL properties of an object must not both be set at the same time.
3. Running two or more instances of the same durable topic subscriber simultaneously contravenes the *Java Message Service Specification, Version 1.1*.
4. The binding options, STANDARD, SHARED, ISOLATED, and FASTPATH, are ignored if the application connects in client mode. The SHARED, ISOLATED, and FASTPATH options are ignored by a queue manager running on z/OS. The connection tag options, SERIALQM, SERIALQSG, RESTRICTQM, and RESTRICTQSG, are supported only by a queue manager running on z/OS. For a more detailed explanation of the connection options, see the *WebSphere MQ Application Programming Guide* and *WebSphere MQ Application Programming Reference*.
5. The CONNTAG property is supported only by a queue manager running on z/OS.
6. If you specify NO, and a message is rolled back after WebSphere MQ classes for JMS has returned control to the subscriber, the subscriber still retains a copy of the message but is not informed of the rollback. In this situation, a subscriber might receive the same message more than once.
7. The values PUTASYNCALLOWED_AS_Q_DEF, PUTASYNCALLOWED_AS_TOPIC_DEF, and PUTASYNCALLOWED_AS_DEST are synonymous and can be used interchangeably.
8. The values READAHEADALLOWED_AS_Q_DEF, READAHEADALLOWED_AS_TOPIC_DEF, and READAHEADALLOWED_AS_DEST are synonymous and can be used interchangeably.
9. The value UNCOMMITTED has an effect only if PROCESSDURATION has the value SHORT. It has no effect if PROCESSDURATION has the value UNKNOWN. If you specify UNCOMMITTED, ensure that a subscriber acknowledges or commits each message individually.
10. The TARGCLIENT property indicates whether the WebSphere MQ RFH2 format is used to exchange information with target applications.

    The MQJMS_CLIENT_JMS_COMPLIANT constant indicates that the RFH2 format is used to send information. Applications that use WebSphere MQ classes for JMS understand the RFH2 format. Set the MQJMS_CLIENT_JMS_COMPLIANT constant when you exchange information with a target WebSphere MQ classes for JMS application.

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

Property dependencies

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

The specific property groups where this can occur are:

- Client properties
- Properties for a real-time connection to a broker
- Exit initialization strings

Client properties

For a connection to a queue manager, the following properties are relevant only if TRANSPORT has the value CLIENT:

- HOSTNAME
- PORT
- CHANNEL
- LOCALADDRESS
- CCDTURL
- CCSID
- COMPHDR

- COMPMSG
- RECEXIT
- RECEXITINIT
- SECEXIT
- SECEXITINIT
- SENDEXIT
- SENDEXITINIT
- SHARECONVALLOWED
- SSLCIPHERSUITE
- SSLCRL
- SSLFIPSREQUIRED
- SSLPEERNAME
- SSLRESETCOUNT

Using the administration tool, you cannot set values for these properties if TRANSPORT has the value BIND.

If TRANSPORT has the value CLIENT, the default value of the BROKERVER property is V1 and the default value of the PORT property is 1414. If you set the value of BROKERVER or PORT explicitly, a later change to the value of TRANSPORT does not override your choices.

Properties for a real-time connection to a broker

Only the following properties are relevant if TRANSPORT has the value DIRECT or DIRECTHTTP:

- BROKERVER
- CLIENTID
- DESCRIPTION
- DIRECTAUTH
- HOSTNAME
- LOCALADDRESS
- MAXBUFFSIZE
- MULTICAST (supported only for DIRECT)
- PORT
- PROXYHOSTNAME (supported only for DIRECT)
- PROXYPORT (supported only for DIRECT)

If TRANSPORT has the value DIRECT or DIRECTHTTP, the default value of the BROKERVER property is V2, and the default value of the PORT property is 1506. If you set the value of BROKERVER or PORT explicitly, a later change to the value of TRANSPORT does not override your choices.

Exit initialization strings

Do not set any of the exit initialization strings without supplying the corresponding exit name. The exit initialization properties are:

- RECEXITINIT
- SECEXITINIT
- SENDEXITINIT

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

The ENCODING property

The ENCODING property comprises three sub-properties, in twelve possible combinations.

The valid values that the `ENCODING` property can take are constructed from the three sub-properties:

integer encoding
: Either normal or reversed

decimal encoding
: Either normal or reversed

floating-point encoding
: IEEE normal, IEEE reversed, or z/OS

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

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

In this string:

- `N` denotes normal
- `R` denotes reversed
- `3` denotes z/OS
- The first character represents *integer encoding*
- The second character represents *decimal encoding*
- The third character represents *floating-point encoding*

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

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

The following examples show valid combinations for `ENCODING`:

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

SSL properties

Enable Secure Sockets Layer (SSL) encryption using the SSLCIPHERSUITE property. You can then change the characteristics of the SSL encryption using several other properties.

When you specify TRANSPORT(CLIENT), you can enable Secure Sockets Layer (SSL) encrypted communication using the SSLCIPHERSUITE property. Set this property to a valid CipherSuite provided by your JSSE provider; it must match the CipherSpec named on the SVRCONN channel named by the CHANNEL property.

However, CipherSpecs (as specified on the SVRCONN channel) and CipherSuites (as specified on ConnectionFactory objects) use different naming schemes to represent the same SSL encryption algorithms. If a recognized CipherSpec name is specified on the SSLCIPHERSUITE property, JMSAdmin issues a warning and maps the CipherSpec to its equivalent CipherSuite. See “SSL CipherSpecs and CipherSuites” on page 145 for a list of CipherSpecs recognized by WebSphere MQ and JMSAdmin.

If you require a connection to use a CipherSuite that is supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS), set the SSLFIPSREQUIRED property of the connection factory to YES. The default value of this property is NO, which means that a connection can use any supported CipherSuite. The property is ignored if SSLCIPHERSUITE is not set.

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

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

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

The example given checks the identifying certificate presented by the server at connect-time. For the connection to succeed, the certificate must have a Common Name beginning QMGR., and must have at least two Organizational Unit names, the first of which is IBM and the second WEBSPHERE. Checking is not case-sensitive.

If SSLPEERNAME is not set, no such checking is performed. SSLPEERNAME is ignored if SSLCIPHERSUITE is not set.

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

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

optionally followed by a single `/`. If `port` is omitted, the default LDAP port of 389 is assumed. At connect-time, the SSL certificate presented by the server is checked against the specified CRL servers. See *WebSphere MQ Security* for more about CRL security.

If SSLCRL is not set, no such checking is performed. SSLCRL is ignored if SSLCIPHERSUITE is not set.

The SSLRESETCOUNT property represents the total number of bytes sent and received by a connection before the secret key that is used for encryption is renegotiated. The number of bytes sent is the number before encryption, and the number of bytes received is the number after decryption. The number of bytes also includes control information sent and received by WebSphere MQ classes for JMS.

For example, to configure a ConnectionFactory object that can be used to create a connection over an SSL enabled MQI channel whose secret key is renegotiated after 4 MB of data have flowed, issue the following command to JMSAdmin:

```
ALTER CF(my.cf) SSLRESETCOUNT(4194304)
```

If the value of SSLRESETCOUNT is zero, which is the default value, the secret key is never renegotiated. The SSLRESETCOUNT property is ignored if SSLCIPHERSUITE is not set.

If you are using an HP or Sun Java 2 Software Development Kit (SDK) or Java Runtime Environment (JRE), do not set SSLRESETCOUNT to a value other than zero. If you do set SSLRESETCOUNT to a value other than zero, a connection fails when it attempts to renegotiate the secret key.

For more information about the secret key that is used for encryption on an SSL enabled channel, see *WebSphere MQ Security*.

Rules for selecting the WebSphere MQ messaging provider mode

The WebSphere MQ messaging provider has two modes of operation: WebSphere MQ messaging provider normal mode and WebSphere MQ messaging provider migration mode. The WebSphere MQ messaging provider normal mode uses all the features of the WebSphere MQ Version 7.0 queue managers to implement JMS. This mode is used only to connect to a WebSphere MQ queue manager and can connect to WebSphere MQ Version 7.0 queue managers in either client or bindings mode. This mode is optimized to use the new WebSphere MQ Version 7.0 function.

If you are not using WebSphere MQ Real-Time Transport, the mode of operation used is determined primarily by the PROVIDERVERSION property of the connection factory. If you cannot change the connection factory you are using, you can use a client configuration property instead called com.ibm.msg.client.wmq.overrideProviderVersion that overrides any setting on the connection factory. This override applies to all connection factories in the JVM but the actual connection factory objects are not modified. You can set PROVIDERVERSION to three possible values: 7, 6, or unspecified:

PROVIDERVERSION=7
Uses the WebSphere MQ messaging provider normal mode.

If you set PROVIDERVERSION to 7 only the WebSphere MQ messaging provider normal mode of operation is available. If the queue manager that is connected to as a result of the other settings in the connection factory is not a Version 7.0 queue manager, the createConnection() method fails with an exception.

The WebSphere MQ messaging provider normal mode uses the sharing conversations feature and the number of conversations that can be shared is controlled by the SHARECNV() property on the server connection channel. If this property is set to 0, you cannot use WebSphere MQ messaging provider normal mode and the createConnection() method fails with an exception.

PROVIDERVERSION=6
Uses the WebSphere MQ messaging provider migration mode.

The WebSphere MQ classes for JMS use the features and algorithms supplied with WebSphere MQ Version 6.0. If you want to connect to WebSphere Event Broker or WebSphere Message Broker using WebSphere MQ Enterprise Transport, you must use this mode. You can connect to a WebSphere MQ Version 7.0 queue manager using this mode, but none of the new features of a Version 7.0 queue manager are used, for example, read ahead or streaming.

PROVIDERVERSION=unspecified
This is the default value and the actual text is "unspecified".

A connection factory that was created with a previous version of WebSphere MQ classes for JMS in JNDI takes this value when the connection factory is used with the new version of WebSphere MQ classes for JMS. The following algorithm is used to determine which mode of operation is used. This algorithm is used when the createConnection() method is called and uses other aspects of the connection factory to determine if WebSphere MQ messaging provider normal mode or WebSphere MQ messaging provider migration mode is required.

- Firstly, an attempt to use WebSphere MQ messaging provider normal mode is made.
- If the queue manager connected is not WebSphere MQ Version 7.0, the connection is closed and WebSphere MQ messaging provider migration mode is used instead.
- If the SHARECNV() property on the server connection channel is set to 0, the connection is closed and WebSphere MQ messaging provider migration mode is used instead.
- If BROKERVER is set to 1 or the new default "unspecified" value, WebSphere MQ messaging provider normal mode continues to be used, and therefore any publish/subscribe operations use the new WebSphere MQ V7.0 features.

  If WebSphere Event Broker or WebSphere Message Broker are used in compatibility mode (and you want to use Version 6.0 publish/subscribe function rather than the WebSphere MQ Version 7 publish/subscribe function), set PROVIDERVERSION to 6 ensure WebSphere MQ messaging provider migration mode is used.
- If BROKERVER is set to V2 and the value of BROKERQMGR is one of the following:
  - If BROKERQMGR is nonblank, this means BROKERQMGR has been explicitly changed from the default, so the assumption is the connection factory really is intended for use with WebSphere Event Broker or WebSphere Message Broker and WebSphere MQ Enterprise Transport. Therefore WebSphere MQ messaging provider migration mode is used.
  - If BROKERQMGR is blank **and** if the specified BROKERCONQ command queue exists and can be opened for output (that is, MQOPEN for output succeeds) **and** PSMODE on the queue manager is set to COMPAT or DISABLED, WebSphere MQ messaging provider migration mode is used.

You can find further guidance about using PROVIDERVERSION in When to use PROVIDERVERSION

When to use PROVIDERVERSION

There are two scenarios where you cannot use the algorithm described in Rules for selecting the WebSphere MQ messaging provider mode; consider using PROVIDERVERSION in these scenarios.

1. If WebSphere Event Broker or WebSphere Message Broker is in compatibility mode, you must specify PROVIDERVERSION for them to work correctly.
2. If you are using WebSphere Application Server Version 6.0.1, WebSphere Application Server Version 6.0.2, or WebSphere Application Server Version 6.1, connection factories are defined using the WebSphere Application Server administrative console.

   In WebSphere Application Server the default value of the BROKERVER property on a connection factory is V2. The default BROKERVER property for connection factories created by using JMSAdmin or WebSphere MQ Explorer is V1. This property is now ″unspecified″ in WebSphere MQ Version 7.0.

If BROKVERVER is set to V2 (either because it was created by WebSphere Application Server or the connection factory has been used for publish/subscribe before) and has an existing queue manager that has a BROKECONQ defined (because it has been used for publish/subscribe messaging before), the WebSphere MQ messaging provider migration mode is used.

However, if you want the application to use peer-to-peer communication and the application is using an existing queue manager that has ever done publish/subscribe, and has a connection factory with BROKERVER set to 2 (if the connection factory was created in WebSphere Application Server this is the default), the WebSphere MQ messaging provider migration mode is used. Using WebSphere MQ messaging provider migration mode in this case is unnecessary; use WebSphere MQ messaging provider normal mode instead. You can use one of the following methods to work around this:

- Set BROKERVER to 1 or unspecified. This is dependent on your application.
- Set PROVIDERVERSION to 7, which is a custom property in WebSphere Application Server Version 6.1. The option to set custom properties in WebSphere Application Server Version 6.1 and later is not currently documented in the WebSphere Application Server Information Center.

  Alternatively, use the client configuration property (see Rules for selecting the WebSphere MQ messaging provider mode for details about how you can specify this system property for all environments), or modify the queue manager connected so it does not have the BROKERCONQ, or make the queue unusable.

WebSphere MQ classes for JMS packages

For details of the Java classes and interfaces in the various packages the comprise WebSphere MQ classes for JMS see the information center or the Javadoc documentation included on the product CDs.

Package com.ibm.jms

This package contains a set of classes that implement the JMS message interfaces.

Package com.ibm.mq

WebSphere MQ classes for Java consist of a set classes that provide an object model that maps directly to the WMQ object model.

Package com.ibm.mq.constants

This package contains the Java versions of the MQ header files, which are implemented as Java interfaces which define constants used with the main MQI.

Package com.ibm.mq.exits

This package comprises a set of classes and interfaces which allow the Java programmer to work with MQ Channel Exits.

Package com.ibm.mq.jmqi

The Java Message Queueing Interface (JMQI) is the interface which represents the native MQI in the Java environment.

Package com.ibm.mq.jms

WebSphere MQ classes for Java Message Service consist of a number of Java classes and interfaces that are based on the Sun javax.jms package of interfaces and classes.

Package com.ibm.msg.client.jms

This package contains interfaces and classes that extend the javax.jms interfaces and classes, providing additional functionality in a way that is independent of WebSphere MQ.

Package com.ibm.msg.client.services

Provides other services and information associated with the Classes for JMS.

Package com.ibm.msg.client.wmq

This package comprises a set of classes and interfaces specific to IBM WebSphere MQ Provider.

Package com.ibm.msg.client.wmq.common

This package comprises a set of helper classes and interfaces specific to IBM WebSphere MQ Provider.

Chapter 3. WebSphere MQ classes for Java

This collection of topics contains the documentation for WebSphere MQ classes for Java.

Getting started with WebSphere MQ classes for Java

This collection of topics gives an overview of WebSphere MQ classes for Java and their uses.

What are WebSphere MQ classes for Java?

WebSphere MQ classes for Java allow you to use WebSphere MQ in a Java environment.

WebSphere MQ classes for Java allow a Java application to:
- Connect to WebSphere MQ as a WebSphere MQ client
- Connect directly to a WebSphere MQ queue manager

WebSphere MQ classes for Java encapsulate the Message Queue Interface (MQI), the native WebSphere MQ API.

WebSphere MQ classes for Java use a similar object model to the C++ and .NET interfaces to WebSphere MQ.

Why should I use WebSphere MQ classes for Java?

A Java application can use either WebSphere MQ classes for Java or WebSphere MQ classes for JMS to access WebSphere MQ resources. There are a number of advantages to using WebSphere MQ classes for Java.

If the following points are significant in your installation, consider using Websphere MQ classes for Java:
- WebSphere MQ classes for Java encapsulate the Message Queue Interface (MQI), the native WebSphere MQ API.
  - If you are familiar with the use of the MQI in procedural languages, you can transfer this knowledge to the Java environment.
  - You can exploit the full range of features of WebSphere MQ, beyond those available through JMS.
- WebSphere MQ classes for Java use a similar object model to the C++ and .NET interfaces to WebSphere MQ. If you are familiar with these interfaces, you can transfer this knowledge to the Java environment.

Connection options for WebSphere MQ classes for Java

WebSphere MQ classes for Java can connect in client or bindings mode.

Programmable options allow WebSphere MQ classes for Java to connect to WebSphere MQ in either of the following ways:
- As a WebSphere MQ client using Transmission Control Protocol/Internet Protocol (TCP/IP)
- In bindings mode, connecting directly to WebSphere MQ using the Java Native Interface (JNI)

Clients cannot be run on z/OS, but clients on other platforms can connect to a WebSphere MQ for z/OS queue manager if the Client Attach Facility is installed.

The following sections describe the client mode and bindings mode connection options in more detail.

Client connection

To connect to a queue manager in client mode, a WebSphere MQ classes for JMS application can run on the same system on which the queue manager is running, or on a different system. In each case, WebSphere MQ classes for JMS connects to the queue manager over TCP/IP.

Bindings connection

When used in bindings mode, WebSphere MQ classes for Java uses the Java Native Interface (JNI) to call directly into the existing queue manager API, rather than communicating through a network. In most environments, connecting in bindings mode provides better performance for WebSphere MQ classes for Java applications than connecting in client mode, by avoiding the overheads of TCP/IP communication.

Prerequisites for WebSphere MQ classes for Java

To use WebSphere MQ classes for Java, you need certain other software products.

For the latest information about the prerequisites for WebSphere MQ classes for Java, see the WebSphere MQ README file.

To develop WebSphere MQ classes for Java applications, you need a Java Development Kit (JDK). Details of the JDKs supported with your operating system can be found on the WebSphere MQ System requirements page.

See http://www-01.ibm.com/software/integration/wmq/requirements/index.html

A suitable JDK is supplied with WebSphere MQ.

To run WebSphere MQ classes for Java applications, you need the following software components:

- A WebSphere MQ queue manager, for applications that connect to a queue manager
- A Java Runtime Environment (JRE), for each system on which you run applications. A suitable JRE is supplied with WebSphere MQ.
- For i5/OS, QShell, which is option 30 of the operating system
- For z/OS, UNIX System Services (USS)

If you require SSL connections to use cryptographic modules that have been FIPS 140-2 certified, you need the IBM Java JSSE FIPS provider (IBMJSSEFIPS). Every IBM JDK and JRE at Version 1.4.2 or later contains IBMJSSEFIPS.

You can use Internet Protocol Version 6 (IPv6) addresses in your WebSphere MQ classes for Java applications if IPv6 is supported by your Java virtual machine (JVM) and the TCP/IP implementation on your operating system.

Installation and configuration of WebSphere MQ classes for Java

This chapter describes the directories and files that are created when you install WebSphere MQ classes for Java, and tells you how to configure WebSphere MQ classes for Java after installation.

What is installed for WebSphere MQ classes for Java

The latest version of WebSphere MQ classes for Java is installed with WebSphere MQ. You might need to override default installation options to make sure this is done.

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

WebSphere MQ for AIX Quick Beginnings
WebSphere MQ for HP-UX Quick Beginnings
WebSphere MQ for i5/OS Quick Beginnings
WebSphere MQ for Linux Quick Beginnings
WebSphere MQ for Solaris Quick Beginnings
WebSphere MQ for Windows Quick Beginnings
WebSphere MQ for z/OS Program Directory

WebSphere MQ classes for Java are contained in the Java archive (JAR) files, com.ibm.mq.jar, and com.ibm.mq.jmqi.jar.

Support for standard message headers, such as Programmable Command Format (PCF), is contained in the JAR file com.ibm.mq.headers.jar.

Support for Programmable Command Format (PCF) is contained in the JAR file com.ibm.mq.pcf.jar.

The following Java library from Sun Microsystems is distributed with WebSphere MQ classes for Java:

- connector.jar (Version 1.0)

The sample application called Postcard is in the JAR file com.ibm.mq.postcard.jar. For more information about this application, see the *Quick Beginnings* information for your operating platform.

The Javadoc tool has been used to generate the HTML pages containing the specifications of the WebSphere MQ classes for Java and WebSphere MQ classes for JMS APIs. The HTML pages are in the doc subdirectory of the WebSphere MQ classes for JMS installation directory. On UNIX systems and Windows, the doc subdirectory contains the individual HTML pages but, on i5/OS and z/OS, the HTML pages are in a file called wmqjms_javadoc.jar.

When installation is complete, files and samples are installed in the locations shown in “Installation directories for WebSphere MQ classes for Java” on page 220.

After installation, on any platform other than Windows, you must update your environment variables as described in “Environment variables relevant to WebSphere MQ classes for Java” on page 220.

Installation directories for WebSphere MQ classes for Java

WebSphere MQ classes for Java files are installed in different locations according to platform.

Table 46 shows where the WebSphere MQ classes for Java files are installed.

Table 46. WebSphere MQ classes for Java installation directories

Platform	Directory
AIX	/usr/mqm/java/lib
HP-UX, Linux, and Solaris	/opt/mqm/java/lib
i5/OS	/QIBM/ProdData/mqm/java/lib
Windows	*install_dir*\java\lib
z/OS	*install_dir*/mqm/V7R0M0/java/lib
Note: *install_dir* is the directory in which you installed WebSphere MQ. On Windows, this directory is normally C:\Program Files\IBM\WebSphere MQ. On z/OS, this directory is likely to be /usr/lpp.	

Some sample applications, such as the Installation Verification Programs (IVPs), are supplied with WebSphere MQ. Table 47 shows where the sample applications are installed. The WebSphere MQ classes for Java samples are in a subdirectory called wmqjava.

Table 47. Samples directories

Platform	Directory
AIX	/usr/mqm/samp/wmqjava/
HP-UX, Linux, and Solaris	/opt/mqm/samp/wmqjava/
i5/OS	/QIBM/ProdData/mqm/java/samples
Windows	*install_dir*\tools\wmqjava\
z/OS	*install_dir*/mqm/V7R0M0/java/samples
Note: *install_dir* is the directory in which you installed WebSphere MQ. On Windows, this directory is normally C:\Program Files\IBM\WebSphere MQ. On z/OS, this directory is likely to be /usr/lpp.	

Environment variables relevant to WebSphere MQ classes for Java

Some environment variables must be set. Specimen values for classpath are given.

For WebSphere MQ classes for Java applications to run, their class path must include the appropriate WebSphere MQ classes for Java directory. To run the sample applications, the class path must also include the appropriate samples directories. This information can be provided in the Java invocation command or in the CLASSPATH environment variable.

Table 48 on page 221 shows the appropriate CLASSPATH setting to use on each platform to run WebSphere MQ classes for Java applications, including the sample applications.

Table 48. CLASSPATH setting to run WebSphere MQ classes for Java applications, including the WebSphere MQ classes for Java sample applications

Platform	CLASSPATH setting
AIX	CLASSPATH=/usr/mqm/java/lib/com.ibm.mq.jar: /usr/mqm/samp/wmqjava:
HP-UX, Linux, and Solaris	CLASSPATH=/opt/mqm/java/lib/com.ibm.mq.jar: /opt/mqm/samp/wmqjava:
i5/OS	CLASSPATH=/QIBM/ProdData/mqm/java/lib/com.ibm.mq.jar: /QIBM/ProdData/mqm/java/samples/wmqjava:
Windows	CLASSPATH=*install_dir*\Java\lib\com.ibm.mq.jar; *install_dir*\tools\wmqjava;
z/OS	CLASSPATH=*install_dir*/mqm/V7R0M0/java/lib/com.ibm.mq.jar: *install_dir*/mqm/V7R0M0/java/samples/wmqjava:
Note: *install_dir* is the directory in which you installed WebSphere MQ. On Windows, this directory is normally C:\Program Files\IBM\WebSphere MQ. On z/OS, this directory is likely to be /usr/lpp.	

The scripts provided with WebSphere MQ classes for Java use the following environment variables:

MQ_JAVA_DATA_PATH
This environment variable specifies the directory for log and trace output.

MQ_JAVA_INSTALL_PATH
This environment variable specifies the directory where WebSphere MQ classes for Java are installed, as shown in Table 46 on page 220.

MQ_JAVA_LIB_PATH
This environment variable specifies the directory where the WebSphere MQ classes for Java libraries are stored, as shown in Table 49 on page 222. Some scripts supplied with WebSphere MQ classes for Java, such as IVTRun, use this environment variable.

On Windows, all the environment variables are set automatically during installation. On any other platform, you must set them yourself. On a UNIX system, you can use the script **setjmsenv** (if you are using a 32-bit JVM) or **setjmsenv64** (if you are using a 64-bit JVM) to set the environment variables. On AIX, these scripts are in the /usr/mqm/java/bin directory and, on HP-UX, Linux, and Solaris, they are in the /opt/mqm/java/bin directory.

On i5/OS, the environment variable QIBM_MULTI_THREADED must be set to Y. You can then run multithreaded applications in the same way that you run single threaded applications.

The WebSphere MQ classes for Java libraries

The location of the WebSphere MQ classes for Java libraries varies according to platform. Specify this location when you start an application.

| To specify the location of the Java Native Interface (JNI) libraries, start your application using a **java** command with the following format:

```
java -Djava.library.path=library_path application_name
```

| where *library_path* is the path to the WebSphere MQ classes for Java libraries,
| which include the JNI libraries. Table 49 on page 222 shows the location of the WebSphere MQ classes for Java libraries for each platform.

Table 49. The location of the WebSphere MQ classes for Java libraries for each platform

Platform	Directory containing the WebSphere MQ classes for Java libraries
AIX	/usr/mqm/java/lib (32-bit libraries) /usr/mqm/java/lib64 (64-bit libraries)
HP-UX Linux (POWER, x86-64 and zSeries s390x platforms) Solaris (x86-64 and Sparc platforms)	/opt/mqm/java/lib (32-bit libraries) /opt/mqm/java/lib64 (64-bit libraries)
Linux (x86 platform)	/opt/mqm/java/lib
Windows	*install_dir*\Java\lib (32-bit libraries) *install_dir*\Java\lib64 (64-bit libraries)
z/OS	*install_dir*/mqm/V7R0M0/java/lib (31-bit and 64-bit libraries)
Note: *install_dir* is the directory in which you installed WebSphere MQ. On Windows, this directory is normally C:\Program Files\IBM\WebSphere MQ. On z/OS, this directory is likely to be /usr/lpp.	

Note:

1. On AIX, HP-UX, Linux (POWER platform), or Solaris, use either the 32-bit libraries or the 64-bit libraries. Use the 64-bit libraries only if you are running your application in a 64-bit Java virtual machine (JVM) on a 64-bit platform. Otherwise, use the 32-bit libraries.
2. On Windows, you can use the PATH environment variable to specify the location of the WebSphere MQ classes for Java libraries instead of specifying their location on the java command.
3. To use WebSphere MQ classes for Java in bindings mode on i5/OS, ensure that the library QMQMJAVA is in your library list.
4. On z/OS, you can use either a 31-bit or 64-bit Java virtual machine (JVM) when running applications in WebSphere Application Server. In other environments on z/OS, you can use only a 31-bit JVM. You do not have to specify which libraries to use and you do not need to modify the system path to use 64-bit support.

STEPLIB configuration on z/OS

On z/OS, the STEPLIB used at runtime must contain the WebSphere MQ SCSQAUTH and SCSQANLE libraries.

From UNIX System Services, you can add these using a line in your `.profile` as shown below, replacing `thlqual` with the high level data set qualifier that you chose when installing WebSphere MQ:

```
export STEPLIB=thlqual.SCSQAUTH:thlqual.SCSQANLE:$STEPLIB
```

In other environments, you typically need to edit the startup JCL to include SCSQAUTH on the STEPLIB concatenation:

```
 STEPLIB DD DSN=thlqual.SCSQAUTH,DISP=SHR
         DD DSN=thlqual.SCSQANLE,DISP=SHR
```

Running WebSphere MQ classes for Java applications under the Java Security Manager

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

The simplest way to do this is to change the policy file supplied with the JRE. On most systems this file is stored in the path `lib/security/java.policy`, relative to your JRE directory. You can edit policy files using your preferred editor or the `policytool` program supplied with your JRE.

| You need to give authority to the com.ibm.mq.jmqi.jar file so that it can:

- Create sockets (in client mode)
- Load the native library (in bindings mode)
- Read various properties from the environment

The system property os.name must be available to the WebSphere MQ classes for Java when running under the Java Security Manager.

Here is an example of a policy file entry that allows WebSphere MQ classes for Java to run successfully under the default security manager. Replace the string `/opt/mqm` in this example with the location where WebSphere MQ classes for Java are installed on your system.

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

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

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

Running WebSphere MQ classes for Java applications under CICS Transaction Server

A WebSphere MQ classes for Java application can be run as a transaction under CICS® Transaction Server.

To run a WebSphere MQ classes for Java application as a transaction under CICS
| Transaction Server for z/OS, perform the following steps:

1. Define the application and transaction to CICS by using the supplied CEDA transaction.
2. Ensure that the WebSphere MQ CICS adapter is installed in your CICS system. (See WebSphere MQ for z/OS System Setup Guide for details.)
3. Ensure that the JVM environment specified in CICS includes the appropriate CLASSPATH and LIBPATH entries.
4. Initiate the transaction by using any of your normal processes.

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

Using WebSphere MQ classes for Java

This collection of topics tells you how to configure your system to run the sample applications to verify your WebSphere MQ classes for Java installation, and how to modify the procedures to run your own applications.

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

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

Before attempting to run any WebSphere MQ classes for Java applications in bindings mode, make sure that you have configured WebSphere MQ as described in the Quick Beginnings book for your platform or the WebSphere MQ for z/OS System Setup Guide.

Configuring your queue manager to accept client connections

| To configure your queue manager to accept incoming connection requests from
| clients, define a server connection channel and start a listener program.

TCP/IP client

| 1. Define a server connection channel. You can either use the WebSphere MQ
| Explorer or the following procedures, depending on your platform:

For i5/OS:

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

b. Define a sample channel called JAVA.CHANNEL by issuing the following command:

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

where `QMGRNAME` is the name of your queue manager, and `SOMEUSERID` is an i5/OS user ID with appropriate authority to the WebSphere MQ resources.

For z/OS:

Note: You must have the Client Attachment feature installed on your target queue manager in order to connect using TCP/IP.

a. Start your queue manager by using the START QMGR command.

b. Define a sample channel called JAVA.CHANNEL by issuing the following command:

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

For other platforms:

a. Start your queue manager by using the `strmqm` command.

b. Type the following command to start the runmqsc program:

```
runmqsc [QMNAME]
```

c. Define a sample channel called JAVA.CHANNEL by issuing the following command:

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

2. Start a listener program with the following commands:

   For UNIX and Windows systems:
   Issue one of the following commands:
   - `runmqlsr -t tcp [-m QMNAME] -p nnnn`

     Note: If you use the default queue manager, you can omit the -m option. If you use the default port, 1414, you can omit the -p option.
   - `START LISTENER TRPTYPE(TCP) PORT(nnnn)`

   Replace the string *nnnn* with your chosen port number.

   For i5/OS:
   Issue the command:
   ```
   STRMQMLSR MQMNAME(QMGRNAME)
   ```
   where `QMGRNAME` is the name of your queue manager.

   For z/OS:
   a. Ensure your channel initiator is started. If not, start it by issuing the START CHINIT command.
   b. Start the listener by issuing the command:
   ```
   START LISTENER TRPTYPE(TCP) PORT(nnnn)
   ```
   Replace the string *nnnn* with your chosen port number.

Verifying your WebSphere MQ classes for Java installation with the sample application

An installation verification program, MQIVP, is supplied with WebSphere MQ classes for Java. You can use this program to test all the connection modes of WebSphere MQ classes for Java.

The program prompts for a number of choices and other data to determine which connection mode you want to verify. Use the following procedure to verify your installation:

1. If you are going to run the program in client mode, configure your queue manager as described in "Configuring your queue manager to accept client connections" on page 224.
2. The user ID associated with the program when it runs must have authority to access certain resources of the queue manager. Grant the following authorities to the user ID:
   - The authority to connect to the queue manager, and the authority to inquire on the attributes of the queue manager object
   - The authority to put messages on the queue SYSTEM.DEFAULT.LOCAL.QUEUE, and the authority to get messages from the queue

   For information about how to grant authorities, see the following books:
   - WebSphere MQ for i5/OS System Administration Guide, if the queue manager is running on i5/OS
   - WebSphere MQ System Administration Guide, if the queue manager is running on a UNIX system or Windows
   - WebSphere MQ for z/OS System Setup Guide, if the queue manager is running on z/OS

If you are going to run the program in client mode, see also WebSphere MQ Clients.

Perform the remaining steps of this procedure on the system on which you are going to run the program.

3. Make sure that you have updated your CLASSPATH environment variable according to the instructions in "Environment variables relevant to WebSphere MQ classes for Java" on page 220.
4. At a command prompt, enter:
   ```
   java -Djava.library.path=library_path MQIVP
   ```
   where *library_path* is the path to the WebSphere MQ classes for Java libraries (see "The WebSphere MQ classes for Java libraries" on page 221).

   The program tries to:
   a. Connect to the queue manager
   b. Open the queue SYSTEM.DEFAULT.LOCAL.QUEUE, put a message on the queue, get a message from the queue, and then close the queue
   c. Disconnect from the queue manager
   d. Return a message if the operations are successful
5. At the prompt marked (§):
   - To use a TCP/IP connection, enter a WebSphere MQ server host name.
   - To use native connection (bindings mode), leave the field blank. (Do not enter a name.)

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

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


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

Note:
1. On z/OS, leave the field blank at prompt marked (§).
2. If you choose server connection, you do not see the prompts marked (§§).
3. On i5/OS, you can issue the command `java MQIVP` only from QShell. Alternatively, you can run the application by using the CL command `RUNJVA CLASS(MQIVP)`.

Solving WebSphere MQ classes for Java problems

Initially, run the installation verification program. You might also have to use the trace facility.

If a program does not complete successfully, run the installation verification program, and follow the advice given in the diagnostic messages. This program is described in “Verifying your WebSphere MQ classes for Java installation with the sample application” on page 225.

If the problems continue and you need to contact the IBM service team, you might be asked to turn on the trace facility. Do this as shown in the following example.

To trace the `MQIVP` program, enter the following command:

```
java -Djava.library.path=library_path MQIVP -trace
```

where *library_path* is the path to the WebSphere MQ classes for Java libraries (see “The WebSphere MQ classes for Java libraries” on page 221).

For more information about how to use trace, see “Tracing WebSphere MQ classes for Java programs” on page 270.

Introduction for programmers

This collection of topics contains general information for programmers.

For more detailed information about writing programs, see “Writing WebSphere MQ classes for Java applications” on page 229.

The WebSphere MQ classes for Java interface

The procedural WebSphere MQ application programming interface is built around verbs such as those listed below:

```
MQBACK, MQBEGIN, MQCLOSE, MQCMIT, MQCONN, MQDISC,
MQGET, MQINQ, MQOPEN, MQPUT, MQSET, MQSUB, MQSUBRQ
```

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

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

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

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

What is new in Websphere MQ Version 7.0?

Websphere MQ classes for Java, as supplied in WebSphere MQ Version 7.0, contains a number of enhancements compared to previous releases.

The following sections summarize the key enhancements.

Embedded publish/subscribe function

The API is extended to support publish/subscribe applications. For more information, see "Publish/subscribe in WebSphere MQ classes for Java" on page 237.

Read ahead

A WebSphere MQ classes for Java client can be configured to use *read ahead*. Read ahead allows messages to be sent to a client before an application requests them. For more information, see "Improving the performance of nonpersistent messages" on page 237.

Asynchronous put

In WebSphere MQ Version 7.0, you can choose to put a message to a queue or topic using the MQPUT or MQPUT1 call without the application waiting for the queue manager to complete the call. In procedural languages, this is done by using the MQPUT or MQPUT1 call, setting the MQPMO_ASYNC_REPONSE option. In WebSphere MQ classes for Java, you set MQPMO_ASYNC_RESPONSE in the MQPutMessageOptions parameter passed to MQDestination.put(). For more information, see "Putting messages asynchronously using WebSphere MQ classes for Java" on page 237

Sharing a communications connection

In WebSphere MQ V7.0, each connection to the queue manager using the same MQI channel can share a single TCP connection. This arrangement means that fewer network resources are required and the total time taken to create multiple connections to the queue manager is reduced, particularly when using SSL because the SSL handshake takes place only once at the start of the TCP connection. For more information, see "Sharing a TCP/IP connection in WebSphere MQ classes for Java" on page 254.

Message properties

WebSphere MQ V7.0 introduces the general availability of *message properties*, previously only available in WebSphere MQ classes for JMS. You can add new properties to any message without affecting applications that currently process that message. For more information, see "Handling message properties" on page 244.

Existing applications which rely on the presence of MQRFH2s containing properties must be changed to use the get*Property methods or to specify MQC.MQGMO_PROPERTIES_FORCE_MQRFH2 to continue receiving properties in the MQRFH2 format. If the application cannot be changed then the queue attribute PropertyControl can be changed to the value MQPROP_FORCE_MQRFH2 so that properties are always returned in MQRFH2 form.

WebSphere MQ message headers

Java classes representing various types of message header are provided, with two helper classes to assist with reading and parsing header content. For more information, see "Handling WebSphere MQ message headers" on page 237.

PCF messages

Java classes are provided to create and parse PCF-structured messages, and to facilitate sending PCF requests and collecting PCF responses. For more information, see “Handling PCF messages” on page 243.

Channel exits

Three new interfaces are defined. Use these to define classes to use as channel exits. The old channel exit interfaces are also still supported, but the new interfaces offer improved functionality and performance. For more information, see “Using channel exits in WebSphere MQ classes for Java” on page 247.

Client configuration files

On all platforms, the WebSphere® MQ client configuration file is used to specify the client configuration options. These apply to WebSphere MQ classes for Java applications as well as to applications using procedural languages. For more information see “Client connections.”

MQC is replaced by MQConstants

A new package, com.ibm.mq.constants, is supplied with WebSphere MQ Version 7.0. This package contains the class MQConstants, which implements a number of interfaces. MQConstants contains definitions of all the constants that were in the MQC interface and a number of new constants. The interfaces in this package closely follow the names of the constants header files used in Websphere MQ.

For example, the interface CMQC contains a constant MQOO_INPUT_SHARED; this corresponds to the header file cmqc.h and the constant MQOO_INPUT_SHARED.

com.ibm.mq.constants can be used with both WebSphere MQ classes for Java and WebSphere MQ classes for JMS.

MQC is still present, and has the constants it previously had; however, for any new applications, you should use the com.ibm.mq.constants package.

Writing WebSphere MQ classes for Java applications

This collection of topics provides information to assist with writing Java applications to interact with WebSphere MQ systems.

To use WebSphere MQ classes for Java to access WebSphere MQ queues, you write Java applications that contain calls that put messages onto, and get messages from, WebSphere MQ queues. For details of individual classes, see “WebSphere MQ classes for Java packages” on page 278.

Connection differences

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

Client connections

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

If you are programming for WebSphere MQ classes for Java for use as a client, be aware of the following differences:

- It supports only TCP/IP.
- It does not read any WebSphere MQ environment variables at startup.
- Information that would be stored in a channel definition and in environment variables can be stored in a class called Environment. Alternatively, this information can be passed as parameters when the connection is made.
- Error and exception conditions are written to a log specified in the MQException class. The default error destination is the Java console.
- Only the following attributes in a WebSphere MQ client configuration file are relevant to WebSphere MQ classes for Java. If you specify other attributes, it has no effect.

Stanza	**Attribute**
ClientExitPath	ExitsDefaultPath
ClientExitPath	ExitsDefaultPath64
ClientExitPath	JavaExitsClasspath
MessageBuffer	MaximumSize
MessageBuffer	PurgeTime
MessageBuffer	UpdatePercentage
TCP	ClntRcvBufSize
TCP	ClntSndBufSize
TCP	Connect_Timeout
TCP	KeepAlive

- If connecting to a queue manager that requires character data to be converted, then the V7 Java client is now capable of doing the conversion if queue manager is unable to do so. The client JVM must support the conversion between the CCSID of the client and that of the queue manager.

When used in client mode, WebSphere MQ classes for Java does not support the MQBEGIN call.

Bindings mode

The bindings mode of WebSphere MQ classes for Java differs from the client mode in three main ways.

If you are programming for WebSphere MQ classes for Java for use in bindings mode, be aware of the following differences from client mode:

- The bindings support the MQBEGIN call
- Fast-path bindings into the WebSphere MQ queue manager are supported
- Most of the parameters provided by the MQEnvironment class not relevant to bindings mode and are ignored

Note: WebSphere MQ for i5/OS and WebSphere MQ for z/OS do not support the use of MQBEGIN to initiate global units of work that are coordinated by the queue manager.

Defining which connection to use

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

Two variables are used:

MQEnvironment.properties
The connection type is determined by the value associated with the key name `MQC.TRANSPORT_PROPERTY`. Possible values are as follows:

MQC.TRANSPORT_MQSERIES_BINDINGS
Connect in bindings mode

MQC.TRANSPORT_MQSERIES_CLIENT
Connect in client mode

MQC.TRANSPORT_MQSERIES
Connection mode is determined by the value of the *hostname* property

MQEnvironment.hostname
Set the value of this variable as follows:

- For client connections, set this to the host name of the WebSphere MQ server to which you want to connect
- For bindings mode, leave this unset, or set it to null

Operations on queue managers

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

Setting up the WebSphere MQ environment

For an application to connect to a queue manager in client mode, the application must specify the channel name, host name, and port number.

The information in this section is not relevant if your application connects to a queue manager in bindings mode.

You can specify the channel name, host name, and port number in one of two ways; either as fields in the MQEnvironment class or as properties of the MQQueueManager object.

If you set fields in MQEnvironment, they apply to your whole application, except where they are overridden by a properties Hashtable. To specify the channel name and host name in MQEnvironment, use the following code:

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

This is equivalent to an MQSERVER environment variable setting of:

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

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

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

If you pass properties to a queue manager object at its creation, they apply only to that queue manager. Create entries in a Hashtable with keys of *hostname*, *channel*,

| and, optionally, *port*, and with appropriate values. To use the default port, 1414,
| you can omit the *port* entry. Create the MQQueueManager using a constructor that
| accepts the properties Hashtable.

Connecting to a queue manager

Connect to a queue manager by creating a new instance of the MQQueueManager class. Disconnect from a queue manager by calling the disconnect() method.

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

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

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

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

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

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

Using a client channel definition table with WebSphere MQ classes for Java

A WebSphere MQ classes for Java client application can use client connection channel definitions stored in a client channel definition table.

As an alternative to creating a client connection channel definition by setting
| certain fields and environment properties in the MQEnvironment class or passing
| them to an MQQueueManager in a properties Hashtable, a WebSphere MQ classes for Java client application can use client connection channel definitions that are stored in a client channel definition table. These definitions are created by WebSphere MQ Script (MQSC) commands or WebSphere MQ Programmable
| Command Format (PCF) commands, or using the WebSphere MQ Explorer. When the application creates an MQQueueManager object, the WebSphere MQ classes for Java client searches the client channel definition table for a suitable client connection channel definition, and uses the channel definition to start an MQI channel. For more information about client channel definition tables and how to construct one, see WebSphere MQ Clients.

To use a client channel definition table, an application must first create a URL object. The URL object encapsulates a uniform resource locator (URL) that identifies the name and location of the file containing the client channel definition table and specifies how the file can be accessed.

For example, if the file ccdt1.tab contains a client channel definition table and is stored on the same system on which the application is running, the application can create a URL object in the following way:

```
java.net.URL chanTab1 = new URL("file:///home/admdata/ccdt1.tab");
```

As another example, suppose the file ccdt2.tab contains a client channel definition table and is stored on a system that is different to the one on which the application is running. If the file can be accessed using the FTP protocol, the application can create a URL object in the following way:

```
java.net.URL chanTab2 = new URL("ftp://ftp.server/admdata/ccdt2.tab");
```

After the application has created a URL object, the application can create an MQQueueManager object using one of the constructors that takes a URL object as a parameter. Here is an example:

```
MQQueueManager mars = new MQQueueManager("MARS", chanTab2);
```

This statement causes the WebSphere MQ classes for Java client to access the client channel definition table identified by the URL object chanTab2, search the table for a suitable client connection channel definition, and then use the channel definition to start an MQI channel to the queue manager called MARS.

Note the following points that apply if an application uses a client channel definition table:

- When the application creates an MQQueueManager object using a constructor that takes a URL object as a parameter, no channel name must be set in the MQEnvironment class, either as a field or as an environment property. If a channel name is set, the WebSphere MQ classes for Java client throws an MQException. The field or environment property specifying the channel name is considered to be set if its value is anything other than null, an empty string, or a string containing all blank characters.
- The queueManagerName parameter on the MQQueueManager constructor can have one of the following values:
  - The name of a queue manager
  - An asterisk (*) followed by the name of a queue manager group
  - An asterisk (*)
  - Null, an empty string, or a string containing all blank characters

  These are the same values that can be used for the *QMgrName* parameter on an MQCONN call issued by a client application that is using Message Queue Interface (MQI). For more information about the meaning of these values therefore, see the WebSphere MQ Application Programming Reference and WebSphere MQ Clients. The way that the WebSphere MQ classes for Java client uses the queueManagerName parameter to search the client channel definition table is also as described in these books. If your application uses connection pooling, see also “Controlling the default connection pool” on page 255.
- When the WebSphere MQ classes for Java client finds a suitable client connection channel definition in the client channel definition table, it uses only the information extracted from this channel definition to start an MQI channel. Any channel related fields or environment properties that the application might have set in the MQEnvironment class are ignored.

  In particular, note the following points if you are using Secure Sockets Layer (SSL):
  - An MQI channel uses SSL only if the channel definition extracted from the client channel definition table specifies the name of a CipherSpec supported by the WebSphere MQ classes for Java client.
  - A client channel definition table also contains information about the location of Lightweight Directory Access Protocol (LDAP) servers that hold certificate revocation lists (CRLs). The WebSphere MQ classes for Java client uses only this information to access LDAP servers that hold CRLs.

  For more information about using SSL with a client channel definition table, see WebSphere MQ Clients.

  Note also the following points if you are using channel exits:

- An MQI channel uses the channel exits and associated user data specified by the channel definition extracted from the client channel definition table in preference to channel exits and data specified using other methods.
- A channel definition extracted from a client channel definition table can specify channel exits that are written in Java, C, or C++. For more information about how to write a channel exit in Java, see "Using channel exits in WebSphere MQ classes for Java" on page 247. For more information about how to write a channel exit in other languages, see "Using channel exits not written in Java with WebSphere MQ classes for Java" on page 251.

Specifying a range of ports for client connections

You can specify a port, or a range of ports, that an application can bind to in either of two ways.

When a WebSphere MQ classes for Java application attempts to connect to a WebSphere MQ queue manager in client mode, a firewall might allow only those connections that originate from specified ports or range of ports. In this situation, you can specify a port, or a range of ports, that the application can bind to. You can do this in the following ways:

- You can set the localAddressSetting field in the MQEnvironment class. Here is an example:
```
MQEnvironment.localAddressSetting = "9.20.0.1(2000,3000)";
```
- You can set the environment property MQC.LOCAL_ADDRESS_PROPERTY. Here is an example:
```
(MQEnvironment.properties).put(MQC.LOCAL_ADDRESS_PROPERTY,
                               "9.20.0.1(2000,3000)");
```
- When you can construct the MQQueueManager object, you can pass a properties Hashtable containing a LOCAL_ADDRESS_PROPERTY with the value ″9.20.0.1(2000,3000)″

In each of these examples, when the application connects to a queue manager subsequently, the application binds to a local IP address and port number in the range 9.20.0.1(2000) to 9.20.0.1(3000).

In a system with more than one network interface, you can also use the localAddressSetting field, or the environment property MQC.LOCAL_ADDRESS_PROPERTY, to specify which network interface must be used for a connection.

Connection errors might occur if you restrict the range of ports. If an error occurs, an MQException is thrown containing the WebSphere MQ reason code MQRC_Q_MGR_NOT_AVAILABLE and the following message:
```
Socket connection attempt refused due to LOCAL_ADDRESS_PROPERTY restrictions
```

An error might occur if all the ports in the specified range are in use, or if the specified IP address, host name, or port number is not valid (a negative port number, for example).

Accessing queues, topics, and processes

To access queues, topics, and processes, use methods of the MQQueueManager class. The MQOD (object descriptor structure) is collapsed into the parameters of these methods.

Queues

To open a queue you can use the accessQueue method of the MQQueueManager class. For example, on a queue manager called queueManager, use the following code:

```
MQQueue queue = queueManager.accessQueue("qName");
```

The accessQueue method returns a new object of class MQQueue.

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

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

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

```
MQQueue queue = new MQQueue(queueManager,
                            "qName");
```

You can specify a number of options when you create queues. For details of these, see the description of the MQQueue class in this manual. Constructing a queue object in this way enables you to write your own subclasses of MQQueue.

Topics

Similarly, you can open a topic using the accessTopic method of the MQQueueManager class and close it using its close() method, and you can create a topic by using the MQTopic constructor.

You can specify a number of options when you create topics. For details of these, see the description of the MQTopic class in this manual. Constructing a topic object in this way enables you to write your own subclasses of MQTopic.

A topic must be opened either for publication or for subscription. The MQQueueManager class has eight accessTopic methods and the Topic class has eight constructors. In each case, four of these have a **destination** parameter and four have a **subscriptionName** parameter (including two that have both). These can only be used to open the topic for subscriptions. The two remaining methods have an **openAs** parameter, and the topic can be opened for either publication or subscription depending on the value of the **openAs** parameter.

To create a topic as a durable subscriber use either an accessTopic method of the MQQueueManager class or an MQTopic constructor that accepts a subscription name and, in either case, set the CMQC.MQSO_DURABLE option.

Processes

To access a process, use the accessProcess method of the MQQueueManager.

The accessProcess method returns a new object of class MQProcess.

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

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

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

Handling messages

Messages are represented by the MQMessage class. You put and get messages using methods of the MQDestination class, which has subclasses of MQQueue and MQTopic.

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

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

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

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

String name = "Charlie Jordan";
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 MQDestination returns a new instance of MQMessage, which represents the message just taken from the queue. It also takes an instance of the MQGetMessageOptions class as a parameter. This class represents the MQGMO structure.

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

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

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

// Extract the message data
```

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

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

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

See the description of class com.ibm.mq.MQMessage later in this book for more details.

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

Improving the performance of nonpersistent messages

To improve performance when browsing messages or consuming nonpersistent messages from a client application, you can use *read ahead*. Client applications using MQGET or asynchronous consume will benefit from the performance improvements when browsing messages or consuming nonpersistent messages.

For general information about the read ahead facility, see the topic on improving performance of non-persistent messages in *WebSphere MQ Application Programming Guide*.

In WebSphere MQ classes for Java, you use the CMQC.MQSO_READ_AHEAD and CMQC.MQSO_NO_READ_AHEAD properties of an MQQueue or MQTopic object to determine whether message consumers and queue browsers are allowed to use read ahead on that object.

Putting messages asynchronously using WebSphere MQ classes for Java

To put a message asynchronously, set MQPMO_ASYNC_RESPONSE.

You put messages onto queues or topics using the put() method of the MQDestination class. To put a message asynchronously, that is, allowing the operation to complete without waiting for a response from the queue manager, you can set MQPMO_ASYNC_RESPONSE in the options field of MQPutMessageOptions. To determine the success or failure of asynchronous puts, use the MQQueueManager.getAsyncStatus call.

Publish/subscribe in WebSphere MQ classes for Java

In WebSphere MQ classes for Java, the topic is represented by the MQTopic class, and you publish to it using the MQTopic.put() methods.

For general information about WebSphere MQ publish/subscribe, see the *Publish/Subscribe User's Guide*.

Handling WebSphere MQ message headers

Java classes are provided representing different types of message header. Two helper classes are also provided.

Header objects are described by the MQHeader interface, which provides general-purpose methods for accessing header fields and for reading and writing message content. Each header type has its own class that implements the MQHeader interface and adds getter and setter methods for individual fields. For example, the MQRFH2 header type is represented by the MQRFH2 class; the MQDLH header type by the MQDLH class, and so on. The header classes perform any necessary data conversion automatically, and can read or write data in any specified numeric encoding or character set (CCSID).

Two helper classes, MQHeaderIterator and MQHeaderList, assist with reading and decoding (parsing) the header content in messages:

- The MQHeaderIterator class works like a java.util.Iterator. For as long as there are more headers in the message, the next() method returns true, and the nextHeader() or next() method returns the next header object.
- The MQHeaderList works like a java.util.List. Like the MQHeaderIterator, it parses header content, but it also allows you to to search for particular headers, add new headers, remove existing headers, update header fields and then write the header content back to a message. Alternatively, you can create an empty MQHeaderList, then populate it with header instances and write it to a message once or repeatedly.

The MQHeaderIterator and MQHeaderList classes use the information in the MQHeaderRegistry to know which WebSphere MQ header classes are associated with particular message types and formats. The MQHeaderRegistry is configured with knowledge of all current WebSphere MQ formats and header types and their implementation classes, and you can also register your own header types.

Support is provided for the following commonly used Websphere MQ headers

- MQRFH – Rules and formatting header
- MQRFH2 – Like MQRFH, used to pass messages to and from a message broker belonging to WebSphere Message Broker. Also used to contain message properties
- MQCIH – CICS Bridge
- MQDLH – Dead letter header
- MQIIH – IMS™ information header
- MQRMH – reference message header
- MQSAPH – SAP header
- MQWIH – Work information header
- MQXQH - Transmission Queue header
- MQDH – Distribution header
- MQEPH – Encapsulated PCF header

You can also define classes representing your own headers.

Printing all the headers in a message

In this example, an instance of MQHeaderIterator parses the headers in an MQMessage that has been received from a queue. The MQHeader objects returned from the nextHeader() method display their structure and contents when their toString method is invoked.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQHeader;
import com.ibm.mq.headers.MQHeaderIterator;
...
MQMessage message = ... // Message received from a queue.
```

```
MQHeaderIterator it = new MQHeaderIterator (message);

while (it.hasNext ())
{
 MQHeader header = it.nextHeader ();

 System.out.println ("Header type " + header.type () + ": " + header);
}
```

Skipping over the headers in a message

In this example, the skipHeaders() method of MQHeaderIterator positions the message read cursor immediately after the last header.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQHeaderIterator;
...
MQMessage message = ... // Message received from a queue.
MQHeaderIterator it = new MQHeaderIterator (message);

it.skipHeaders ();
```

Finding the reason code in a dead-letter message

In this example, the read method populates the MQDLH object by reading from the message. After the read operation, the message read cursor is positioned immediately after the MQDLH header content.

Messages on the queue manager's dead-letter queue are prefixed with a dead-letter header (MQDLH). To decide how to handle these messages - for example, to determine whether to retry or discard them - a dead-letter handling application must look at the reason code contained in the MQDLH.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQDLH;
...
MQMessage message = ... // Message received from the dead-letter queue.
MQDLH dlh = new MQDLH ();

dlh.read (message);

System.out.println ("Reason: " + dlh.getReason ());
```

All header classes also provide a convenience constructor to initialize themselves directly from the message in a single step. So the code in this example could be simplified as follows:

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQDLH;
...
MQMessage message = ... // Message received from the dead-letter queue.
MQDLH dlh = new MQDLH (message);

System.out.println ("Reason: " + dlh.getReason ());
```

Reading and removing the MQDLH from a dead-letter message

In this example, MQDLH is used to remove the header from a dead-letter message.

A dead-letter handling application will typically resubmit messages that have been rejected if their reason code indicates a transient error. Before resubmitting the message, it must remove the MQDLH header.

This example performs the following steps (see the comments in the example code):

1. The MQHeaderList reads the entire message, and each header encountered in the message becomes an item in the list.
2. Dead-letter messages contain an MQDLH as their first header, so this can be found in the first item of the header list. The MQDLH has already been populated from the message when the MQHeaderList is built, so there is no need to invoke its read method.
3. The reason code is extracted using the getReason() method provided by the MQDLH class.
4. The reason code has been inspected, and indicates that it is appropriate to resubmit the message. The MQDLH is removed using the MQHeaderList remove() method.
5. The MQHeaderList writes its remaining content to a new message object. The new message now contains everything in the original message except the MQDLH and can be written to a queue. The **true** argument to the constructor and to the write method indicates that the message body is to be held within the MQHeaderList, and written out again.
6. The format field in the message descriptor of the new message now contains the value that was previously in the MQDLH format field. The message data matches the numeric encoding and CCSID set in the message descriptor.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQDLH;
import com.ibm.mq.headers.MQHeaderList;
...
MQMessage message = ... // Message received from the dead-letter queue.
MQHeaderList list = new MQHeaderList (message, true); // Step 1.
MQDLH dlh = (MQDLH) list.get (0); // Step 2.
int reason = dlh.getReason (); // Step 3.
...
list.remove (dlh); // Step 4.

MQMessage newMessage = new MQMessage ();

list.write (newMessage, true); // Step 5.
newMessage.format = list.getFormat (); // Step 6.
```

Printing the content of a message

This example uses MQHeaderList to print out the content of a message, including its headers.

The output contains a view of all the header contents as well as the body of the message. The MQHeaderList class decodes all the headers in one go, whereas the MQHeaderIterator steps through them one at a time under application control. You might use this technique to provide a simple debugging tool when writing Websphere MQ applications.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQHeaderList;
...
MQMessage message = ... // Message received from a queue.

System.out.println (new MQHeaderList (message, true));
```

This example also prints out the message descriptor fields, using the MQMD class. The copyFrom() method of the com.ibm.mq.headers.MQMD class populates the header object from the message descriptor fields of the MQMessage rather than by reading the message body.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQMD;
import com.ibm.mq.headers.MQHeaderList;
```

```
...
MQMessage message = ...
MQMD md = new MQMD ();
...
md.copyFrom (message);
System.out.println (md + "\n" + new MQHeaderList (message, true));
```

Finding a specific type of header in a message

This example uses the indexOf(String) method of MQHeaderList to find an MQRFH2 header in a message, if one is present.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQHeaderList;
import com.ibm.mq.headers.MQRFH2;
...
MQMessage message = ...
MQHeaderList list = new MQHeaderList (message);
int index = list.indexOf ("MQRFH2");

if (index >= 0)
{
 MQRFH2 rfh = (MQRFH2) list.get (index);
 ...
}
```

Analyzing an MQRFH2 header

This example shows how to access a known field value in a named folder, using the MQRFH2 class.

The MQRFH2 class provides a number of ways to access not only the fields in the fixed part of the structure, but also the XML-encoded folder contents that are carried within the NameValueData field. This example shows how to access a known field value in a named folder - in this instance, the Rto field in the jms folder, which represents the reply queue name in an MQ JMS message.

```
MQRFH2 rfh = ...

String value = rfh.getStringFieldValue ("jms", "Rto");
```

To discover the contents of an MQRFH2 (as opposed to requesting specific fields directly), you can use the getFolders method to return a list of MQRFH2.Element, which represents the structure of a folder that could contain fields and other folders. Setting a field or folder to null removes it from the MQRFH2. When you manipulate the NameValueData folder contents in this way, the StrucLength field is automatically updated accordingly.

Reading and writing byte streams other than MQMessage objects

These examples use the header classes to parse and manipulate WebSphere MQ header content when the data source is not an MQMessage object.

You can use the header classes to parse and manipulate WebSphere MQ header content even when the data source is something other than an MQMessage object. The MQHeader interface implemented by every header class provides the methods `int read (java.io.DataInput message, int encoding, int characterSet)` and `int write (java.io.DataOutput message, int encoding, int characterSet)`. The com.ibm.mq.MQMessage class implements the java.io.DataInput and java.io.DataOutput interfaces. This means that you can use the two MQHeader methods above to read and write MQMessage content, overriding the encoding and CCSID specified in the message descriptor. This is useful for messages that contain a chain of headers in different encodings.

You can also obtain DataInput and DataOutput objects from other data streams, for example file or socket streams, or byte arrays carried in JMS messages. The java.io.DataInputStream and java.io.DataOutputStream classes implement DataInput and DataOutput respectively. This example reads WebSphere MQ header content from a byte array:

```
import java.io.*;
import com.ibm.mq.headers.*;
...
byte [] bytes = ...
DataInput in = new DataInputStream (new ByteArrayInputStream (bytes));
MQHeaderIterator it = new MQHeaderIterator (in, CMQC.MQENC_NATIVE,
  CMQC.MQCCSI_DEFAULT);
```

The line starting MQHeaderIterator could be replaced with

```
MQDLH dlh = new MQDLH (in, CMQC.MQENC_NATIVE, CMQC.MQCCSI_DEFAULT);
// or any other header type
```

This example writes to a byte array using a DataOutputStream:

```
MQHeader header = ... // Could be any header type
ByteArrayOutputStream out = new ByteArrayOutputStream ();

header.write (new DataOutputStream (out), CMQC.MQENC_NATIVE, CMQC.MQCCSI_DEFAULT);
byte [] bytes = out.toByteArray ();
```

When you work with streams in this way, be careful to use the correct values for the encoding and characterSet arguments. When reading headers, specify the encoding and CCSID with which the byte content was originally written. When writing headers, specify the encoding and CCSID that you want to produce. The data conversion is performed automatically by the header classes.

Creating classes for new header types

You can create Java classes for header types not supplied with WebSphere MQ classes for Java.

To add a Java class representing a new header type that you can use in the same way as any header class supplied with WebSphere MQ classes for Java, you create a class that implements the MQHeader interface. The simplest approach is to extend the com.ibm.mq.headers.impl.Header class. This example produces a fully-functional class representing the MQTM header structure. You do not have to add individual getter and setter methods for each field, but it is a useful convenience for users of the header class. The generic getValue and setValue methods that take a string for the field name will work for all fields defined in the header type. The inherited read, write and size methods will enable instances of the new header type to be read and written and will calculate the header size correctly based upon its field definition. The type definition is created just once, however many instances of this header class are created. To make the new header definition available for decoding using the MQHeaderIterator or MQHeaderList classes, you would register it using the MQHeaderRegistry. Note however that the MQTM header class is in fact already provided in this package and registered in the default registry.

```
import com.ibm.mq.headers.impl.Header;
import com.ibm.mq.headers.impl.HeaderField;
import com.ibm.mq.headers.CMQC;

public class MQTM extends Header {
 final static HeaderType TYPE = new HeaderType ("MQTM");
 final static HeaderField StrucId = TYPE.addMQChar ("StrucId", CMQC.MQTM_STRUC_ID);
 final static HeaderField Version = TYPE.addMQLong ("Version", CMQC.MQTM_VERSION_1);
 final static HeaderField QName = TYPE.addMQChar ("QName", CMQC.MQ_Q_NAME_LENGTH);
 final static HeaderField ProcessName = TYPE.addMQChar ("ProcessName",
      CMQC.MQ_PROCESS_NAME_LENGTH);
 final static HeaderField TriggerData = TYPE.addMQChar ("TriggerData",
      CMQC.MQ_TRIGGER_DATA_LENGTH);
 final static HeaderField ApplType = TYPE.addMQLong ("ApplType");
 final static HeaderField ApplId = TYPE.addMQChar ("ApplId", 256);
 final static HeaderField EnvData = TYPE.addMQChar ("EnvData", 128);
 final static HeaderField UserData = TYPE.addMQChar ("UserData", 128);

 protected MQTM (HeaderType type){
  super (type);
 }
 public String getStrucId () {
  return getStringValue (StrucId);
 }
 public int getVersion () {
  return getIntValue (Version);
 }
 public String getQName () {
  return getStringValue (QName);
 }
 public void setQName (String value) {
  setStringValue (QName, value);
 }
 // ...Add convenience getters and setters for remaining fields in the same way.
}
```

Handling PCF messages

Java classes are provided to create and parse PCF-structured messages, and to facilitate sending PCF requests and collecting PCF responses.

Classes PCFMessage & MQCFGR represent arrays of PCF parameter structures. They provide convenience methods for adding and retrieving PCF parameters.

PCF parameter structures are represented by the classes MQCFH, MQCFIN, MQCFIN64, MQCFST, MQCFBS, MQCFIL, MQCFIL64 MQCFSL, and MQCFGR. These share basic operational interfaces:

- Methods to read and write message content: read (), write (), and size ()
- Methods to manipulate parameters: getValue (), setValue (), getParameter () and others
- The enumerator method .nextParameter (), which parses PCF content in an MQMessage

The PCF filter parameter is used in inquire commands to provide a filter function. It in encapsulated in the following classes:

- MQCFIF – integer filter
- MQCFSF – string filter
- MQCFBF – byte filter

Two agent classes, PCFAgent and PCFMessageAgent are provided to manage the connection to a Queue Manager, the command server queue, and an associated response queue. PCFMessageAgent extends PCFAgent and should normally be used in preference to it. The PCFMessageAgent class converts the received MQMessages and passes them back to the caller as a PCFMessage array. PCFAgent simply returns an array of MQMessages, which you have to parse before use.

Handling message properties

Function calls to process message handles have no equivalent in WebSphere MQ classes for Java. To set, return, or delete message handle properties, use methods of the MQMessage class.

For general information about message properties, see .

In WebSphere MQ classes for Java access to messages is through the MQMessage class. Message handles are therefore not provided in the Java environment and there is no equivalent to the WebSphere MQ function calls MQCRTMH, MQDLTMH, MQMHBUF, and MQBUFMH

To set message handle properties in the procedural interface, you use the call MQSETMP. In WebSphere MQ classes for Java, use the appropriate method of the MQMessage class:

- setBooleanProperty
- setByteProperty
- setBytesProperty
- setShortProperty
- setIntProperty
- setInt2Property
- setInt4Property
- setInt8Property
- setLongProperty
- setFloatProperty
- setDoubleProperty
- setStringProperty
- setObjectProperty

These are sometimes referred to collectively as the *set*property* methods.

To return the value of message handle properties in the procedural interface, you use the call MQINQMP. In WebSphere MQ classes for Java, use the appropriate method of the MQMessage class:

- getBooleanProperty
- getByteProperty
- getBytesProperty
- getShortProperty
- getIntProperty
- getInt2Property
- getInt4Property
- getInt8Property
- getLongProperty
- getFloatProperty
- getDoubleProperty
- getStringProperty
- getObjectProperty

These are sometimes referred to collectively as the *get*property* methods.

To delete the value of message handle properties in the procedural interface, you use the call MQDLTMP. In WebSphere MQ classes for Java, use the deleteProperty method of the MQMessage class.

Handling errors in WebSphere MQ classes for Java

Handle errors arising from WebSphere MQ classes for Java using Java `try` and `catch` blocks.

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

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

The WebSphere MQ call reason codes reported back in Java exceptions are documented in *WebSphere MQ for z/OS Messages and Codes* for z/OS and *WebSphere MQ Messages* for all other platforms.

Exceptions that are thrown while a WebSphere MQ classes for Java application is running are also written to the log. However, an application can call the MQException.logExclude() method to prevent exceptions associated with a specific reason code from being logged. You might want to do this in situations where you expect many exceptions associated with a specific reason code to be thrown, and you do not want the log to be filled with these exceptions. For example, if your application attempts to get a message from a queue each time it iterates around a loop and, for most of these attempts, you expect no suitable message to be on the queue, you might want to prevent exceptions associated with the reason code MQRC_NO_MSG_AVAILABLE from being logged. If an application has previously prevented exceptions associated with a specific reason code from being logged, it can allow these exceptions to be logged again by calling the method MQException.logInclude().

Sometimes the reason code does not convey all details associated with the error. For each exception that is thrown, an application should check the linked exception. The linked exception itself might have another linked exception, and so the linked exceptions form a chain leading back to the original underlying problem. A linked exception is implemented by using the chained exception mechanism of the java.lang.Throwable class, and an application obtains a linked exception by calling the Throwable.getCause() method. From an exception that is an instance of MQException, MQException.getCause() retrieves the underlying instance of com.ibm.mq.jmqi.JmqiException, and getCause from this exception retrieves the underlying java.lang.Exception that caused the error.

Getting and setting attribute values in WebSphere MQ classes for Java

getXXX() and setXXX() methods are provided for many common attributes. Others can be accessed using the generic inquire() and set() methods.

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

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

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

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

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

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

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

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

queue.inquire(selectors,intAttrs,charAttrs);

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

Multithreaded programs in Java

The Java runtime environment is inherently multithreaded. WebSphere MQ classes for Java allows a queue manager object to be shared across multiple threads but ensures that all access to the target queue manager is synchronized.

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

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

With the C based WebSphere MQ client, this would cause a problem, because there are limitations to the sharing of handles across multiple threads. WebSphere MQ classes for Java relaxes this constraint, allowing a queue manager object (and its associated queue, topic and process objects) to be shared across multiple threads.

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

Using channel exits in WebSphere MQ classes for Java

An overview of how to use channel exits in an application using the WebSphere MQ classes for Java.

The following topics describe how to write a channel exit in Java, how to assign it, and how to pass data to it. They then describe how to use channel exits written in C and how to use a sequence of channel exits.

Your application must have the correct security permission to load the channel exit class.

Creating a channel exit in WebSphere MQ classes for Java

You can provide your own channel exits by defining a Java class that implements an appropriate interface.

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

- WMQSendExit
- WMQReceiveExit
- WMQSecurityExit

Note: Channel exits are supported for client connections only; they are not supported for bindings connections.

Any SSL encryption defined for a connection is performed *after* send and security exits have been invoked. Similarly, decryption is performed *before* receive and security exits are invoked.

The following sample defines a class that implements all three interfaces:

```
public class MyMQExits implements WMQSendExit, WMQReceiveExit, WMQSecurityExit {
    // Default constructor
  public MyMQExits(){
  }
    // This method comes from the send exit interface
  public ByteBuffer channelSendExit(MQCXP channelExitParms,
                                    MQCD channelDefinition,
                                    ByteBuffer agentBuffer)
  {
    // Fill in the body of the send exit here
  }
    // This method comes from the receive exit interface
  public ByteBuffer channelReceiveExit(MQCXP channelExitParms,
                                       MQCD channelDefinition,
                                       ByteBuffer agentBuffer)
  {
    // Fill in the body of the receive exit here
  }
    // This method comes from the security exit interface
  public ByteBuffer channelSecurityExit(MQCXP channelExitParms,
                                        MQCD channelDefinition,
                                        ByteBuffer agentBuffer)
  {
    // Fill in the body of the security exit here
  }
}
```

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

Any exit class you write must have a constructor. This can be either the default constructor or one that takes a string argument. If it takes a string then the user data will be passed into the exit class when it is created. If the exit class contains both a default constructor and a single argument constructor, the single argument constructor has priority.

For the send and security exits, your exit code must return the data that you want to send to the server. For a receive exit, your exit code must return the modified data that you want WebSphere MQ to interpret.

The simplest possible exit body is:

```
{ return agentBuffer; }
```

Do not close the queue manager from within a channel exit.

Using existing channel exit classes

In versions of WebSphere MQ earlier than 7.0, you would implement these exits using the interfaces MQSendExit, MQReceiveExit, and MQSecurityExit, as in the following example. This method remains valid, but the new method is preferred for improved functionality and performance.

```
public class MyMQExits implements MQSendExit, MQReceiveExit, MQSecurityExit {
    // Default constructor
  public MyMQExits(){
  }
    // 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
  }
}
```

Assigning a channel exit in WebSphere MQ classes for Java

You can assign a channel exit using WebSphere MQ classes for Java.

There is no direct equivalent to the WebSphere MQ channel in WebSphere MQ classes for Java. Channel exits are assigned to an MQQueueManager. For example, having defined a class that implements the WMQSecurityExit interface, an application can use the security exit in one of four ways:

- By assigning an instance of the class to the MQEnvironment.channelSecurityExit field before creating an MQQueueManager object
- By setting the MQEnvironment.channelSecurityExit field to a string representing the security exit class before creating an MQQueueManager object
- By creating a key/value pair in the properties Hashtable passed to MQQueueManager with a key of MQC.SECURITY_EXIT_PROPERTY
- Using a client channel definition table (CCDT)

Any exit assigned by setting the MQEnvironment.channelSecurityExit field to a string, creating a key/value pair in the properties Hashtable, or using CCDT, must be written with a default constructor. An exit assigned as an instance of a class does not need a default constructor, depending on the application.

An application can use a send or a receive exit in a similar way. For example, the following code fragment shows you how to use the security, send, and receive exits that are implemented in the class MyMQExits, which was defined previously, using MQEnvironment:

```
MyMQExits myexits = new MyMQExits();
MQEnvironment.channelSecurityExit = myexits;
MQEnvironment.channelSendExit = myexits;
MQEnvironment.channelReceiveExit = myexits;
 :
MQQueueManager jupiter = new MQQueueManager("JUPITER");
```

If more than one method is used to assign a channel exit, the order of precedence is as follows:

- If the URL of a CCDT is passed to the MQQueueManager, the contents of the CCDT determine the channel exits to be used and any exit definitions in MQEnvironment or the properties Hashtable are ignored.
- If no CCDT URL is passed, exit definitions from MQEnvironment and the Hashtable are merged
  - If the same exit type is defined in both MQEnvironment and the Hashtable, the definition in the Hashtable is used.

- If equivalent old and new types of exit are specified (for example the sendExit field, which can only be used for the type of exit used in versions of Websphere MQ earlier than Version 7.0, and the channelSendExit field, which can be used for any send exit), the new exit (channelSendExit) is used rather than the old exit.

If you have declared a channel exit as a String, you must enable Websphere MQ to locate the channel exit program. You can do this in various ways, depending on the environment in which the application is running and on how the channel exit programs are packaged.

- For an application that is running in an application server, you must store the files in the directory shown in Table 50 or packaged in JAR files referenced by exitClasspath.
- For an application that is not running in an application server, the following rules apply:
  - If your channel exit classes are packaged in separate JAR files, these JAR files must be included in the exitClasspath.
  - If your channel exit classes are not packaged in JAR files, the class files can be stored in the directory shown in Table 50 or in any directory in the JVM system Classpath or exitClasspath.

The exitClasspath property can be specified in four ways. In order of priority, these are as follows:

1. The system property com.ibm.mq.exitClasspath (usually defined on the command line using the -D option)
2. The exitPath stanza of the mqclient.ini file
3. A Hashtable entry with the key MQC.EXIT_CLASSPATH_PROPERTY
4. The MQEnvironment variable exitClasspath

Separate multiple paths using the java.io.File.pathSeparator character.

Table 50. The directory for channel exit programs

Platform	Directory
AIX, HP-UX, Linux, and Solaris	/var/mqm/exits (32-bit channel exit programs) /var/mqm/exits64 (64-bit channel exit programs)
Windows	*install_data_dir*\exits
Note: *install_data_dir* is the directory that you chose for the WebSphere MQ data files during installation. The default directory is C:\Program Files\IBM\WebSphere MQ.	

Passing data to channel exits in WebSphere MQ classes for Java

You can pass data to channel exits and return data from channel exits to your application.

The agentBuffer parameter

For a send exit, the *agentBuffer* parameter contains the data that is about to be sent. For a receive exit or a security exit, the *agentBuffer* parameter contains the data that has just been received. You do not need a length parameter, because the expression agentBuffer.limit() indicates the length of the array.

For the send and security exits, your exit code must return the data that you want to send to the server. For a receive exit, your exit code must return the modified data that you want WebSphere MQ to interpret.

The simplest possible exit body is:

```
{ return agentBuffer; }
```

Channel exits are called with a buffer that has a backing array. For best performance, the exit should return a buffer with a backing array.

User data

If an application connects to a queue manager by setting channelSecurityExit, channelSendExit, or channelReceiveExit, 32 bytes of user data can be passed to the appropriate channel exit class when it is called, using the channelSecurityExitUserData, channelSendExitUserData, or channelReceiveExitUserData fields. This user data is available to the channel exit class but is refreshed each time the exit is called. Any changes made to the user data in the channel exit will therefore be lost. If you want to make persistent changes to data in a channel exit, use the MQCXP exitUserArea. Data in this field is maintained between invocations of the exit.

If the application sets securityExit, sendExit, or receiveExit, no user data can be passed to these channel exit classes.

If an application uses a client channel definition table to connect to a queue manager, any user data specified in a client connection channel definition is passed to channel exit classes when they are called. For more information about using a client channel definition table, see “Using a client channel definition table with WebSphere MQ classes for Java” on page 232.

Using channel exits not written in Java with WebSphere MQ classes for Java

How to use channel exit programs written in C from a Java application.

In WebSphere MQ Version 7.0, you can specify the name of a channel exit program written in C as a String passed to the channelSecurityExit, channelSendExit, or channelReceiveExit fields in the MQEnvironment object or properties Hashtable.

Specify the exit program name in the format `library(function)` and ensure that the location of the exit program is included in the path environment variable.

For information about how to write a channel exit in C, see *WebSphere MQ Intercommunication*.

Using external exit classes

In versions of WebSphere MQ earlier than Version 7.0, three classes were provided to enable you to use channel exits written in languages other than Java:

- MQExternalSecurityExit, which implements the MQSecurityExit interface
- MQExternalSendExit, which implements the MQSendExit interface
- MQExternalReceiveExit, which implements the MQReceiveExit interface

The use of these classes remains valid but the new method is preferred.

To use a security exit that is not written in Java, an application first had to create an MQExternalSecurityExit object. The application specified, as parameters on the MQExternalSecurityExit constructor, the name of the library containing the security exit, the name of the entry point for the security exit, and the user data to be passed to the security exit when it was called. Channel exit programs that are not written in Java were stored in the directory shown in Table 50 on page 250.

Using a sequence of channel send or receive exits in WebSphere MQ classes for Java

A WebSphere MQ classes for Java application can use a sequence of channel send or receive exits that are run in succession.

To use a sequence of send exits, an application can create either a List or a String containing the send exits. If a List is used, each element of the List can be any of the following:

- An instance of a user defined class that implements the WMQSendExit interface
- An instance of a user defined class that implements the MQSendExit interface (for a send exit written in Java)
- An instance of the MQExternalSendExit class (for a send exit not written in Java)
- An instance of the MQSendExitChain class
- An instance of the String class

A List cannot contain another List.

The application can use a sequence of receive exits in a similar manner.

If a String is used, it must consist of one or more comma-separated exit definitions, each of which can be the name of a Java class, or a C program in the format `library(function)`.

The application then assigns the List or String object to the MQEnvironment.channelSendExit field before creating an MQQueueManager object.

The context of information passed to exits is solely within the domain of the exits. For example, if a Java exit and a C exit are chained, the presence of the Java exit has no effect on the C exit.

Using exit chain classes

In versions of WebSphere MQ earlier than Version 7.0, two classes were provided to allow sequences of exits:

- MQSendExitChain, which implements the MQSendExit interface
- MQReceiveExitChain, which implements the MQReceiveExit interface

The use of these classes remains valid but the new method is preferred. Using the WebSphere MQ Classes for Java interfaces means that your application still has a dependency on com.ibm.mq.jar If the new set of interfaces in the com.ibm.mq.exits package are used there is no dependency on com.ibm.mq.jar.

To use a sequence of send exits, an application created a list of objects, where each object was one of the following:

- An instance of a user defined class that implements the MQSendExit interface (for a send exit written in Java)
- An instance of the MQExternalSendExit class (for a send exit not written in Java)
- An instance of the MQSendExitChain class

The application created an MQSendExitChain object by passing this list of objects as a parameter on the constructor. The application would then have assigned the MQSendExitChain object to the MQEnvironment.sendExit field before creating an MQQueueManager object.

Channel compression in WebSphere MQ classes for Java

Compressing the data that flows on a channel can improve the performance of the channel and reduce network traffic. WebSphere MQ classes for Java use the compression function built into WebSphere MQ.

Using function supplied with WebSphere MQ, you can compress the data that flows on message channels and MQI channels and, on either type of channel, you can compress header data and message data independently of each other. By default, no data is compressed on a channel. For a full description of channel compression, including how it is implemented in WebSphere MQ, see *WebSphere MQ Intercommunication*.

A WebSphere MQ classes for Java application specifies the techniques that can be used for compressing header or message data on a client connection by creating a java.util.Collection object. Each compression technique is an Integer object in the collection, and the order in which the application adds the compression techniques to the collection is the order in which the compression techniques are negotiated with the queue manager when the client connection starts. The application can then assign the collection to the hdrCompList field, for header data, or the msgCompList field, for message data, in the MQEnvironment class. When the application is ready, it can start the client connection by creating an MQQueueManager object.

The following code fragments illustrate the approach just described. The first code fragment shows you how to implement header data compression:

```
Collection headerComp = new Vector();
headerComp.add(new Integer(MQC.MQCOMPRESS_SYSTEM));
:
MQEnvironment.hdrCompList = headerComp;
:
MQQueueManager qMgr = new MQQueueManager(QM);
```

The second code fragment shows you how to implement message data compression:

```
Collection msgComp = new Vector();
msgComp.add(new Integer(MQC.MQCOMPRESS_RLE));
msgComp.add(new Integer(MQC.MQCOMPRESS_ZLIBHIGH));
:
MQEnvironment.msgCompList = msgComp;
:
MQQueueManager qMgr = new MQQueueManager(QM);
```

In the second example, the compression techniques are negotiated in the order RLE, then ZLIBHIGH, when the client connection starts. The compression technique that is selected cannot be changed during the lifetime of the MQQueueManager object.

The compression techniques for header and message data that are supported by both the client and the queue manager on a client connection are passed to a channel exit as collections in the hdrCompList and msgCompList fields respectively of an MQChannelDefinition object. The actual techniques that are currently being used for compressing header and message data on a client connection are passed to a channel exit in the CurHdrCompression and CurMsgCompression fields respectively of an MQChannelExit object.

Note that, if compression is used on a client connection, the data is compressed before any channel send exits are processed and decompressed after any channel receive exits are processed. The data passed to send and receive exits is therefore in a compressed state.

For more information about specifying compression techniques, and about which compression techniques are available, see Class com.ibm.mq.MQEnvironment and Interface com.ibm.mq.MQC .

Sharing a TCP/IP connection in WebSphere MQ classes for Java

Multiple instances of an MQI channel can be made to share a single TCP/IP connection.

If a channel is defined with the SHARECNV (SharingConversations) parameter set to a value greater than 1, then that number of conversations can share a channel instance. If more than one suitable channel is defined in a client channel definition table (CCDT), the AFFINITY and CLNTWGHT channel attributes influence which channel definition is used. See the related topics below for more details of these properties.

In WebSphere MQ classes for Java, you use the MQEnvironment.sharingConversations variable to control the number of conversations that can share a single TCP/IP connection.

Connection pooling in WebSphere MQ classes for Java

WebSphere MQ classes for Java allows spare connections to be pooled for reuse.

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

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

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

To prevent connections from using too much resource, you can limit the total number of connections that an MQSimpleConnectionManager object can handle, and you can limit the size of the connection pool. Setting limits is useful if there are conflicting demands for connections within a JVM.

By default, the getMaxConnections() method returns the value zero, which means that there is no limit to the number of connections that the MQSimpleConnectionManager object can handle. You can set a limit by using the setMaxConnections() method. If you set a limit and the limit is reached, a request for a further connection might cause an MQException to be thrown, with a reason code of MQRC_MAX_CONNS_LIMIT_REACHED.

Controlling the default connection pool

This example shows how to use the default connection pool.

Consider the following example application, MQApp1:

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

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

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

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

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

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

         MQEnvironment.removeConnectionPoolToken(token);

      }
}
```

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

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

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

If an application uses a client channel definition table to connect to a queue manager, the MQQueueManager constructor first searches the table for a suitable client connection channel definition. If one is found, the constructor searches the default connection pool for a connection that can be used for the channel. If the constructor cannot find a suitable connection in the pool, it then searches the client channel definition table for the next suitable client connection channel definition, and proceeds as described previously. If the constructor completes its search of the client channel definition table and fails to find any suitable connection in the pool, the constructor starts a second search of the table. During this search, the constructor tries to create a new connection for each suitable client connection channel definition in turn, and uses the first connection that it manages to create.

The default connection pool stores a maximum of ten unused connections, and keeps unused connections active for a maximum of five minutes. The application can alter this (for details, see "Supplying a different connection pool" on page 257).

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

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

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

The default connection pool and multiple components

This example shows how to add or remove MQPoolTokens from a static set of registered MQPoolToken objects.

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

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

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

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

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

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

class MQApp3_Thread extends Thread
{
      long time;

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

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

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

Supplying a different connection pool

This example shows how to use the class **com.ibm.mq.MQSimpleConnectionManager** to supply a different connection pool.

This class provides basic facilities for connection pooling, and applications can use this class to customize the behavior of the pool.

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

The following code fragment demonstrates this behavior:

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

```
:
qmgr.disconnect();

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

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

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

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

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

The following application demonstrates this:

```
import com.ibm.mq.*;
public class MQApp4
{
      public static void main(String []args)
      {
         MQSimpleConnectionManager myConnMan=new MQSimpleConnectionManager();
         myConnMan.setActive(MQSimpleConnectionManager.MODE_AUTO);
         myConnMan.setTimeout(3600000);
         myConnMan.setMaxConnections(75);
         myConnMan.setMaxUnusedConnections(50);
         MQEnvironment.setDefaultConnectionManager(myConnMan);
         MQApp3.main(args);
      }
}
```

The bold lines create and configure an MQSimpleConnectionManager object. The configuration does the following:

- Ends connections that are not used for an hour
- Limits the number of connections managed by myConnMan to 75
- Limits the number of unused connections in the pool to 50
- Sets MODE_AUTO, which is the default. This means that the pool is active only if it is the default connection manager, and there is at least one token in the set of MQPoolTokens held by MQEnvironment.

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

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

Supplying your own ConnectionManager

WebSphere MQ classes for Java provides a partial implementation of the J2EE Connector Architecture, allowing implementations of javax.resource.spi.ConnectionManager to be used.

| Applications and middleware providers can provide alternative implementations of connection pools. WebSphere MQ classes for Java provides a partial implementation of the J2EE Connector Architecture. Implementations of **javax.resource.spi.ConnectionManager** can either be used as the default Connection Manager or be specified on the MQQueueManager constructor.

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

The ConnectionManager interface defines only one method:

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

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

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

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

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

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

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

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

This connection handle can be returned from allocateConnection().

A ConnectionManager must register an interest in the ManagedConnection through:

```
mc.addConnectionEventListener()
```

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

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

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

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

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

JTA/JDBC coordination using WebSphere MQ classes for Java

WebSphere MQ classes for Java supports the MQQueueManager.begin() method, which allows WebSphere MQ to act as a coordinator for a database which provides a JDBC type 2 or JDBC type 4 compliant driver.

Currently this support is available on AIX, HP-UX, Solaris, and Windows with DB2 or Oracle databases.

Configuring JTA/JDBC coordination

In order to use the XA-JTA support, you must use the special JTA switch library. The method for using this library varies depending on whether you are using Windows or one of the other platforms.

Configuring on Windows:

The XA library is supplied as a DLL with a name of the format `jdbcxxx.dll`.

On Windows systems, the XA library is supplied as a complete DLL. The name of this DLL is `jdbcxxx.dll` where `xxx` indicates the database for which the switch
| library has been compiled. This library is in the `java\lib\jdbc` or `java\lib64\jdbc` directory of your WebSphere MQ classes for Java installation.

Configuring JTA/JDBC coordination on platforms other than Windows:

Object files are supplied. Link the appropriate one using the supplied makefile, and declare it to the queue manager using the configuration file.

For each database management system, WebSphere MQ provides two object files. You must link one object file to create a 32-bit switch library, and link the other object file to create a 64-bit switch library. For DB2, the name of each object file is jdbcdb2.o and, for Oracle, the name of each object file is jdbcora.o.

You must link each object file using the appropriate makefile supplied with WebSphere MQ. A switch library requires other libraries, which might be stored in different locations on different systems. However, a switch library cannot use the library path environment variable to locate these libraries because the switch library is loaded by the queue manager, which runs in a setuid environment. The supplied makefile therefore ensures that a switch library contains the fully qualified path names of these libraries.

To create a switch library, enter a **make** command with the following format. To create a 32-bit switch library, enter the command in the /java/lib/jdbc directory of your WebSphere MQ installation. To create a 64-bit switch library, enter the command in the /java/lib64/jdbc directory.

```
make DBMS
```

where *DBMS* is the database management system for which you are creating the switch library. The valid values are `db2` for DB2 and `oracle` for Oracle.

Here is an example of a **make** command:

```
make db2
```

Note the following points:

- To run 32-bit applications, you must create both a 32-bit and a 64-bit switch library for each database management system that you are using. To run 64-bit applications, you need create only a 64-bit switch library. For DB2, the name of each switch library is jdbcdb2 and, for Oracle, the name of each switch library is jdbcora. The makefiles ensure that 32-bit and 64-bit switch libraries are stored in different WebSphere MQ directories. A 32-bit switch library is stored in the /java/lib/jdbc directory, and a 64-bit switch library is stored in the /java/lib64/jdbc directory.
- Because you can install Oracle anywhere on a system, the makefiles use the ORACLE_HOME environment variable to locate where Oracle is installed.

After you have created the switch libraries for DB2, Oracle, or both, you must declare them to your queue manager. If the queue manager configuration file (qm.ini) already contains XAResourceManger stanzas for DB2 or Oracle databases, you must replace the SwitchFile entry in each stanza by one of the following:

For a DB2 database

```
SwitchFile=jdbcdb2
```

For an Oracle database

```
SwitchFile=jdbcora
```

Do not specify the fully qualified path name of either the 32-bit or 64-bit switch library. Specify only the name of the library.

If the queue manager configuration file does not already contain XAResourceManager stanzas for DB2 or Oracle databases, or if you want to add additional XAResourceManager stanzas, see the WebSphere MQ System Administration Guide for information about how to construct an XAResourceManager stanza. However, each SwitchFile entry in a new XAResourceManger stanza must be exactly as described previously for a DB2 or Oracle database. You must also include the entry `ThreadOfControl=PROCESS`.

After you have updated the queue manager configuration file, and made sure that all appropriate database environment variables have been set, you can restart the queue manager.

Using JTA/JDBC coordination

Code your API calls as in the supplied example.

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

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

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

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

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

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

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

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

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

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

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

To use the JTA/JDBC support, you must include the following statement in your application:

|
```
MQEnvironment.properties.put(MQC.THREAD_AFFINITY_PROPERTY, new Boolean(true));
```

Known problems and limitations with JTA/JDBC coordination

| There are certain problems and limitations of JTA/JDBC support, some depending
| on the database management system in use.

Because this support makes calls to JDBC drivers, the implementation of those JDBC drivers can have significant impact on the system behavior. In particular, tested JDBC drivers behave differently when the database is shut down while an application is running. **Always** avoid abruptly shutting down a database while there are applications holding open connections to it.

Multiple XAResourceManager stanzas
The use of more than one XAResourceManager stanza in a queue manager configuration file, qm.ini, is not supported. Any XAResourceManager stanza other than the first is ignored.

DB2

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

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

Refer to the DB2 documentation for more information.

The XAResourceManager stanza must be configured to use ThreadOfControl=PROCESS. For DB2 version 8.1 and higher this does not match the default thread of control setting for DB2, so toc=p must be specified in the XA Open String. An example XAResourceManager stanza for DB2 with JTA/JDBC coordination is as follows:

```
XAResourceManager:
     Name=jdbcdb2
     SwitchFile=jdbcdb2
     XAOpenString=uid=userid,db=dbalias,pwd=password,toc=p
     ThreadOfControl=PROCESS
```

This does not prevent the Java applications that use JTA/JDBC coordination from being multithreaded themselves.

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

Secure Sockets Layer (SSL) support

WebSphere MQ classes for Java client applications support Secure Sockets Layer (SSL) encryption. You require a JSSE provider to use SSL encryption.

WebSphere MQ classes for Java client applications using TRANSPORT(CLIENT) support Secure Sockets Layer (SSL) encryption. SSL provides communication encryption, authentication, and message integrity. It is typically used to secure communications between any two peers on the Internet or within an intranet.

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

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

If your WebSphere MQ classes for Java client application uses a client channel definition table to connect to a queue manager, see "Using a client channel definition table with WebSphere MQ classes for Java" on page 232.

Enabling SSL in WebSphere MQ classes for Java

To enable SSL, you specify a CipherSuite. There are two ways of doing this.

SSL is supported only for client connections. To enable SSL, you must specify the CipherSuite to use when communicating with the queue manager, and this must match the CipherSpec set on the target channel. Additionally, the named CipherSuite must be supported by your JSSE provider. However, CipherSuites are distinct from CipherSpecs and so have different names. "SSL CipherSpecs and CipherSuites" on page 269 contains a table mapping the CipherSpecs supported by WebSphere MQ to their equivalent CipherSuites as known to JSSE.

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

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

Note that, although the channel has a CipherSpec of RC4_MD5_EXPORT, the Java application must specify a CipherSuite of SSL_RSA_EXPORT_WITH_RC4_40_MD5. For more information about CipherSpecs and CipherSuites, see WebSphere MQ Security. See "SSL CipherSpecs and CipherSuites" on page 269 for a list of mappings between CipherSpecs and CipherSuites.

An application can also specify a CipherSuite by setting the environment property MQC.SSL_CIPHER_SUITE_PROPERTY.

If you require a client connection to use a CipherSuite that is supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS), an application can set the sslFipsRequired field in the MQEnvironment class to `true`. Alternatively, the application can set the environment property MQC.SSL_FIPS_REQUIRED_PROPERTY. The default value is `false`, which means that a client connection can use any CipherSuite that is supported by WebSphere MQ.

If an application uses more than one client connection, the value of the sslFipsRequired field that is used when the application creates the first client connection determines the value that is used when the application creates any subsequent client connection. This means that, when the application creates a subsequent client connection, the value of the sslFipsRequired field is ignored. You must restart the application if you want to use a different value for the sslFipsRequired field.

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

Using the distinguished name of the queue manager

The queue manager identifies itself using an SSL certificate, which contains a *Distinguished Name* (DN). A WebSphere MQ classes for Java client application can use this DN to ensure that it is communicating with the correct queue manager.

A DN pattern is specified using the sslPeerName variable of MQEnvironment. For example, setting:

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

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

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

An application can also specify the distinguished name of the queue manager by setting the environment property MQC.SSL_PEER_NAME_PROPERTY. For more information about distinguished names, see WebSphere MQ Security.

Using certificate revocation lists

Specify the certificate revocation lists to use through the java.security.cert.CertStore class. WebSphere MQ classes for Java will then check certificates against the specified CRL.

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

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

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

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

Having created a CertStoreParameters instance, use the static constructor on CertStore to create a CertStore of type LDAP:

```
CertStore cs = CertStore.getInstance("LDAP", csp);
```

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

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

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

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

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

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

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

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

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

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

Renegotiating the secret key used for encryption

A WebSphere MQ classes for Java client application can control when the secret key that is used for encryption on a client connection is renegotiated, in terms of the total number of bytes sent and received.

. The application can do this in either of the following ways:

- By setting the sslResetCount field in the MQEnvironment class.
- By setting the environment property MQC.SSL_RESET_COUNT_PROPERTY in a Hashtable object. The application then assigns the hashtable to the `properties` field in the MQEnvironment class, or passes the hashtable to an MQQueueManager object on its constructor.

If the application uses more than one of these ways, the usual precedence rules apply. See Class com.ibm.mq.MQEnvironment for the precedence rules.

The value of the sslResetCount field or environment property MQC.SSL_RESET_COUNT_PROPERTY represents the total number of bytes sent and received by the WebSphere MQ classes for Java client code before the secret key is renegotiated. The number of bytes sent is the number before encryption, and the number of bytes received is the number after decryption. The number of bytes also includes control information sent and received by the WebSphere MQ classes for Java client.

If the reset count is zero, which is the default value, the secret key is never renegotiated. The reset count is ignored if no CipherSuite is specified.

In some environments, you must not set the reset count to a value other than zero. If you do set the reset count to a value other than zero, a client connection fails when it attempts to renegotiate the secret key. These environments are:

- an HP or Sun V1.4.2 JDK
- any V1.4.2 JDK when using FIPS mode

- any V5.0 or later JDK

For more information about the secret key that is used for encryption on an SSL enabled channel, see *WebSphere MQ Security*.

Supplying a customized SSLSocketFactory

If you use a customized JSSE Socket Factory, set the MQEnvironment.sslSocketFactory to the customized factory object. Details vary between different JSSE implementations.

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

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

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

When you have an SSLSocketFactory object, set the MQEnvironment.sslSocketFactory to the customized factory object. For example:

```
javax.net.ssl.SSLSocketFactory sf = sslContext.getSocketFactory();
MQEnvironment.sslSocketFactory = sf;
```

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

Bear in mind the effect of TCP/IP connection sharing when using custom SSLSocketFactories. If connection sharing is possible then a new socket will not be requested of the SSLSocketFactory supplied, even if the socket produced would be different in some way in the context of a subsequent connection request. For example, if a different client certificate is to be presented on a subsequent connection, then connection sharing must not be allowed.

Making changes to the JSSE keystore or truststore

If you change the JSSE keystore or truststore, you must perform certain actions for the changes to take effect.

If you change the contents of the JSSE keystore or truststore, or change the location of the keystore or truststore file, WebSphere MQ classes for Java applications that are running at the time do not automatically pick up the changes. For the changes to take effect, the following actions must be performed:

- The applications must close all their connections, and destroy any unused connections in connection pools.

- If your JSSE provider caches information from the keystore and truststore, this information must be refreshed.

After these actions have been performed, the applications can then recreate their connections.

Depending on how you design your applications, and on the function provided by your JSSE provider, it might be possible to perform these actions without stopping and restarting your applications. However, stopping and restarting the applications might be the simplest solution.

Error handling when using SSL

A number of reason codes can be issued by WebSphere MQ classes for Java when connecting to a queue manager using SSL.

These are explained in the following list:

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

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

MQRC_SSL_INITIALIZATION_ERROR
An MQCONN or MQCONNX call was issued with SSL configuration options specified, but an error occurred during the initialization of the SSL environment.

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

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

MQRC_UNSUPPORTED_CIPHER_SUITE
The CipherSuite named in sslCipherSuite was not recognized by the JSSE provider. A full list of CipherSuites supported by the JSSE provider can be obtained by a program using the SSLSocketFactory.getSupportedCipherSuites() method. A list of CipherSuites that can be used to communicate with WebSphere MQ can be found in "SSL CipherSpecs and CipherSuites" on page 269.

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

MQRC_SSL_CERT_STORE_ERROR
None of the supplied CertStores could be searched for the certificate presented by the queue manager. The MQException.getCause() method returns the error that occurred while searching the first CertStore attempted. If the causal exception is NoSuchElementException, ClassCastException, or NullPointerException, check that the Collection specified on the sslCertStores property contains at least one valid CertStore object.

SSL CipherSpecs and CipherSuites

Whether a WebSphere MQ classes for Java application can establish a connection to a queue manager depends on the CipherSpec specified at the server end of the MQI channel and the CipherSuite specified at the client end.

The following table lists the CipherSpecs supported by WebSphere MQ and their equivalent CipherSuites. The table also indicates whether a WebSphere MQ classes for Java application can establish a connection to a queue manager if a CipherSpec is specified at the server end of the MQI channel and the equivalent CipherSuite is specified at the client end.

For each combination of CipherSpec and CipherSuite, whether a WebSphere MQ classes for Java application can connect to a queue manager depends on the value of the sslFipsRequired field in the MQEnvironment class, or on the value of the environment property MQC.SSL_FIPS_REQUIRED_PROPERTY.

At the server end of an MQI channel, the name of a CipherSpec can be specified as the value of the SSLCIPH parameter on a DEFINE CHANNEL CHLTYPE(SVRCONN) command. At the client end of an MQI channel, a WebSphere MQ classes for Java application can set the sslCipherSuite field in the MQEnvironment class, or set the environment property MQC.SSL_CIPHER_SUITE_PROPERTY.

Table 51. CipherSpecs supported by WebSphere MQ and their equivalent CipherSuites

CipherSpec	Equivalent CipherSuite	Connection possible if FIPS is not required?[1]	Connection possible if FIPS is required?[1]
NULL_MD5	SSL_RSA_WITH_NULL_MD5	Yes	No
NULL_SHA	SSL_RSA_WITH_NULL_SHA	Yes	No
RC4_MD5_EXPORT	SSL_RSA_EXPORT_WITH_RC4_40_MD5	Yes	No
RC4_MD5_US	SSL_RSA_WITH_RC4_128_MD5	Yes	No
RC4_SHA_US	SSL_RSA_WITH_RC4_128_SHA	Yes	No
RC2_MD5_EXPORT	SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	Yes	No
DES_SHA_EXPORT	SSL_RSA_WITH_DES_CBC_SHA	Yes	No
RC4_56_SHA_EXPORT1024	SSL_RSA_EXPORT1024_WITH_RC4_56_SHA	No	No
DES_SHA_EXPORT1024	SSL_RSA_EXPORT1024_WITH_DES_CBC_SHA	No	No
TRIPLE_DES_SHA_US	SSL_RSA_WITH_3DES_EDE_CBC_SHA	Yes	No
TLS_RSA_WITH_AES_128_CBC_SHA	SSL_RSA_WITH_AES_128_CBC_SHA	No	Yes
TLS_RSA_WITH_AES_256_CBC_SHA	SSL_RSA_WITH_AES_256_CBC_SHA	No	Yes
AES_SHA_US[2]			
| TLS_RSA_WITH_DES_CBC_SHA	SSL_RSA_WITH_DES_CBC_SHA	No	No[3]
TLS_RSA_WITH_3DES_EDE_CBC_SHA	SSL_RSA_WITH_3DES_EDE_CBC_SHA	No	Yes
FIPS_WITH_DES_CBC_SHA	SSL_RSA_FIPS_WITH_DES_CBC_SHA	Yes	No[4]
FIPS_WITH_3DES_EDE_CBC_SHA	SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA	Yes	No

Notes:

1. In a WebSphere MQ classes for Java application, indicate that only FIPS-certified algorithms are to be used by setting the sslFipsRequired field in

the MQEnvironment class to `true` and indicate that non-FIPS-certified algorithms can also be used by setting the sslFipsRequired field to `false`. Alternatively, set the environment property MQC.SSL_FIPS_REQUIRED_PROPERTY.
2. This CipherSpec has no equivalent CipherSuite.
3. This CipherSpec was FIPS 140-2 certified prior to 19th May 2007.
4. This CipherSpec was FIPS 140-2 certified prior to 19th May 2007. The name FIPS_WITH_DES_CBC_SHA is historical and reflects the fact that this CipherSpec was previously FIPS-compliant.

Running WebSphere MQ classes for Java applications

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

Use the command:

```
java -Djava.library.path=library_path MyClass
```

where *library_path* is the path to the WebSphere MQ classes for Java libraries (see "The WebSphere MQ classes for Java libraries" on page 221).

Tracing WebSphere MQ classes for Java programs

WebSphere MQ classes for Java includes a trace facility, which you can use to produce diagnostic messages if you suspect that there might be a problem with the code.

(You 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(int); // start trace
|  ...                              // these commands will be traced
| MQEnvironment.disableTracing();   // turn tracing off again
```

where *int* is an integer. The value of the integer is ignored.

| You can also invoke trace using `MQEnvironment.enableTracing(int,`
| `OutputStream);` but the *OutputStream* argument is also ignored.

| All tracing configuration is controlled using Websphere MQ common services as
| described in *WebSphere MQ System Administration Guide*.

Environment-dependent behavior

WebSphere MQ classes for Java allow you to create applications that can run against different versions of WebSphere MQ. This collection of topics describes the behavior of the Java classes dependent on these different versions.

WebSphere MQ classes for Java provides a core of classes, which provide consistent function and behavior in all the environments. Features outside this core depend on the capability of the queue manager to which the application is connected.

Except where noted here, the behavior exhibited is as described in the Application Programming Reference appropriate to the queue manager.

Core classes in WebSphere MQ classes for Java

WebSphere MQ classes for Java contains a core set of classes, which can be used in all environments.

The following set of classes are considered core classes, and can be used in all environments with only the minor variations listed in “Restrictions and variations for core classes” on page 272.

- MQEnvironment
- MQException
- MQGetMessageOptions

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

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

  Excluding:
  - groupId
  - messageFlags
  - messageSequenceNumber
  - offset
  - originalLength
- MQPoolServices
- MQPoolServicesEvent
- MQPoolServicesEventListener
- MQPoolToken
- MQPutMessageOptions

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

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

Note:

1. Some constants are not included in the core (see “Restrictions and variations for core classes” for details); do not use them in completely portable programs.
2. Some platforms do not support all connection modes. On these platforms, you can use only the core classes and options that relate to the supported modes. (See “Connection options for WebSphere MQ classes for Java” on page 217.)

Restrictions and variations for core classes

The core classes generally behave consistently across all environments, even if the equivalent MQI calls normally have environment differences. The behavior is as if a Windows or UNIX WebSphere MQ queue manager is used, except for the following minor restrictions and variations.

MQGMO_* values

Certain MQGMO_* values are not supported by all queue managers.

Use of the following MQGMO_* values might result in an MQException being thrown from an MQQueue.get():

MQGMO_SYNCPOINT_IF_PERSISTENT
MQGMO_MARK_SKIP_BACKOUT
MQGMO_BROWSE_MSG_UNDER_CURSOR
MQGMO_LOCK
MQGMO_UNLOCK
MQGMO_LOGICAL_ORDER
MQGMO_COMPLETE_MESSAGE
MQGMO_ALL_MSGS_AVAILABLE
MQGMO_ALL_SEGMENTS_AVAILABLE
MQGMO_UNMARKED_BROWSE_MSG
MQGMO_MARK_BROWSE_HANDLE
MQGMO_MARK_BROWSE_CO_OP
MQGMO_UNMARK_BROWSE_HANDLE
MQGMO_UNMARK_BROWSE_CO_OP

Additionally, MQGMO_SET_SIGNAL is not supported when used from Java.

MQPMRF_* values

These are used only when putting messages to a distribution list, and are supported only by queue managers supporting distribution lists. For example, z/OS queue managers do not support distribution lists.

MQPMO_* values

Certain MQPMO_* values are not supported by all queue managers

Use of the following MQPMO_* values might result in an MQException being thrown from an MQQueue.put() or an MQQueueManager.put():

MQPMO_LOGICAL_ORDER
MQPMO_NEW_CORREL_ID
MQPMO_NEW_MESSAGE_ID
MQPMO_RESOLVE_LOCAL_Q

MQCNO_FASTPATH_BINDING

This value is ignored on queue managers that do not support it, or when using a TCP/IP client connection.

MQRO_* values

Certain report options are ignored by some queue managers.

The following report options can be set but are ignored by some queue managers. This can affect applications connected to a queue manager that honors the report options when the report message is generated by a remote queue manager that does not. Avoid relying on these options if there is a possibility that a queue manager involved does not support them.

MQRO_EXCEPTION_WITH_FULL_DATA
MQRO_EXPIRATION_WITH_FULL_DATA
MQRO_COA_WITH_FULL_DATA
MQRO_COD_WITH_FULL_DATA
MQRO_DISCARD_MSG
MQRO_PASS_DISCARD_AND_EXPIRY

Miscellaneous differences with z/OS

WebSphere MQ for z/OS behaves differently from WebSphere MQ on other platforms in some areas.

Message priority
When a message is put with a priority greater than MaxPriority, a z/OS queue manager rejects the put with MQCC_FAILED and MQRC_PRIORITY_ERROR. Other platforms complete the put with MQCC_WARNING and MQRC_PRIORITY_EXCEEDS_MAXIMUM, and treat the message as if it were put with MaxPriority.

BackoutCount
A z/OS queue manager returns a maximum BackoutCount of 255, even if the message has been backed out more than 255 times.

Default dynamic queue prefix
When connected to a z/OS queue manager using a bindings connection, the default dynamic queue prefix is CSQ.*. Otherwise, the default dynamic queue prefix is AMQ.*.

MQQueueManager constructor
Client connect is not supported on z/OS. Attempting to connect with client options results in an MQException with MQCC_FAILED and MQRC_ENVIRONMENT_ERROR. The MQQueueManager constructor might also fail with MQRC_CHAR_CONVERSION_ERROR (if it fails to initialize conversion between the IBM-1047 and ISO8859-1 code pages), or MQRC_UCS2_CONVERSION_ERROR (if it fails to initialize conversion between the queue manager's code page and Unicode). If your application fails with one of these reason codes, ensure that the National Language Resources component of Language Environment® is installed, and ensure that the correct conversion tables are available.

Conversion tables for Unicode are installed as part of the z/OS C/C++ optional feature. See the *z/OS C/C++ Programming Guide*, SC09-4765, for more information about enabling UCS-2 conversions.

| **Application termination**
| If a Java application ends before issuing an MQDISC call, WebSphere MQ
| for z/OS performs its standard task cleanup, which includes committing
| any outstanding unit of work if the thread terminated normally, or backing
| it out if the thread terminated abnormally. When an Exception or Error is
| thrown in a Java application, and is not caught by the application code, the
| Java launcher stops the JVM and returns a nonzero return code to its caller.
| Because this does not result in an abnormal termination of the thread, it

might later terminate normally, causing MQ to commit any outstanding unit of work. To ensure that WebSphere MQ work is always resolved as required, write your Java applications to handle any error situations, and explicitly resolve outstanding WebSphere MQ work before terminating the JVM.

Features outside the core

WebSphere MQ classes for Java contain certain functions that are specifically designed to use API extensions that are not supported by all queue managers. This collection of topics describes how they behave when using a queue manager that does *not* support them.

MQQueueManager constructor option

Some of the MQQueueManager constructors include an optional integer argument. Some values of this argument are not accepted on all platforms.

Where an MQQueueManager constructor include an optional integer argument, it maps onto the MQI's MQCNO options field, and is used to switch between normal and fast path 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 fast path option MQC.MQCNO_FASTPATH_BINDING is honored only with a bindings connection to a queue manager that supports it. In other environments, it is ignored.

MQQueueManager.begin() method

This method can be used only against a WebSphere MQ queue manager on UNIX or Windows systems in bindings mode. Otherwise, it fails with MQRC_ENVIRONMENT_ERROR.

See "JTA/JDBC coordination using WebSphere MQ classes for Java" on page 260 for more details.

MQGetMessageOptions fields

Some queue managers do not support the Version 2 MQGMO structure, so you must set some fields to their default values.

When using a queue manager that does not support the Version 2 MQGMO structure, leave the following fields set to their default values:
  GroupStatus
  SegmentStatus
  Segmentation

Also, the MatchOptions field supports only MQMO_MATCH_MSG_ID and MQMO_MATCH_CORREL_ID. If you put unsupported values into these fields, the subsequent MQDestination.get() fails with MQRC_GMO_ERROR. If the queue manager does not support the Version 2 MQGMO structure, these fields are not updated after a successful MQDestination.get().

Distribution lists

Not all queue managers allow you to open an MQDistributionList.

The following classes are used to create distribution lists:
  MQDistributionList
  MQDistributionListItem

MQMessageTracker

You can create and populate MQDistributionLists and MQDistributionListItems in any environment, but not all queue managers allow you to open an MQDistributionList. In particular, z/OS queue managers do not support distribution lists. Attempting to open an MQDistributionList when using such a queue manager results in MQRC_OD_ERROR.

MQPutMessageOptions fields

If a queue manager does not support distribution lists, certain MQPMO fields are treated differently.

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

knownDestCount
unknownDestCount
invalidDestCount
recordFields

These fields are primarily intended for use with distribution lists. However, a queue manager that supports distribution lists also fills in the DestCount fields after an MQPUT to a single queue. For example, if the queue resolves to a local queue, knownDestCount is set to 1 and the other two count fields are set to 0.

If the queue manager does not support distribution lists, these values are simulated as follows:

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

The recordFields variable is used with distribution lists. A value can be written into recordFields at any time, regardless of the environment. It is ignored if the MQPutMessageOptions object is used on a subsequent MQDestination.put() or MQQueueManager.put(), rather than MQDistributionList.put().

MQMD fields

Certain MQMD fields concerned with message segmentation should be left at their default value when using a queue manager that does not support segmentation.

The following MQMD fields are largely concerned with message segmentation:

GroupId
MsgSeqNumber
Offset
MsgFlags
OriginalLength

If an application sets any of these MQMD fields to values other than their defaults, and then does a put() or get() on a queue manager that does not support these, the put() or get() raises an MQException with MQRC_MD_ERROR. A successful put() or get() with such a queue manager always leaves the MQMD fields set to their default values. Do not send a grouped or segmented message to a Java application that runs against a queue manager that does not support message grouping and segmentation.

If a Java application attempts to get() a message from a queue manager that does not support these fields, and the physical message to be retrieved is part of a group of segmented messages (that is, it has non-default values for the MQMD fields), it is retrieved without error. However, the MQMD fields in the MQMessage are not updated, the MQMessage format property is set to MQFMT_MD_EXTENSION, and the true message data is prefixed with an MQMDE structure that contains the values for the new fields.

Restrictions under CICS Transaction Server

In the CICS Transaction Server for z/OS environment, only the main (first) thread is allowed to issue CICS or WebSphere MQ calls.

It is therefore not possible to share MQQueueManager or MQQueue objects between threads in this environment, or to create a new MQQueueManager on a child thread.

"Miscellaneous differences with z/OS" on page 273 identifies some restrictions and variations that apply to the WebSphere MQ classes for Java when running against a z/OS queue manager. Additionally, when running under CICS, the transaction control methods on MQQueueManager are not supported. Instead of issuing MQQueueManager.commit() or MQQueueManager.backout(), applications use the JCICS task synchronization methods, Task.commit() and Task.rollback(). The Task class is supplied by JCICS in the com.ibm.cics.server package.

Running WebSphere MQ classes for Java applications within Java EE

There are certain restrictions and design considerations that must be taken into account before using WebSphere MQ classes for Java in JEE

WebSphere MQ classes for Java has restrictions when used within a JEE environment. There are also additional considerations that should be taken into account when designing, implementing and managing a WebSphere MQ classes for Java application that runs inside a JEE environment. These restrictions and considerations are outlined in the following sections.

JTA transactions restrictions

The only supported transaction manager for applications using WebSphere MQ classes for Java is WebSphere MQ itself. Although an application under JTA control can make use WebSphere MQ classes for Java, any work performed through these classes will not be controlled by the JTA units of work. They will instead form local units of work separate from those managed by the application server through the JTA interfaces. In particular, any rollback of the JTA transaction will not result in a rollback of any sent or received messages. This restriction applies to application or bean managed transactions and to container managed transactions, and all JEE containers. To perform messaging work directly with WebSphere MQ inside application server-coordinated transactions, WebSphere MQ classes for JMS must be used instead.

Thread creation

WebSphere MQ classes for Java spawns threads internally for various operations. For example, when running in BINDINGS mode to make calls directly on a local queue manager, the calls are made on a 'worker' thread created internally by WebSphere MQ classes for Java. Other threads can be spawned internally, for example to clear unused connections from a connection pool or to remove subscriptions for terminated publish/subscribe applications.

Some JEE applications (for instance those running in EJB and Web containers) must not spawn new threads; instead all work must be performed on the main application threads managed by the application server. When applications use WebSphere MQ classes for Java, the application server might not be able to distinguish between application code and the WebSphere MQ classes for Java code, so the threads described above will cause the application to be non-compliant with the container specification. WebSphere MQ classes for JMS does not break these JEE specifications and so should be used instead.

Security restrictions

Security policies implemented by an application server might prevent certain operations that are undertaken by the WebSphere MQ classes for Java API, such as creating and operating new threads of control (as described in the preceding sections).

For example, application servers typically run with Java Security disabled by default, and allow it to be enabled through some application server-specific configuration (some application servers also allow more detailed configuration of the policies used within Java Security). When Java Security is enabled, WebSphere MQ classes for Java might break the Java Security policy threading rules defined for the application server, and the API might not be able to create all the threads that it needs in order to function. To prevent problems with thread management, the use of WebSphere MQ classes for Java is not supported in environments where Java Security is enabled.

Application isolation considerations

An intended benefit of running applications within a JEE environment is *application isolation*. Changes to one application should not adversely affect other applications running in the same application server. The design and implementation of WebSphere MQ classes for Java predate the JEE environment, and WebSphere MQ classes for Java can be used in a manner which does not support the concept of application isolation. Specific examples of considerations in this area include:

- The use of static (JVM process-wide) settings within the MQEnvironment class, such as:
  - the userid and password to be used for connection identification and authentication
  - the hostname, port and channel used for client connections
  - SSL configuration for secured client connections

  Modifying any of the MQEnvironment properties for the benefit of one application also affect other applications making use of the same properties. When running in a multi-application environment such as JEE, each application should use its own distinct configuration through the creation of MQQueueManager objects with a specific set of properties, rather than defaulting to the properties configured in the process-wide MQEnvironment class.
- The MQEnvironment class introduces a number of static methods which act globally on all applications using WebSphere MQ classes for Java within the same JVM process, and there is no way to override this behavior for particular applications. Examples include:
  - configuring SSL properties, such as the location of the key store
  - configuring client channel exits

  - enabling or disabling diagnostic tracing
  - managing the default connection pool used to optimize the use of connections to queue managers

  Invoking such methods affects all applications running in the same JEE environment.
- Connection pooling is enabled to optimize the process of making multiple connections to the same queue manager. The default connection pool manager is process-wide, and shared by multiple applications. Changes to connection pool configuration, such as replacing the default connection manager for one application using the MQEnvironment.setDefaultConnectionManager() method therefore affects other applications running in the same JEE application server.
- SSL is configured for applications using WebSphere MQ classes for Java using the MQEnvironment class and MQQueueManager object properties, and is not integrated with the managed security configuration of the application server itself. You must ensure that you configure WebSphere MQ classes for Java approriately to provide your required level of security, and not use the application server configuration.

WebSphere MQ classes for Java packages

For details of the Java classes and interfaces in the various packages the comprise WebSphere MQ classes for Java see the information center or the Javadoc documentation included on the product CDs.

Package com.ibm.mq

This is the main package of Java classes and interfaces for WebSphere MQ classes for Java.

Package com.ibm.mq.constants

This package contains a single class which allows an application to look up WebSphere MQ constants by name or by value.

Package com.ibm.mq.exits

This package contains a collection of classes and interfaces which allow the Java programmer to work with WebSphere MQ channel exits.

Package com.ibm.mq.headers

This package contains a collection of classes for constructing WebSphere MQ headers and writing them into WebSphere MQ messages, for reading and decoding (parsing) headers in WebSphere MQ messages, and for manipulating the content of headers. Java classes for all WebSphere MQ header types are provided, and the framework also allows you to define classes representing other header types.

Package com.ibm.mq jmqi

This package contains a collection of classes and interfaces which implement the Java Message Queueing Interface (JMQI), the interface which represents the native MQI in the Java environment.

Package com.ibm.mq.pcf

This package contains a collection of classes that represent PCF parameters, PCF parameter structures, and arrays of PCF parameter structures. The methods of these classes allow you to read and write message content, manipulate parameter values, parse PCF content in an MQMessage, and so on.

It also contains agent classes, which provide methods to connect to, send requests to, and receive responses from the WebSphere MQ Command Server.

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-k,u
Tokyo 106, Japan

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

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

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

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

AIX	CICS	DB2
FFST	First Failure Support Technology	i5/OS
IBM	IBMLink	IMS
Language Environment	OS/390	OS/400
POWER	S/390	WebSphere
z/OS	zSeries	

Index

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

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

You can send your comments to IBM in any of the following ways:
- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44-1962-816151
  - From within the U.K., use 01962-816151
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:
- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

SC34-6935-01

Spine information:

IBM WebSphere MQ Using Java Version 7.0