



WebSphere MQ
Programming
Using Base Classes for Java
(Course Code MQ09)

Student Notebook

ERC 2.0

IBM Learning Services
Worldwide Certified Material

Trademarks

IBM® is a registered trademark of International Business Machines Corporation.

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

AIX	AS/400	CICS
DB2	IBM	IMS
MQSeries	OS/2	OS/390
OS/400	RACF	SupportPac
WebSphere	z/OS	

Linux is a registered trademark of Linus Torvalds in the United States and other countries.

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

Java, Java Development Kit, JDBC, JDK and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

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

June 2002 Edition

The information contained in this document has not been submitted to any formal IBM test and is distributed on an “as is” basis without any warranty either express or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the sa Customers attempting to adapt these techniques to their own environments do so at their own risk. The original reposi

© Copyright International Business Machines Corporation 2000, 2002. All rights reserved.

This document may not be reproduced in whole or in part without the prior written permission of IBM.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure i

Contents

Trademarks	ix
Course Description	v
Agenda	vii
Unit 1. Introduction	1-1
Unit Objectives	1-2
1.1 Introduction to Java	1-3
What is Java?	1-4
JVM - Java Virtual Machine	1-5
JDK - Java Development Kit	1-6
Java Packages	1-8
What is a Java Class?	1-10
Class Syntax	1-12
Method Syntax	1-13
Constructors	1-15
1.2 Introduction to MQSeries	1-17
MQSeries - Commercial Messaging	1-18
The Three Styles of Communication	1-19
Queues	1-21
Queue Managers	1-22
Publish/Subscribe	1-23
MQI - Messaging and Queuing Interface	1-25
Building an MQSeries Java Application	1-27
MQSeries for Java Classes	1-28
Environment	1-29
SupportPacs	1-30
1.3 Summary	1-33
Unit Summary	1-34
Unit 2. Queue Manager Connection	2-1
Unit Objectives	2-2
2.1 How to Connect to a Queue Manager	2-3
What is a Queue Manager?	2-4
Connection Type	2-5
Connection Type: Client Connection	2-6
Connection Type: Bindings Mode	2-7
What's the Advantage?'	2-8
Java Basics: Application vs. Applet	2-9
Defining Which Connection to Use	2-10
Other Useful MQEnvironment Variables	2-11
Connecting to a Queue Manager	2-13
Completion and Reason Codes	2-14

Testing the Connection	2-15
Disconnecting from a Queue Manager	2-16
Example	2-17
2.2 Checkpoint and Summary	2-19
Unit Checkpoint	2-20
Unit Summary	2-21
Unit 3. Working with Queues	3-1
Unit Objectives	3-2
3.1 Working with Queues	3-3
Accessing Queues	3-4
Open Options	3-6
Reason Codes when Opening a Queue	3-7
Closing a Queue	3-8
Alias Queues	3-10
Model Queues	3-11
Dynamic Queues	3-12
Dynamic Queue Names	3-13
3.2 Checkpoint and Summary	3-15
Unit Checkpoint	3-16
Unit Summary	3-17
Unit 4. Error Handling	4-1
Unit Objectives	4-2
4.1 Error Handling	4-3
MQException Class	4-4
MQException.log	4-5
MQException.completionCode and MQException.reasonCode	4-6
MQException.exceptionSource	4-7
Try / Catch Blocks	4-8
4.2 Checkpoint and Summary	4-9
Unit Checkpoint	4-10
Unit Summary	4-11
Unit 5. Messaging and Queuing	5-1
Unit Objectives	5-2
5.1 The Message Object	5-3
Message = Header + Application Data	5-4
Constructing a Message	5-5
The MQMessage Object	5-6
User Data Formats	5-7
write and writeString	5-8
writeChar, writeChars and writeUTF	5-10
Numeric Data Formats	5-11
Other Data Formats	5-13
Changing the Buffer Location	5-15
5.2 Putting a Message	5-17

Message Descriptor Properties	5-18
Priority	5-19
Persistence	5-20
messageld	5-21
Put Message Options	5-22
PMO: options	5-23
Putting Messages	5-24
PMO: resolvedQueueName / resolvedQueueManagerName	5-25
Put Message Buffer Considerations	5-27
5.3 Getting a Message	5-29
Getting Messages	5-30
Processing the Message	5-31
Creating the Message Buffer	5-32
Get Message Options	5-34
GMO: options	5-35
Issue the Get Request	5-37
Retrieving Message Length	5-39
Retrieving User Data	5-40
Catering for the Exception	5-42
5.4 Checkpoint and Summary	5-45
Unit Checkpoint	5-46
Unit Checkpoint	5-47
Unit Checkpoint	5-48
Unit Checkpoint	5-49
Unit Summary	5-50
Unit 6. Messages Types	6-1
Unit Objectives	6-2
6.1 Requests and Replies	6-3
Message Types	6-4
How to use Message Types	6-5
Request	6-7
replyToQueueManagerName/replyToQueueName	6-8
Reply	6-9
Retrieving the Reply Queue and Reply Queue Manager Names	6-10
6.2 Reports	6-11
Report Messages	6-12
Exception Reports	6-13
Expiry Reports	6-15
COA and COD Reports	6-17
Feedback	6-19
COPY_MSG_ID_TO_CORREL_ID	6-20
6.3 Checkpoint and Summary	6-21
Unit Checkpoint	6-22
Unit Summary	6-23

Unit 7. Retrieval of Messages	7-1
Unit Objectives	7-2
7.1 Message Id and Correlation ID	7-3
messageld and correlationId	7-4
messageld and correlationId	7-6
messageld and correlationId	7-7
Using messageld and correlationId	7-9
Using messageld and correlationId	7-11
7.2 Waiting for Replies	7-13
Wait	7-14
GMO: wait option and waitInterval	7-16
Wait with waitInterval	7-17
Wait Example	7-18
7.3 Message Groups	7-19
Introduction	7-20
The Message Group Variables	7-21
Message Groups and the groupId Variable	7-22
Putting a Message to a Group	7-23
Getting a Message from a Group	7-25
Match Options	7-27
Spanning Units of Work	7-28
7.4 Message Segments	7-31
Segmentation by the Queue Manager	7-32
Message Segmentation Variables	7-34
Segmentation by the Program	7-35
Selective Reassembly	7-37
7.5 Checkpoint and Summary	7-39
Unit Checkpoint	7-40
Unit Checkpoint	7-41
Unit Summary	7-42
Unit 8. More on Messages	8-1
Unit Objectives	8-2
8.1 Triggering	8-3
Trigger Types	8-4
Triggering Characteristics	8-6
Process	8-7
Initiation Queue	8-8
Trigger Monitor	8-9
Implementation of Triggering	8-10
8.2 Inquire and Set Attributes	8-11
Inquire and Set Attributes	8-12
Why Inquire and Set?	8-13
Inquire Attributes	8-14
Set Attributes	8-15
8.3 Data Conversion	8-17
Data Formats	8-18

Selecting the Data Format	8-19
Data Conversion by Write or Read Methods	8-21
Requesting Conversion	8-22
8.4 Distribution Lists	8-25
What are Distribution Lists?	8-26
Creating a Distribution List	8-28
Opening a Distribution List	8-29
Putting a Message onto a Distribution List	8-30
Error Handling	8-31
Problem Determination	8-33
8.5 Checkpoint and Summary	8-35
Unit Checkpoint	8-36
Unit Summary	8-37
Unit 9. Security	9-1
Unit Objectives	9-2
9.1 Local Security	9-3
User Identification	9-4
9.2 Context Variables	9-7
The Context Variables	9-8
Context Properties of a Message	9-9
Manipulating the Context Variables	9-11
Pass Context Example Code	9-13
9.3 Alternate User ID	9-15
Alternate User ID	9-16
9.4 Checkpoint and Summary	9-19
Unit Checkpoint	9-20
Unit Summary	9-21
Unit 10. Units of Work	10-1
Unit Objectives	10-2
10.1 Local Units of Work	10-3
Unit of Work	10-4
Implementing the Local UOW Processing	10-6
Commit	10-8
Backout	10-9
10.2 Global Units of Work	10-11
Implementing the Global UOW Processing	10-12
Global UOW Example Code	10-13
Considerations for Global UOW Processing	10-14
10.3 Checkpoint and Summary	10-15
Unit Checkpoint	10-16
Unit Summary	10-17
Unit 11. Exits	11-1
Unit Objectives	11-2
11.1 Exits	11-3

Channel Exits	11-4
Client Channel Exits	11-5
MQSendExit	11-6
MQReceiveExit	11-7
MQSecurityExit	11-8
MQChannelExit and MQChannelDefinition	11-9
Example Code	11-10
11.2 Checkpoint and Summary	11-13
Unit Checkpoint	11-14
Unit Summary	11-15
Unit 12. Multithreading	12-1
Unit Objectives	12-2
12.1 Multithreading	12-3
Multithreaded Programs	12-4
Thread Synchronization	12-5
Multithreading Example	12-6
12.2 Checkpoint and Summary	12-7
Unit Checkpoint	12-8
Unit Summary	12-9
Appendix A. Checkpoint Solutions	A-1
Appendix B. Bibliography	B-1
Appendix C. Glossary of terms and abbreviations	C-1

Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM® is a registered trademark of International Business Machines Corporation.

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

AIX	AS/400	CICS
DB2	IBM	IMS
MQSeries	OS/2	OS/390
OS/400	RACF	SupportPac
WebSphere	z/OS	

Linux is a registered trademark of Linus Torvalds in the United States and other countries.

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

Java, Java Development Kit, JDBC, JDK and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

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

Course Description

WebSphere MQ Programming Using Base Classes for Java

Duration: Three days

Purpose

This classroom class teaches Java application programmers to develop basic MQSeries applications written in Java. The course contains extensive machine exercises.

Audience

The class is for Java programmers who want to learn to write basic MQSeries applications.

Prerequisites

Students must know basic Java application programming and also basic MQSeries information.

Objectives

After completing this course, you should be able to:

- Design and develop application programs for MQSeries Java environment
- Develop Java applets and Java applications
- Connect and disconnect the application to a queue manager
- Work with messages
- Handle and manage exception conditions
- Manipulate message delivery
- Use triggering functions
- Write messages on a distribution list

Curriculum relationship

- MQ01 MQSeries Technical Introduction (classroom)
- MQ82 MQSeries Technical Introduction (CBT)

- JA30 Intro to Developing OO Applications with Java for OO Developers
- JA32 Intro to Developing OO Applications with Java for 3GL Developers
- JA34 Developing and Testing OO Applications with Java

Agenda

Day 1

Welcome
Course Introduction and Administration
Unit 1 - Introduction
Unit 2 - Queue Manager Connection
Exercise 1 - Customize the Queue Manager
Unit 3 - Working with Queues
Exercise 2 - Basic Put and Get (Part 1)
Unit 4 - Error Handling
Unit 5 - Messaging and Queuing (Part 1)

Day 2

Day 1 - Review
Unit 5 - Messaging and Queuing (Part 2)
Exercise 2 - Basic Put and Get (Part 2)
Unit 6 - Message Types
Unit 7 - Retrieval of Messages
Exercise 3 - Request and Reply
Unit 8 - More on Messages
Exercise 4 - Triggered Server Application

Day 3

Day 2 - Review
Unit 9 - Security
Unit 10 - Units of Work
Exercise 5 - Get and Reply under Syncpoint
Unit 11 - Exits
Unit 12 - Multithreading
Exercise 6 - Put it Together
“End of Course” Evaluations

Unit 1. Introduction

What This Unit is About

In this unit, you will be introduced to Java and MQSeries.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Understand the basic Java concepts
- Understand the basic MQSeries concepts

References

SC34-5456 *MQSeries Using Java*

[http://www.ibm.com/software/ts/mqseries/messaging/
WebSphere MQ](http://www.ibm.com/software/ts/mqseries/messaging/WebSphereMQ)

<http://www.java.sun.com>
The Source for Java Technology

Objectives

- Learn the basic **Java** concepts
- A Review of **MQSeries**

Figure 1-1. Unit Objectives

MQ092.0

Notes:

1.1 Introduction to Java

What is Java?

- An *Object Oriented* programming language developed by Sun Microsystems
- A set of standardized *Class Libraries* (packages), that support :
 - creating graphical user interfaces
 - communicating over networks
 - controlling multimedia data
- Java requires a *virtual machine* (run-time environment)

Figure 1-2. What is Java?

MQ092.0

Notes:

Java is an object oriented programming language, developed by Sun Microsystems. As such, it incorporates many of the object-oriented characteristics of more traditional object oriented languages like C++ or Smalltalk.

Like other OO languages, Java comes with a set of class libraries containing the code which is used by your program to communicate with users, and with other machines.

Java applications are bundled up into deliverable chunks of code called "packages".

What makes Java different from the other members of the object oriented family is its ability to "run anywhere". In theory, once written, a Java program can run on any machine which has a Java Virtual Machine (JVM). The Java Virtual Machine provides the run-time environment in which a Java program executes. Most Web browsers come with a JVM built in, so that special Java programs called applets can be run inside the browser. It is this aspect of Java which makes it so attractive for the writers of distributed applications.

JVM - Java Virtual Machine

- Java is compiled to *byte-codes* which are interpreted by the Java Virtual Machine (JVM)
- The **JVM** runs standalone, or as part of another software environment, on many operating systems

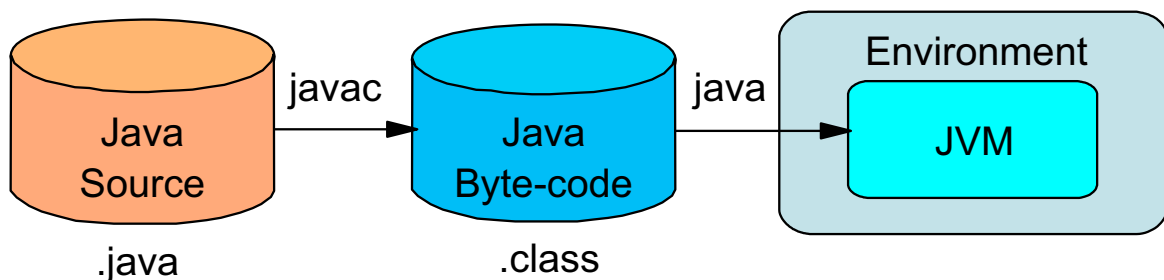


Figure 1-3. JVM - Java Virtual Machine

MQ092.0

Notes:

JVM is the processor on which Java's byte-codes run. It is the instruction set that the interpreter understands.

The Java compiler, `javac`, takes as input the java source code and produces as output, java byte-codes. When the program is executed, these byte-codes are interpreted by the JVM, which executes them by issuing instructions understood by the machine on which the JVM is running. The Java compiler creates one `.class` file for each class definition.

Because the JVM is not unique to any particular hardware/operating system, `.class` files are portable to any implementation of the JVM. Most Web browsers come with a JVM built into them, to allow them to run special Java programs called *applets*. An applet is a Java program which has been written to be downloadable to a Web Browser, and which usually only supports a subset of the Java classes.

The JVM includes a byte-code verifier to validate byte-codes as they are imported into the JVM. It also includes a Class Loader, which can be used to perform security checking on classes loaded over the network.

JDK - Java Development Kit

- Package of **tools** for writing Java programs
- Three principal tools are :
 - javac** -- **compiles** Java source into bytecode

 - java** -- **executes** stand-alone applications

 - appletviewer** -- Tool for **testing applets**

Figure 1-4. JDK - Java Development Kit

MQ092.0

Notes:

JDK is a package of tools for writing Java programs.

Other parts of the JDK are:

- javap - disassembles .class (bytecode) file
- jar - creates archive file
- javadoc - produces HTML documentation from source
- jar signer - prepares jar files to be authenticated
- javah - creates header and stub for inter language linking
- native2ascii - converts native to Unicode encoded file
- jdb - a rudimentary debugger
- rmic - creates stub and skeleton for RMI
- keytool - creates pairs of keys used to "sign" and authenticate programs
- rmiregistry - starts remote object registry naming server
- policytool - defines authentication criteria and allowed functions of "trusted" programs
- serialVer - creates unique id for serialization

The javap command can be used to extract and list a brief description of a .class file's API, but without any comments. It can also list the java bytecode with the -c flag.

The javadoc command was used to generate the apidocs using the `"/** ... */"` (documentation comments).

Java Packages

java.applet	Applet development
java.awt	Abstract Window Toolkit
java.awt.event	Input event processing
java.io	Input/output streams
java.lang	Core language support
java.net	Networking support
java.util	Miscellaneous utility support
javax.swing	Release 2 GUI support

Figure 1-5. Java Packages

MQ092.0

Notes:

java.applet provides the base class for all Applets. It also provides interfaces to the Web Browser and the services it exports.

java.awt is the GUI class library. It provides support for the development of User-Interfaces.

java.awt.event provides support events and processing input. New for version 1.1.

java.io provides support for stream based I/O. It also supports files and file descriptors. Two major divisions exist for byte and character streams. Character streams are new for Version 1.1.

java.lang is the set of fundamental classes which make up Java's environment. These include String, Thread, System, Runtime,...

java.net provides support for operating in a networked environment, e.g. Sockets, URLs, etc.

java.util includes useful miscellaneous classes including some basic container classes.

javax.swing contains new light-weight GUI objects Jxxxxxx.

There are many more packages included with Version 1.2 of the Java Development Kit. See the API documentation for more information on these and the other packages.

What is a Java Class?

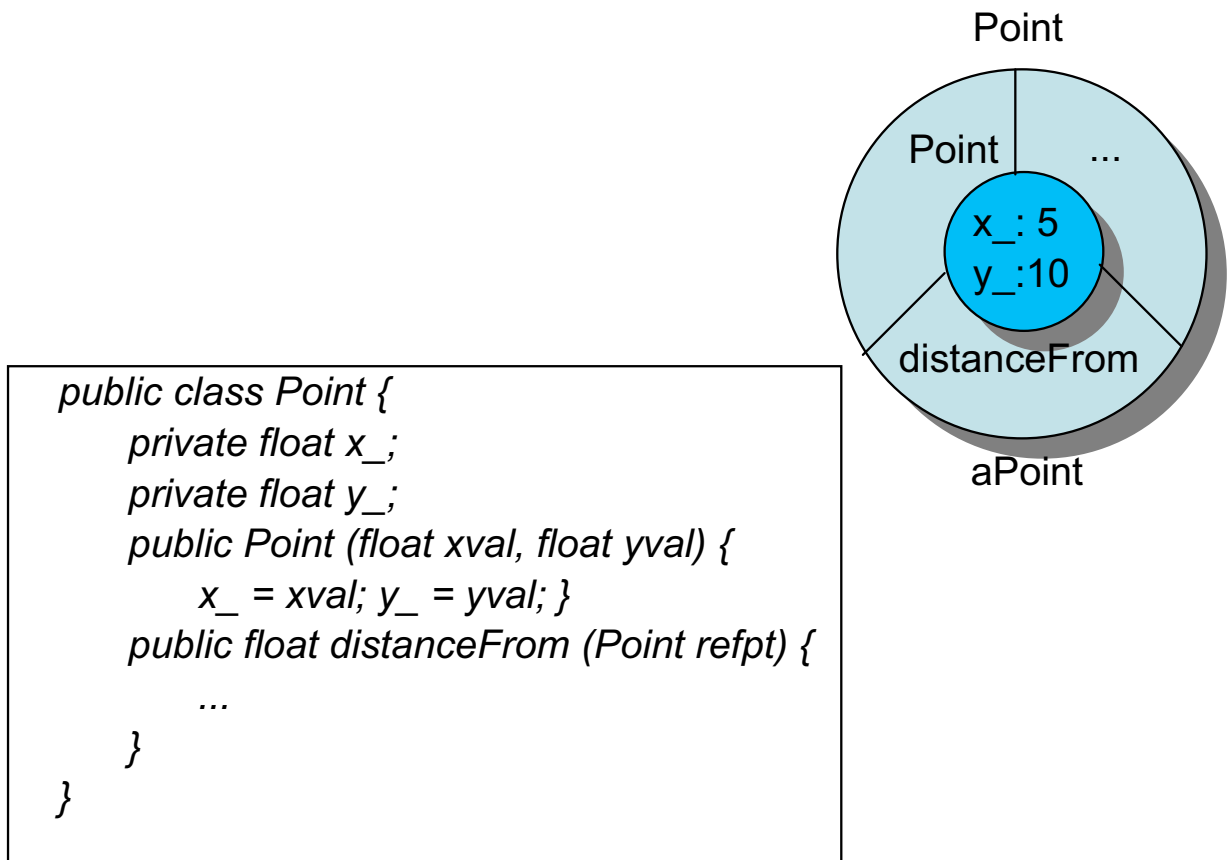


Figure 1-6. What is a Java Class?

MQ092.0

Notes:

The Java language is a class-based object-oriented language, implementing inheritance.

Your Java program will probably create a class, which does **three** things:

1. Encapsulates your new code so that the implementation is hidden from the users of your code.
2. Inherits the characteristics of another, higher class, of which yours is a subset
3. Provides one or more methods which cause your code to perform some action related to an instantiation of your class.

When you define a class, you define how an object of that class will look and behave once instantiated.

In our example here, we have a class `Point`, which has one public function, the `distanceFrom` function. Once we have written our Java class file, any other Java program can instantiate two `Point` objects, passing values for the `x` and `y` coordinates, and then invoke the `distanceFrom` method of one of the points, passing the other as an input parameter, to find out the distance between the two points.

A sample invocation of distanceFrom could be:

```
public static void main(String[] args) {
    Point p = new Point(0,0);
    Point q = new Point(5,12);
    System.out.println("distanceFrom p to q  =" +p.distanceFrom(q));
}
```

The distanceFrom function could be implemented as:

```
public float distanceFrom(Point refpt) {
    double dx = X_-refpt.getX_
    double dy = y_-refpt.getY_
    return math.sqrt(Math.pow(dx,2.0)+Math.pow(dy,2.0));
}
```

Class Syntax

- `[modifier] class <ClassName> [extends <ClassName>] [implements <intfNames>] { ... }`

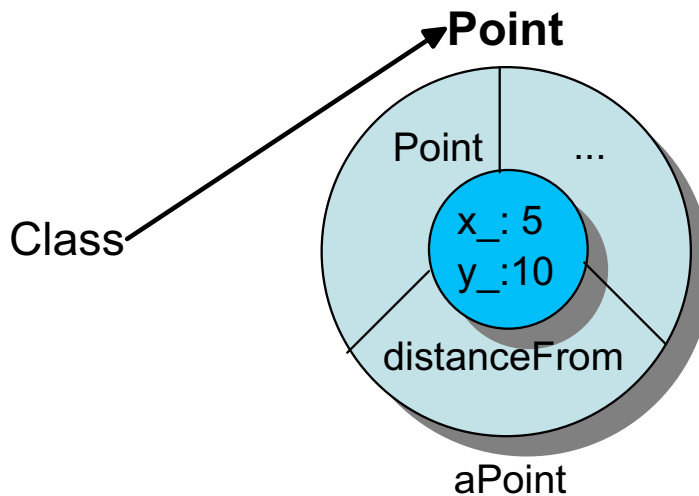


Figure 1-7. Class Syntax

MQ092.0

Notes:

Modifiers - A class can be specified as any of:

- abstract - No objects can be instantiated. Only an abstract class may declare abstract methods
- final - No subclasses
- public - Accessibility from outside of its package

Extends - (Optional) Name of the class which is the immediate superclass

Implements - (optional) name of the interface(s) implemented by this class. If the class is not abstract, then every method of each interface must be defined by some superclass or by itself.

Method Syntax

- *[modifier]* <ClassName> <methName> ([<args>])
[throws <ExceptNameList>] { ... }

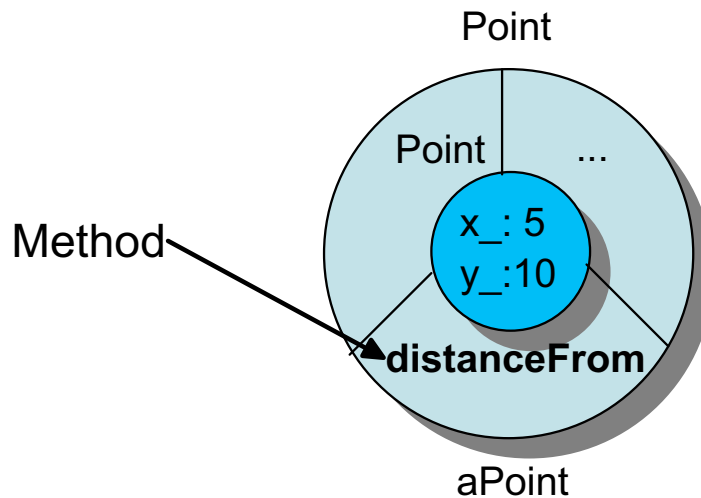


Figure 1-8. Method Syntax

MQ092.0

Notes:

Modifiers - In addition to access specifiers:

- static - class method. Does not require an object of class. Only has access to static fields and methods
- abstract - no implementation provided, subclass must implement or redeclare abstract
- final - cannot be overridden. (A private method is effectively final)
- native - declares a signature for code to be implemented outside of Java
- synchronized - states that a thread must acquire a monitor lock prior to execution

Throws list - Declares any exceptions that result from its execution

Again, many combinations of the modifiers are legal.

Like static (class) variables, a static method may be called by using the class name as the "scope" resolution. For example:

```
public class X
{
    public static void f1() { /* ... */ }
}
```

```
//in some code block somewhere invoke static method f1.
X.f1();
```

Unlike class variables, a static method may also be called with an object as the target, e.g.,

```
public class Y extends X { /* ... */ }

Y y = new Y();
y.f1();          // invoke static method f1
```

Note, these are different. Static methods can be dispatched using "Class." or "object.," but the behavior is the static type of the object reference. Thus, above, `y.f1()` looks at the runtime type of `y` to determine which `f1` method to invoke. In the case of `X.f1()`, the method is statically dispatched.

Constructors

- `[modifier] <ClassName> ([<args>]) [throws <ExceptNameList>] { ... }`

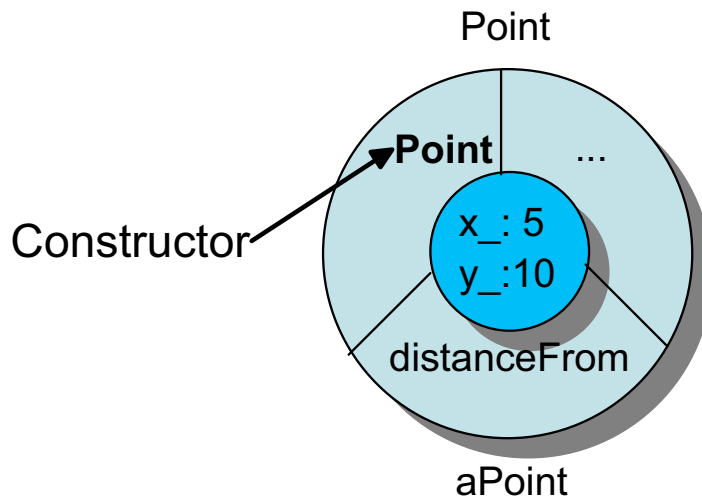


Figure 1-9. Constructors

MQ092.0

Notes:

Every class in Java has at least one constructor method, which has the same name as the class. When the constructor method of a class is invoked, all necessary initialization for a new instance of that class is performed. If no constructor is defined for a class, Java uses a default constructor that takes no arguments, and performs no special initialization for that class.

Java allows you to define more than one constructor for a class, so that you can use the one which is most suited to the circumstances in which you are working.

You can use the usual modifiers, but in general you would expect constructors to be public methods.

1.2 Introduction to MQSeries

MQSeries - Commercial Messaging

- A single, multi-platform API
- Assured message delivery
- Faster application development
- Time independent processing
- Application parallelism

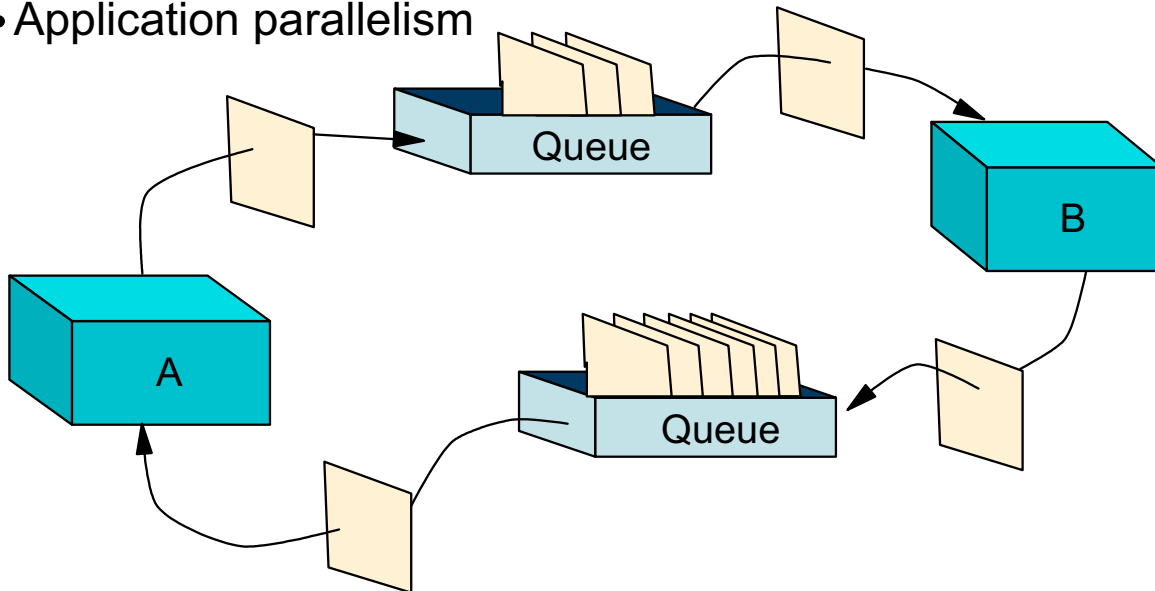


Figure 1-10. MQSeries - Commercial Messaging

MQ092.0

Notes:

In a commercial messaging environment, programs send data to other programs by putting the data in the form of messages onto queues.

Programs do not have to be concerned with the availability of the receiving program or the network that might lie in between.

Once a program receives a successful return code from the call that puts the message on the queue, it is free to continue to do other work. The delivery of the message becomes the responsibility of the queue manager.

The getting application can be started when a message arrives on its queue. This is called event-driven or message-driven processing.

The Three Styles of Communication

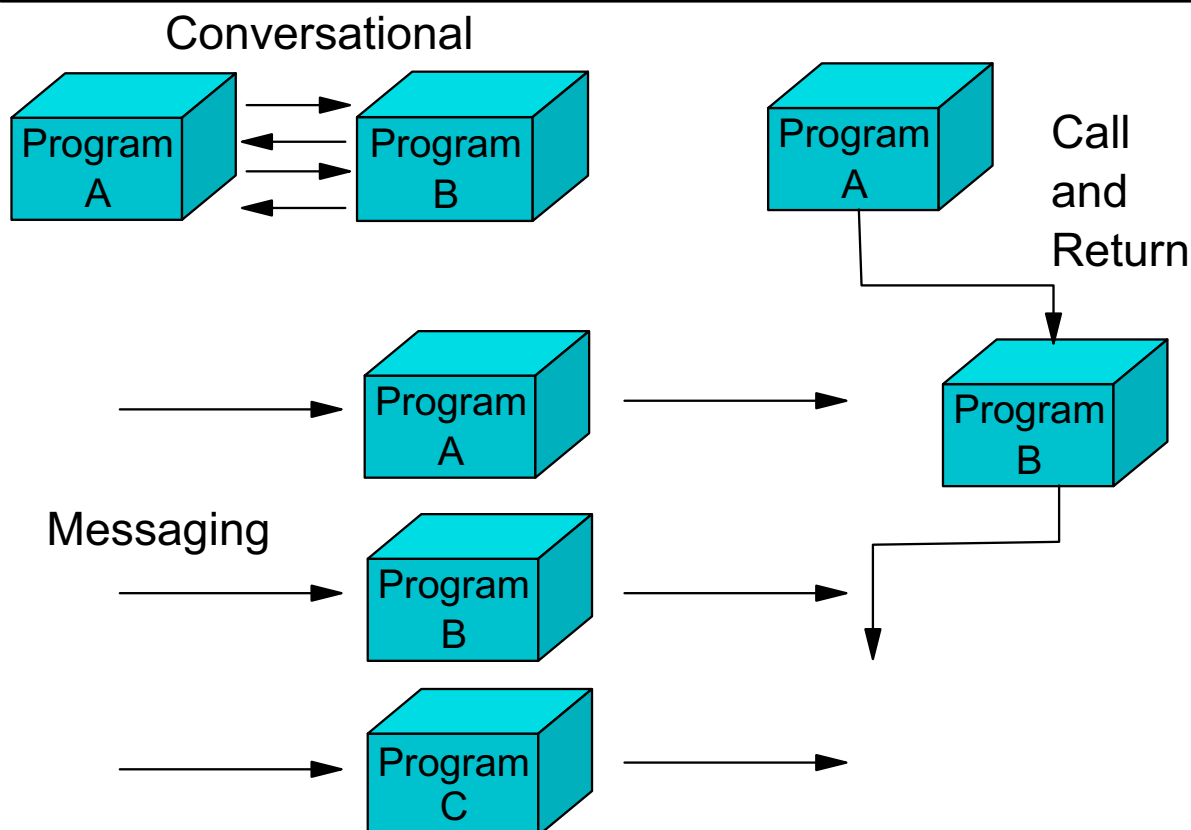


Figure 1-11. The Three Styles of Communication

MQ092.0

Notes:

Conversational or transaction oriented communication is characterized by two or more programs executing in a cooperative manner to perform a transaction. They communicate with each other through an architected interface. While one program is waiting for a reply from another program with which it is communicating, it may continue with other processing. APPC, CPI-C and the sockets interface of TCP/IP are examples of this type of communication.

The call and return style is similar, except that the interface is structured to resemble a call-and-return mechanism. When one program calls another program, the former is blocked and cannot perform any other processing. Remote procedure call (RPC) is an example of this style of communication.

The messaging style implies that communicating programs can execute independently. An executing program receives input in the form of messages and outputs its results also as messages. A message that is the output from one program becomes the input to another program, but there is no requirement that the latter must be executing when the former outputs the message. Contrast this with the conversational and call-return styles where all

cooperating partners must be executing at the same time. The messaging style is used by MQSeries.

Queues

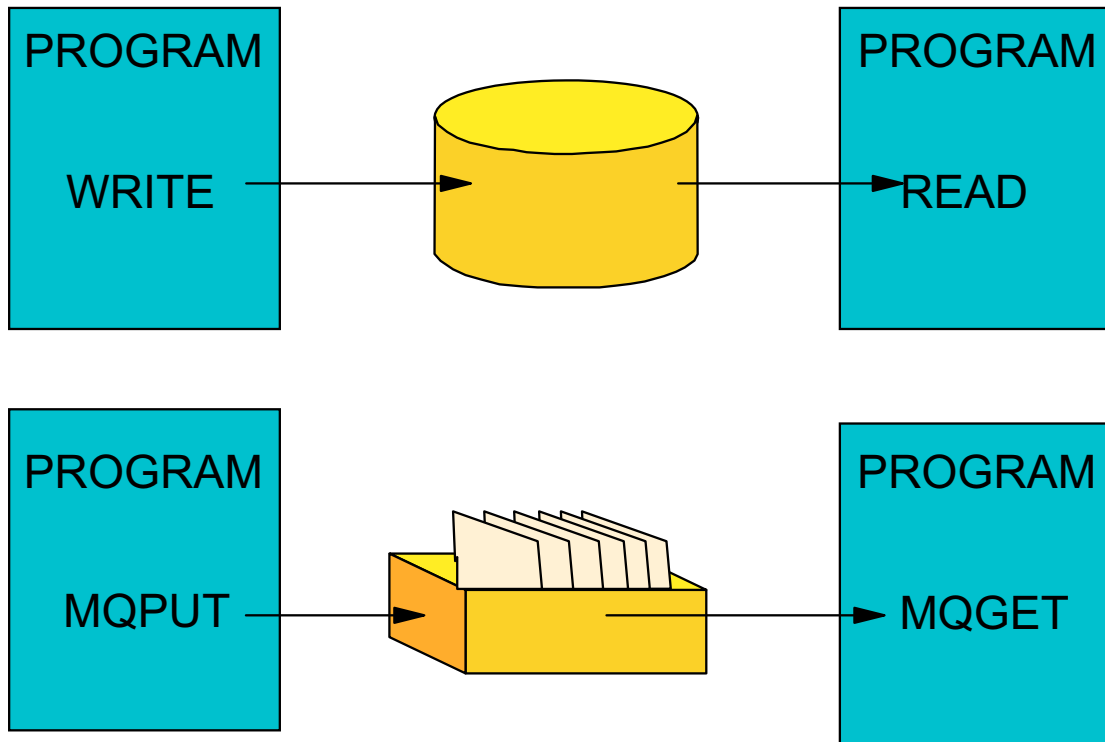


Figure 1-12. Queues

MQ092.0

Notes:

Queues are as easy to access as files. However, there are some major differences to be aware of:

- GETting a message from a queue can delete it from the queue
- When a message is PUT on a queue in syncpoint, it becomes visible to another process as soon as it is committed.

Typically, records are written to a file and when the file is closed, the records are made available to be read. The immediate nature of message delivery means that messages can still be PUT by the initiating application while the retrieving application is GETting message. This can allow for shorter overall processing.

Queue Managers

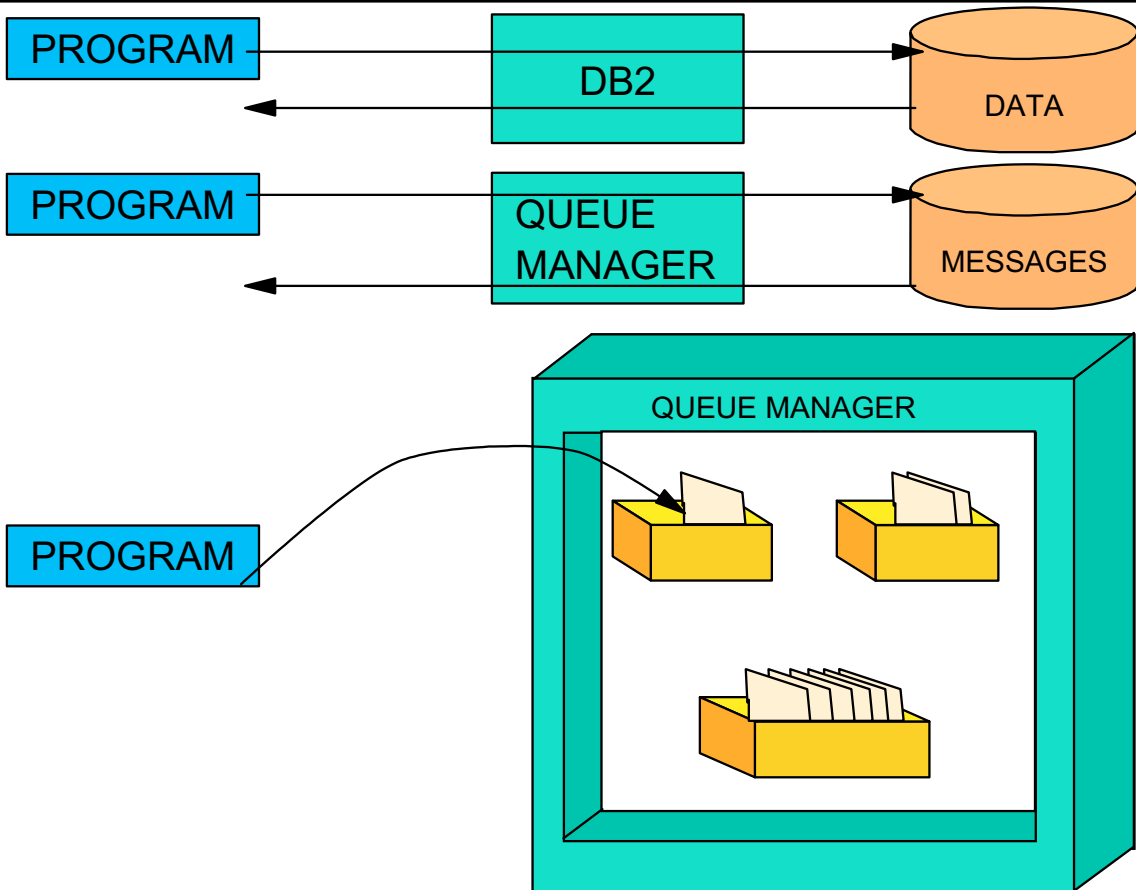


Figure 1-13. Queue Managers

MQ092.0

Notes:

Queues are controlled by a queue manager.

Queue managers provide:

- interface to messages on queues (the MQI)
- security and authorization control
- administration control

Publish/Subscribe

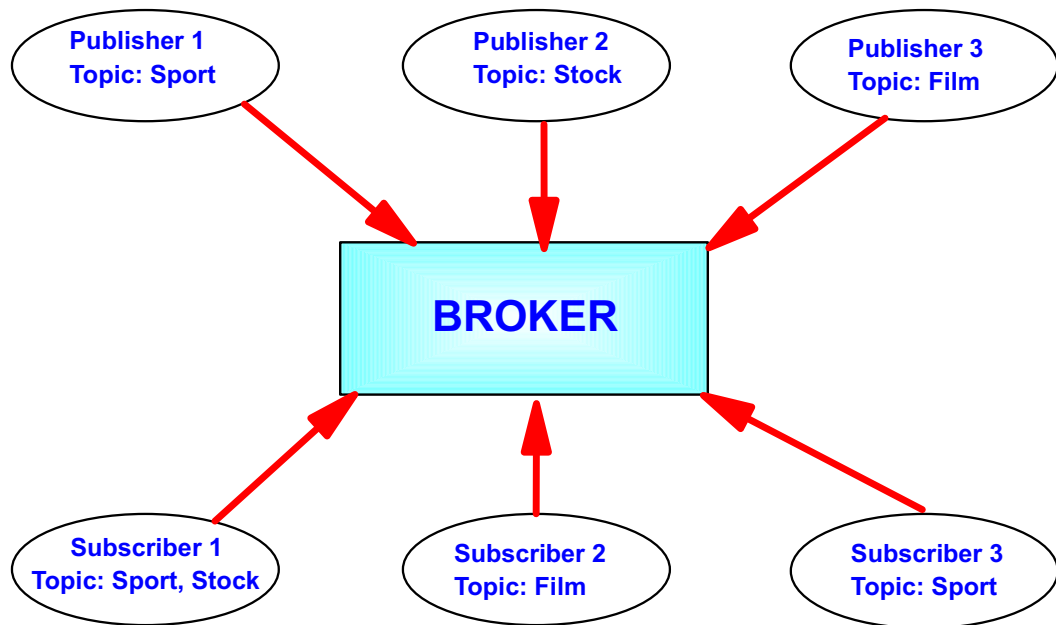


Figure 1-14. Publish/Subscribe

MQ092.0

Notes:

With MQSeries Publish/Subscribe, you eliminate the need for an application to know specifically where to send data. All it does is send information it wants to share to a standard destination managed by MQSeries Publish/Subscribe, and let MQSeries Publish/Subscribe deal with the distribution. The target application does not have to know anything about the source of the information it receives.

Publishers supply information about a subject, without having any knowledge about the applications that are interested in the information.

Subscribers decide what information they are interested in, and wait to receive that information. Subscribers can receive information from many different publishers, and the information can also be sent to other subscribers. The information is sent in an MQSeries message, and the subject of the information is identified by a topic. The publisher specifies the topic when it publishes the information, and the subscriber specifies the topics on which it wishes to receive publications. The subscriber is only sent information about those topics it subscribes to.

Interactions between publishers and subscribers are all controlled by a broker. The broker receives messages from publishers, and subscription requests from subscribers (to a range of topics). The broker's job is to route the published data to the target subscribers.

The broker uses standard MQSeries facilities to do this, so applications can use all the features that are available to existing MQSeries applications. This means that you can use persistent messages to get once-only, assured delivery, and that messages can be part of a transactional units-of-work to ensure that messages are delivered to the subscriber only if they are committed by the publisher.

MQI - Messaging and Queuing Interface

- **Simple "CALL" interface**

- **"Major" calls**

- MQCONN
- MQOPEN
- MQPUT
- MQPUT1
- MQGET
- MQCLOSE
- MQDISC

- **"Minor" calls**

- MQINQ
- MQSET
- MQCONNX
- MQBEGIN
- MQCMIT
- MQBACK

Figure 1-15. MQI - Messaging and Queuing Interface

MQ092.0

Notes:

A brief look at the calls:

- MQCONN Connect to a queue manager
- MQOPEN Open an MQSeries object
- MQPUT Place a message on a queue
- MQPUT1 Put a single message on a queue (not available with MQSeries for Java classes)
 - No MQOPEN required
 - No MQCLOSE required
 - Use for single messages only
- MQGET Retrieve a message from a queue
- MQCLOSE Close an MQSeries object
- MQDISC Disconnect from a queue manager

- MQINQ Inquire about attributes of an MQSeries object
- MQSET Set some specific queue attributes
- MQCONNX Connect to a queue manager using some options
- MQBEGIN Begin a unit of work coordinated by the queue manager
 - Only on Version 5 queue managers
 - May involve external resource managers
- MQCMIT Syncpoint notification for all PUTs and GETs since last syncpoint
 - Not on z/OS using CICS or IMS
 - Single phase commit process only
- MQBACK Backout notification for all PUTs and GETs since last syncpoint
 - Not on z/OS using CICS or IMS
 - Single phase commit process only

Building an MQSeries Java Application

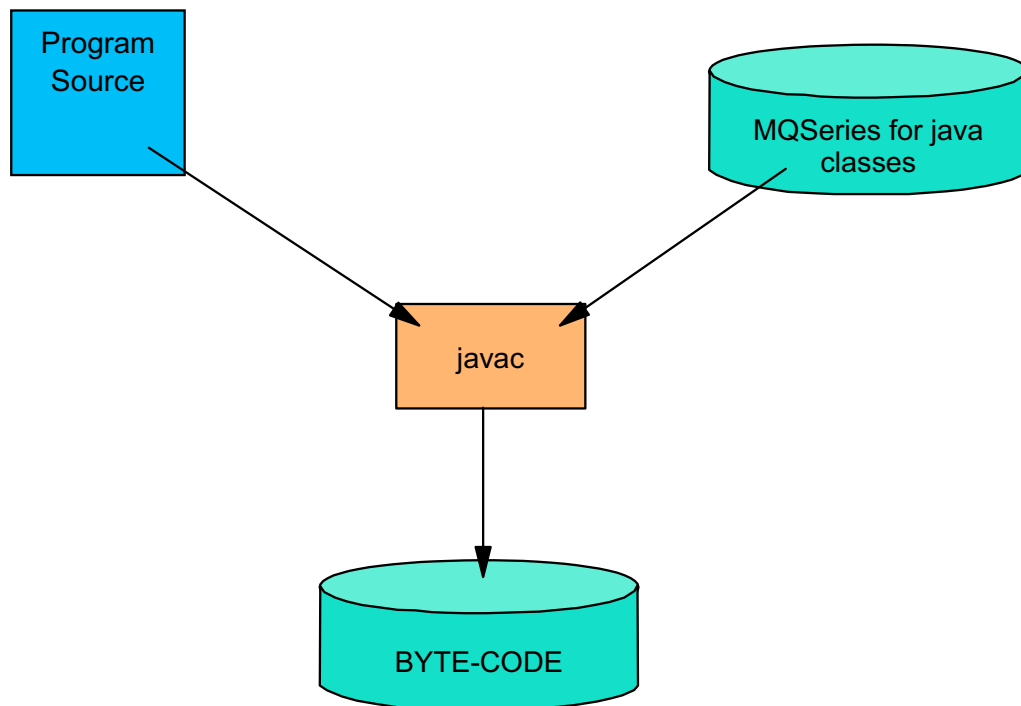


Figure 1-16. Building an MQSeries Java Application

MQ092.0

Notes:

The procedure to follow for successful compilation is slightly different for each Java development environment, you should refer to the products documentation.

MQSeries for Java Classes

MQSeries for Java Classes are included in the following .jar files:

<i>com.ibm.mq.jar</i>	This code includes support for all the connection options.
<i>com.ibm.mq.iiop.jar</i>	This code supports only the Visibroker connection .
<i>com.ibm.mqbind.jar</i>	This code supports only the bindings connection .

Figure 1-17. MQSeries for Java Classes

MQ092.0

Notes:

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

Environment

To run MQSeries Base Classes forJava, the following software is required:

- [MQSeries](#) for the server platform
- [Java Development Kit](#) for the server platform
- [Java Development Kit](#), or [Java Runtime Environment](#), or Java-enabled [Webbrowser](#) for client platforms

To run Pub/Sub applications, also the following is needed:

- MQSeries [Publish/Subscribe](#)
- or [WebSphere MQ Integrator](#)

Figure 1-18. Environment

MQ092.0

Notes:

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

- MQSeries for the server platform you wish to use.
- Java Development Kit (JDK) for the server platform.
- Java Development Kit, or Java Runtime Environment (JRE), or Java-enabled Web browser for client platforms.
- For z/OS & OS/390, OS/390 Version 2 Release 9 or higher, or z/OS, with UNIX System Services (USS).
- For OS/400, the AS/400 Developer Kit for Java, 5769-JV1, and the Qshell Interpreter, OS/400 (5769-SS1) Option 30.

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

SupportPacs

- **MA88**: MQSeries classes for Java and MQSeries classes for Java Message Service
- **MA0C**: MQSeries - Publish/Subscribe
- **MS0B**: MQSeries Java classes for PCF

Figure 1-19. SupportPacs

MQ092.0

Notes:

SupportPac MA88 provides support for developing MQSeries Java applications through the MQSeries classes for Java and the MQSeries classes for Java Messaging Service (JMS).

SupportPac MA0C is a product extension that helps to distribute information to where it is wanted. Providers and consumers of information don't need to know anything about each other. Information is sent to MQSeries Publish/Subscribe, which looks after the distribution.

MQSeries Publish/Subscribe is available for Microsoft Windows NT, Windows 2000, AIX, HP-UX, Linux and Sun Solaris.

SupportPac MS0B contains a set of Java classes representing PCF header structures as well as an agent that can be used to simplify the task of communicating with a target queue manager and thus enable the use of MQSeries Programmable Command Formats for queue manager administration.

The MQSeries Programmable Command Formats (PCF) provide the capability to perform administration tasks on a queue manager by sending and receiving MQSeries messages of

a special format. PCF request messages are sent to the queue manager's command queue, where they are processed by the command server and replies returned to the designated reply-to queue.

1.3 Summary

Summary

You should now understand the basic Java concepts:

- JVM, JDK
- Classes
- Methods

and the basic MQSeries concepts:

- Messaging
- Queue managers
- Queues

Notes:

Unit 2. Queue Manager Connection

What This Unit is About

In this unit, you will learn about queue managers and how to connect to them using java applets or applications.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Choose which connection mode to use
- Set up the MQ Environment
- Connect to and disconnect from a queue manager
- Work with the MQQueueManager and MQEnvironment classes
- Understand the differences between Applet and Application/Servlet

How You Will Check Your Progress

Accountability:

- Checkpoint
- Machine exercises

References

SC34-5456 *MQSeries Using Java*

GC33-1632 *MQSeries Clients*

<http://www.ibm.com/software/ts/mqseries/messaging/>
WebSphere MQ

Objectives

- Bindings mode vs Client connection mode
- Applet vs Application/Servlet
- Setting up the MQ environment
- Connecting to a queue manager
- The MQQueueManager class
- Disconnecting from a queue manager

Figure 2-1. Unit Objectives

MQ092.0

Notes:

2.1 How to Connect to a Queue Manager

What is a Queue Manager?

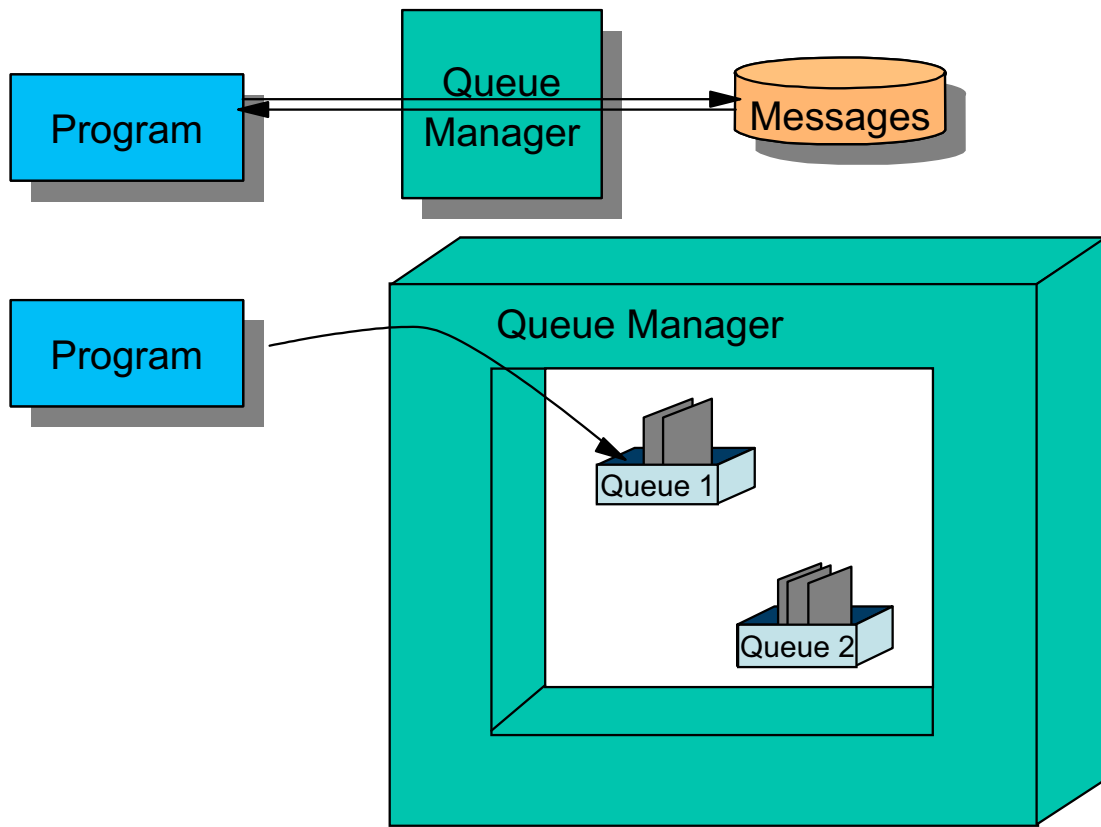


Figure 2-2. What is a Queue Manager?

MQ092.0

Notes:

Queues are controlled by a queue manager.

Queue managers provide:

- interface to messages on queues (the MQI)
- security and authorization control
- administration control

A queue manager is managed using the `MQQueueManager` class.

Connection Type

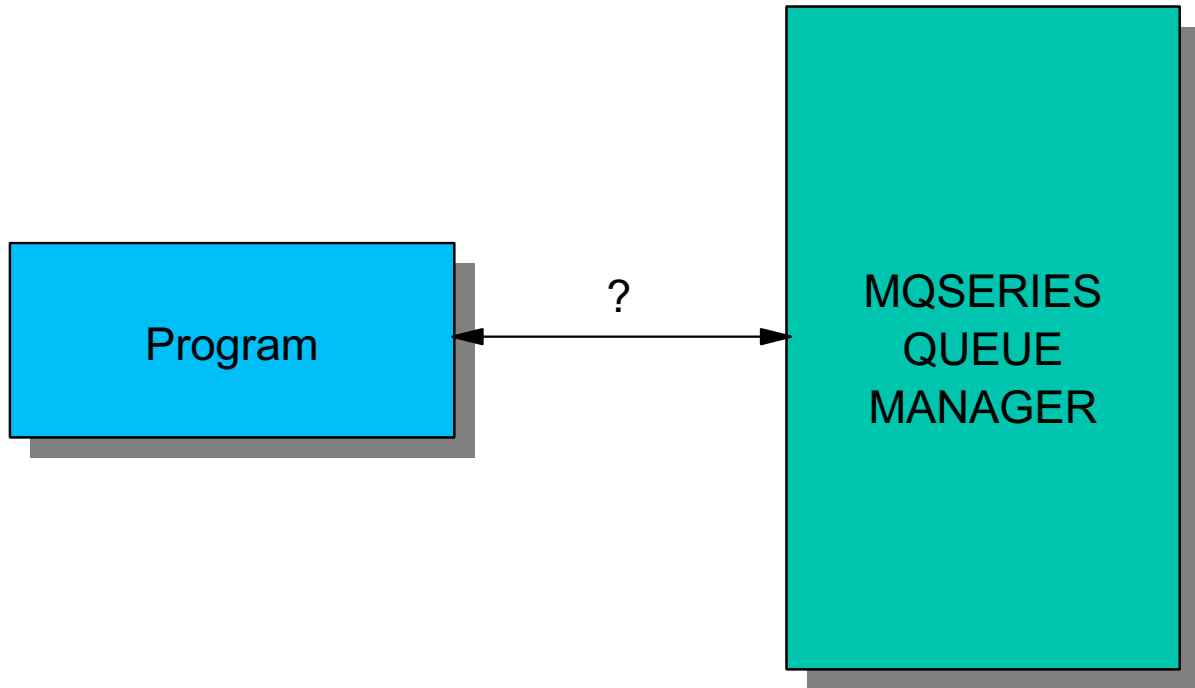


Figure 2-3. Connection Type

MQ092.0

Notes:

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

There are two connection modes for connecting to a queue manager:

1. Client connections (as an MQSeries client using TCP/IP)
2. Bindings mode (connecting directly to MQSeries).

Connection Type: Client Connections

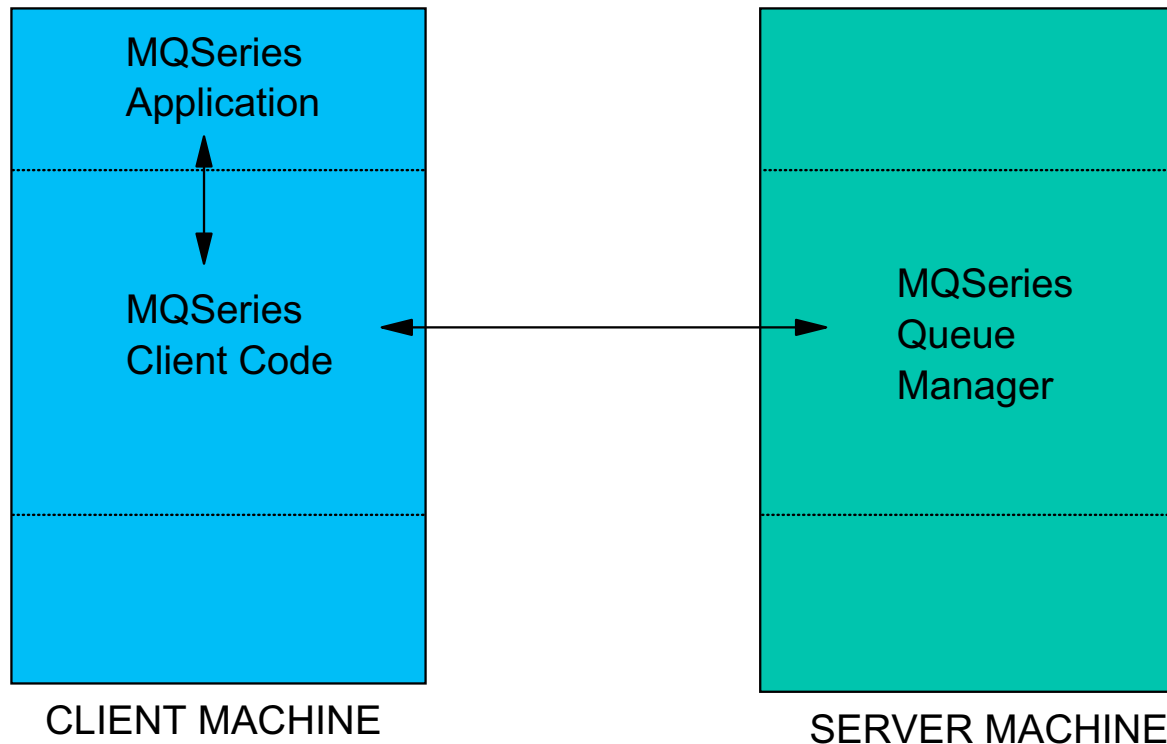


Figure 2-4. Connection Type: Client Connection

MQ092.0

Notes:

An application that runs in the MQSeries client environment must first be linked with the relevant client library. When the application issues an MQI (Message Queue Interface) call, the MQSeries client code directs the request to a queue manager, where it is processed and from where a reply is sent back to the MQSeries client. The link between the application and the MQSeries client code is established dynamically at runtime.

The MQI is available to applications running on the client platform; the queues and other MQSeries objects are held on a queue manager that is installed on a server machine.

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

Connection Type: Bindings Mode

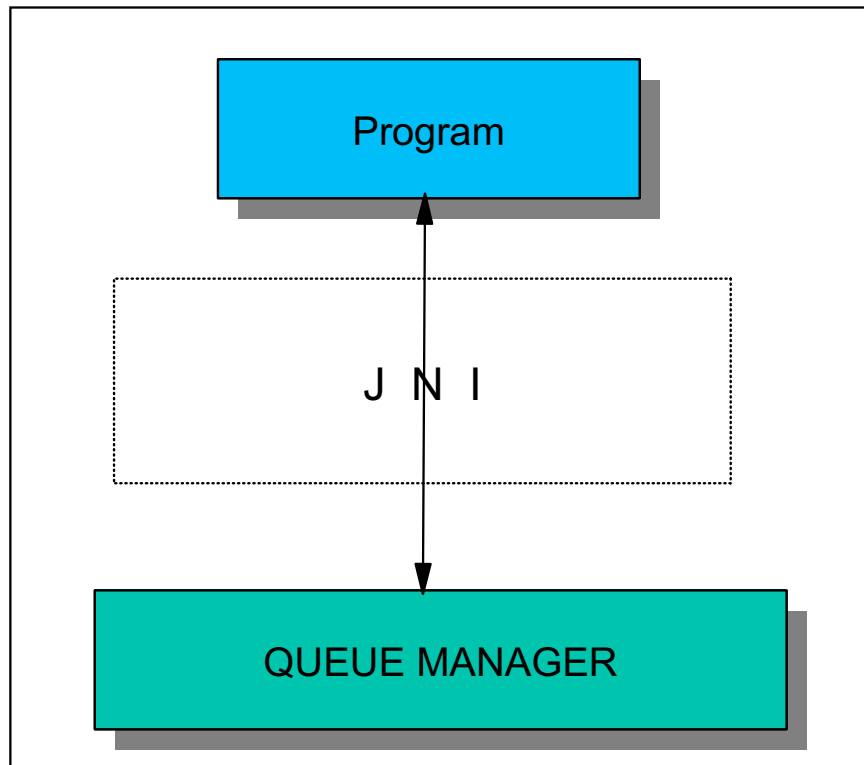


Figure 2-5. Connection Type: Bindings Mode

MQ092.0

Notes:

This mode is also known as the "native connection" mode.

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

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

In bindings mode, most of the parameters provided by the MQEnvironment class are ignored. This class shall be further exposed in the following pages.

What's the Advantage?

- **Java Client**

- End-users don't need to run MQSeries code
- Downloadable applets
- Easy maintenance and high reusability
- Allows everyone to participate fully in transactions

- **Java Bindings**

- High productivity and performance development option
- Applications communicate directly with the queue manager

Figure 2-6. What's the Advantage?

MQ092.0

Notes:

The most important advantage to using the MQSeries for Java Client is that the end-user does not need to run any MQSeries code. It opens the MQSeries network to all participants. The MQSeries for Java Client allows Internet applications easy access to MQSeries networks, via downloadable applets.

The most important advantage to using the MQSeries for Java Bindings is that they enable MQSeries server-side applications to be written using the Java programming language.

Java Basics: Application vs Applet

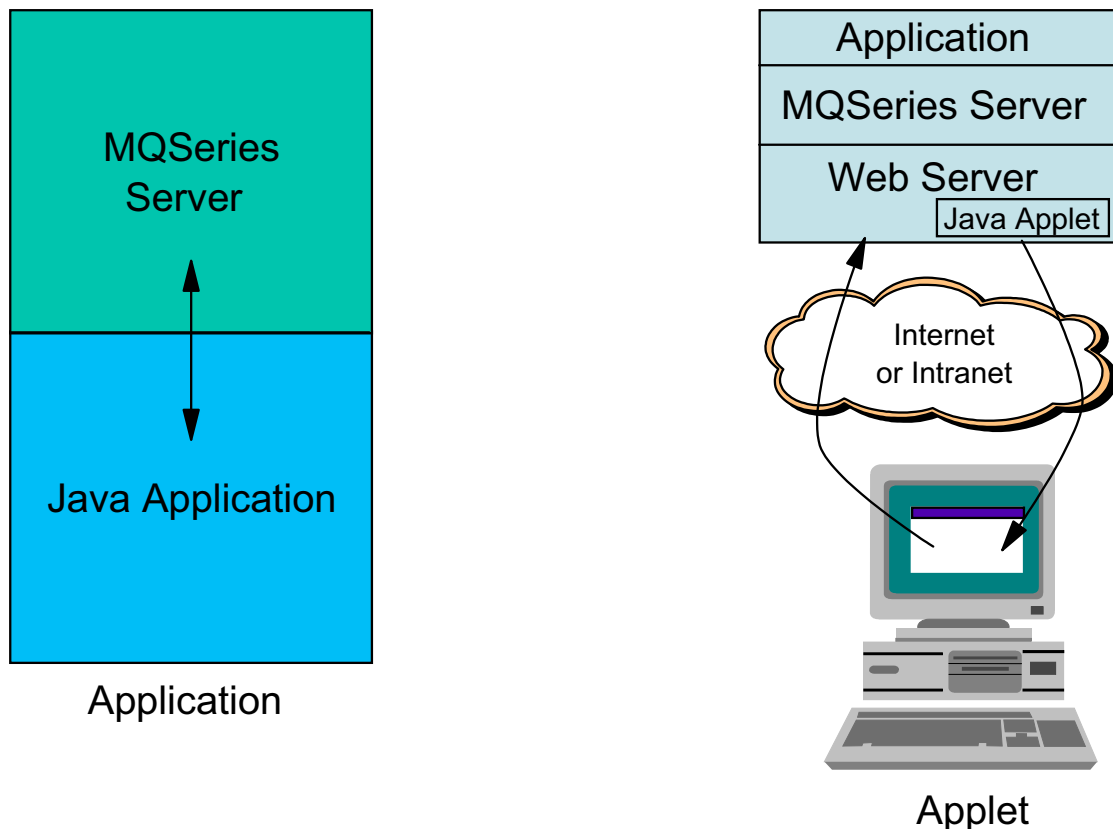


Figure 2-7. Java Basics: Application vs. Applet

MQ092.0

Notes:

You can create two different types of programs with Java: applications and applets.

Applets are restricted programs: they are intended to be delivered over the Internet. Applets are small, and do not have access to all Java functions. This restriction provides the security required to avoid intentional data corruption and viruses.

Applications do not have to be run in a browser or an applet viewer, which is why they are often used as servlets, running on the MQ Server.

The principal difference between the two is that an application requires a `main()` method to run, whereas an applet does not.

Defining Which Connection to Use

Class MQEnvironment

public static java.util.Hashtable properties

key:

- MQC.TRANSPORT_PROPERTY

values:

- MQC.TRANSPORT_MQSERIES_BINDINGS
- MQC.TRANSPORT_MQSERIES_CLIENT
- MQC.TRANSPORT_MQSERIES

Figure 2-8. Defining Which Connection to Use

MQ092.0

Notes:

The connection is determined by the key/value pairs contained in the properties variable of the MQEnvironment class.

For client connections, use:

```
MQEnvironment.properties.put (MQC.TRANSPORT_PROPERTY,  
                               MQC.TRANSPORT_MQSERIES_CLIENT)
```

For bindings connections, use:

```
MQEnvironment.properties.put (MQC.TRANSPORT_PROPERTY,  
                               MQC.TRANSPORT_MQSERIES_BINDINGS)
```

The default value for MQC.TRANSPORT_PROPERTY is MQC.TRANSPORT_MQSERIES. In this case, the connection type is selected according to the value of the "hostname" variable of the MQEnvironment class (if the hostname is not set, then bindings mode will be used).

Other Useful MQEnvironment Variables

MQEnvironment.hostname

MQEnvironment.port

MQEnvironment.channel

MQEnvironment.userID

MQEnvironment.password

Figure 2-9. Other Useful MQEnvironment Variables

MQ092.0

Notes:

For client connections, it is often necessary to set these variables before trying to connect to a queue manager.

hostname	public static String hostname The TCP/IP hostname of the machine on which the MQSeries server resides. If the hostname is not set, and no overriding properties are set, bindings mode is used to connect to the local queue manager.
port	public static int port The port to connect to. This is the port on which the MQSeries server is listening for incoming connection requests. The default value is 1414.
channel	public static String channel

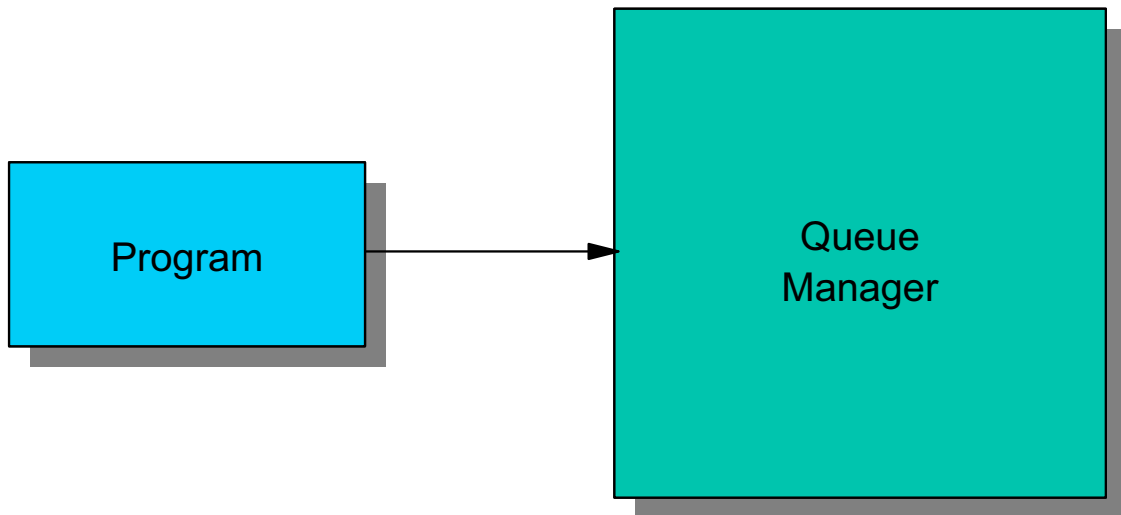
The name of the channel to connect to on the target queue manager. You must set this variable before connecting to a queue manager in client mode.

userID / password

public static String userID public static String password

User ID and password to connect to the security system on the server machine.

Connecting to a Queue Manager



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

Figure 2-10. Connecting to a Queue Manager

MQ092.0

Notes:

To connect to a queue manager, all you need to do is create a new instance of the `MQQueueManager` class:

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

If the queue manager name is left blank (null or ""), a connection is made to the default queue manager.

Completion and Reason Codes

- Completion Codes :
 - 0: MQCC_NONE
 - 1: MQCC_WARNING
 - 2: MQCC_FAILED
- Reason Codes
 - 0: MQRC_NONE
 - 2002: MQRC_ALREADY_CONNECTED
 - 2103: MQRC_ANOTHER_Q_MGR_CONNECTED
 - 2162: MQRC_Q_MGR_STOPPING
 - 2035: MQRC_NOT_AUTHORIZED
 - 2059: MQRC_Q_MGR_NOT_AVAILABLE

Figure 2-11. Completion and Reason Codes

MQ092.0

Notes:

All connection attempts will result in one of the three listed completion codes:

- 0** a successful call completed
- 1** warning - the call completed but reason code indicates certain conditions
- 2** failed - the call did not complete successfully

The programmer needs to handle exceptions to check of completion codes and reason codes.

The list of reason codes is an example of some of the most common that might occur when a connection is attempted.

Testing the Connection

Class MQQueueManager

- Method: public boolean isConnected()

Figure 2-12. Testing the Connection

MQ092.0

Notes:

You can use the `isConnected` method of the `MQQueueManager` class to test the connection to a queue manager:

```
public boolean isConnected()
```

Returns the value `True` if the connection to the queue manager is still open.

Disconnecting from a Queue Manager

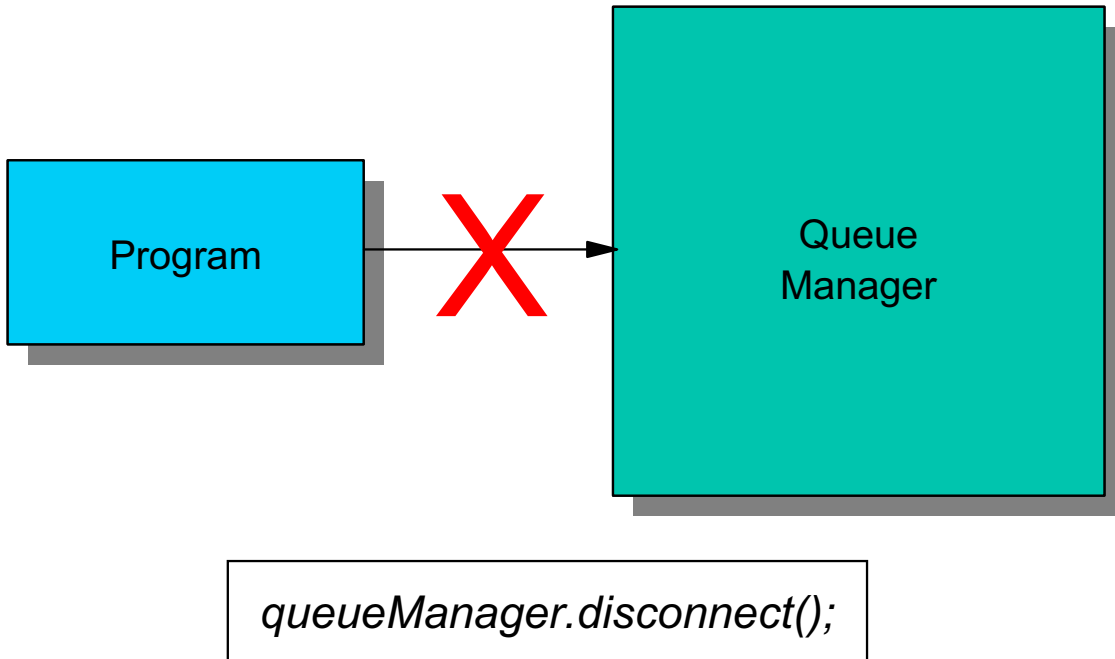


Figure 2-13. Disconnecting from a Queue Manager

MQ092.0

Notes:

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

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

(`queueManager` being the name of the `MQQueueManager` object)

Calling the `disconnect` method causes all open queues and processes that you have accessed through that queue manager to be closed. It is good programming practice, however to close these resources yourself when you have finished using them (you will learn how to do this in Unit 3).

When you have disconnected from a queue manager, the only way to reconnect is to create a new `MQQueueManager` object.

Example

```
private MQQueueManager qMgr; // define a queue Manager object

...

// Set up MQSeries environment
MQEnvironment.hostname = "my_hostname";
MQEnvironment.channel = "my_server_channel";
MQEnvironment.port = 1415;

...

qMgr = new MQQueueManager("my_queue_manager_name"); // Create connection

...

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

Figure 2-14. Example

MQ092.0

Notes:

The above example shows how to connect to a queue manager using the client connection mode.

2.2 Checkpoint and Summary

Unit Checkpoint

1. What is the name of the MQSeries class that is used to manage a queue manager?
2. Which connection type uses the Java Native Interface to connect to a queue manager?
 - a. Client Mode
 - b. Bindings Mode
3. If the "hostname" environment variable is not set, what connection mode will be used by default?
 - a. Client Mode
 - b. Bindings Mode
4. What is the correct syntax to connect to a queue manager?
 - a. `queueManager = new MQQueueManager("qMgrName");`
 - b. `queueManager.connect();`

Notes:

Summary

You should now know how to use the following MQSeries classes to connect to and disconnect from a specific queue manager :

- MQEnvironment
- MQQueueManager

You should also be acquainted with the different connection modes and understand the differences between an MQ applet and an MQ application

Notes:

Unit 3. Working with Queues

What This Unit is About

In this unit, you will learn how to work with queues, and how to use the different queue types available.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Open and close a queue
- Set open and close options
- Work with alias queues, model queues and dynamic queues

How You Will Check Your Progress

Accountability:

- Checkpoint
- Machine exercises

References

SC34-5456 *MQSeries Using Java*

GC33-1632 *MQSeries Clients*

SC33-1673 *MQSeries Application Programming Reference*

<http://www.ibm.com/software/ts/mqseries/messaging/>
WebSphere MQ

Objectives

- Opening and closing queues
- Setting open and close options
- Queue Types

Notes:

3.1 Working with Queues

Accessing Queues

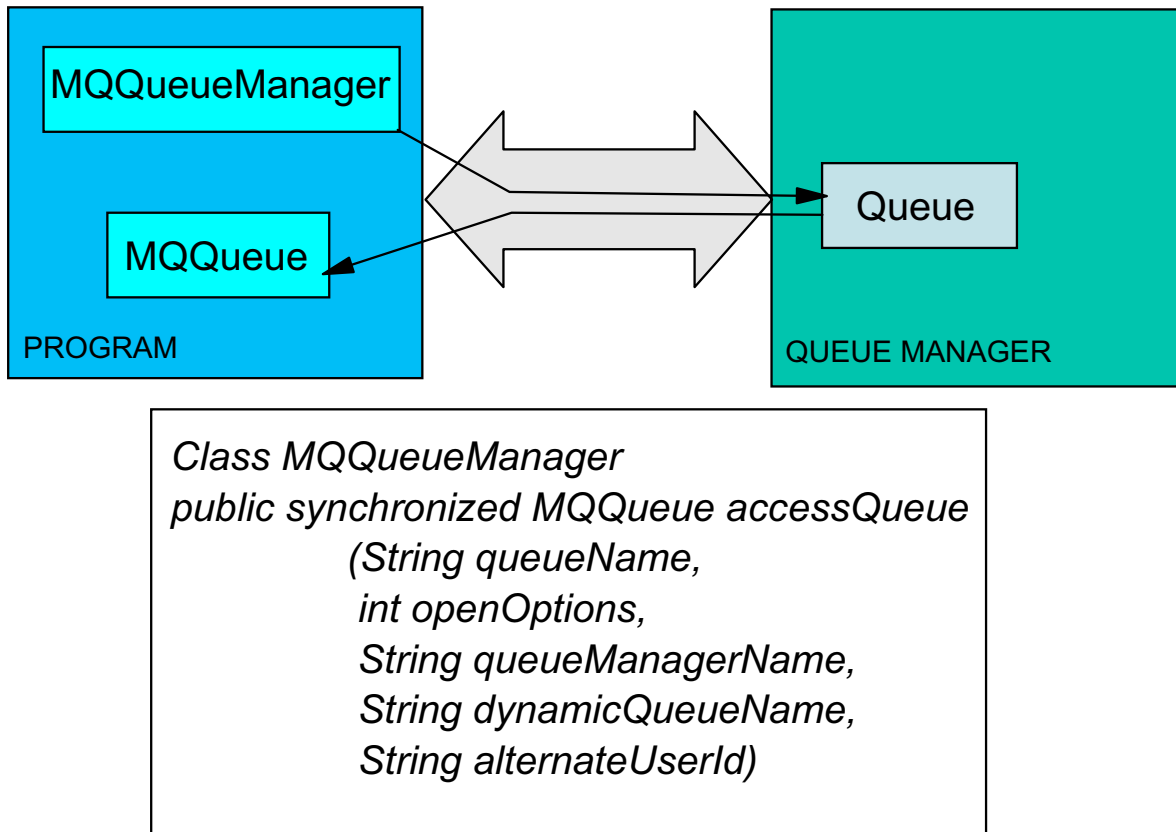


Figure 3-2. Accessing Queues

MQ092.0

Notes:

Queues are accessed using the `accessQueue` method of the `MQQueueManager` class. The `accessQueue` method returns a new object of class `MQQueue`. For example, to open a queue on a queue manager "queueManager", use the following code:

```

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

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

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

Open Options

- **Open Options** specify operations required
 - MQC.MQOO_INPUT_SHARED
 - MQC.MQOO_INPUT_EXCLUSIVE
 - MQC.MQOO_INPUT_AS_Q_DEF
 - MQC.MQOO_OUTPUT
 - MQC.MQOO_BROWSE
- Object can be opened for a **combination of options**
 - Options must be valid for object type
 - User must be authorized for all options requested
 - Specified options must not be contradictory

Figure 3-3. Open Options

MQ092.0

Notes:

Open options specify what a program wants to do with the object it is opening. Those listed are some of the most common but there are others. If more than one option is required the values can be added together or combined using the bitwise OR operator.

For instance, a common pair of options is MQC.MQOO_INPUT_SHARED and MQC.MQOO_BROWSE. Generally, a browse will be done to determine if there are any messages that meet some processing criteria. If so, the application will want to take the message off the queue for processing. However, combining MQC.MQOO_INPUT_EXCLUSIVE and MQC.MQOO_INPUT_SHARED is not valid.

Note that the program can defer to the queue definition for the open options to be used.

Reason Codes when Opening a Queue

0 : MQRC_NONE

2035 : MQRC_NOT_AUTHORIZED

2042 : MQRC_OBJECT_IN_USE

2045 : MQRC_OPTION_NOT_VALID_FOR_TYPE

2046 : MQRC_OPTIONS_ERROR

2195 : MQRC_UNEXPECTED_ERROR

Figure 3-4. Reason Codes when Opening a Queue

MQ092.0

Notes:

There are many reason codes that can be returned if an open call is unsuccessful. The Application Programming Reference contains a description of each along with recommended actions.

Closing a Queue

```
closeOptions = MQC.MQCO_NONE;  
queue.close();
```

Figure 3-5. Closing a Queue

MQ092.0

Notes:

When you have finished using the queue, close it using the `close()` method of `MQQueue` class, as in the above example.

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

Closing a queue will return a completion code and a reason code. These are documented in the Application Programming Reference.

You can also set close options before closing a queue using the `closeOptions` variable of the `MQQueue` object:

```
public int closeOptions
```

Set this attribute to control the way the resource is closed. The default value is `MQC.MQCO_NONE`, and this is the only permissible value for all resources other than permanent dynamic queues.

For these queues, the following additional values are permissible:

MQC.MQCO_DELETE

Delete the queue if there are no messages.

MQC.MQCO_DELETE_PURGE

Delete the queue, purging any messages on it.

Alias Queues

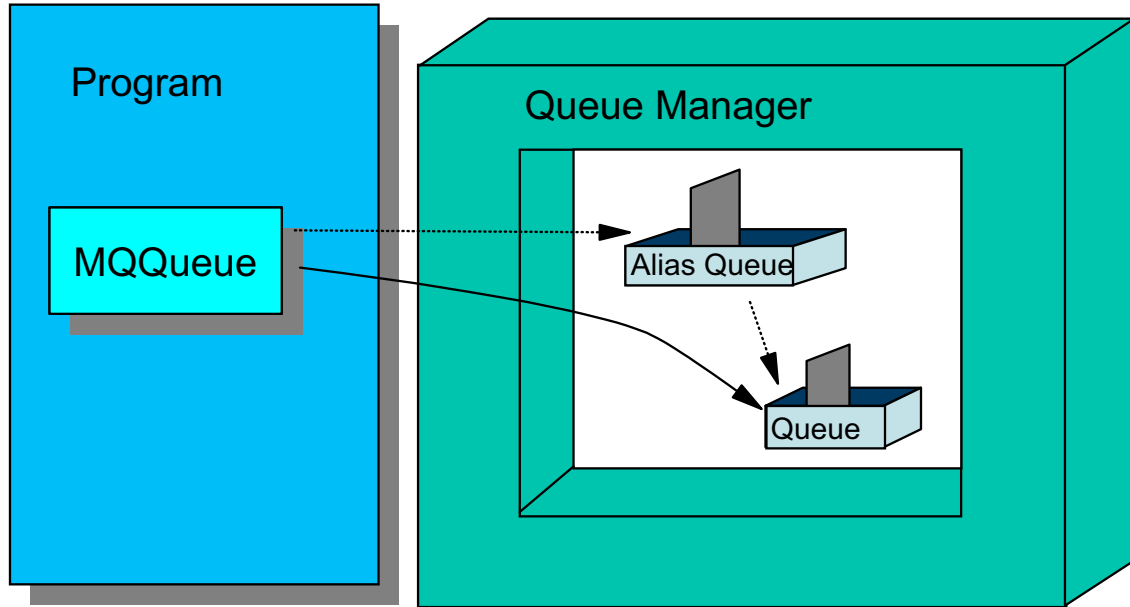


Figure 3-6. Alias Queues

MQ092.0

Notes:

An alias queue is simply a definition. It allows a local or remote definition to be referred to by another name. An alias queue can actually have some different properties from the underlying queue it is pointing to. For instance, a queue can allow Getting and Putting messages, while its alias only allows Getting.

It is important to note that the program thinks that the alias queue it is working with is a "real" queue, not simply a pointer to another queue.

The advantage of using aliases is that you can change queue names and properties without having to update programs, as long as the aliases still point to the correct base queue.

Model Queues

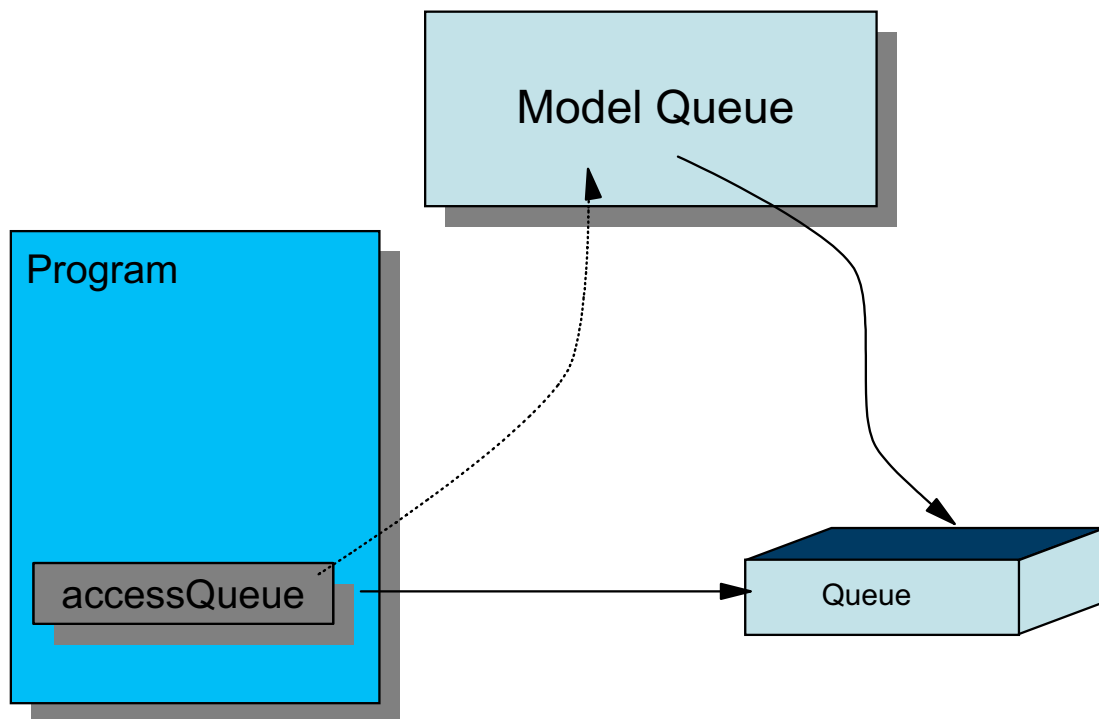


Figure 3-7. Model Queues

MQ092.0

Notes:

When an administrator defines a model queue, that definition is nothing more than a template. When used as the `queueName` in the `accessQueue` method, the result is that a queue is dynamically created that has all the attributes of the model. The model itself is never actually used for anything else.

Thus, a program is simply creating a queue that looks like the model. We will talk about names and types of dynamic queues in the next few pages.

Dynamic Queues

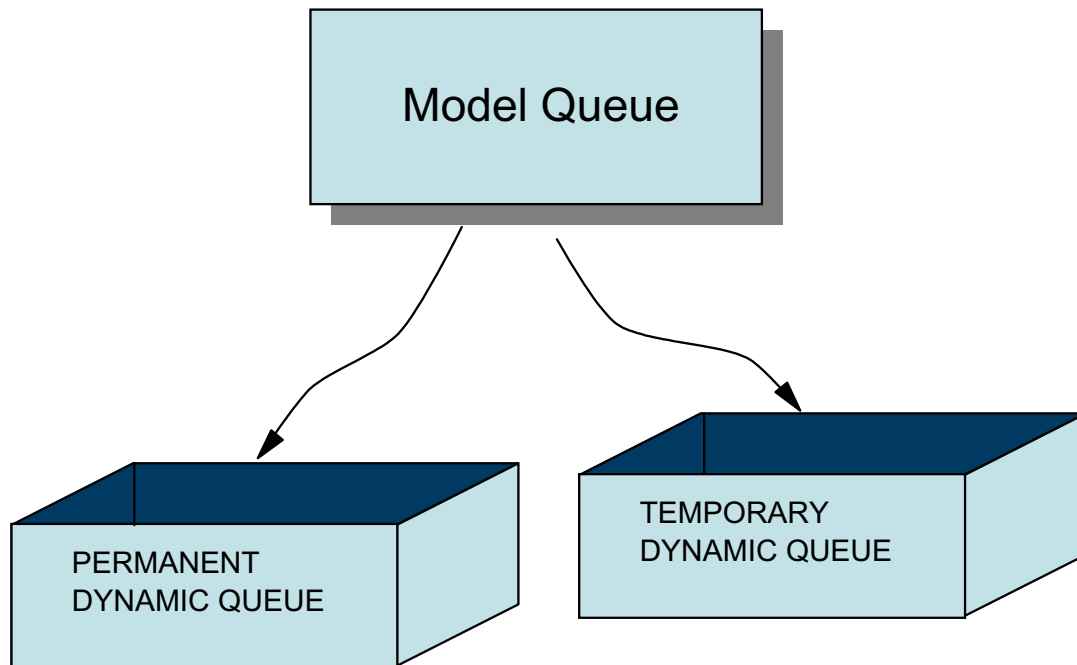


Figure 3-8. Dynamic Queues

MQ092.0

Notes:

When a model queue is defined by the administrator, one of the attributes will ultimately determine whether the dynamic queue your program creates will be long-lived or not.

The queue type can be either temporary or permanent:

A temporary dynamic queue will **ONLY** last until the execution of the program that created it ends (normally or abnormally) or until that creating execution closes it. There is no way to keep the temporary dynamic queue beyond that point.

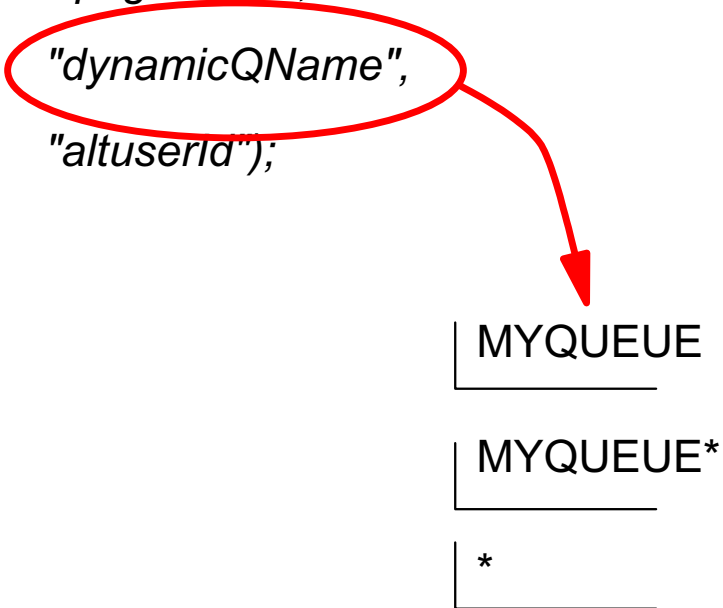
A permanent dynamic queue is created in exactly the same way but does **NOT** automatically disappear. It must be explicitly deleted (using a close option of delete or by the administrator with a delete command) or it simply becomes another local queue. Once created, there is nothing that MQSeries does to keep track of the dynamically created permanent dynamic queues that would allow them to be identified as different from any other local queues.

Dynamic Queue Names

```

qMgr.accessQueue ("qName",
                  openOptions,
                  "qMgrName",
                  "dynamicQName",
                  "altuserid");

```



MYQUEUE
 MYQUEUE*
 *

Figure 3-9. Dynamic Queue Names

MQ092.0

Notes:

Remember the `accessQueue` method. One of the fields that we saw was the dynamic queue name. It has several potential values:

XXX* - where XXX is some prefix that the programmer wants at the beginning of a unique name that will be generated by the queue manager

* alone will mean that there is no prefix that might be used. The name will be unique but might not be one that will be of much value.

"XXXXX" where the entire name is provided (no trailing *). This means that the queue manager will play no part in generating a unique name. The name, as provided, will be used as the queue name.

3.2 Checkpoint and Summary

Unit Checkpoint

1. What is the correct syntax to open a queue?
 - a. `queue.open();`
 - b. `MQQueue queue = queueManager.accessQueue(...);`
2. You can combine several open options. True/False?
3. Several alias queues can represent one single local queue. True/False?
4. You can put messages onto a model queue. True/False?
5. Which of the following names for a dynamic queue will generate a queue with a unique name?
 - a. `ABC.*`
 - b. `ABC.#`
 - c. `ABC.%`
 - d. `*`

Notes:

Summary

- You should now know how to open and close a queue
- You should also understand which options you need to set before opening and closing a queue
- Finally, you should know which types of queue to use according to your requirements (local queues, alias queues or dynamic queues through a model queue)

Notes:

Unit 4. Error Handling

What This Unit is About

In this unit, you will discover the MQException class, and its variables. You will learn how to use this class in order to trap MQSeries exceptions, and react accordingly.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Handle MQSeries exceptions in Java
- Retrieve completion codes and reason codes
- Use the MQException class

How You Will Check Your Progress

Accountability:

- Checkpoint

References

SC34-5456

MQSeries Using Java

SC33-1673

MQSeries Application Programming Reference

Objectives

- Error Handling in Java
- MQException Class
- Completion Codes and Reason Codes

Figure 4-1. Unit Objectives

MQ092.0

Notes:

4.1 Error Handling

MQException Class

```
public class MQException  
  
extends Exception
```

Variables:

- log
- completionCode
- reasonCode
- exceptionSource

Figure 4-2. MQException Class

MQ092.0

Notes:

Methods in the Java interface do not return a completion code and a reason code. Instead, they throw an exception whenever the completion code and reason code resulting from an MQSeries call are not both zero. This simplifies the program logic so that you do not have to check the return codes after each call to MQSeries.

Whenever an MQSeries error occurs, an MQException is thrown. This class contains definitions of completion code and reason code constants. Constants beginning with MQCC_ are MQSeries completion codes and constants beginning with MQRC_ are MQSeries reason codes. The MQSeries Application Programming Reference contains a full description of these errors and their probable causes.

MQException.log

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

default value:

- System.err

no logging:

- null

Figure 4-3. MQException.log

MQ092.0

Notes:

This variable of the MQException class indicates the stream to which exceptions are logged. (The default is System.err) If you set this to null, no logging occurs.

MQException.completionCode and MQException.reasonCode

```
public int completionCode
```

```
MQException.MQCC_NONE → 0
```

```
MQException.MQCC_WARNING → 1
```

```
MQException.MQCC_FAILED → 2
```

```
public int reasonCode
```

```
MQException.MQRC_NONE → 0
```

```
MQException.MQRC_XXXXXX → 2000 - 2xxx
```



See Application Programming Reference

Figure 4-4. MQException.completionCode and MQException.reasonCode

MQ092.0

Notes:

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

0 (MQCC_NONE) - a successful call completed

1 (MQCC_WARNING) - the call completed but reason code indicates certain conditions

2 (MQCC_FAILED) - the call did not complete successfully.

reasonCode returns the MQSeries reason code describing the error. The possible values are:

0 (MQRC_NONE) - a successful call completed

2000 to 2xxx - reason code returned for a completion code of 1 or 2.

For a full explanation of the reason codes refer to the MQSeries Application Programming Reference.

MQException.exceptionSource

```
public Object exceptionSource
```

Figure 4-5. MQException.exceptionSource

MQ092.0

Notes:

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

Try / Catch Blocks

```
try {  
    // MQ Calls are inserted here  
}  
catch (MQException ex) {  
    System.out.println ("An error has occurred: Completion code" +  
        ex.completionCode + ", Reason code " + ex.reasonCode);  
}
```

Figure 4-6. Try / Catch Blocks

MQ092.0

Notes:

You can decide at which point in your program you want to deal with the possibility of failure by surrounding your code with 'try' and 'catch' blocks, as in the example above.

The 'catch' block is only executed if one of the instructions within the try block gives rise to a non-zero completion code or reason code.

4.2 Checkpoint and Summary

Unit Checkpoint

1. What is the name of the variable that identifies the reason for the MQException?

2. What is the name of the variable that identifies the object that has thrown the MQException?

Notes:

Summary

By now, you should understand how to :

- Trap MQSeries exceptions
- Retrieve completion codes and reason codes
- Handle the error

Notes:

Unit 5. Messaging and Queuing

What This Unit is About

In this unit, you shall learn about the MQMessage object, and how to put and get messages to and from a queue. This unit also covers message formats, put and get options, and gives you an overlook on setting and getting message attributes.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Construct an MQMessage object
- Put messages onto a queue, in different formats, with various options
- Get messages from a queue, in different formats, with various options
- Retrieve and set message attributes

How You Will Check Your Progress

Accountability:

- Checkpoint
- Machine exercises

References

SC34-5456 *MQSeries Using Java*

SC33-1673 *MQSeries Application Programming Reference*

<http://www.ibm.com/software/ts/mqseries/messaging/>
WebSphere MQ

Objectives

- Construct a message object
- Put messages onto queues
- Get messages from queues
- Retrieve and set message attributes

Notes:

5.1 The Message Object

Message = Header + Application Data

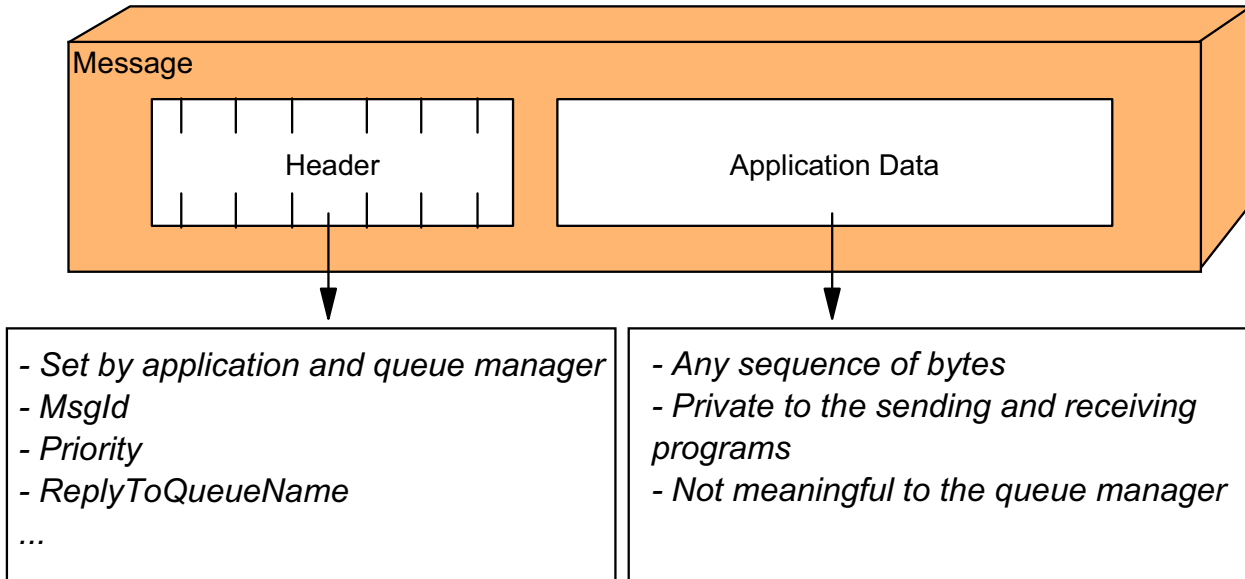


Figure 5-2. Message = Header + Application Data

MQ092.0

Notes:

A message has two parts: the application data and a header called the message descriptor. The message descriptor contains information about the message which is used by both MQSeries and the receiving application. Some of the fields it contains are:

- The type of message
- An identifier for the message
- The priority of the message (Priority)
- The identifier of the coded character set of the character data within the application data
- The name of the queue to which a reply should be sent...

There is no limitation on the contents of the application data, but there is a maximum allowable length for the message whose value is platform dependent.

Constructing a Message

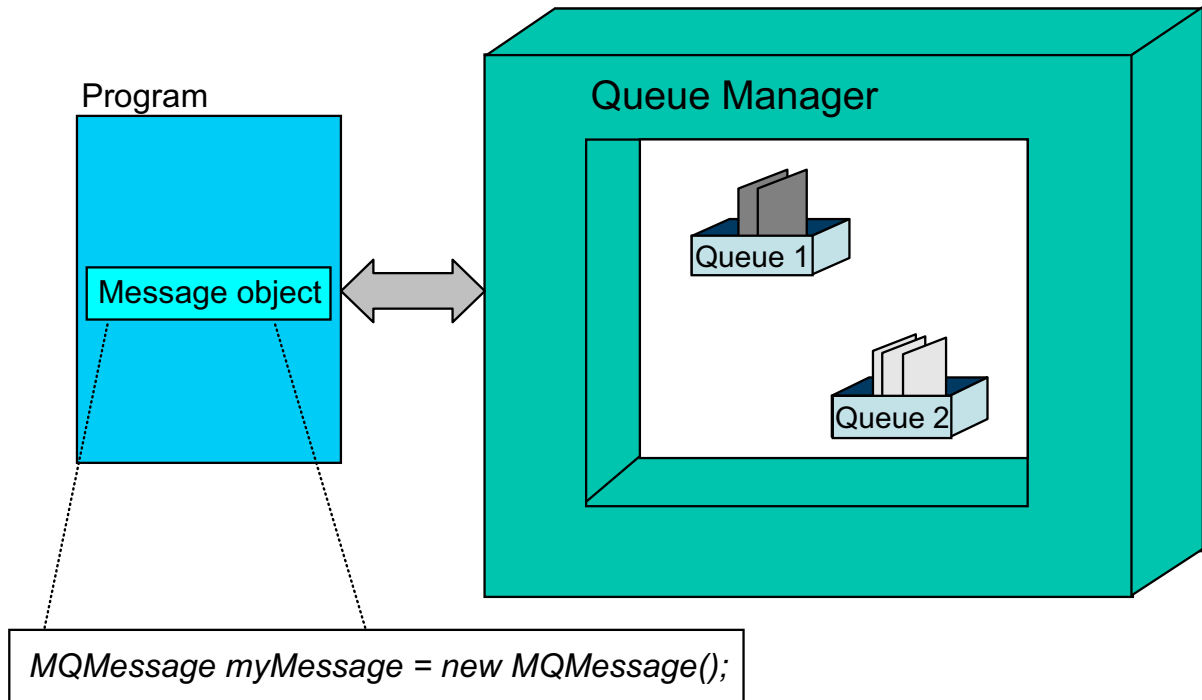


Figure 5-3. Constructing a Message

MQ092.0

Notes:

The first step in constructing a message is to create a new message object. This is done by instantiating the `MQMessage` class giving the new object a name. In this example, we have named the new object `myMessage`.

As an alternative to creating a new object an existing message object can be reused, but first it must be reset. This is done with the `clearMessage` method: the `clearMessage` method will reset the `MQMessage` variables to their default settings, clear the message buffer and reset the buffer cursor to the start of the buffer.

The MQMessage Object

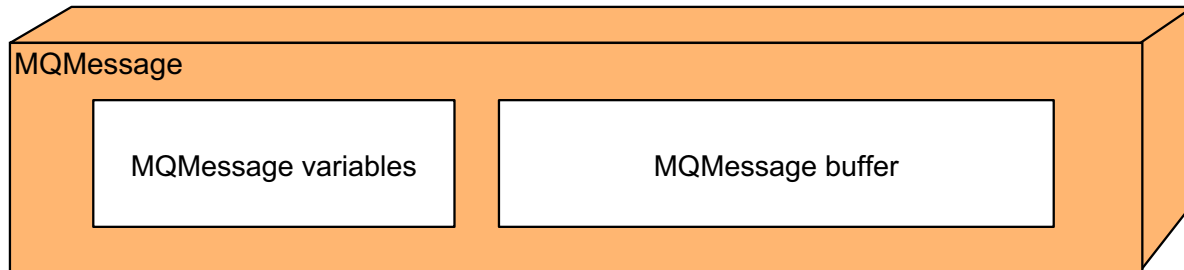


Figure 5-4. The MQMessage Object

MQ092.0

Notes:

The MQMessage object represents the message descriptor as variables and the user message data as the buffer.

There is a group of readXXX methods for reading data from a message, and a group of writeXXX methods for writing data into a message. The format of numbers and strings used by these read and write methods can be controlled by the encoding and characterSet member variables.

The remaining member variables contain control information that accompanies the application message data when a message travels between sending and receiving applications. The application can set values into the member variable before putting a message to a queue and can read values after retrieving a message from a queue.

User Data Formats

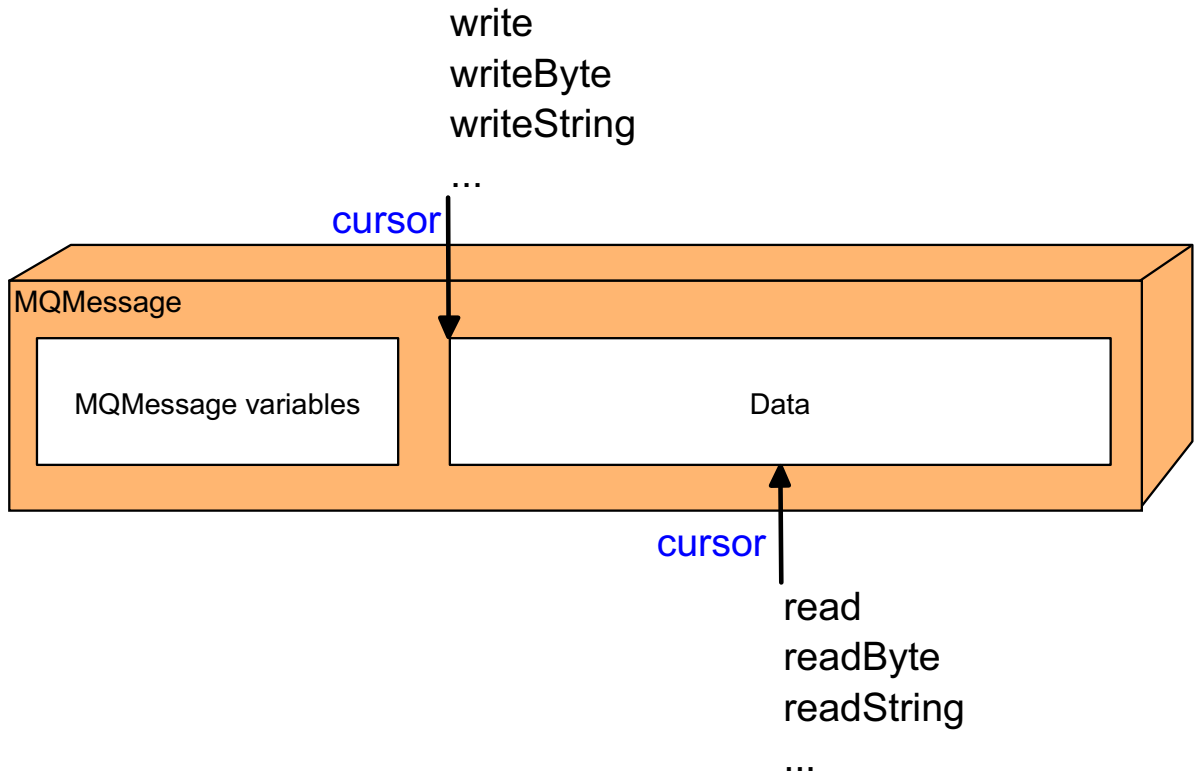


Figure 5-5. User Data Formats

MQ092.0

Notes:

The next step is to move the required user data into the message buffer of the MQMessage object. The data can be written using the writeXXX methods.

The message buffer maintains a cursor to indicate the current location for the write methods to place the data.

write and writeString

```
int oneByte = "R";
myMessage.write(oneByte);

byte[] strData = new byte[24];
strData = "Melbourne.Australia";
myMessage.write(strData);

myMessage.write(strData, 0,9);
myMessage.write(strData, 10,9);

string strData = "hello world!"
myMessage.writeString(strData);
```

Figure 5-6. write and writeString

MQ092.0

Notes:

The write method is overloaded to allow for either a single parameter or three parameters. If a single parameter is specified then the data is either a single integer or a string array:

```
public void write(int b)
```

Throws IOException.

Write a byte into the message buffer at the current position

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

Throws IOException.

Write an array of bytes into the message buffer at the current position.

When three parameters are specified, the first is the data, the second is the offset for the data and the third parameter is the length of the data:

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

Throws IOException

Write a series of bytes into the message buffer at the current position. *len* bytes will be written, taken from offset *off* in the array *b*.

public void writeString(String str)

Throws IOException

The writeString method will write the specified string into the message buffer at the current position, converting it to the code set identified by the characterSet member variable.

writeChar, writeChars and writeUTF

```
int confirm = 0;  
myMessage.writeChar(confirm);
```

```
String confirm = "yes";  
myMessage.writeChars(confirm);
```

```
String name = "Bill";  
myMessage.writeUTF(name);
```

Figure 5-7. writeChar, writeChars and writeUTF

MQ092.0

Notes:

public void writeChar(int v)

Throws IOException

The writeChar method will write a Unicode character into the message buffer at the current position.

public void writeChars(String s)

Throws IOException

The writeChars method will write a string as a sequence of Unicode characters into the message buffer at the current position.

public void writeUTF(String str)

Throws IOException

The writeUTF method will write a UTF (Universal Text Format) string, prefixed by a 2-byte length field, into the message buffer at the current position.

The value of the buffer cursor is taken as the current starting position for the data transfer, with the method updating the cursor on completion of the data move.

Numeric Data Formats

```
int v = 5;
myMessage.writeInt(v);

short years = 24;
myMessage.writeDecimal2(years);

int secondsToDay = 86400;
myMessage.writeDecimal4(secondsToDay);

long secondsToDate = 777182400;
myMessage.writeDecimal8(secondsToDate);
```

Figure 5-8. Numeric Data Formats

MQ092.0

Notes:

public void writeInt(int v)

Throws IOException

The writeInt method will write an integer into the message buffer at the current position.

public void writeDecimal2(short v)

Throws IOException

The writeDecimal2 method will write a 2-byte packed decimal format number in the range -999 to +999 into the message buffer at the current position.

public void writeDecimal4(int v)

Throws IOException

The writeDecimal4 method will write a 4-byte packed decimal format number in the range -9 999 999 to +9 999 999 into the message buffer at the current position.

public void writeDecimal8(long v)

Throws IOException

The writeDecimal8 method will write a 8-byte packed decimal format number in the range -999 999 999 to +999 999 999 into the message buffer at the current position.

You can also use the `writeDouble` and `writeFloat` methods to write floating point numbers.

Other Data Formats

```
int days = 8766;  
myMessage.writeByte(days);
```

```
String days = "days";  
myMessage.writeBytes(days);
```

```
boolean agree = false;  
myMessage.writeBoolean(agree);
```

```
AnyObject myObject = new AnyObject();  
myMessage.writeObject(myObject);
```

Figure 5-9. Other Data Formats

MQ092.0

Notes:

public void writeByte(int v)

Throws IOException

The writeByte method will write the specified byte into the message buffer at the current position.

public void writeBytes(string v)

Throws IOException

The writeBytes method will write the specified string to the message buffer as a sequence of bytes. Each character in the string is written out in sequence by discarding its high eight bits.

public void writeBoolean(boolean v)

Throws IOException

The writeBoolean method will write the specified boolean into the message buffer at the specified position.

public void writeObject(Object obj)

Throws IOException

The writeObject method will write the specified object to the message buffer at the specified position. The class of the object, the signature of the class, and the values of the non-transient and non static fields of the class and all its supertypes are all written.

Changing the Buffer Location

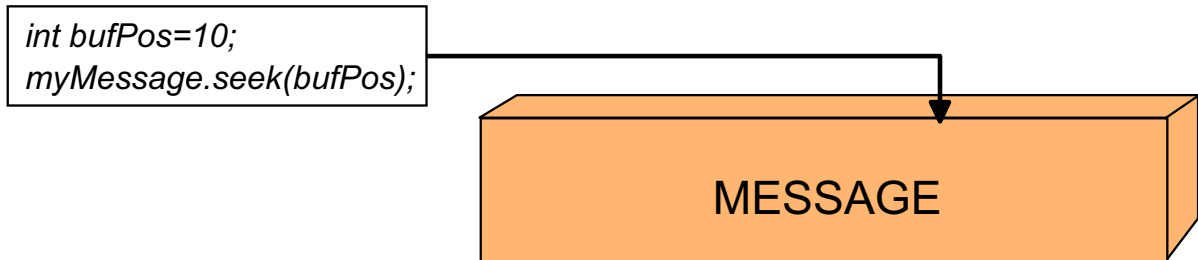


Figure 5-10. Changing the Buffer Location

MQ092.0

Notes:

The buffer location for the data being stored can be altered by the use of the `seek` or `setDataOffset` methods:

The `seek` method will move the cursor to the absolute position in the message buffer given by the integer parameter. The `setDataOffset` method is a synonym for `seek()`, and is provided for cross-language compatibility with the other MQSeries APIs.

Subsequent read and write methods will work from this new position within the message buffer. The `seek` and `setDataOffset` methods throw `IOException` if the `MQMessage` object is not open. The `seek` and `setDataOffset` methods also throw `EOFException` if the specified position is outside the message data length.

5.2 Putting a Message

Message Descriptor Properties

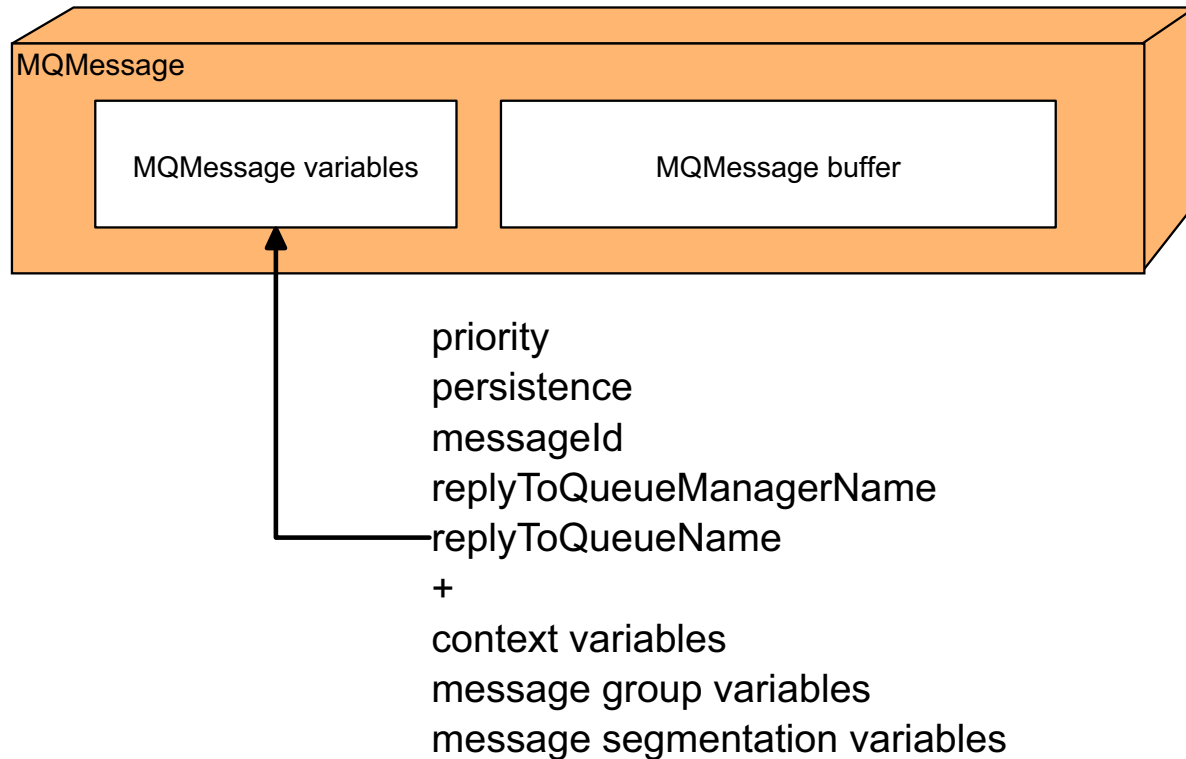


Figure 5-11. Message Descriptor Properties

MQ092.0

Notes:

The put method will take an instance of the MQMessage object as the first parameter.

The variables of this object are also known as the message descriptor properties. They may be altered as a result of this method.

The values they have immediately after the completion of this method are the values that were put onto the MQSeries queue.

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

Performing a put may update the messageld and correlationId.

On successful completion of the put request the named variables of the MQMessage object have been updated.

Priority

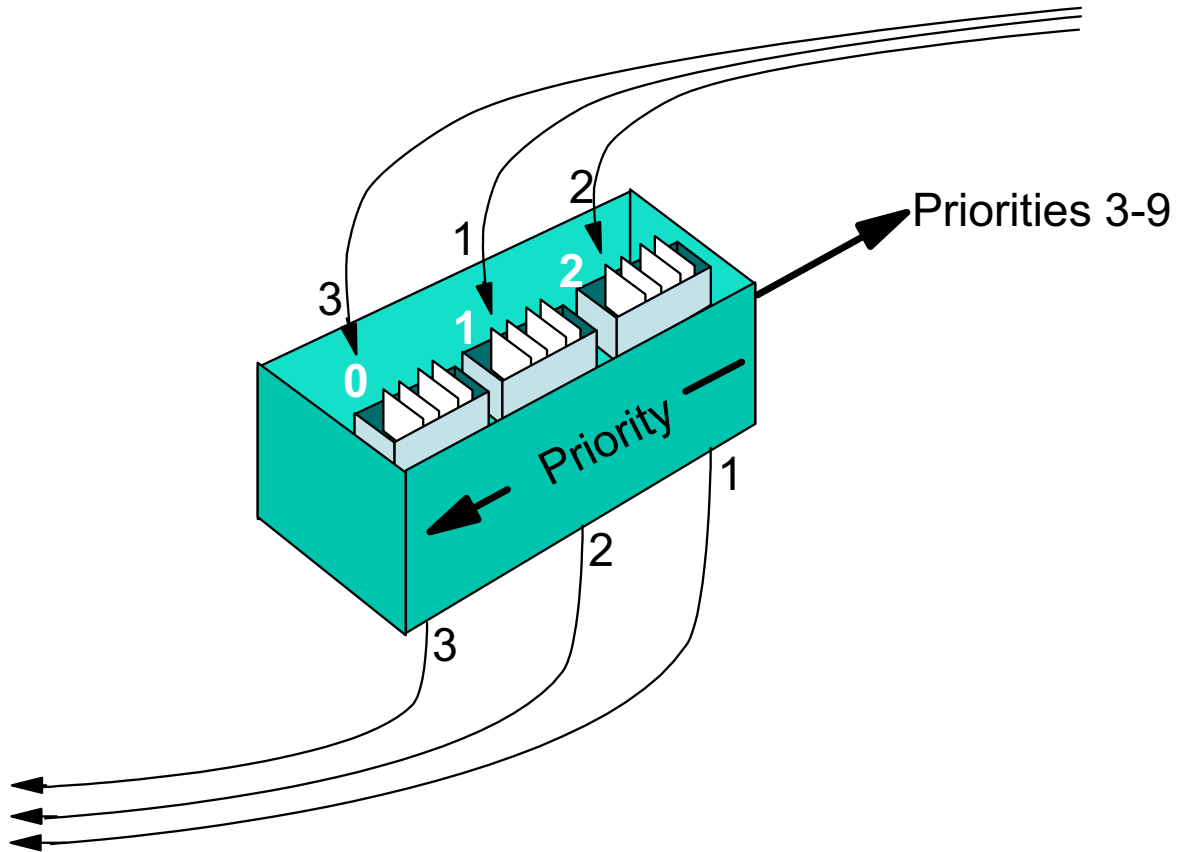


Figure 5-12. Priority

MQ092.0

Notes:

It is possible to have messages delivered in a different order than that in which they were put on a queue. One of the easiest ways to do this is using delivery by priority.

The valid range of values for this variable is zero to nine, with nine being the highest and zero being the lowest.

If the priority variable is set to the special value of `MQC.MQPRI_PRIORITY_AS_Q_DEF` then the message will inherit the default message priority value from the queue definition.

Persistence

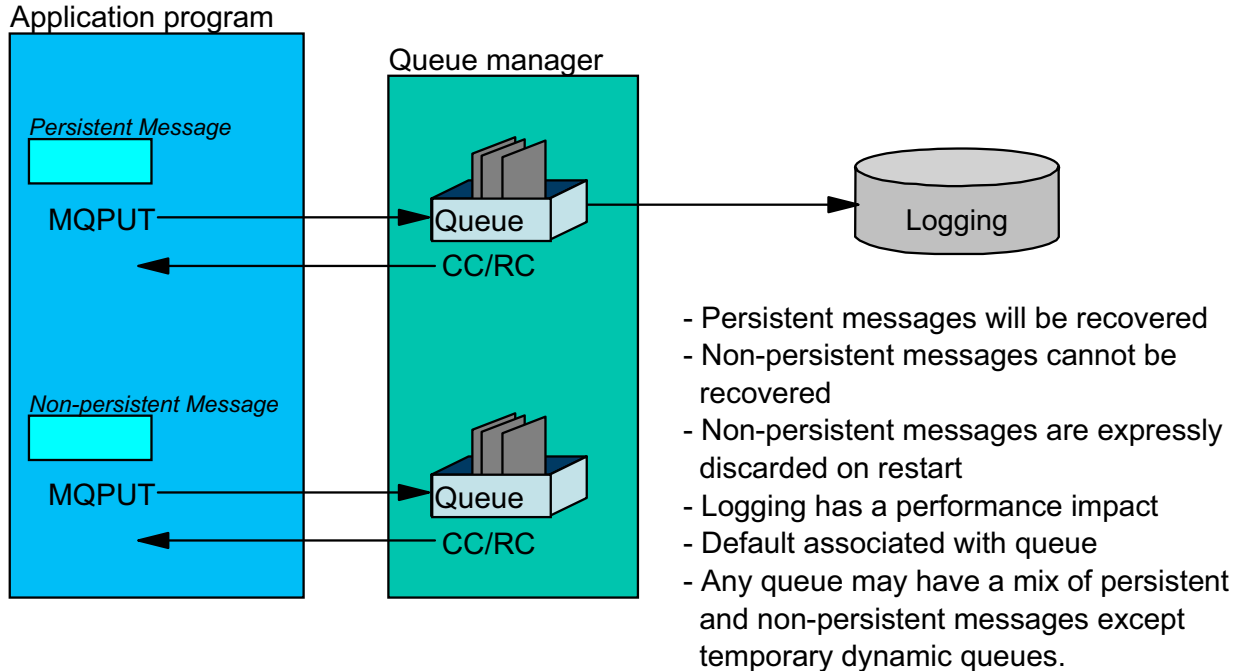


Figure 5-13. Persistence

MQ092.0

Notes:

When messages are put on a queue with persistence, at restart time, the queue manager will recover the messages by replaying its logs. At the same time, ALL messages that were put as non-persistent will be explicitly deleted at restart.

The possible returned value for this variable is either `MQC.MQPER_PERSISTENT` or `MQC.MQPER_NOT_PERSISTENT`.

If at the time of the put request the persistence variable is set to `MQC.MQPER_PERSISTENCE_AS_Q_DEF` the queue manager will update the value to reflect the persistence attribute of the queue definition, and return the new value on completion of the put request. When the `MQMessage` object is instantiated the persistence variable inherits the default value of `MQC.MQPER_PERSISTENCE_AS_Q_DEF`.

messageld

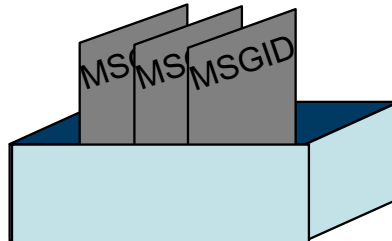


Figure 5-14. messageld

MQ092.0

Notes:

The messageld variable can later be used to retrieve a specific message from a queue. If the messageld variable is set to MQC.MQMI_NONE when the put is issued, the queue manager will generate a unique message identifier.

Put Message Options

```
MQPutMessageOptions myPMO = new MQPutMessageoptions();
```

```
myPMO.options  
myPMO.contextReference  
myPMO.recordFields  
myPMO.resolvedQueueName  
myPMO.resolvedQueueManagerName  
myPMO.knownDestCount  
myPMO.unknownDestCount  
myPMO.invalidDestCount
```

Figure 5-15. Put Message Options

MQ092.0

Notes:

The Put Message Options (MQPMO) control how a message is put on a queue: a PMO object contains variables that influence the operation of the put request, and is updated by the put request.

PMO: options

```
int myOutOptions = MQC.MQPMO_ NONE;  
myPMO.options(myOutOptions);
```

```
int myOtherOptions = MQC.MQPMO_FAIL_IF_QUIESCING |  
MQC.MQPMO_NEW_MSG_ID;  
myPMO.options(myOtherOptions);
```

Figure 5-16. PMO: options

MQ092.0

Notes:

The settings of the options variable controls the selectable functions performed by the queue manager on receipt of the put request.

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

The purpose of the options will be detailed in subsequent modules of this course.

Putting Messages

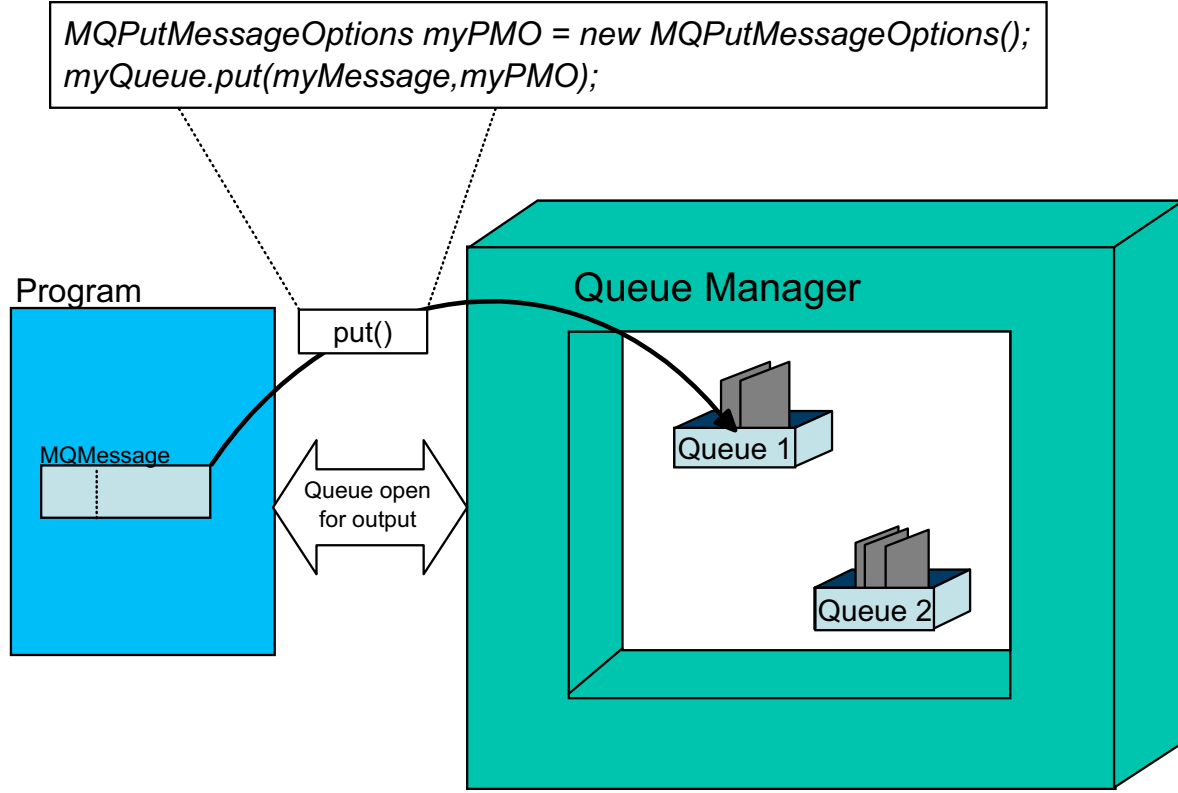


Figure 5-17. Putting Messages

MQ092.0

Notes:

To put a message on a queue, the queue must have been opened with the output option.

The parameters to the put method are the message object and the put message options object. The put method will use these objects to construct the underlying MQI put call.

The variables of the MQMessage object will be used to construct the message descriptor and will be updated on successful completion of the put request.

PMO: resolvedQueueName / resolvedQueueManagerName

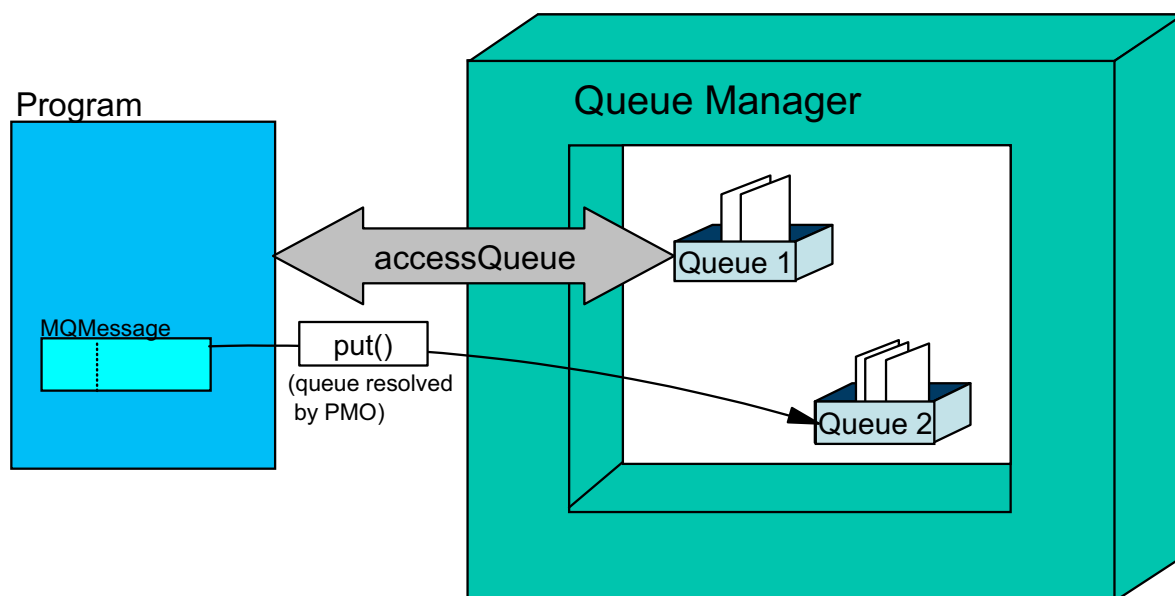


Figure 5-18. PMO: resolvedQueueName / resolvedQueueManagerName

MQ092.0

Notes:

The resolvedQueueName variable is an output field that is set by the queue manager to the name of the queue on which the message is placed.

This may be different from the name used on the accessQueue request if the named queue was an alias or model queue.

For example:

```
MQQueue myOutputQueue = qMgr.accessQueue("ABC");
System.out.println("the queue we opened to put our messages on is: " +
    myOutputQueue.name());
myOutputQueue.put(myMessage, myPMO);
if (myPMO.resolvedQueueName != "ABC") {
    System.out.println("the queue we have put our message to is named: " +
        resolvedQueueName); }
```

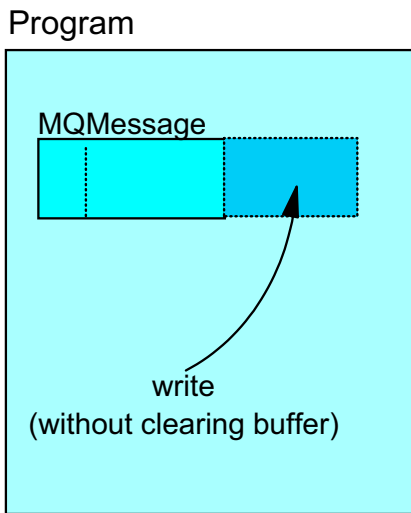
The resolvedQueueManagerName variable is an output field set by the queue manager to the name of the queue manager that owns the queue specified by the remote queue name.

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

For example:

```
int oOpts = MQC.MQOO_OUTPUT;
String yRR = "ATLANTA";
MQQueue myOutputQueue = qMgr.accessQueue("ABC", oOpts, yRR, null, null);
myOutputQueue.put(myMessage, myPMO);
if (myPMO.resolvedQueueManagerName != yRR) {
    System.out.println("the queue manager that owns the queue our message is
being put on is named: "
+ resolvedQueueManagerName); }
```

Put Message Buffer Considerations



```
myMessage.clearMessage();
```

Figure 5-19. Put Message Buffer Considerations

MQ092.0

Notes:

The put method does not clear the message buffer, therefore if the following sequence was performed, the second message may not be as expected.

```
myMessage.writeString("Atlanta");
myOutputQueue.put(myMessage,myPMO);
myMessage.writeString("Melbourne");
myOutputQueue.put(myMessage,myPMO);
```

The first message contains "Atlanta" and the second "AtlantaMelbourne".

If the MQMessage buffer is to be reused with different message variables, then the clearMessage method should be used after each put request.

If the successive messages are to be put with the same message variables then use the seek method to reposition the buffer cursor to the beginning of the buffer.

5.3 Getting a Message

Getting Messages

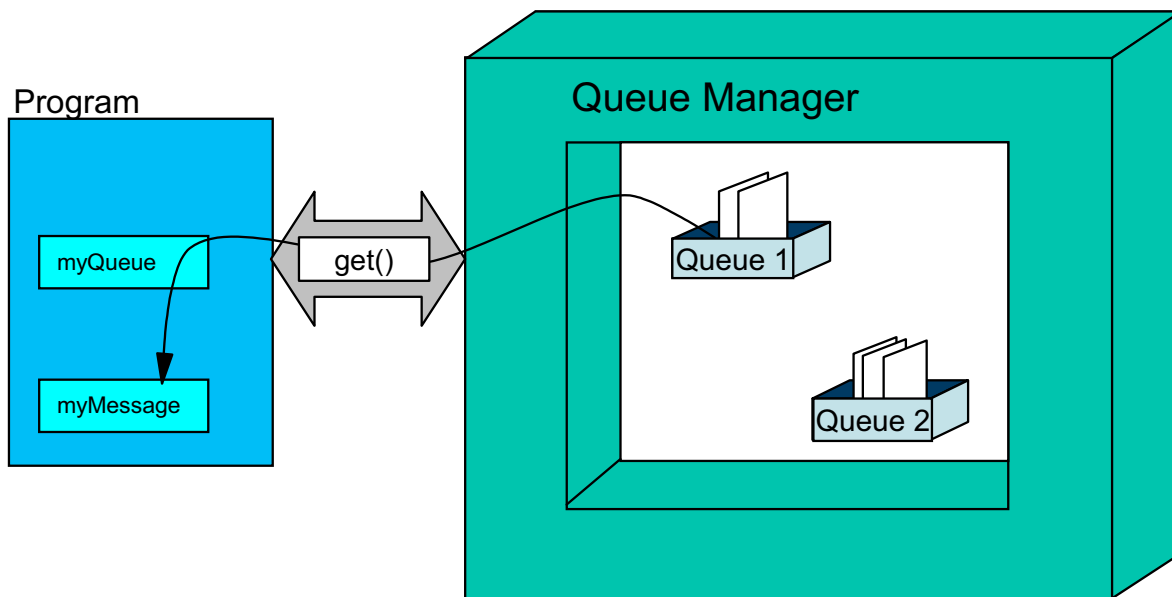


Figure 5-20. Getting Messages

MQ092.0

Notes:

The `get` will retrieve a message from the opened queue and place it into the named message object. All the properties of the message, known as fields of the Message Descriptor, are available as properties of the message object. The size of the user message buffer is indicated either on the `get` request, by a method of the `MQMessage` class or automatically by the `get` method.

Options specified on the `get` dictate which message is returned and what if any exception handling is to be handled by the request.

The `get` request is a method of the `MQQueue` class, if issued before the `accessQueue` request is issued it will throw an `MQException`.

Processing the Message

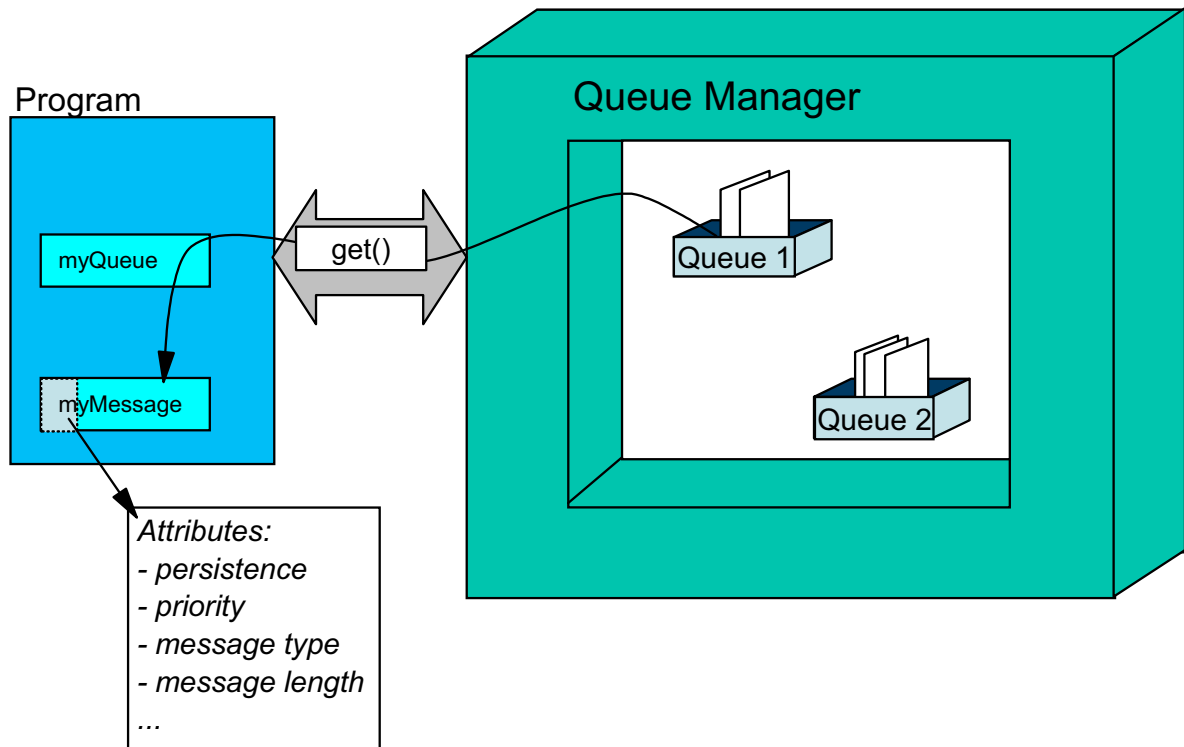


Figure 5-21. Processing the Message

MQ092.0

Notes:

The message is placed in the specified MQMessage buffer on successful completion of the get request. The MQMessage class provides methods for the reading and writing of the user portion of the message data, and the setting or viewing of the message attributes.

The provided methods allow for the user data portion to be manipulated by field type, by number of characters or even as a variable length binary image.

The length and format of the user data is stored as attributes of the message and are available by MQMessage method calls.

Creating the Message Buffer

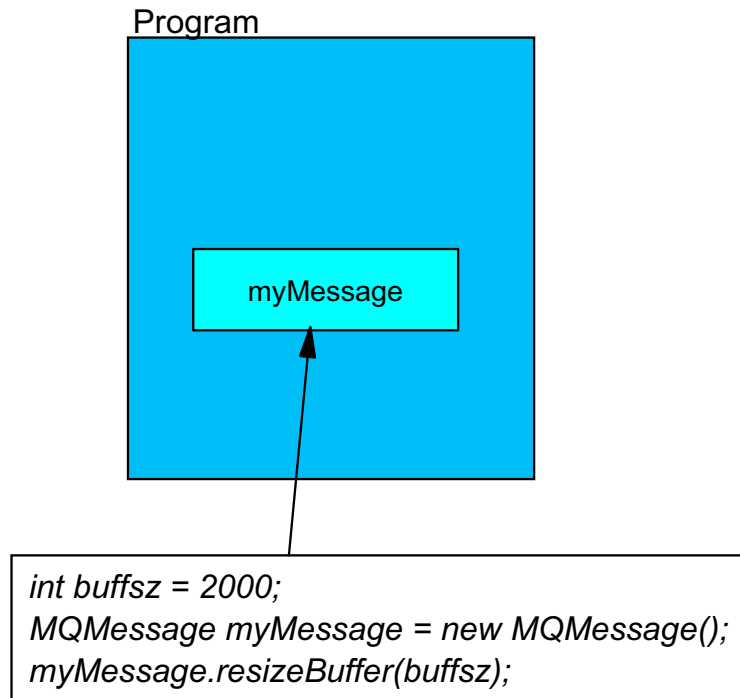


Figure 5-22. Creating the Message Buffer

MQ092.0

Notes:

The first thing to do before getting a message is to create a message buffer:

```
int buffsz = 2000;
```

An integer is named and set to 2000 as an example. The required value would be the size of a user message to be received. An integer value is required by the resize method.

```
MQMessage myMessage = new MQMessage();
```

The Java program must provide an object to store the user data and message descriptor fields. This object is created by the constructor of the MQMessage class. The constructor has no parameters, it is instantiated with default values.

The fields of the descriptor are named to provide the program with the ability to set them as required before the get is issued.

MQMessage represents both the message descriptor and the data for an MQSeries message.

```
myMessage.resizeBuffer(buffsz);
```

The program can set the size of the user data area, at any time, with the `resizeBuffer` method or use a parameter on the `get` request. If the existing buffer contains data, the `resizeBuffer` request may result in the loss of data. `MQMessage` represents both the message descriptor and the data for an MQSeries message.

Get Message Options

Variables:

- options
- waitInterval
- resolvedQueueName
- matchOptions
- groupStatus
- segmentStatus
- segmentation

Figure 5-23. Get Message Options

MQ092.0

Notes:

Like the PMO object for Put Message Options, there is a GMO object that allows for definition of options to be used when the get is issued.

The Get Message Options object supplies information that controls the functions performed by the get request. It will subsequently be named on a get request.

GMO: options

```
MQC.MQGMO_NONE
MQC.MQGMO_NO_WAIT
MQC.MQGMO_WAIT
MQC.MQGMO_BROWSE_FIRST
MQC.MQGMO_BROWSE_NEXT
MQC.MQGMO_MSG_UNDER_CURSOR
MQC.MQGMO_FAIL_IF_QUIESCING
```

Figure 5-24. GMO: options

MQ092.0

Notes:

The options variable controls the actions performed by the get request. There are many values that can be specified for this variable. If more than one value is required they can be added together or combined using the bitwise OR operator.

MQC.MQGMO_NONE

The options variable is set to this value when the program does not require special actions to be performed on the get request. No other option would be specified with this option, as it would nullify this specification.

MQC.MQGMO_NO_WAIT

Upon receiving the get request, the queue manager will search the associated queue for a message that matches the selection criteria, as specified by the associated variables of this GetMessageOptions object. If no qualifying message is found, this option will result in the program regaining control with an MQException being thrown. The reasonCode field of the MQException class will be set to no message available.

MQC.MQGMO_WAIT

The specification of this option results in the queue manager blocking response to the program if there is no qualifying message currently on the queue.

MQC.MQGMO_BROWSE_FIRST

This option indicates to the queue manager that the get request is not to be a destructive read request but instead, the qualifying message is to be retrieved without deletion. To use this option the queue must have been opened with the get and browse options.

MQC.MQGMO_BROWSE_NEXT

This request moves the browse cursor, established by the browse first request, on to the next qualifying message on the queue, and retrieves that message. The retrieved message is not marked for deletion and is not locked, unless the associated lock option was also specified. Therefore the message potentially remains available to other programs for retrieval.

MQC.MQGMO_MSG_UNDER_CURSOR

This option results in the message, that had previously been returned on a browse first or browse next request, being re-obtained with the standard get request and is subsequently marked for deletion. The use of this option requires that a browse request had previously been issued, else an MQException will be thrown.

MQC.MQGMO_FAIL_IF QUIESCING

This option indicates to the queue manager that this request can be terminated, thereby throwing an MQException, if it is received during a time when the queue manager is in the process of shutting down. The program would be expected to honor this response by closing any open queues and disconnecting from the queue manager.

Issue the Get Request

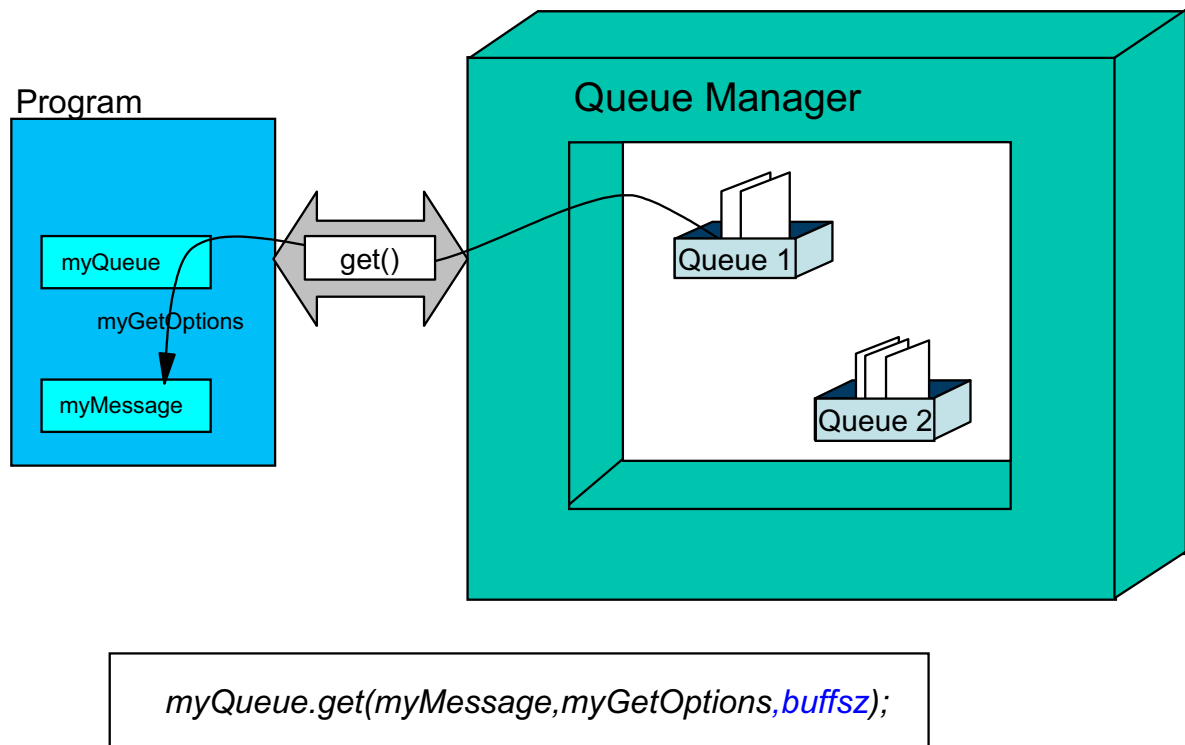


Figure 5-25. Issue the Get Request

MQ092.0

Notes:

The `get` method retrieves a message from the associated queue, it takes a `MQMessage` object as the first parameter. A number of fields in the `MQMessage` object are treated as input, in the processing performed by the underlying `MQSeries` queue manager in response to the `get` request.

If the `get` throws an `MQException` and the associated completion code is set to failed, the fields and message buffer of the `MQMessage` object are unchanged, as no message has been returned. If the `get` request does not throw an `MQException` or does, but the completion code is set to warning. The message fields (also known as member variables or Message Descriptor) and the message buffer of the associated `MQMessage` object are completely replaced with the message descriptor and message data from the incoming message.

The second parameter of the `get` method is the name of the associated `getMessageOptions` object. The fields of this object control the actions of the `MQGET` request, in particular the `options` field controls the type of `get` request and another field controls the method of message identification.

Other fields within the `getMessageOptions` object are updated upon successful completion of the `get` request and are in addition to the characteristics of the selected message as stored in the fields of the `MQMessage` object.

The third and last parameter of the `get` request is the size of the largest message that can be returned by this request.

If a value is not supplied the default action will be to adjust the size of the buffer of the specified `MQMessage` object to accommodate the selected message. The use of this parameter can therefore be to restrict the size of the message buffer, and thereby stop a very large message from being returned.

This parameter overrides the default size and the value set by the `resizeBuffer` request.

Retrieving Message Length

```
int tml;  
myMessage.getTotalMessageLength(tml);  
System.out.println("The message length is: " + tml );
```

```
int ml;  
myMessage.getMessageLength(ml);  
System.out.println("The length of the user data in the  
message is: " + ml);
```

Figure 5-26. Retrieving Message Length

MQ092.0

Notes:

The `getTotalMessageLength` method returns an integer. The value is the total number of bytes of the message, as stored on the queue from which it has been retrieved (or attempted to be retrieved). If the `get` method fails, with a message-truncated error code, the value returned by this method is the total size of the message on the queue.

The `getMessageLength` returns an integer. The value is set to the length of the user data portion of the message returned in the associated `MQMessage` object.

This method throws an `IOException` if there is no message in the buffer or the buffer size has been set to zero.

Retrieving User Data

```
myMessage.readString(int)
myMessage.readFloat()
myMessage.readFully(byte,offset,number)
myMessage.readInt()
myMessage.readObject()
myMessage.readUTF()
myMessage.readDecimal2()
myMessage.readDecimal4()
myMessage.readDecimal8()
```

Figure 5-27. Retrieving User Data

MQ092.0

Notes:

readString()

The readString method of the MQMessage class will retrieve the specified number of characters from the message buffer. The data will be retrieved from the current buffer position as indicated by the cursor. This method throws IOException if there is no message and EOFException if the requested length plus the current cursor position exceeds the total message length.

readFloat()

The readFloat method will return a floating point number from the current position in the message buffer. The cursor will then be incremented past the item in the message buffer.

If the referenced item is not a floating point number an IOException will be raised.

readFully()

This method will treat the message data as bytes, the data will be copied without translation from the input location to the specified output location.

The `readFully` method is overloaded, the first form has only one parameter: the output byte array. The second form has three parameters, the first is the output byte array, the second is the byte displacement from the current message buffer position and the third parameter is the number of bytes to be copied.

In the case of the first form of the method, the number of bytes copied is equal to the size of the receiving byte array, and the data is taken from the current message buffer position.

The first form throws `Exception`, `EOFException`.

The second form throws `IOException`, `EOFException`.

`readInt()`

The `readInt` method will return an integer from the current message buffer position. The method `readInt4` is a synonym for `readInt` and is provided for cross-language MQSeries API compatibility.

The methods throw exceptions of `IOException` and `EOFException`.

`readObject()`

The `readObject` method will read an object from the current position in the message buffer.

The class of the object, the signature of the class, and the value of the non-transient and non-static fields of the class are all read and copied to the output specification.

This method throws an `OptionalDataException`, a `ClassNotFoundException` or an `IOException`.

`readUTF()`

Reads in a string that has been encoded using a modified UTF-8 format from the current position in the message buffer, and returns it as a Unicode string. This method blocks until all the bytes are read, the end of the stream is detected, or an exception is thrown.

The first two bytes of the UTF format is a short binary, the value is the number of bytes actually written out, not the length of the string. Following the length, each character of the string is in the UTF-8 encoding for the character.

`readDecimalx()`

The `readDecimal2` method reads a 2-byte packed decimal number in the range (-999. +999). The `readDecimal4` method reads a 4-byte packed decimal number in the range (-9999999. +9999999).

The `readDecimal8` method reads an 8-byte packed decimal number in the range (-9999999999999999 to +9999999999999999).

The methods all throw `IOException` and `EOFException`.

Catering for the Exceptions

- MQException:
 - no message available
 - message is too big
 - not open for get
 - options error
 - message has been truncated
- IOException
- EOFException

Figure 5-28. Catering for the Exception

MQ092.0

Notes:

Above are some of the most common exceptions returned by a get. Here is a description of these exceptions.

MQRC_NO_MESSAGE_AVAILABLE

The get request failed to find a message on the current queue. This can be caused if the queue is empty or a specific message is being requested but is yet to be placed on the queue. The situation can also be caused if the message is on the queue but has been locked by another process. The standard processing would be to try the get request again after waiting the required period of time.

MQRC_TRUNCATED_MSG_FAILED

When receiving a message the get method will acquire a message buffer large enough for the selected message. This feature is overridden by the third parameter of the get. If specified it will restrict the maximum size of the buffer to the value that is specified. Therefore if a message larger than this value is selected for return to the program the request will fail with this reason code.

MQRC_NOT_OPEN_FOR_INPUT

The get request is a method of the MQQueue object. The accessQueue method opens the associated MQSeries queue object. For the get request to succeed, the open options specified on the accessQueue method must include an input option. Else an exception will be raised and the reason code will be set to indicate that the queue was not opened for input.

MQRC_OPTIONS_ERROR

The options specified on the get request must be consistent, else the get request will fail with this MQException. If more than one the input options is specified, this exception can be thrown. If the browse option is specified but an associated input option is not coded, then this exception can be thrown.

MQRC_TRUNCATED_MSG_ACCEPTED

An optional parameter of the get request is an integer setting the maximum size of a message that can be received by this request. This parameter will override the resizeBuffer request or the default nature of the MQMessage buffer size mechanism. If the user message exceeds this value the request will fail and no message data will be received. But if the get message option of MQC.MQGMO_ACCEPT_TRUNCATED_MSG is specified, then the request will raise a warning instead of a failure and the portion of the message that does not exceed the specified maximum size will be received and the remainder discarded.

IOException or EOFException

The IOException and EOFException conditions thrown by a number of MQMessage methods relate to the violation of message data integrity. This can be caused by the truncation of message data, due to size limitations. The incorrect specification of data types. Or message buffer cursor adjustment errors.

5.4 Checkpoint and Summary

Unit Checkpoint

1. What is the name of the MQSeries class that incorporates the variables and data of a message?
2. What is the name of the MQSeries class that incorporates the variables and options required by the put method?
- 3.

```
String quartet = "notes";  
MQMessage myMessage = new MQMessage();  
MQPutMessageOptions myPMO = new MQPutMessageOptions();  
_____._____ (quartet);
```

In this example, what is the name of the object that contains the user message buffer?

4. In the above example, what method will be used to write the string into the message buffer?

Notes:

Unit Checkpoint

5.

```
String quartet = "notes";  
int theAnswer = 42;  
MQMessage myMessage = new MQMessage();  
MQPutMessageOptions myPMO = new MQPutMessageOptions();  
myMessage.writeString(quartet);  
myMessage._____ (theAnswer);
```

What method will be used to move the cursor to position fortytwo within the buffer?

6. What open option should be specified to allow messages to be put on the queue?

Notes:

Unit Checkpoint

7.

```
String quartet = "notes";
int oOpts = MQC.MQOO_OUTPUT;
MQQueue myOutputQueue = qMgr.accessQueue("amore", oOpts);
MQMessage myMessage = new MQMessage();
MQPutMessageOptions myPMO = new MQPutMessageOptions();
myMessage.writeString(quartet);
_____.put(myMessage, myPMO);
```

The put is a method of which object?

8. What open option will be required to open the queue to get messages, while sharing the queue with other programs?
9. What is the name of the method to specify the size of the message buffer as 2000?
10. What is the name of the class that encapsulates the options required by the get request?

Notes:

Unit Checkpoint

11.

```
MQQueueManager eThompson;  
eThompson = new MQQueueManager("Atlanta");  
int oOpts = MQC.MQOO_INPUT_SHARED;  
MQQueue myInputQueue = eThompson.accessQueue("Wildwood", oOpts);  
MQMessage myMessage = new MQMessage();  
myMessage.resizeBuffer(2000);  
MQGetMessageOptions myGetOpts = new MQGetMessageOptions();  
myGetOpts.options = MQC.MQGMO_ACCEPT_TRUNCATED_MSG |  
    MQC.MQGMO_BROWSE_FIST;  
_____.get(...
```

In this example, what is the name of the object that has a method to get messages from the queue?

Notes:

Summary

You should now know how to:

- Construct and manage an MQMessage object
- Put messages onto queues, in different formats, with various options
- Get messages from queues, in different formats, with various options
- Retrieve and set message attributes

Notes:

Unit 6. Messages Types

What This Unit is About

This unit describes the message types provided by MQSeries.

What You Should Be Able to Do

After completing this unit, you should be able to

- Describe the message types
- Work with request and replies
- Use the report function of MQSeries

How You Will Check Your Progress

Accountability:

- Checkpoint
- Machine exercises

References

SC34-5456 *MQSeries Using Java*

SC33-1673 *MQSeries Application Programming Reference*

<http://www.ibm.com/software/ts/mqseries/messaging/>
WebSphere MQ

Objectives

- Understand different message styles
- Work with request and replies
- Use the MQSeries report function

Figure 6-1. Unit Objectives

MQ092.0

Notes:

6.1 Requests and Replies

Message Types

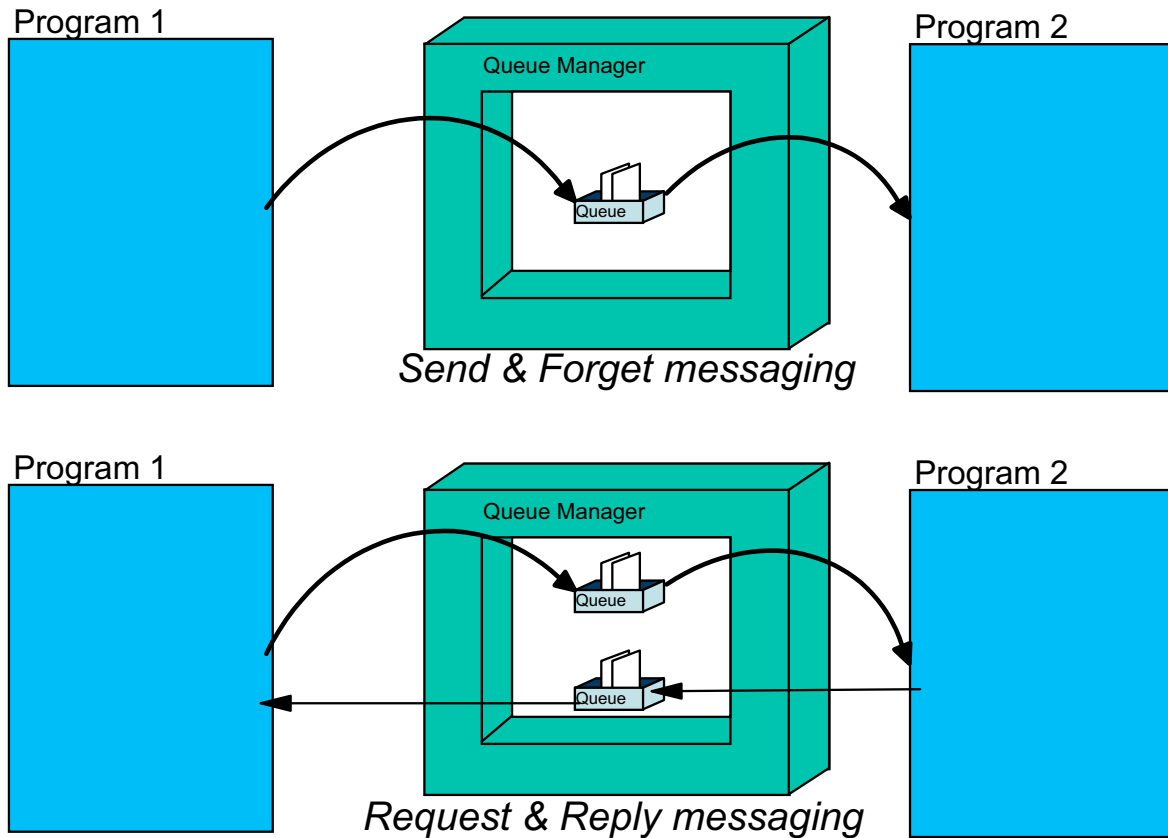


Figure 6-2. Message Types

MQ092.0

Notes:

The message types used by a messaging system are indicators that are used to show the purpose or intent of the message. They basically fall into two categories, either the send and forget style of messaging or the request and reply style.

With the send and forget style, the client program will create a message indicating it is a datagram message type and put it on the servers input queue. The server program will get and process the message. When finding that the message is a datagram, the server program will not generate a reply message, instead it will complete the processing of this message.

With the request and reply style, the client program will create a message indicating it is a request message type and put it on the servers input queue. The server program will get and process the message. When finding that the message is a request, the server program will generate a reply message and put it on the clients input queue.

How to use Message Types

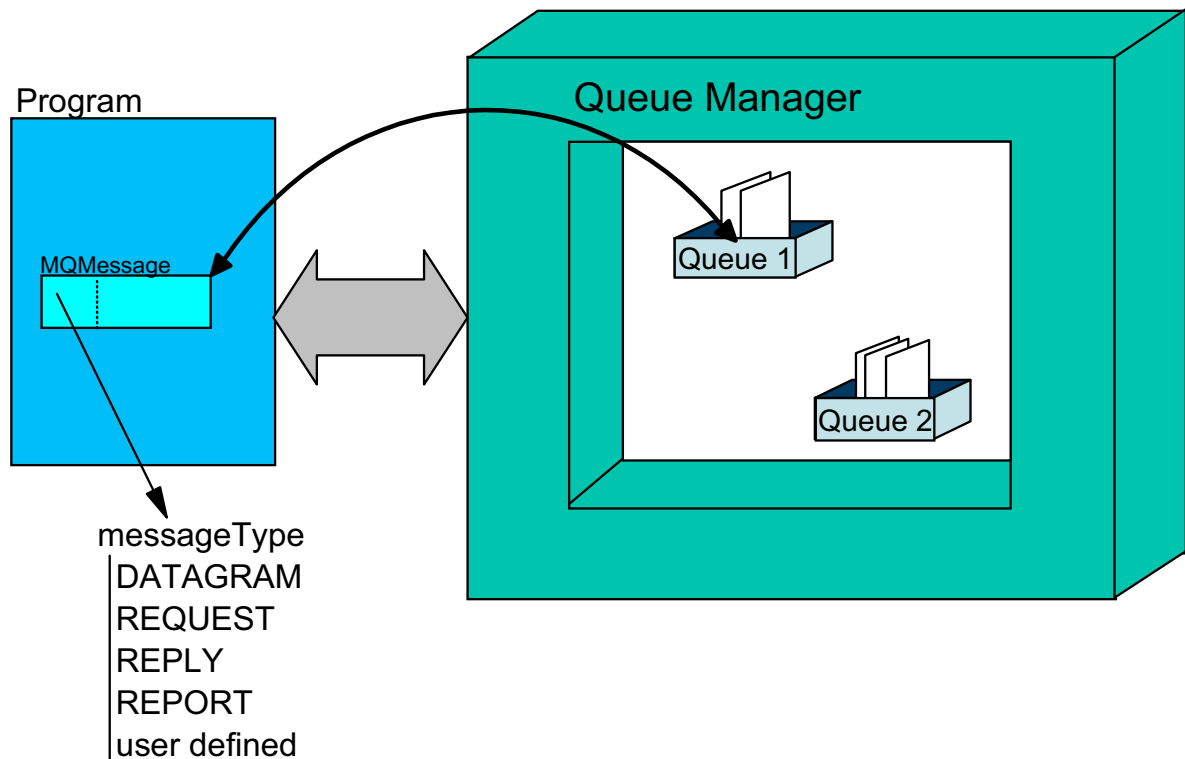


Figure 6-3. How to use Message Types

MQ092.0

Notes:

One of the MQMessage variables is named the messageType, it is an integer that is set by the message creator to indicate to the receiver the purpose of the message.

The value of the messageType variable is either ignored or acted upon based on the logic of the receiver program. This variable is not a message selection option, therefore a message can not be selected for retrieval based upon the value of this variable. Currently MQSeries has four system defined values - DATAGRAM, REQUEST, REPLY, REPORT - and a range assigned for user defined values.

```
myMessage.messageType =MQC.MQMT_DATAGRAM;
```

If the program creating the message does not assign a value to the messageType variable, it will by default be set to message type of datagram. This setting, indicates to the receiving program, that the sending program has not indicated any special messaging properties or actions that need to be performed by the receiver program. A datagram is synonymous with a broadcast or newsletter form of messaging, where the originator is not expecting an interaction or reply from the receiver of this message.

`myMessage.messageType =MQC.MQMT_REQUEST;`

The setting of the `messageType` variable to message type of request, indicates to the receiver of this message that a reply message is expected to be created and sent to the reply queue as indicated by the `replyToQueueName` variable of this message. When the message is created and put to the queue with this message type, the queue manager will enforce that there must be a valid value in the `replyToQueueName` variable. This form of messaging is synonymous with a conversational style of communications.

`myMessage.messageType =MQC.MQMT_REPLY;`

The setting of the message type variable to the message type of reply, indicates that the message is an answer or reply to another message. Presumably the original message was a message type of request, therefore indicating that it required a reply message. The reply message is put to the queue identified by the 'replyToQueueName' variable of the request type message.

`myReply.messageType =MQC.MQMT_REPORT;`

The setting of the message type variable to a message type of report, indicates that this message is a status message rather than a reply message. This type of message is generated by the queue manager when the original message has specified reports are required. The queue manager will send the report type messages to the queue named by the `replyToQueueName` field of the original message. Report type messages can also be created by programs, to indicate a processing condition has been raised. Presumably the original message was a message type of request, therefore indicating that it required a response message.

Request

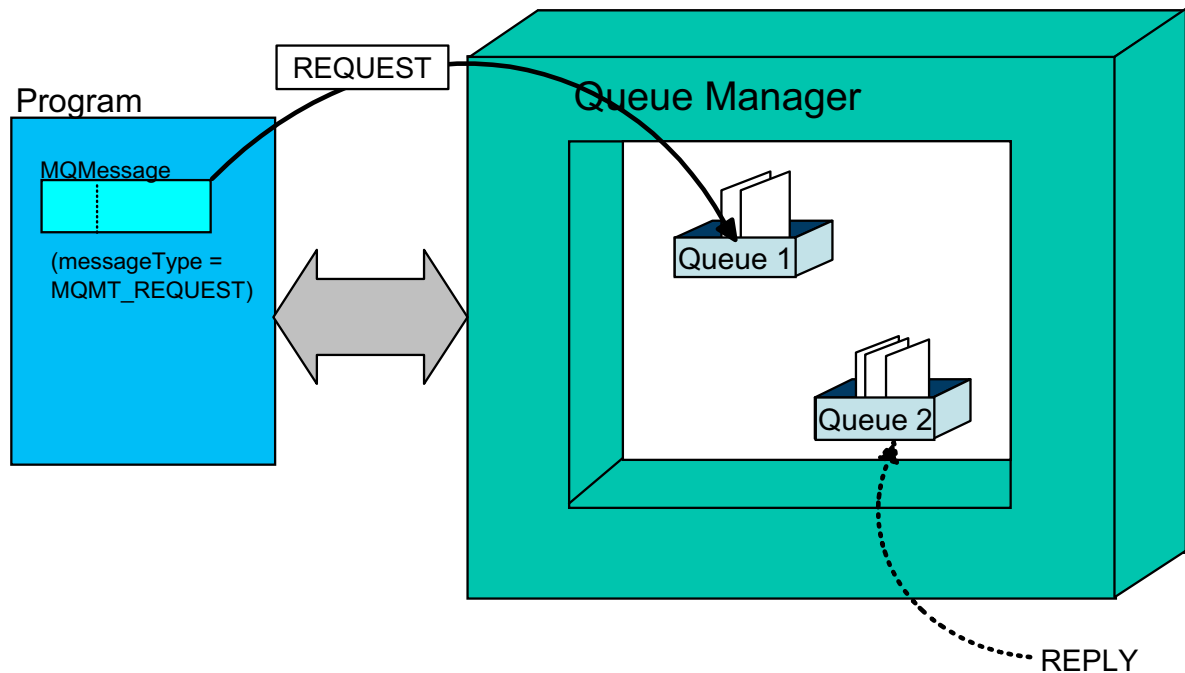


Figure 6-4. Request

MQ092.0

Notes:

When some type of reply is desired, an application should set the `messageType` variable to `MQMT_REQUEST`. Also, the `replyToQueueName` variable should contain the name of a queue that will be monitored for responses.

When an application retrieves a message from a queue, it can determine the message type, and if a request, can use the `replyToQueueName` and `replyToQueueManagerName` variables to know where to send any replies.

The program that initially sent the request can then look for replies on the `replyToQueue` that it specified.

replyToQueueManagerName/replyToQueueName

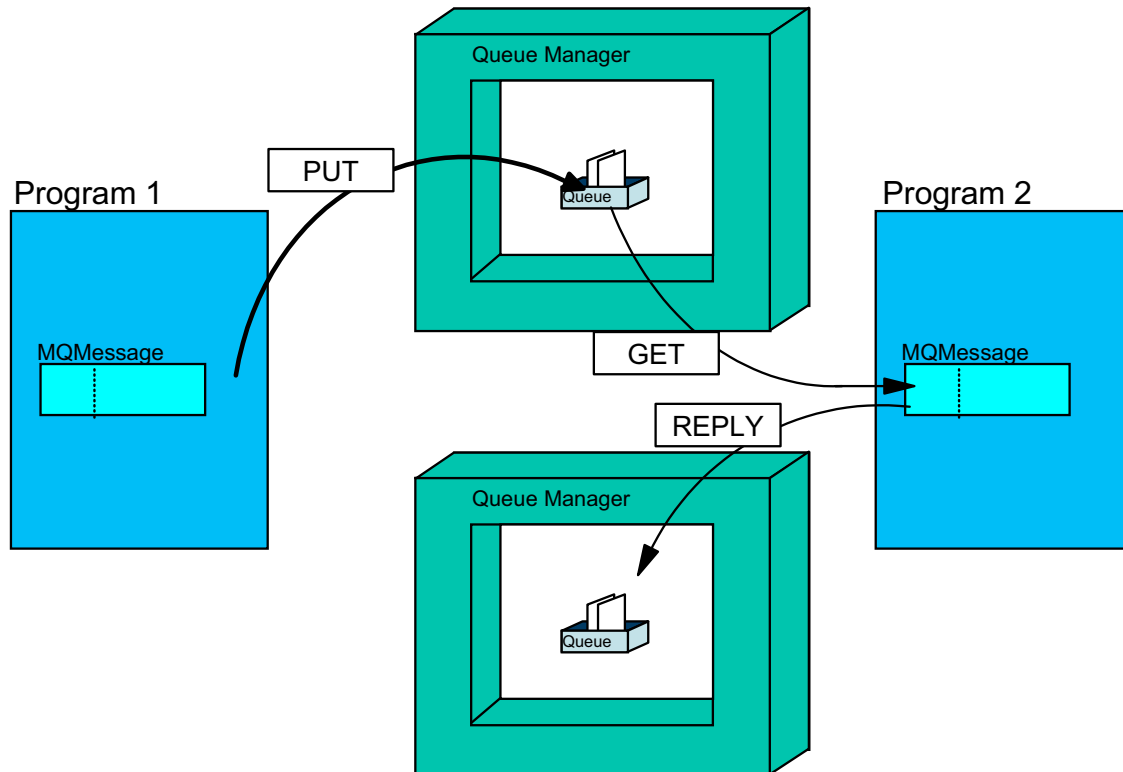


Figure 6-5. replyToQueueManagerName/replyToQueueName

MQ092.0

Notes:

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

The `replyToQueueManagerName` variable is the name of the queue manager that hosts the queue that has been specified as the value of the `replyToQueueName` variable.

If this variable is set to "" on input to the put request, the queue manager will validate the value of the `replyToQueueName` variable against the current queue manager and extract from the resolving definition the name of the hosting queue manager. This subsequent value will be stored as the `replyToQueueManagerName` and returned to the program on successful completion of the put request.

Reply

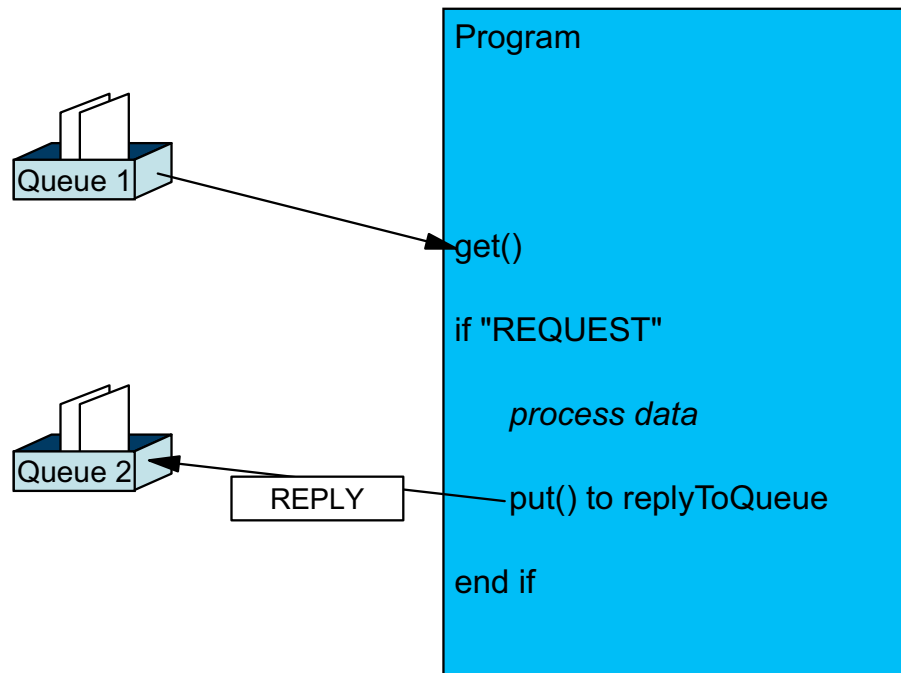


Figure 6-6. Reply

MQ092.0

Notes:

When a program issues an MQGET, it might be designed to interrogate the messageType to validate that the message is a request. If not, some error process might be invoked if this getting program always expects request type messages.

Once the message type is deemed to be valid, business processing is completed and some type of response message can be formulated and sent to the replyToQueue.

Note that for the reply, the messageId and correlationId are usually handled as follows:

3. correlationId takes the value of the request message's messageId
4. messageId is null.

Retrieving the Reply Queue and Reply Queue Manager Names

```
myInputQueue.get(myMessage, gmo);

replyQueueName = myMessage.replyToQueueName;
replyQueueManagerName = myMessage.replyToQueueManagerName;

myReplyQueue = qMgr.accessQueue(
    replyQueueName,
    openOptions,
    replyQueueManagerName,
    null,          // no dynamic queue name
    null);        // no alternate userId
```

Figure 6-7. Retrieving the Reply Queue and Reply Queue Manager Names

MQ092.0

Notes:

The `replyToQueueName` variable of the `MQMessage` object is defined as a string. The value is the name of the message queue for the replies to this message.

The `replyToQueueManagerName` variable of the `MQMessage` object is defined as a string. The value is the name of the queue manager for the replies to this message.

6.2 Reports

Report Messages

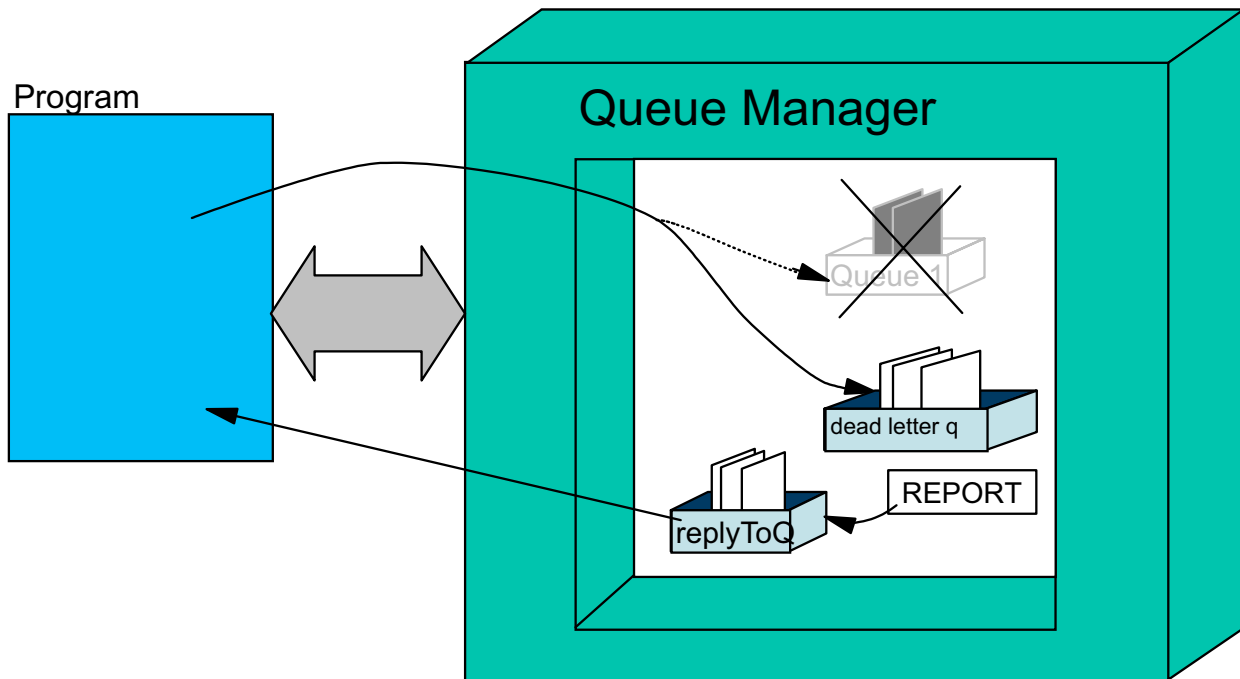


Figure 6-8. Report Messages

MQ092.0

Notes:

All queue managers have the ability to have messages that are undeliverable placed on an undeliverable message queue called a dead letter queue.

When a message is placed on the dead letter queue because it can not be delivered, the queue manager takes no action to notify the sender unless one of the report options specified when the message was put was `MQRO_EXCEPTION_....` (where `....` can be nothing, `WITH_DATA` or `WITH_FULL_DATA`). In the case where one of these report options is specified, the queue manager will build a report message (messageType will be `MQMT_REPORT`). The Feedback field in the message descriptor of the report message (represented by the feedback variable of the `MQMessage` object) will contain the reason code associated with the failed attempt to deliver the message. The report message will be placed on the `replyToQueue` as specified in the undeliverable message.

Exception Reports

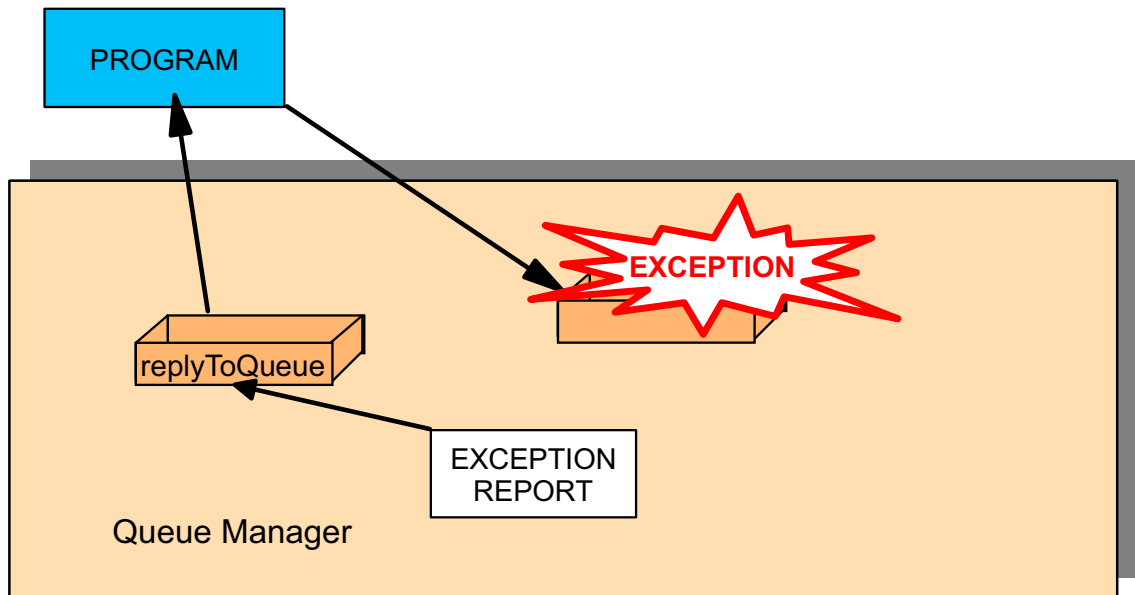


Figure 6-9. Exception Reports

MQ092.0

Notes:

When an application puts a message on a queue, if the queue is local, the completion and reason code will immediately tell the program whether the message is successfully delivered. However, the nature of asynchronous processing means that a message that is successfully placed on a transmission queue for subsequent delivery across a network results in a successful completion and reason code being returned to the putting application.

If the application wishes to have notification that a message is undeliverable, the message descriptor report field will need to include one of the following report options:

- MQC.MQRO_EXCEPTION
- MQC.MQRO_EXCEPTION_WITH_DATA
- MQC.MQRO_EXCEPTION_WITH_FULL_DATA

Each of the above will cause a report message to be generated. The first will result in a message descriptor with a zero length message. The second will include the first 100 bytes of the original message. The last will include the entire original message. In all cases, the

messageType will be MQMT_REPORT and the feedback field will contain the reason code that tells you why the message could not be delivered.

Expiry Reports

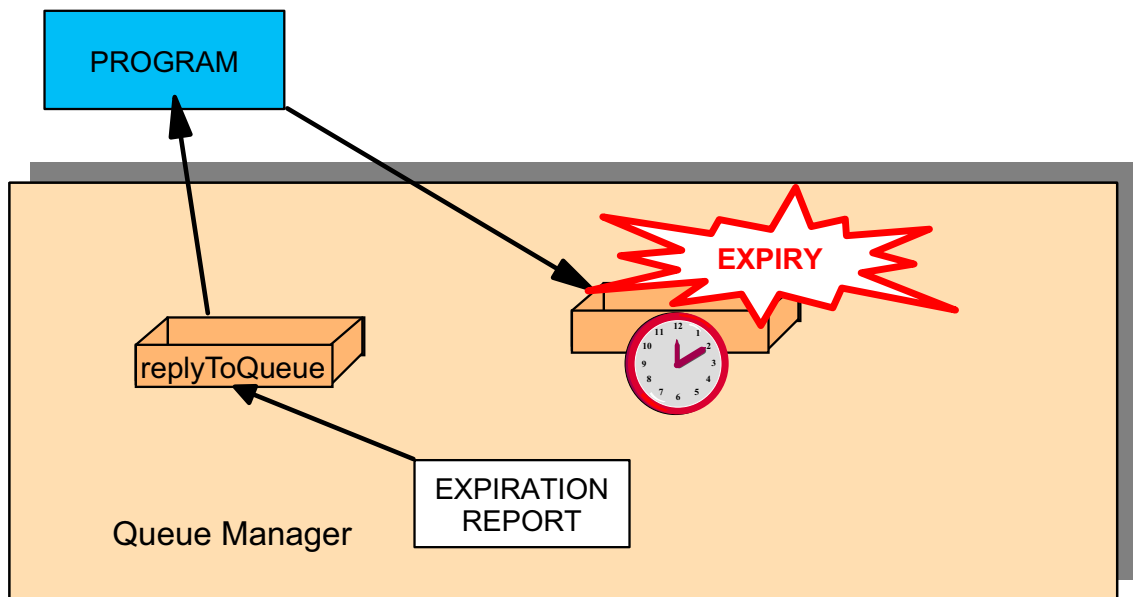


Figure 6-10. Expiry Reports

MQ092.0

Notes:

When the report field is set to request expiration reports, there are three types of the expiration reports that can be specified:

- MQC.MQRO_EXPIRATION
- MQC.MQRO_EXPIRATION_WITH_DATA
- MQC.MQRO_EXPIRATION_WITH_FULL_DATA

This type of report is generated by the queue manager if the message is discarded prior to delivery to a program because its expiry time has passed. If this option is not set, no report message is generated if a message is discarded for this reason.

The report option without the specification of data will generate a message that does not have a message buffer. The report option with the full data option will generate a message that contains the complete original failing message. While the report option with just the data option will only generate a message buffer with the first 100 bytes of the failing message.

An expiration report will only be built when the message is read from the queue, even if the message has already been expired for a long time.

COA and COD Reports

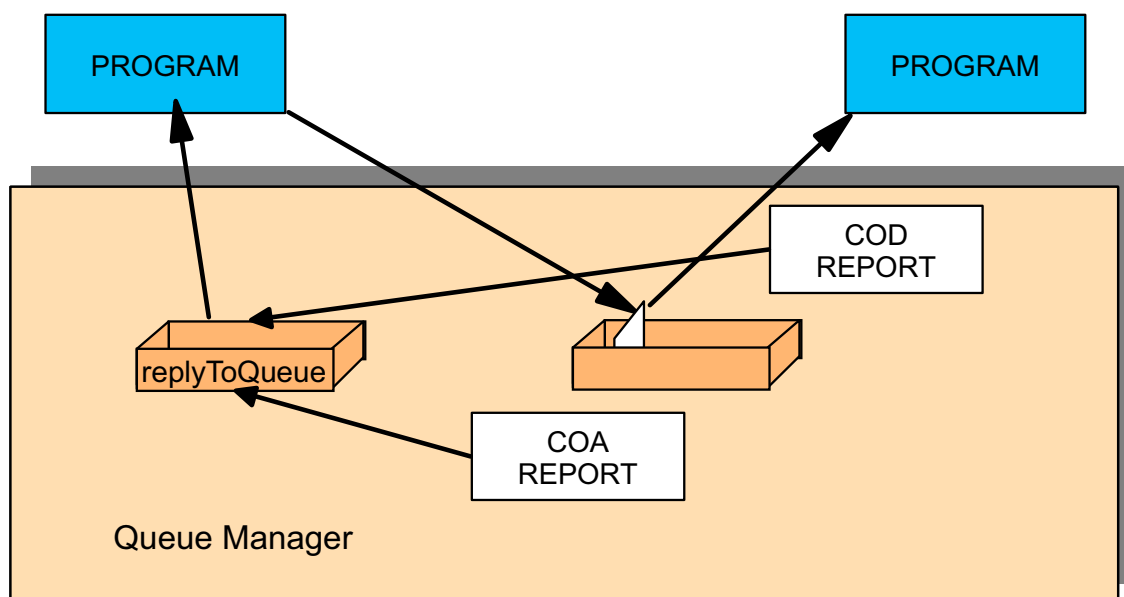


Figure 6-11. COA and COD Reports

MQ092.0

Notes:

A COA report is generated by the queue manager that owns the destination queue, when the message is placed on the destination queue. Message data from the original message is not included with the report message, unless the with data options are specified. If the message is put as part of a unit of work, and the destination queue is a local queue, the COA report message generated by the queue manager becomes available for retrieval only if and when the unit of work is committed.

A COA report is not generated if the Format field in the message descriptor is MQFMT_XMIT_Q_HEADER or MQFMT_DEAD_LETTER_HEADER. This prevents a COA report being generated if the message is put on a transmission queue, or is undeliverable and put on a dead-letter queue.

A COD report is generated by the queue manager when an application retrieves the message from the destination queue in a way that causes the message to be deleted from the queue.

If the message is retrieved as part of a unit of work, the report message is generated within the same unit of work, so that the report is not available until the unit of work is committed.

If the unit of work is backed out, the report is not sent. A COD report is not generated if the Format field in the message descriptor is MQFMT_DEAD_LETTER_HEADER. This prevents a COD report being generated if the message is undeliverable and put on a dead-letter queue.

Feedback

public int feedback

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

Figure 6-12. Feedback

MQ092.0

Notes:

The feedback field of the message object is used with a message of type MQC.MQMT_REPORT to indicate the nature of the report, and is only meaningful with that type of message.

The following feedback codes are defined by the system for general use:

MQC.MQFB_EXPIRATION
MQC.MQFB_COA
MQC.MQFB_COD
MQC.MQFB_QUIT
MQC.MQFB_PAN
MQC.MQFB_NAN

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

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

COPY_MSG_ID_TO_CORREL_ID

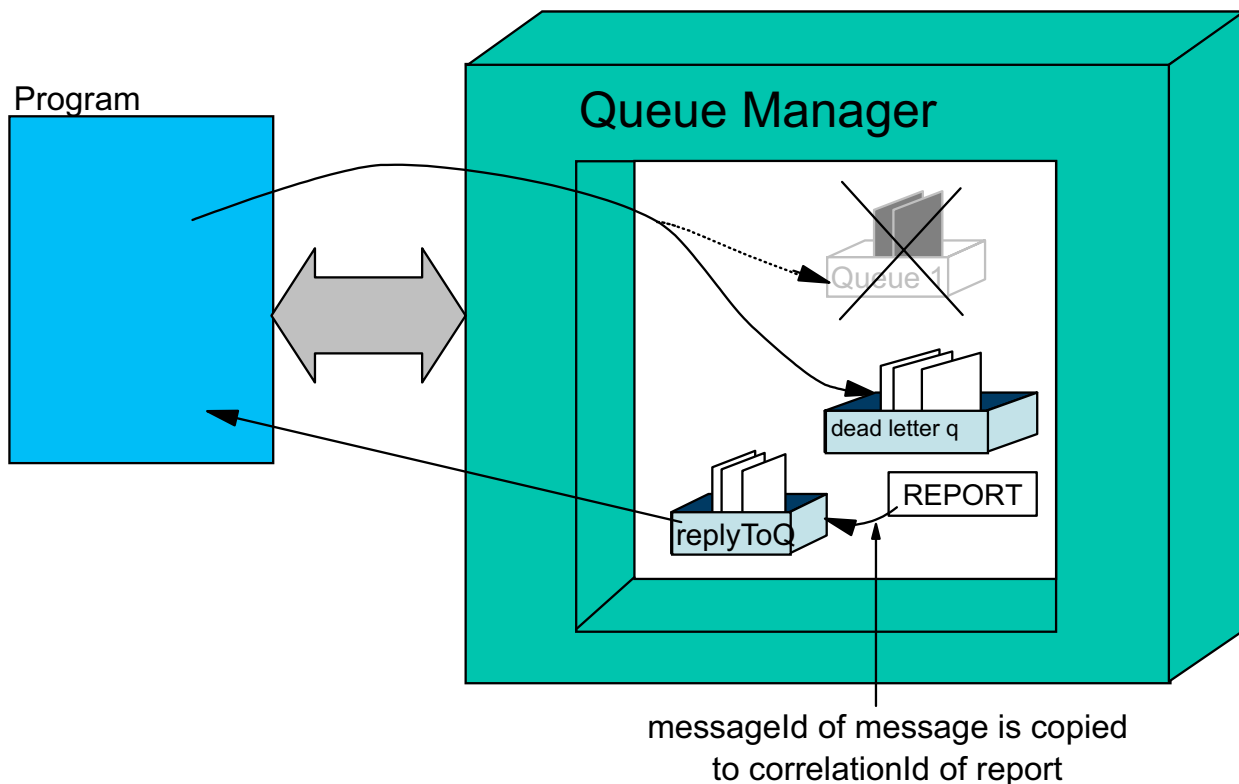


Figure 6-13. COPY_MSG_ID_TO_CORREL_ID

MQ092.0

Notes:

MQC.MQRO_COPY_MSG_ID_TO_CORREL_ID

This is the default report action, and indicates that if a report or reply is generated as a result of this message, the MessageId of this message is to be copied to the CorrelationId of the report or reply message.

MQC.MQRO_PASS_CORREL_ID

If a report or reply is generated as a result of this message, the CorrelationId of this message is to be copied to the CorrelationId of the report or reply message.

Servers replying to requests or generating report messages are recommended to check whether the MQC.MQRO_PASS_MSG_ID or MQC.MQRO_PASS_CORREL_ID options were set in the original message. If they were, the servers should take the action described for those options.

If neither is set, the servers should take the corresponding default action.

6.3 Checkpoint and Summary

Unit Checkpoint

1. What is the name of the variable of the MQMessage object that holds the type of message indicator?

2.

```
public void start(){  
    MQMessage myMessage = new MQMessage();  
    myMessage.writeString("Goodbye and thanks for all the Fish");  
    myMessage.messageType = MQC.MQMT_DATAGRAM;  
}
```

Could a report message be generated by this message?

3. What is the value that the messageType variable should be set to, to indicate that the message needs a reply?

4. What is the name of the variable of the MQMessage object that must be set when a message type of request is specified?

Notes:

Summary

You should now know how to:

- Work with different types of messages
- Send request messages
- Formulate and send replies
- Request reply messages

Notes:

Unit 7. Retrieval of Messages

What This Unit is About

This unit describes the possibilities to manipulate the message retrieval.

What You Should Be Able to Do

After completing this unit, you should be able to

- Get messages selectively using `messageId` and `correlationId`
- Code synchronous programs, using the `wait` function
- Group messages and retrieve groups of messages
- Handle segmented messages

How You Will Check Your Progress

Accountability:

- Checkpoint
- Machine exercises

References

SC34-5456

MQSeries Using Java

SC33-1673

MQSeries Application Programming Reference

<http://www.ibm.com/software/ts/mqseries/messaging/>
WebSphere MQ

Objectives

- Get messages selectively using messageId and correlationId
- Code synchronous programs, using the wait function
- Group messages and retrieve a group of messages
- Use message segmentation techniques

Notes:

7.1 Message Id and Correlation ID

messageId and correlationId

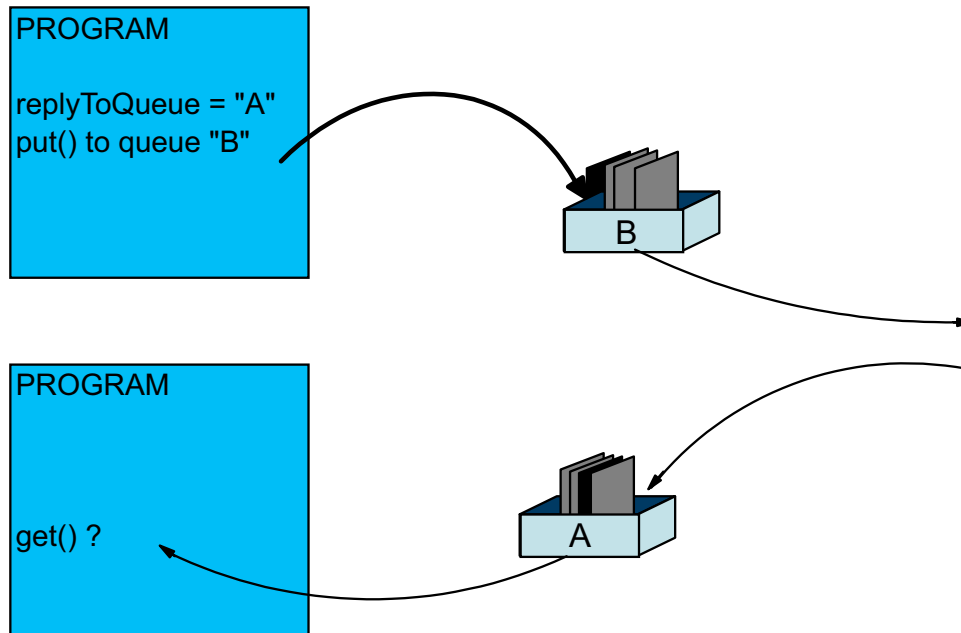


Figure 7-2. messageId and correlationId

MQ092.0

Notes:

Consider the above picture:

A continuously running server application is used to process requests for information.

- Several copies of this application may be running at one time
- Each request placed on the queue is for a unique customer
- The replyToQueue is always the same

That same program will process replies in response to each request

- The application must ensure that the reply it processes is for the request it has just sent, ignoring any that are not
- The replyToQueue may have messages that are related to requests from other copies of the application that are running at the same time

If an application simply issues GETs, we know the messages are destructively taken from the queue. The application can't remove other replies that are not for its request. It is possible to use the BROWSE function and check each message, but that can be wasteful since additional GETs would be issued just to examine each message.

Two fields in the message descriptor allow an application to selectively retrieve messages. The `messageId` and `correlationId` fields can be used in our scenario to ensure that the reply message retrieved is the one that matches the request sent.

messageld and correlationId

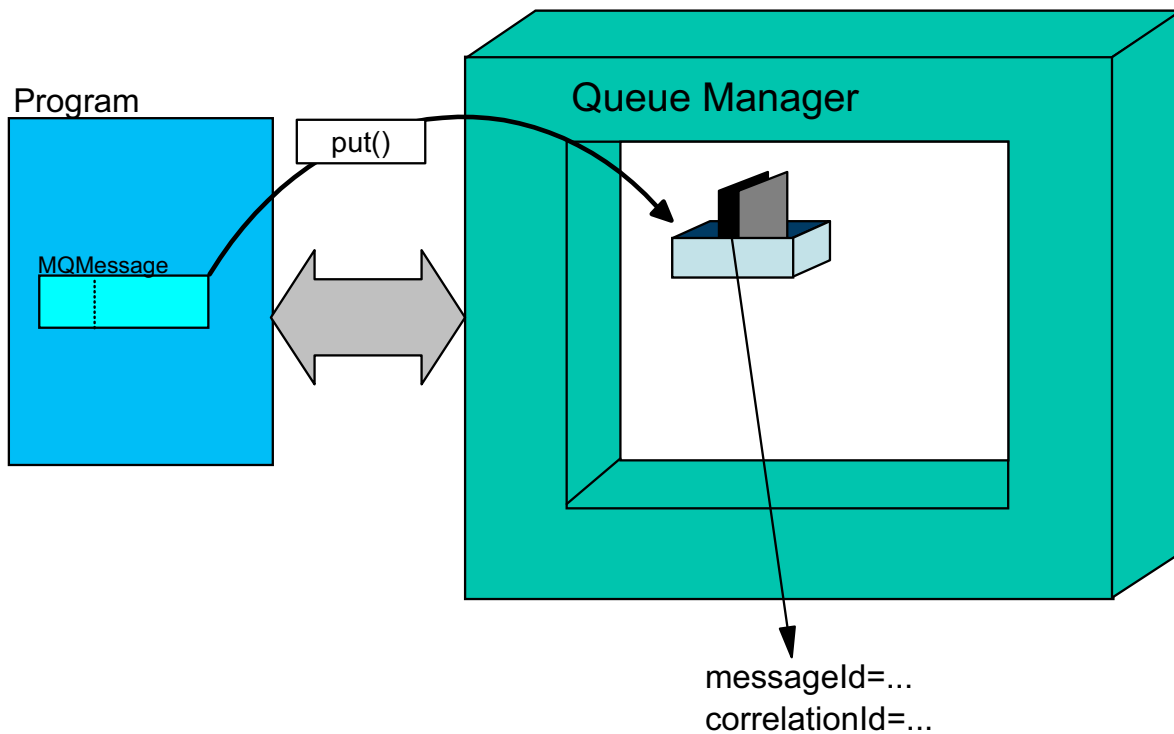


Figure 7-3. messageld and correlationId

MQ092.0

Notes:

After the successful completion of the put request the messageId variable will always have a value other than null but the correlationId variable may have a value of null.

The two variables are the primary identifiers of a message and will be referenced during the message selection processing performed by the get process.

The values expressed in these variables do not have to be unique, but if each message is to be treated as a separate entity, then each message should have a unique value for one or both of the message identifiers. This can be achieved by specifying MQPMO_NEW_MSG_ID or MQPMO_NEW_CORREL_ID in the Put Message Options.

messageId and correlationId

```
public byte messageId[]
```

```
public byte correlationId[]
```

Figure 7-4. messageId and correlationId

MQ092.0

Notes:

The messageId variable is a collection of 24 bytes, the value is not subject to data conversion as the message moves over a network of disparate nodes.

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

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

The correlationId variable is a collection of 24 bytes, the value is not subject to data conversion as the message moves over a network of disparate nodes.

For an `MQQueue.put` method, this specifies the correlation identifier to use. The default value is `MQC.MQCI_NONE`.

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

Using messageId and correlationId

```
myMessage.messageId = "custno27";  
myMessage.correlationId = "answer";  
myOutputQueue.put(myMessage);  
  
myReply.messageId = "custno27";  
myInputQueue.get(myReply);
```

Figure 7-5. Using messageId and correlationId

MQ092.0

Notes:

If the program sets the message id variable or the correlation id variable prior to the put request then the queue manager will not reset the values. The specified values will not be checked for uniqueness or consistency. The values are considered to be a pattern of bits and are designated by the queue manager as a byte field. They will not be subjected to character set conversion, if the message moves onto a node with a different character set code page setting. The message will then be identifiable by these value as it moves over the network, but only on nodes with the same character representation.

If the program wants to select a specific message, not just the next available message. It must set the message id and / or correlation id prior to the get request. But for this to be successful the server program must be performing compatible processing when putting the message.

If the program sets the message identifier variable prior to the get request, the queue manager will search the associated queue for the first available message that matches that messageId. If a matching message is not found the request will throw an MQException indicating that there is no message available.

The same applies for the correlation identifier.

If both the message and correlation identifiers are set prior to the get request, then a message will only be returned if it matches both of these values.

Using messageId and correlationId

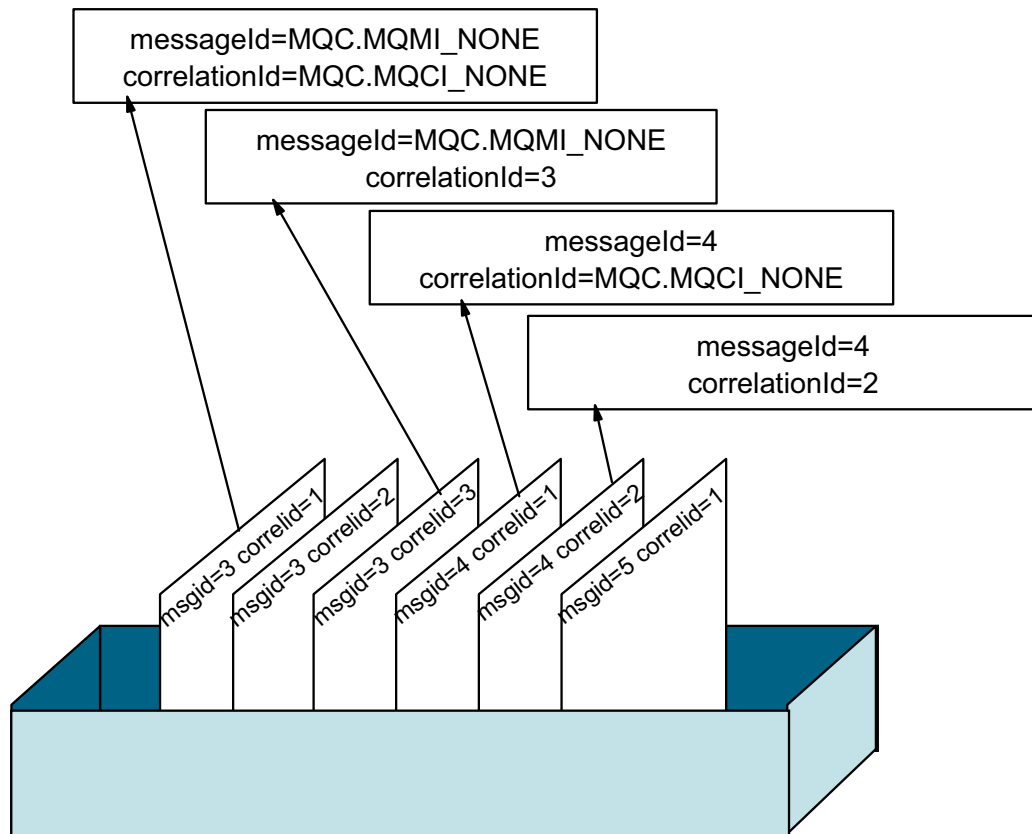


Figure 7-6. Using messageId and correlationId

MQ092.0

Notes:

Let's examine the examples:

1. This is asking for the first available message regardless of the values of messageId and correlationId
2. This one requests a message that has correlationId of 3, regardless of the value of the messageId field
3. This MQGET will succeed if it finds a message with messageId of 4, regardless of the value in correlationId
4. Unless a message with messageId of 4 AND correlationId of 2 is found, this call would not succeed

On completion of the four MQGETs in the above example, if no other messages have arrived, two messages would remain on the queue. Which ones?

Be aware that queues with many messages could cause a performance impact if an application chooses to use MQGET with messageId and correlationId: the queue manager will actually scan the queue sequentially looking for a match.

You can use the matchOptions to bypass this scan.

7.2 Waiting for Replies

Wait

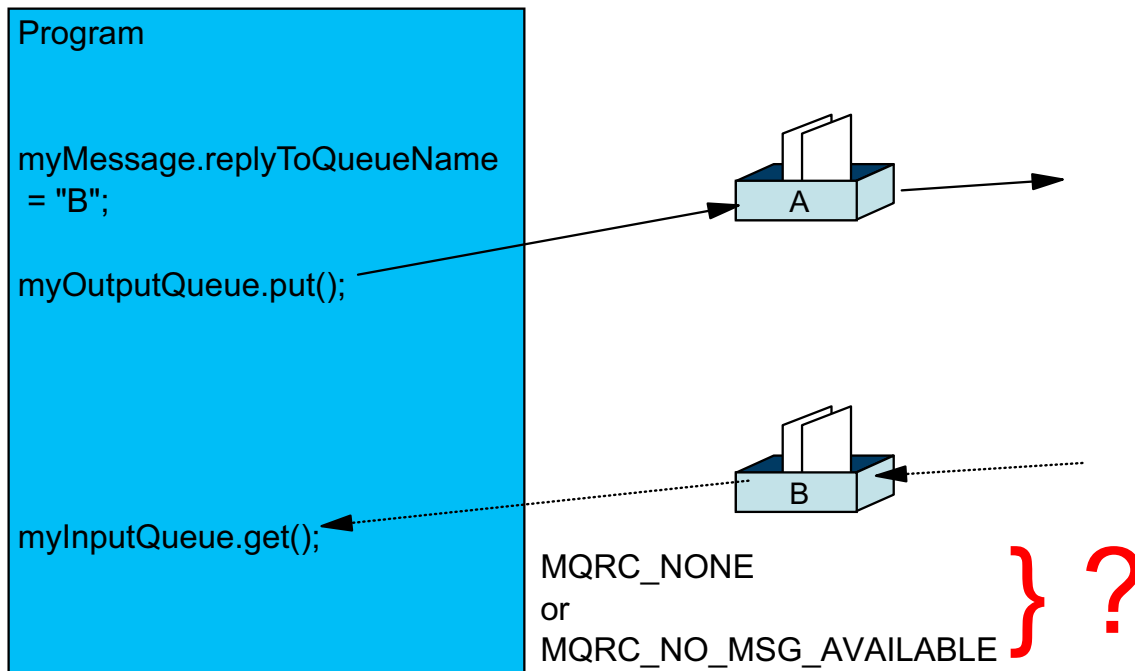


Figure 7-7. Wait

MQ092.0

Notes:

Because MQSeries is asynchronous by nature, it is probable that there may be a delay between the issuance of a request and any reply that is returned. The MQGET will fail with MQRC_NO_MSG_AVAILABLE if a message retrieval is attempted and none is available. So, to allow the program to wait for a message to arrive, there is a Get Message Options option of MQGMO_WAIT.

By specifying the MQGMO_WAIT option as one of the Get Message Options, an application can issue an MQGET and will essentially "go on hold" until a message arrives that satisfies the delivery requirements.

Some conditions that the Wait option allows for are:

- messages to be delivered across a network
- another program to be triggered to process the request
- a slower program to complete processing

If an application uses the MQGMO_WAIT, it is highly recommended that the Get Message Option of MQGMO_FAIL_IF_QUIESCING be used as well. This will allow the call to be

terminated if the queue manager is quiescing (MQRC_Q_MGR QUIESCING), possibly eliminating a locked condition.

The Wait can also terminate if conditions regarding the queue's ability to deliver messages changes. For instance, if the MQSeries administrator changes the Get attribute for the queue from "allowed" to "inhibited", the call would then fail with MQRC_GET_INHIBITED.

Using the Wait option can cause the generally asynchronous nature of MQSeries to become synchronous. Careful consideration should be given to its use.

GMO: wait option and waitInterval

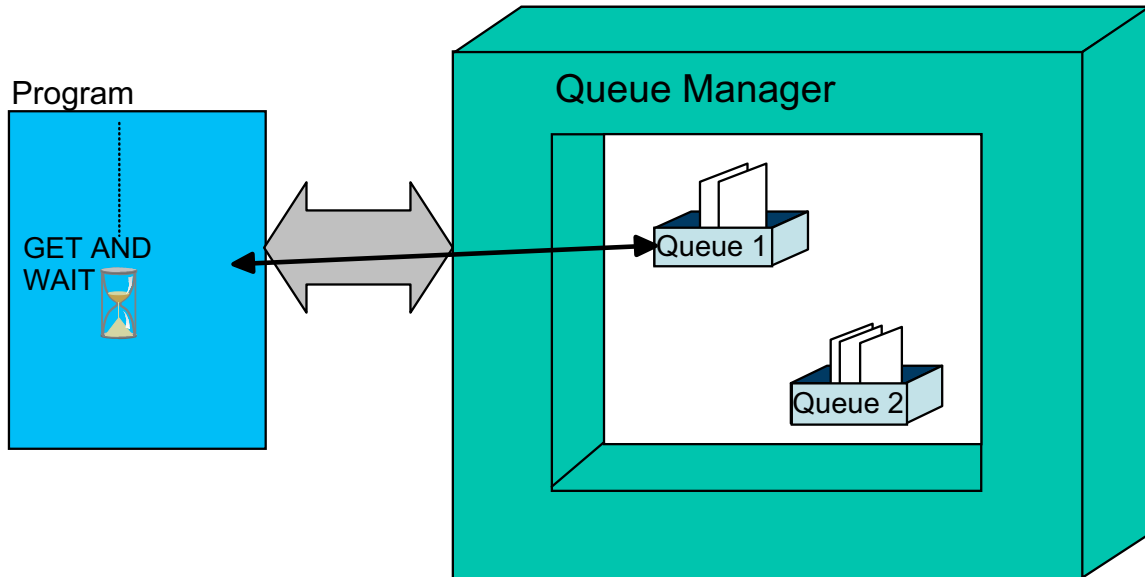


Figure 7-8. GMO: wait option and waitInterval

MQ092.0

Notes:

The maximum time (in milliseconds) that an `MQQueue.get` request is to wait for a suitable message to arrive. A value of `MQC.MQWI_UNLIMITED` indicates that an unlimited wait is required.

Wait with waitInterval

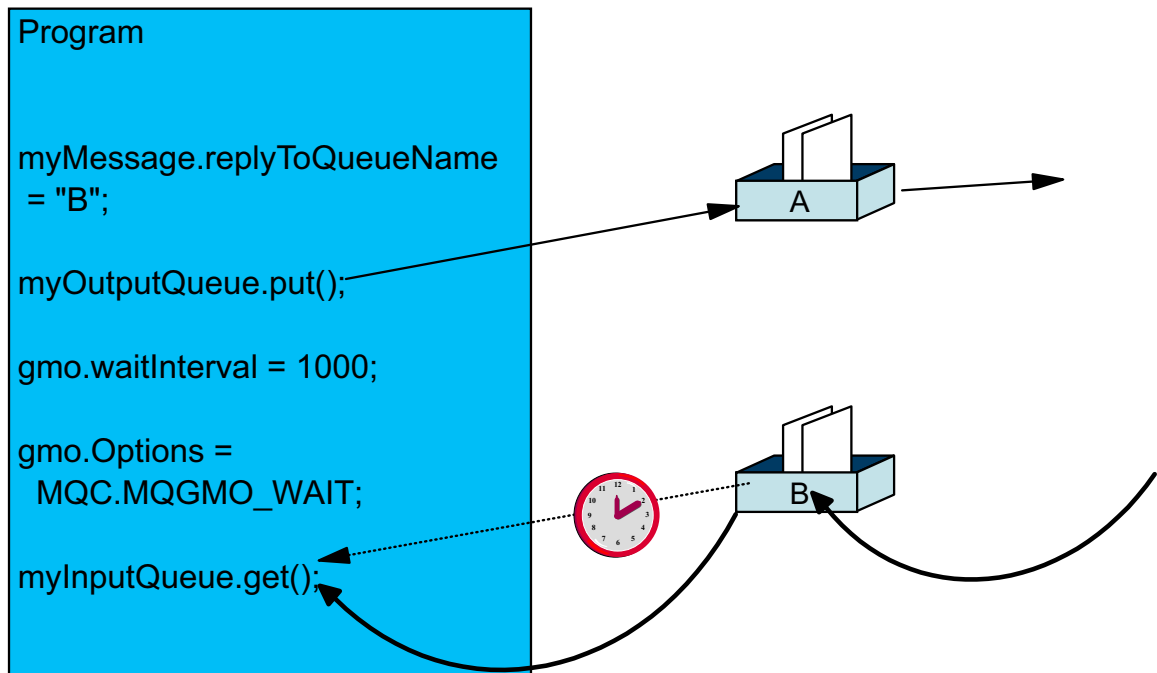


Figure 7-9. Wait with waitInterval

MQ092.0

Notes:

If a message is available when the MQGET is issued, the wait never takes effect. If not, the MQGMO_WAIT option along with a value given by waitInterval allow the program to establish a reasonable time to wait for a message. If none arrives during that time, the MQGET completes with a failing completion code and a reason code of MQRC_NO_MSG_AVAILABLE. During the wait interval, if a message arrives, the wait is immediately satisfied.

Remember, by not specifying a waitInterval, an application could wait forever since the default is MQWI_UNLIMITED.

Wait Example

```
MQMessage retrievedMessage = new MQMessage();

MQGetMessageOptions gmo = new MQGetMessageOptions();
gmo.options = MQC.MQGMO_WAIT;
gmo.waitInterval = MQC.MQWI_UNLIMITED;

m_queueD.get(retrievedMessage,
             gmo,
             100);           // max message size
```

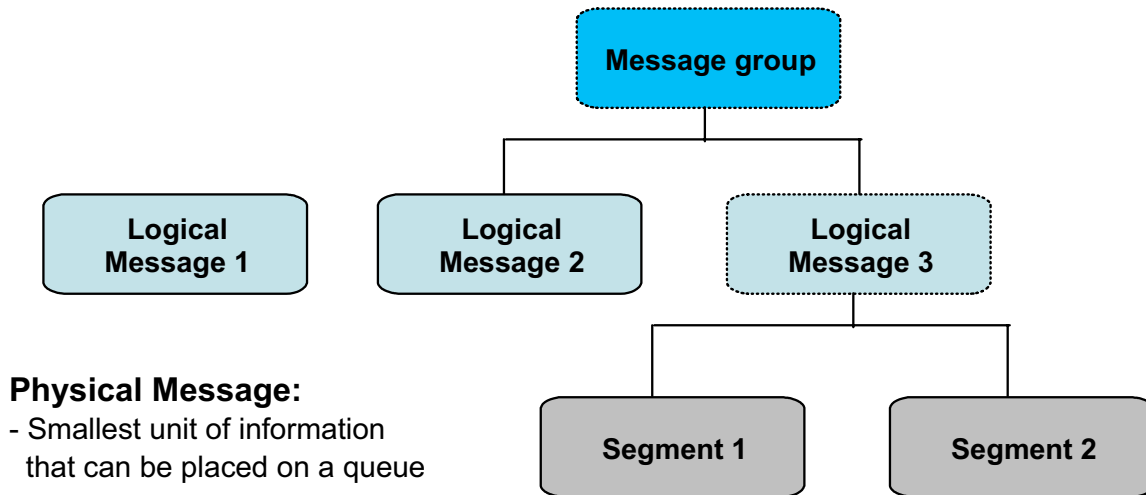
Figure 7-10. Wait Example

MQ092.0

Notes:

7.3 Message Groups

Introduction



Physical Message:

- Smallest unit of information that can be placed on a queue

Logical Message:

- Single unit of application information
- One or more physical messages called segments
- Segments are anywhere in the queue

Message group:

- A set of one or more logical messages, composed of one or more physical messages

Figure 7-11. Introduction

MQ092.0

Notes:

Messages can occur in groups. This enables ordering of messages as well as segmentation of large messages within the same group.

Message segments will be discussed in the next topic within this unit. For now, it is enough to know that a segment is one physical message that, when taken together with other related segments, make up a logical message. Segments are useful when it is necessary to handle messages that are too large for the putting or getting application or for the queue manager.

Logical messages can actually be a physical message as well. If not made up of segments, then a logical message is the smallest unit of information, by definition, a physical message.

A message group is made up of one or more logical messages, consisting of one or more physical messages that have some relationship. We will explore the possible relationships as we proceed.

The Message Group Variables

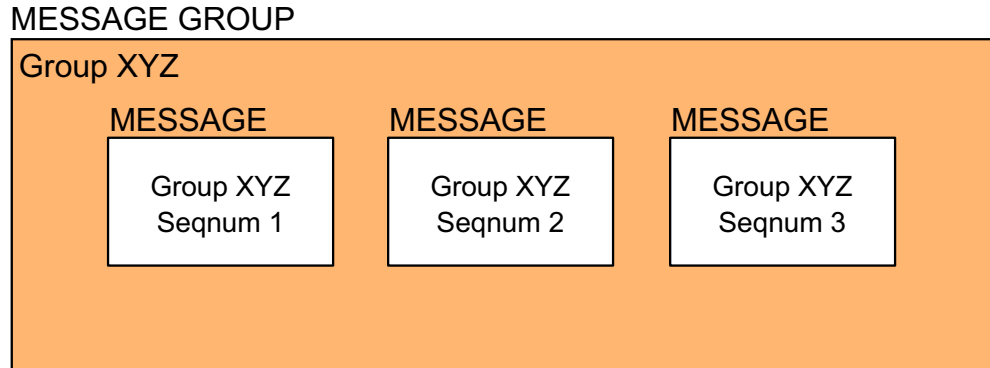


Figure 7-12. The Message Group Variables

MQ092.0

Notes:

The message group variables detail the relationship of this message within a group of messages. They include the name of the group and the position of this message within the chronological order of the group.

These variables are controlled by settings within the `putMessageOptions` object.

Message Groups and the groupId Variable

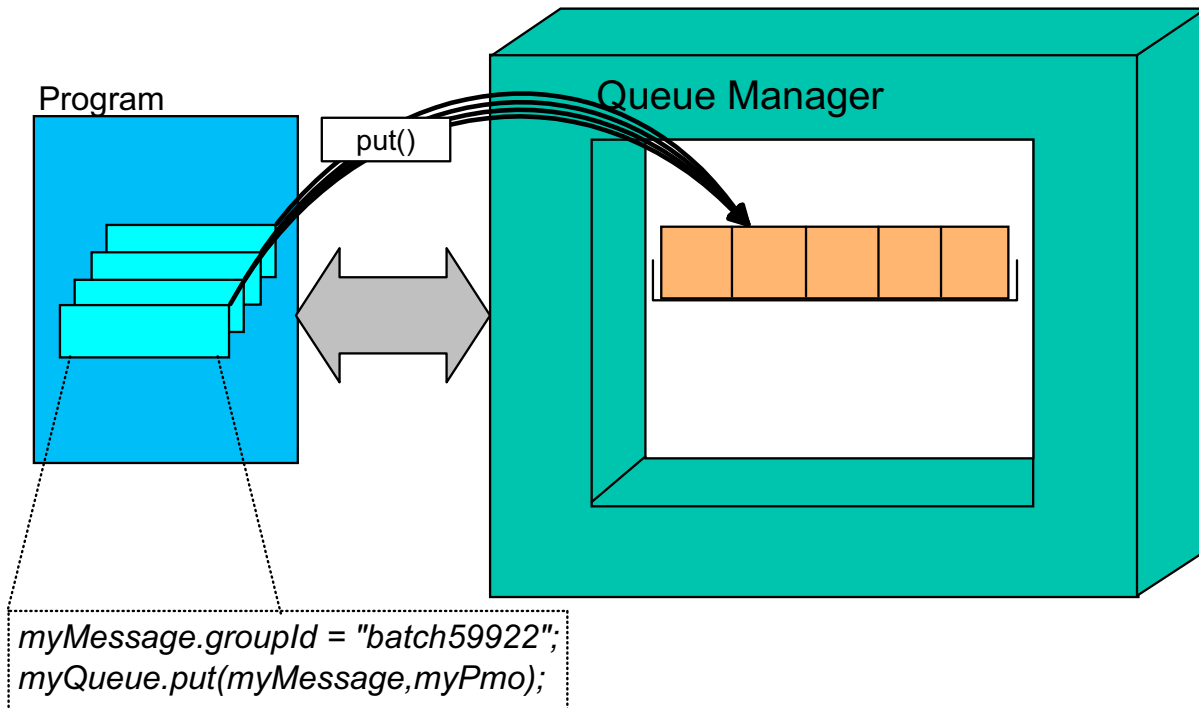


Figure 7-13. Message Groups and the groupId Variable

MQ092.0

Notes:

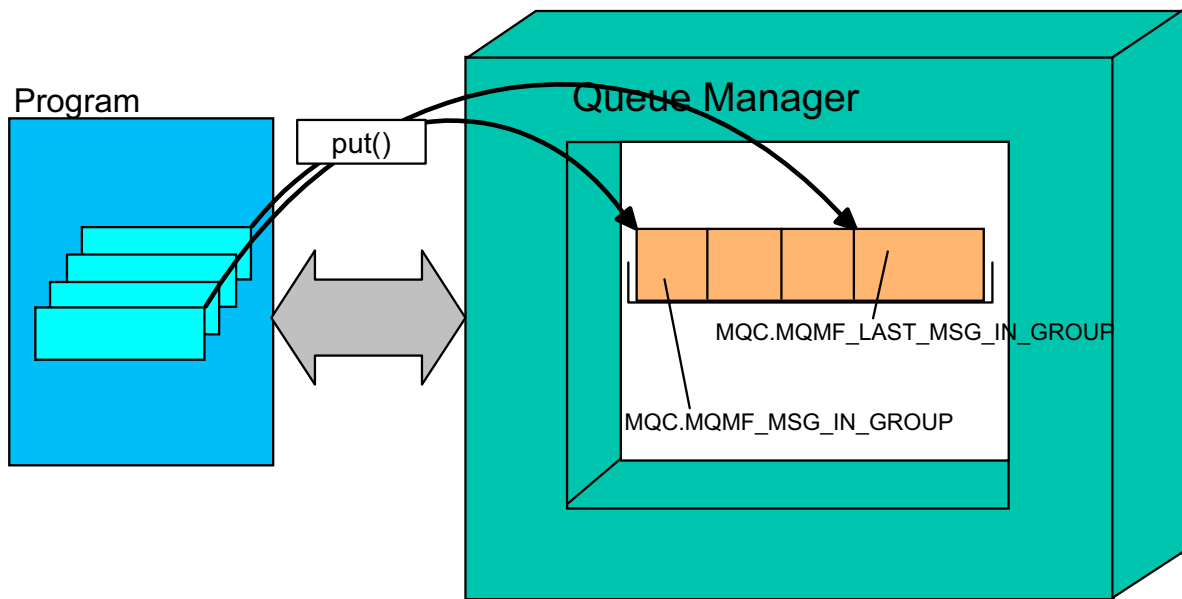
Each message put on a queue has potentially a unique identifier in the messageId variable. This is created by the queue manager when the value assigned by the program is MQC.MQMI_NONE. The correlationId variable can also contain a value, it would be set by the program to a user defined value.

But what happens when a program needs to put a batch of messages on a queue? It needs to identify that the messages are to be treated as a batch, yet still allowing for each message to be individually monitored. The use of a common messageId or correlationId variable may not provide the required level of identification, association and uniqueness.

To solve this problem, MQSeries introduces another message identifier in the form of the groupId variable. This variable can be set to a user value or left to the system to define a unique value.

Several messages with the same groupId form a Message Group.

Putting a Message to a Group



```

myMessage.groupId = MQC.MQGI_NONE;
myMessage.messageFlags=MQC.MQMF_MSG_IN_GROUP;
myPmo.options = MQC.MQPMO_LOGICAL_ORDER;
myOutQue.put(myMessage, myPmo);

```

Figure 7-14. Putting a Message to a Group

MQ092.0

Notes:

On a put request MQSeries will ignore the value of the groupId variable when the MQMessage object does not include the required value in the messageFlags variable.

The default value for the messageFlags variable is MQC.MQMF_NONE this indicates that the message is not a member of a group.

When the message is a member of a group, the messageFlags variable can take the following values:

MQMF_MSG_IN_GROUP indicates that the message is a member of the group identified by the groupId variable.

MQMF_LAST_MSG_IN_GROUP indicates that the message is the last member of the group or is the only message within the group identified by the groupId variable.

Messages put to this queue by this program will be added to the group while the messageFlags value remains set to MQMF_MSG_IN_GROUP. When the message group is to be closed the program specifies the messageFlags value of

MQMF_LAST_MSG_IN_GROUP. Any subsequent messages put to the queue will either be in a new group or not in a group.

In the above example, the groupid is set to MQC.MQGI_NONE. This special value tells the queue manager to allocate a unique groupid and return this value to the program on completion of the put request.

Note also the put message option MQC.MQPMO_LOGICAL_ORDER: this is to indicate to the queue manager to allocate the next logical sequence number for the message that is being put.

Getting a Message from a Group

```
myGmo.waitInterval = 15000;
myGmo.options = MQC.MQGMO_ALL_MSG_AVAILABLE |
                MQC.MQGMO_WAIT;
myInQue.get(yourMessage, myGmo);
if (myGmo.groupStatus == MQC.MQGS_LAST_MSG_IN_GROUP) {...}
if (myGmo.groupStatus == MQC.MQGS_MSG_IN_GROUP) {...}
if (myGmo.groupStatus == MQC.MQGS_NOT_IN_GROUP) {...}
```

Figure 7-15. Getting a Message from a Group

MQ092.0

Notes:

When a program wishes to get a message it issues the get request. If the program is not sensitive to the message group facility it will retrieve the next available message regardless of its group status.

If the program is sensitive to the message group facility it will include on the get request the get message option of MQGMO_ALL_MSG_AVAILABLE. This option indicates to the queue manager that if the selected message is in a group, do not retrieve the message unless the message flag of LAST_MSG_IN_GROUP has been received for this group. To allow for the message group to be received, it is recommended that when using the 'all msg available' option, that the get message option of wait and an appropriate wait interval be also coded.

On successful retrieval of a message the MQGetMessageOptions object has been updated to reflect the status of the message. The groupStatus variable will indicate if the message is in a group, if it is the last message in the group or it is not in a group. If a group consists of only one message, when it is retrieved the groupStatus will be LAST_MSG_IN_GROUP.

The program can request the queue manager to return each message in strict logical order. This order is based on the value of the message sequence number variable. The get message option of MQGMO_LOGICAL_ORDER must be specified on the options variable of the relevant MQGetMessageOptions object. The order is maintained by the queue manager outside the control of the program. If the logical order option is not specified the messages will be returned in the order as specified by the delivery sequence property of the queue definition.

Match Options

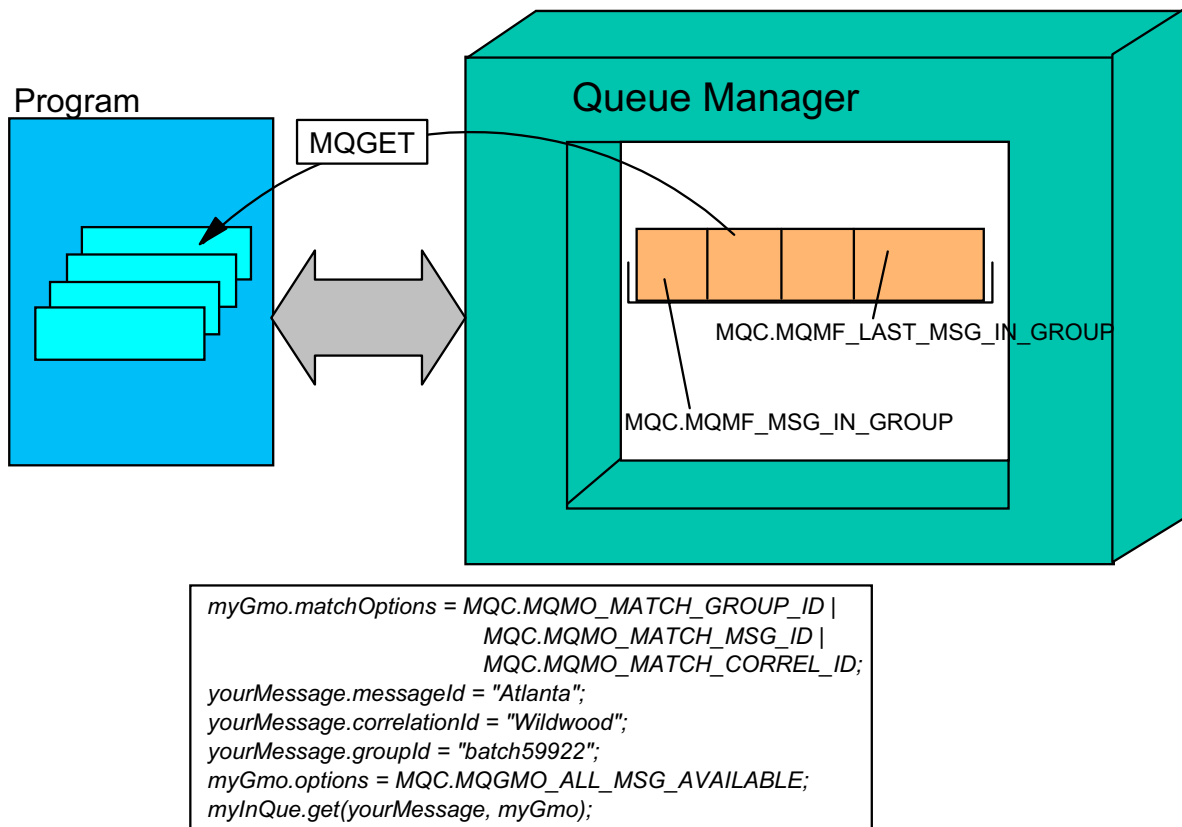


Figure 7-16. Match Options

MQ092.0

Notes:

The MQGetMessageOptions variable `matchOptions` can also be set to allow the program to select the message group it wishes to retrieve messages from. This variable also indicates the message identification selection criteria.

When the `matchOptions` variable is set to `MQMO_MATCH_GROUP_ID`

the value of the `MQMessage` variable `groupId` will be used as the message selection identifier.

The `matchOptions` can also include the specification of both `MQMO_MATCH_MSG_ID` and `MQMO_MATCH_CORREL_ID` in which case a message will not be retrieved unless it matches all three of these values.

Spanning Units of Work

- **Putting application**
 - **Some of the group may be committed before a failure**
 - **Status information must be saved**
 - **Simplest way to use a STATUS queue**
 - **Status information consists of groupid and MsgSeqNum**
 - **STATUS queue is empty if a complete group has been successfully put**
 - **MQPMO_LOGICAL_ORDER cannot be used on the first put of the restart application**
- **Getting application**
 - **STATUS queue again**
 - **First GET of the restart application must match on groupid and MsgSeqNum**

Figure 7-17. Spanning Units of Work

MQ092.0

Notes:

The MQPMO_LOGICAL_ORDER option affects units of work as follows:

If the first physical message in the unit of work specifies MQPMO_SYNCPOINT, then all subsequent physical messages in the group or logical message must use the same option. However, they need not be put within the same unit of work. This permits spreading a message group or logical message that has many physical messages into smaller units of work.

Conversely, if the first physical message has specified MQPMO_NO_SYNCPOINT, then none of the subsequent physical messages within the group or logical message can do otherwise.

If these conditions are not met, the MQPUT will fail with MQRC_INCONSISTENT_UOW. The conditions described for the MQPUT are the same when using the MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT options.

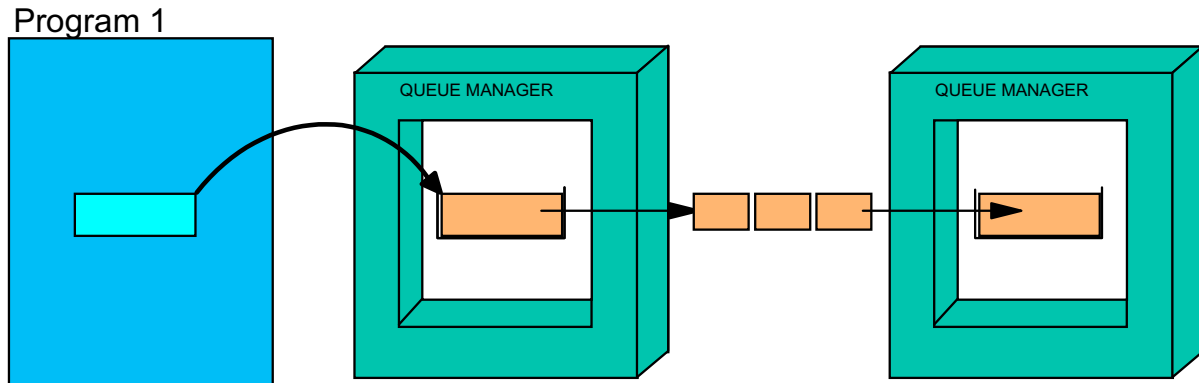
Since it is possible to split a group or logical message over multiple units of work, it is possible to have some of the physical messages but not all committed when a failure

occurs. It is the responsibility of the application to keep track of the status information associated with a group or logical messages that spans units of work. By keeping track of the groupId and the MsgSeqNumber on a special status queue (using syncpoint options for the MQPUT and MQGET), an accurate picture of what has been completed can be kept.

If a restart is done, this is the information that can be used, as discussed earlier, without specifying the Get or Put Message Options option for logical order to essentially restart proper processing of a group or a logical message.

7.4 Message Segments

Segmentation by the Queue Manager



```
myMessage.messageFlags = MQMF_SEGMENTATION_ALLOWED;
myOutQ.put(myMessage,myPmo);
```

Figure 7-18. Segmentation by the Queue Manager

MQ092.0

Notes:

Each message on a queue is variable in length, up to a maximum of 100 megabytes. The MQSeries administrator can specify a smaller maximum size for each queue and also the maximum number of messages that can be stored on a queue. Each queue has a maximum physical size of 2 gigabytes. A message can not be put on a queue if it exceeds any of these settings.

Messages that are destined for a queue that is remotely hosted travel to the remote queue manager via a transmission queue. The message must be able to be stored on the transmission queue else the put request will fail. The MQSeries component that moves messages between queue managers is called the message channel agent (MCA).

When moving messages between queue managers. If the message is larger than can be received by a remote queue, the MCA can break down the message into segments and store each segment as a separate message on the remote queue. Specifying `MQMF_SEGMENTATION_ALLOWED` as the value of the `messageFlag` variable allows the queue manager to segment the message.

If the message is too big for the remote queue and the message indicates that it is not allowed to be segmented, then the message will be put to the dead letter queue on the receiving queue manager. The messageFlags variable set to the specification of MQMF_SEGMENTATION_INHIBITED prevents the message being broken into segments by the queue manager.

Each segmented message contains the same message header, updated to indicate segmentation, followed by the next portion of the message data. Each message sent is known as a physical message. If the message is a segment of a larger message the complete message is referred to as a logical message.

Message Segmentation Variables

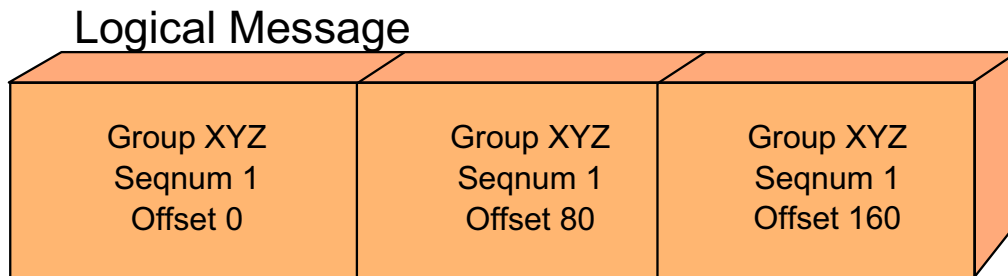


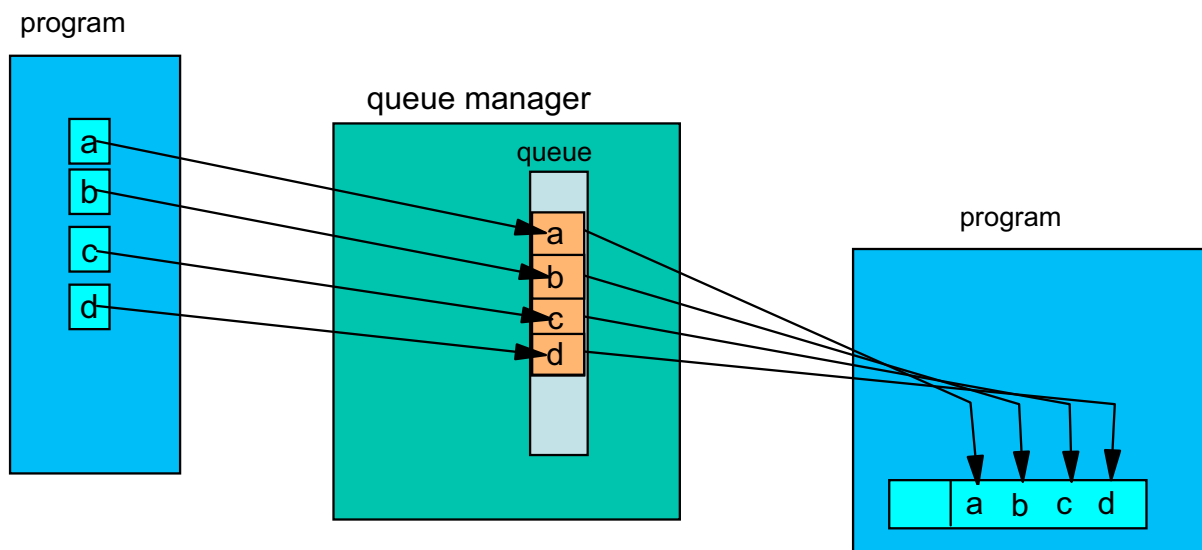
Figure 7-19. Message Segmentation Variables

MQ092.0

Notes:

The message segmentation variables relate to the segmenting of the user buffer information. This involves the creating of multiple physical messages that relate to the same logical message.

Segmentation by the Program



```
myGmo.options = MQC.MQGMO_COMPLETE_MSG
                MQC.MQGMO_WAIT;
myGmo.waitInterval = 15000;
myOutQ.get(yourMessage,myGmo);
```

Figure 7-20. Segmentation by the Program

MQ092.0

Notes:

A program may also segment a message, by putting many messages with the same header all indicating segmentation, followed by the relevant portion of the message data. When getting a message, the program can indicate that if the message is segmented it is to be reassembled into one message. The header from the first message will provide the header for the reassembled message. A program can issue get requests indicating that if the message is segmented it is not to be reassembled, but retrieved as a message. Each message retrieved this way will indicate that it is segmented.

Application segmentation is used for two reasons:

1. Queue manager segmentation is not sufficient because the application buffer is not large enough to handle the full message
2. Data conversion must be performed by sender channels and the putting program needs to split the message on specific boundaries to enable successful conversion of the segments.

The application can include the Put Message Options of MQPMO_LOGICAL_ORDER. The same caution applies to using unit of work processing. As each segment is built and the

MQPUT is executed, the application needs to make sure the messageFlags field in the message descriptor includes either MQMF_SEGMENT or MQMF_SEGMENT_LAST. The queue manager will assign and maintain the groupId, MsgSeqNumber and offset.

If the application does not specify MQPMO_LOGICAL_ORDER, then the program is responsible for ensuring the assignment of a new groupId, as well as proper MsgSeqNumbers and offsets.

If MQGMO_LOGICAL_ORDER is specified in the get message options, all remaining segments for this logical message shall be processed. Not specifying this option can be used for recovery purposes.

Finally, any intermediate applications that are simply passing the data should not use the MQGMO_LOGICAL_ORDER option to ensure the offset field is not corrupted.

Selective Reassembly

- **matchOptions** field in the get message options
 - For selective MQGETs
 - **MQMO_MATCH_MSG_ID**
 - **MQMO_MATCH_CORREL_ID**
 - **MQMO_MATCH_GROUP_ID**
 - **MQMO_MATCH_OFFSET** and **MQMO_MATCH_MSG_SEQ_NUM**
 - Only if **MQGMO_LOGICAL_ORDER** is not specified

- **SegmentStatus** field in the **GMO**
 - Returned on the MQGET
 - **MQSS_NOT_A_SEGMENT**
 - **MQSS_SEGMENT**
 - **MQSS_LAST_SEGMENT**

Figure 7-21. Selective Reassembly

MQ092.0

Notes:

It is possible to control which segment is retrieved by a program as well as have total control over message retrieval by a program. The match options field of the get message options structure allows for this.

The following is very much like the process to retrieve a complete message group. Now we want to learn how to retrieve a complete logical message.

If an application wishes to retrieve a particular logical message, it can begin retrieval of message from the groupId (using match option **MQMO_MATCH_GROUP_ID**), the logical message (using **MQMO_MATCH_MSG_SEQ_NUM**) starting the **MQMD.offset** set to zero, and including **MQMO_MATCH_OFFSET** in the match options of the get message options structure. The **MQMO_MATCH_MSG_SEQ_NUM** is not valid if combined with the **MQGMO_LOGICAL_ORDER**.

Finally, an application can determine if it has processed the final segment of a logical message by checking another field in the get message options structure after a message is retrieved. The **MQGMO.segmentStatus** field would contain a value represented by the symbolic **MQSS_LAST_SEGMENT**. If not, and a group was being processed, the value

would be MQSS_SEGMENT. The groupStatus field could be checked to see if all logical messages within the group had been processed.

7.5 Checkpoint and Summary

Unit Checkpoint

1. The program needs to request the retrieval of a message with the correlation identifier of "abc". What Java object needs to be set?
2. What variable of this object needs to be set?
3. If the program is to get a message, but only if it is currently available, what MQSeries classes for Java object will need to be modified?
4. If the program is to get a message and is willing to wait if the message is currently unavailable, what value will the option variable of the GMO object need to be set to?
5. If the program is to get a message and is willing to wait for a maximum of 7.5 seconds if the message is currently unavailable, what value will be coded on the waitInterval variable?

Notes:

Unit Checkpoint

6. What is the name of the variable that identifies the group status of a message?
7. What is the value that must be specified in the messageFlags variable to identify the message is to belong to a group?
8. What is the messageFlags setting to allow for the segmentation of the message?
9. What is the options specification that will result in only one message being returned, regardless of segmentation?

Notes:

Summary

You should now know how to:

- Get messages selectively using `messageId` and `correlationId`
- Code synchronous programs, using the `wait` and the `waitInterval` functions
- Send batches of messages using message groups
- Segment and reassemble a message

Notes:

Unit 8. More on Messages

What This Unit is About

This unit describes the more advanced features provided by MQSeries to work with messages, including triggering, inquire and set of MQSeries object attributes, data conversion, and sending messages to a distribution list.

What You Should Be Able to Do

After completing this unit, you should be able to

- Trigger programs using the MQSeries trigger function
- Inquire and set MQSeries object attributes
- Handle data conversion
- Write programs that use distribution lists

How You Will Check Your Progress

Accountability:

- Checkpoint
- Machine exercises

References

SC34-5456 *MQSeries Using Java*

SC33-1673 *MQSeries Application Programming Reference*

<http://www.ibm.com/software/ts/mqseries/messaging>
WebSphere MQ

Objectives

- Trigger programs using the MQSeries trigger function
- Inquire attributes of MQSeries objects
- Set queue attributes
- Handle data conversion
- Send messages to a distribution list

Notes:

8.1 Triggering

Triggering Types

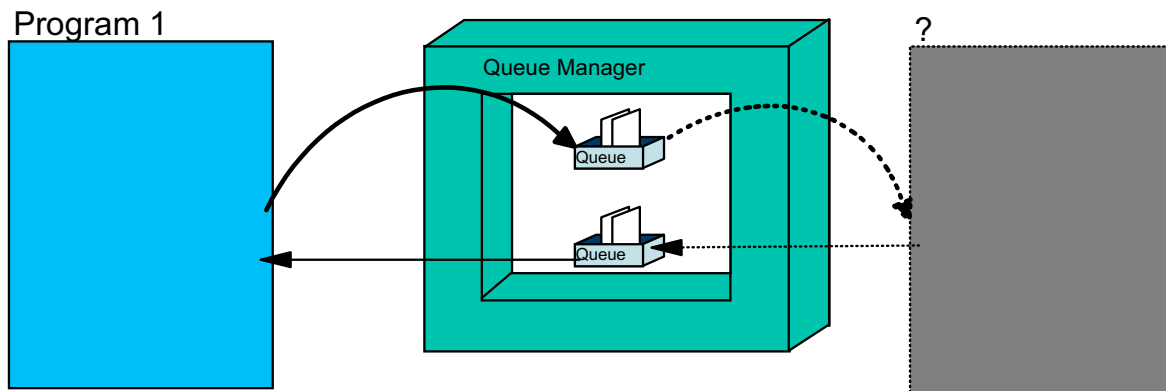


Figure 8-2. Trigger Types

MQ092.0

Notes:

A client program puts a message on a queue and then waits for the response.

Who generates the response?

A server program.

How is the server program started?

The server program may be started automatically, by a feature of MQSeries known as triggering.

There are three types of triggering supported by MQSeries:

The first type is where the server program is activated when the number of eligible messages on the queue reaches a specified value. This form of triggering is known as 'trigger on depth', and is suitable for batch type processing.

The second type of triggering is where the server program is activated when the first eligible message arrives on the queue. This form of triggering is known as 'trigger on first', and is suitable for most types of processing and especially where the arrival of messages is

infrequent. An eligible message is a message that has the priority variable set to a value equal to or greater than the trigger message priority setting of the queue.

The third type of triggering is where an instance of the server program is to be activated each time a message is put on the queue. This form of triggering is known as 'trigger on every', and is suitable for highly available and scalable systems, especially where there is a requirement for each message to be processed by a separate server.

Triggering Characteristics

- Trigger Control
- Trigger Data
- Trigger Depth
- Trigger Message Priority
- Trigger Type

Figure 8-3. Triggering Characteristics

MQ092.0

Notes:

A queue can only support one form of triggering at any time. The type of triggering is controlled by the queue definition as stored on the queue manager. The program can inquire and change the triggering characteristics by the use of the `getxxx` and `setxxx` methods of the `MQQueue` class. To support these methods, the queue must have been opened with the `inquire` and `set` options, else their use will result in an `MQException` being thrown.

Process

```
define process(abcdef) applcid('Rus1')
```

Figure 8-4. Process

MQ092.0

Notes:

The process object is defined to the queue manager by the administration utilities, it details the program name and associated attributes. The name of this object is referenced by the process attribute of a local queue. The process object can be referenced by many local queues, but a local queue can only reference one process object.

You can manage this object using the MQProcess class:

```
public MQProcess (MQQueueManager qMgr,  
                 String processName,  
                 int openOptions,  
                 String queueManagerName,  
                 String alternateUserId)
```

Initiation Queue

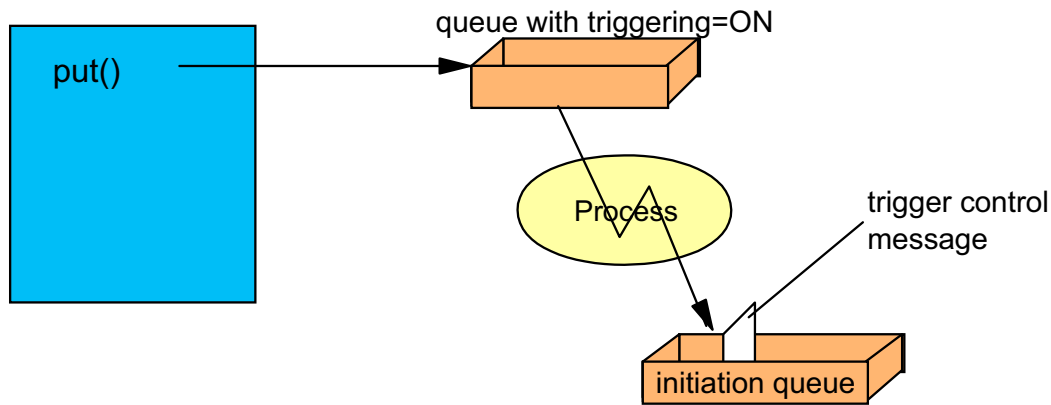


Figure 8-5. Initiation Queue

MQ092.0

Notes:

When defining a queue, you must associate this queue with two other MQSeries objects in order to allow triggering:

The first object that the queue definition refers to is the name of another queue to receive the trigger control messages for this queue. This queue is known as an initiation queue. The name of this queue is contained in the initiation queue attribute of the current queue. To extract this value the inquire method must be used. An MQException will be thrown by this method if the queue is not open or the queue was not opened with the inquire option.

The second object that the queue definition refers to is a process definition. This object describes the attributes of the required server program.

Trigger Monitor

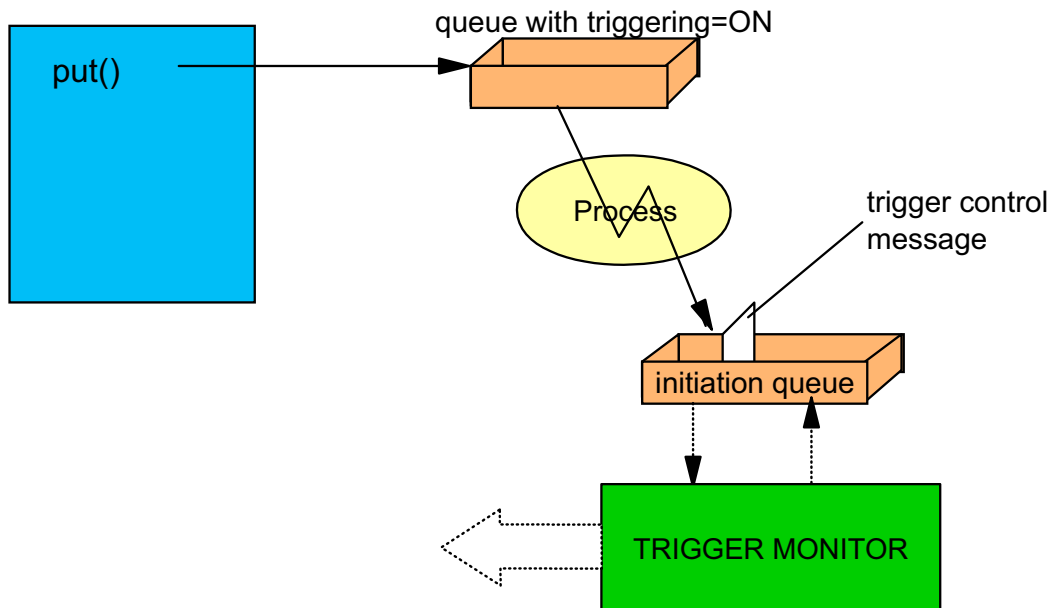


Figure 8-6. Trigger Monitor

MQ092.0

Notes:

The initiation queue must be serviced by a special type of program referred to as a trigger monitor. This program will get messages from the assigned input queue and expect each message to be a MQSeries trigger control message. This format is documented in the structure named MQTM. The message is constructed from information obtained by the queue manager from the input queue and the associated process definition.

The purpose of the trigger monitor program is to activate the required program, as documented by the MQTM message. The MQSeries server product code includes a suitable trigger monitor for general use.

Implementation of Triggering

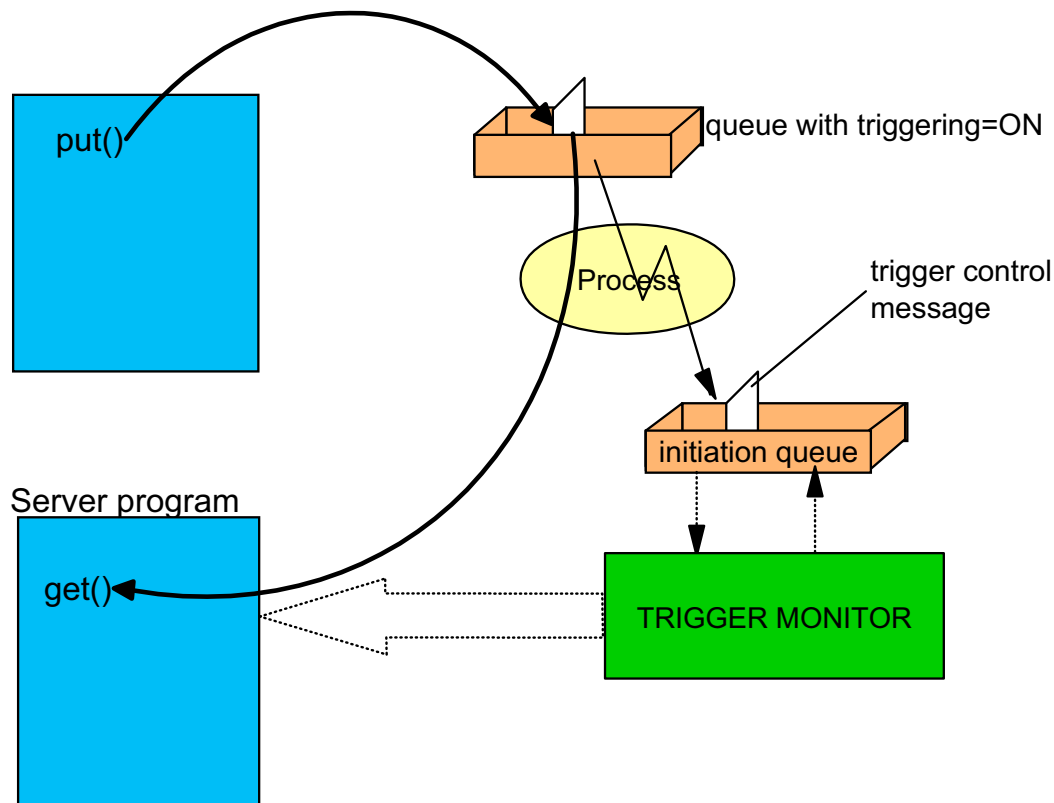


Figure 8-7. Implementation of Triggering

MQ092.0

Notes:

When a program puts a message on a queue, the queue manager will automatically evaluate the queue for a triggering condition. If a trigger condition is raised, the queue manager will create the trigger control message, from the queue definition and the associated process definition and write this new message to the associated initiation queue. Note that no part of the user message that has been placed on the application queue is included in the special control message written to the initiation queue.

The trigger monitor is waiting on the get request, it now receives the message and activates the identified program.

The server program now opens and gets messages from the application queue. This program receives the name of the application queue from the trigger monitor via activation arguments (in fact, the trigger monitor passes information to the started program via a structure called MQTMC2, which contains the name of the triggered queue. This structure is not available for Java, but is described in the Application Programming Reference).

8.2 Inquire and Set Attributes

Inquire and Set Attributes

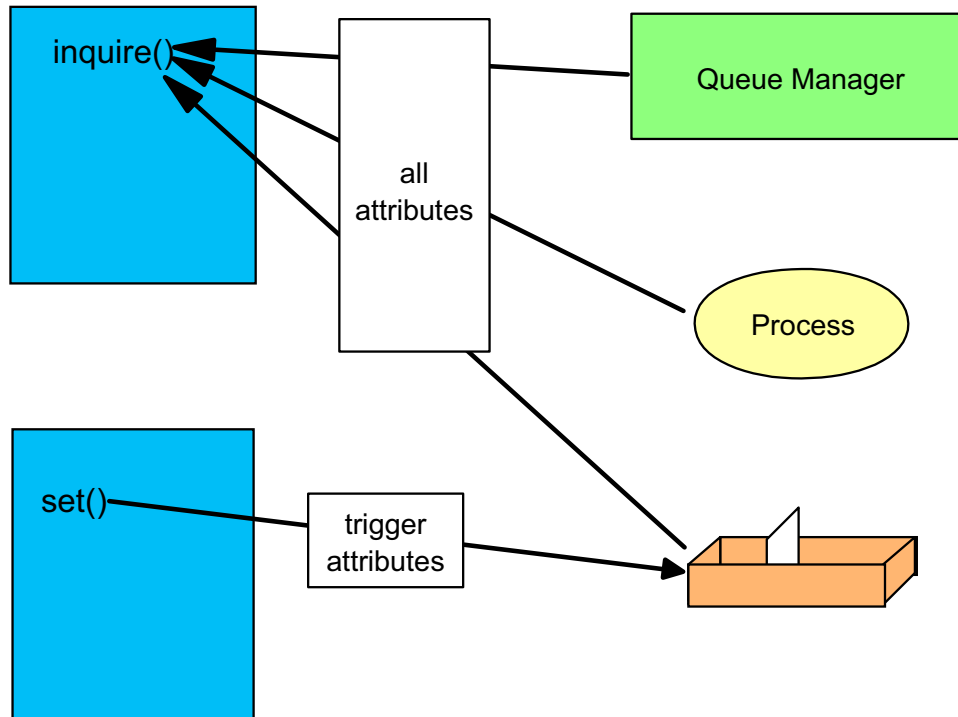


Figure 8-8. Inquire and Set Attributes

MQ092.0

Notes:

Inquire and set methods are used to perform some of the administrative functions within a program.

Inquire allows you to get all of the attributes of any queue, process, queue manager, or namelist. Set allows you to change attributes, but only some attributes of a queue.

Before you use the inquire or set method your application must be connected to the queue manager, and the object must be open, with the open or set open option.

Why Inquire and Set?

- To query the maximum message length for a queue to set up the input buffer size before the get()
- To query the maximum depth and the current depth of a queue
- To query the name of the dead letter queue
- To change a queue with TRIGTYPE=DEPTH from NOTRIGGER to TRIGGER

Figure 8-9. Why Inquire and Set?

MQ092.0

Notes:

Within a program, you use this information to discover the maximum message length for a particular queue, to get the name of the application program pointed to by a process, to find the name of the dead letter queue, or to find the list of attributes for a namelist.

The set method allows to change the queue attributes associated with triggering.

Inquire Attributes

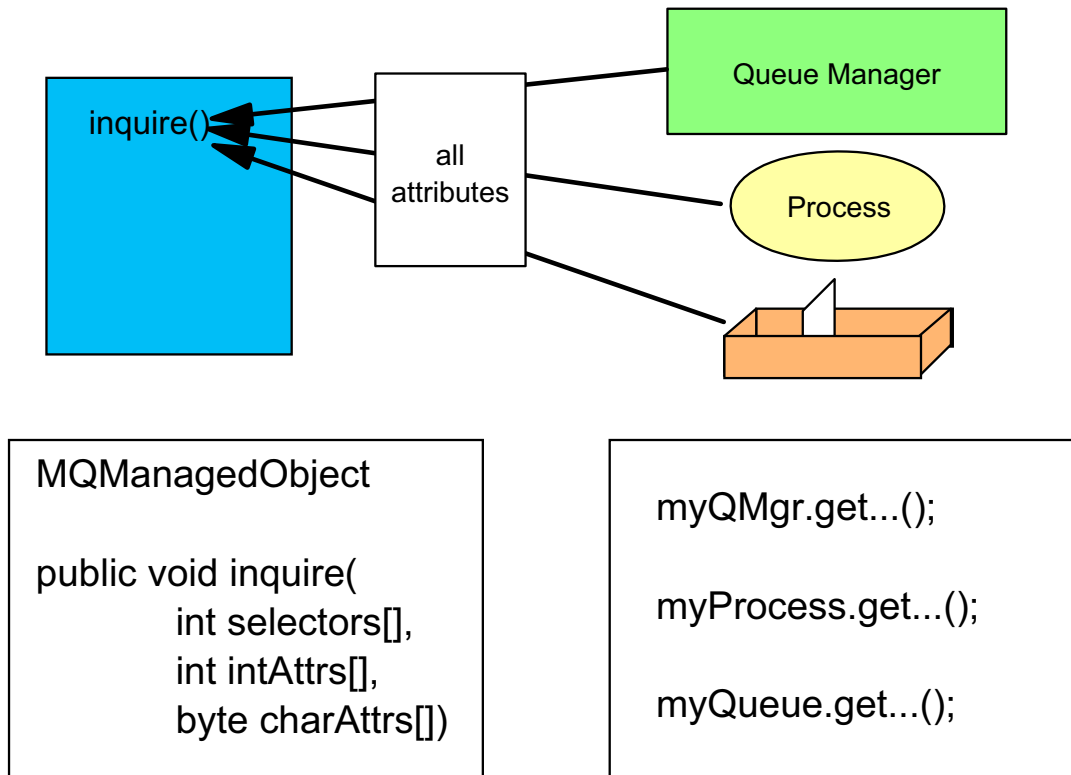


Figure 8-10. Inquire Attributes

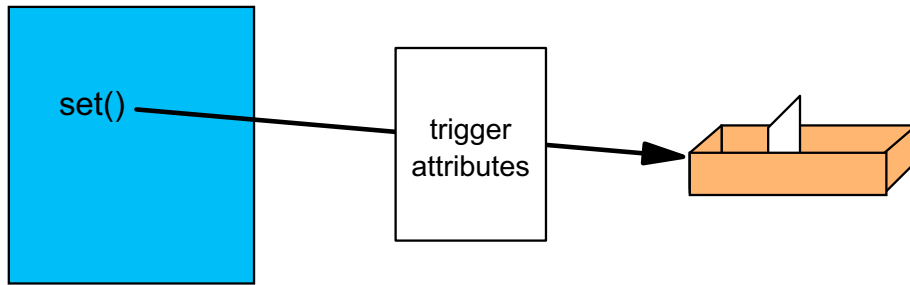
MQ092.0

Notes:

Inquire is a method of the MQManagedObject class. So MQQueue, MQQueueManager, and MQProcess inherit it.

Many of the more common attributes can be queried using the getXXX() methods defined in MQManagedObject, MQQueueManager, MQQueue, and MQProcess.

Set Attributes



MQManagedObject

```
public void set(
    int selectors[],
    int intAttrs[],
    byte charAttrs[])
```

```
myQueue.setInhibitGet(...);
myQueue.setInhibitPut(...);
myQueue.setTriggerControl(...);
myQueue.setTriggerData(...);
myQueue.setTriggerDepth(...);
myQueue.setTriggerMessagePriority(...);
myQueue.setTriggerType(...);
```

Figure 8-11. Set Attributes

MQ092.0

Notes:

Set is a method of the MQManagedObject class. So MQQueue inherits it.

All the queue attributes that can be changed by the program can be set using the setXXX methods defined in MQQueue.

8.3 Data Conversion

Data Formats

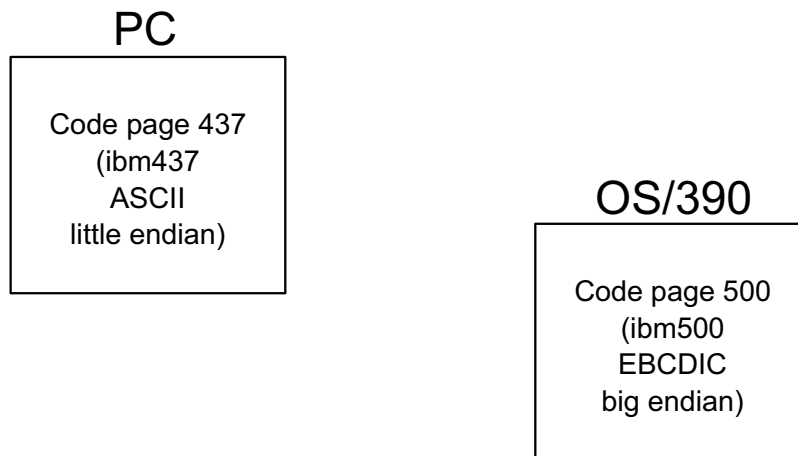


Figure 8-12. Data Formats

MQ092.0

Notes:

When data is moved between different operating system environments it must be checked, to see if the data was stored using the same or similar code pages.

Each operating system has to identify and store the character set coding being used. The value will indicate what code page is being used, whether the data is stored in ASCII or EBCDIC format and what the local language code is set to.

Numeric data is potentially represented differently by each of the operating system environments. Big endian and little endian are standard identifiers for the representation of numeric data. Where big endian compliant operating environments store the data in what is termed as normal format. Little endian compliant operating environments store the data in the reverse order to big endian data. Also note that the system/390 environment has a different length for the storage of large floating point numbers compared to the IEEE standard format.

Java overcomes this by having a standard format for the representation of numeric data regardless of the operating environment.

Selecting the Data Format

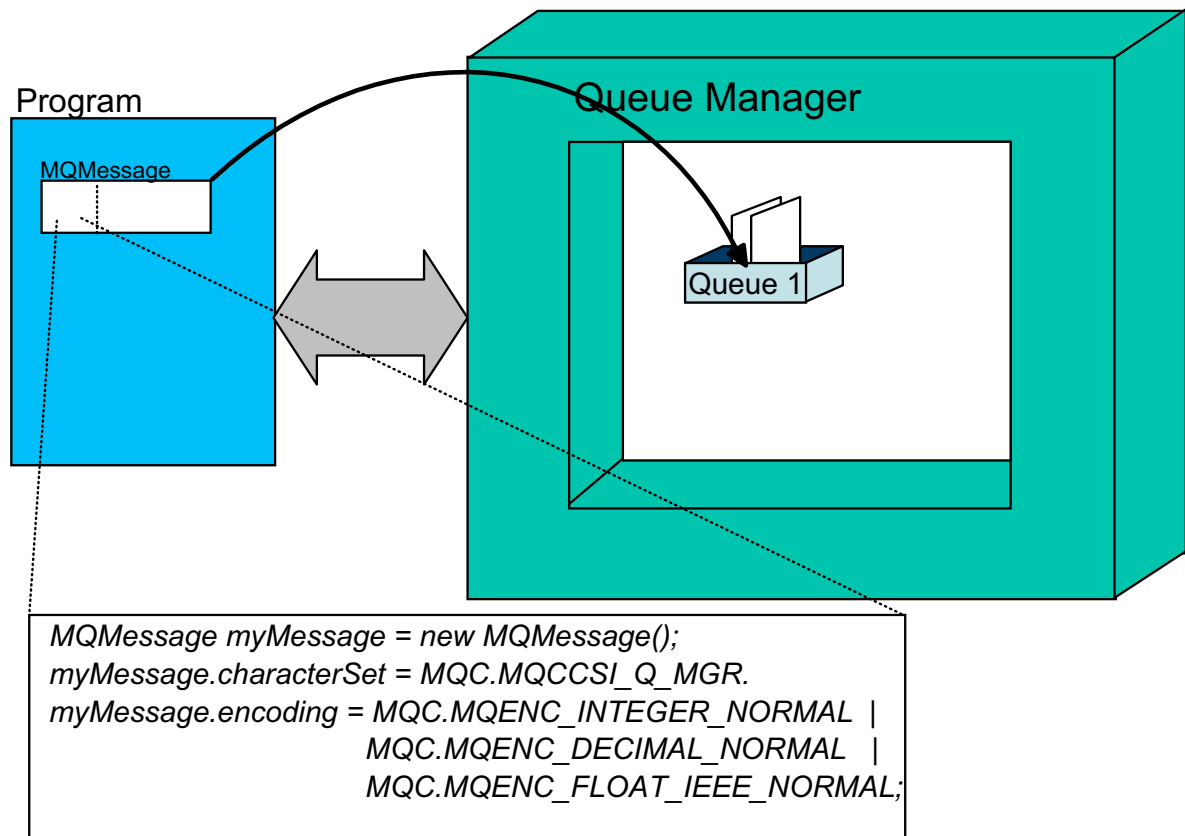


Figure 8-13. Selecting the Data Format

MQ092.0

Notes:

On each operating system environment a local identifier is used to indicate the manner in which data is represented. This indicator is obtained and stored by the queue manager during queue manager creation. The value is made available to the program via the `getCharacterSet` method of the `MQQueueManager` class:

```

int ourCodePage = qMgr.getCharacterSet();
System.out.println("the queue manager is using code page numbered:
" + ourCodePage);

```

When the `MQMessage` object is instantiated the `characterSet` variable is set to the default value of `MQC.MQCCSI_Q_MGR`. This value indicates that the `writexxx` methods, associated with the storing of character based data, are to store that data in the code page format as indicated by the queue manager `characterSet` variable.

The MQSeries product identifies internally the manner in which numeric data is stored by the operating environment. When an `MQMessage` object is instantiated, the `encoding` variable will be set to the default value of `MQC.MQENC_NATIVE`. This value indicates that

integer, decimal and floating point data when written to the buffer by the supplied writexxx methods will be stored in big-endian format.

The queue manager automatically converts the message variables of the message that has been selected, before it is placed in the MQMessage object. This conversion is performed using the characterSet and encoding of the originating and destination systems. The exception to this conversion is that all byte type variables are not converted. The byte type variables are: messageId, correlationId, accountingToken and groupId. As these variables are not converted, care must be taken when analyzing or reporting their value.

Data Conversion by Write or Read Methods

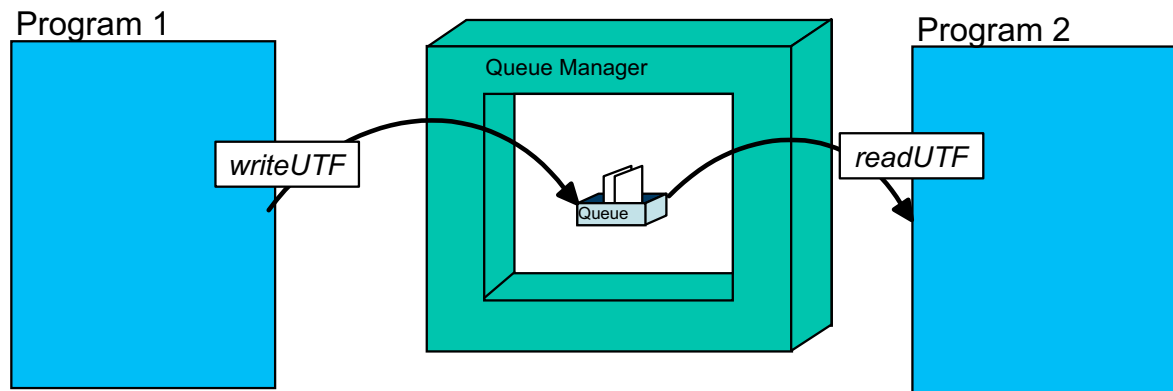


Figure 8-14. Data Conversion by Write or Read Methods

MQ092.0

Notes:

The data written to the message buffer by these writexxx methods will automatically be subject to character and numeric conversion, based on the values of the encoding and characterSet variables. If these two variables are set to values other than the default then the data will be converted into the encoding and characterSet as indicated, if that conversion is supported by this implementation of the MQSeries product.

The data will also be subjected to conversion analysis as it is retrieved by the readxxx methods.

Using the writeUTF() method automatically encodes the length of the string as well as its Unicode bytes. This is then the simplest way to send string data if the message data is to be read by another Java program using the readUTF() method.

Requesting Conversion

```
MQPutMessageOptions myPMO = new MQPutMessageOptions();  
myMessage.format = 'ABC';  
myMessage.encoding = MQC.MQENC_INTEGER_REVERSED |  
                    MQC.MQENC_DECIMAL_REVERSED |  
                    MQC.MQENC_FLOAT_IEEE_REVERSED;  
myMessage.characterSet = 1253;  
myOutputQueue.put(myMessage, myPmo);
```

```
MQGetMessageOptions myGMO = new MQGetMessageOptions();  
myGmo.options = MQC.MQGMO_CONVERT;  
myInputQueue.get(myMessage, myGmo);
```

Figure 8-15. Requesting Conversion

MQ092.0

Notes:

The program can request that MQSeries should check and convert the message data if required. MQGMO_CONVERT is the get message option to request this feature. If the code pages of the sender and the MQMessage object are found to be different, the data contained within the message will be converted to the code page as identified by the characterSet variable of the MQMessage object. This process occurs before the queue manager completes the get request.

The conversion of the message data, as performed by the queue manager in response to the get message option of convert, is controlled by the value of the format variable. The value is set by the message sender to indicate to the receiver the nature of the message data. The format variable is an 8 character value, if the value is set to an MQSeries provided literal, then the queue manager will convert the data using the knowledge it has of that format.

The possible values of the format variable are:

MQC.MQFMT_NONE

This is the default value for the format variable, as there is no value the queue manager will

be unable to convert the user message data. A get request with the convert option will result in an MQException being thrown, if the message encoding or characterSet variables indicate conversion is required.

MQC.MQFMT_STRING

When the format variable is set to this value it indicates to the queue manager that the user message data consists entirely of characters. The queue manager will convert the data character by character, using the sending and receiving character code set conversion tables.

User declared value.

When the value of the format variable is not recognized by the queue manager, it is assumed to be the name of a user supplied conversion exit program. The value will be used in a dynamic link request, if the link fails then the conversion request will fail and an MQException will be thrown by the interface.

8.4 Distribution Lists

What are Distribution Lists?

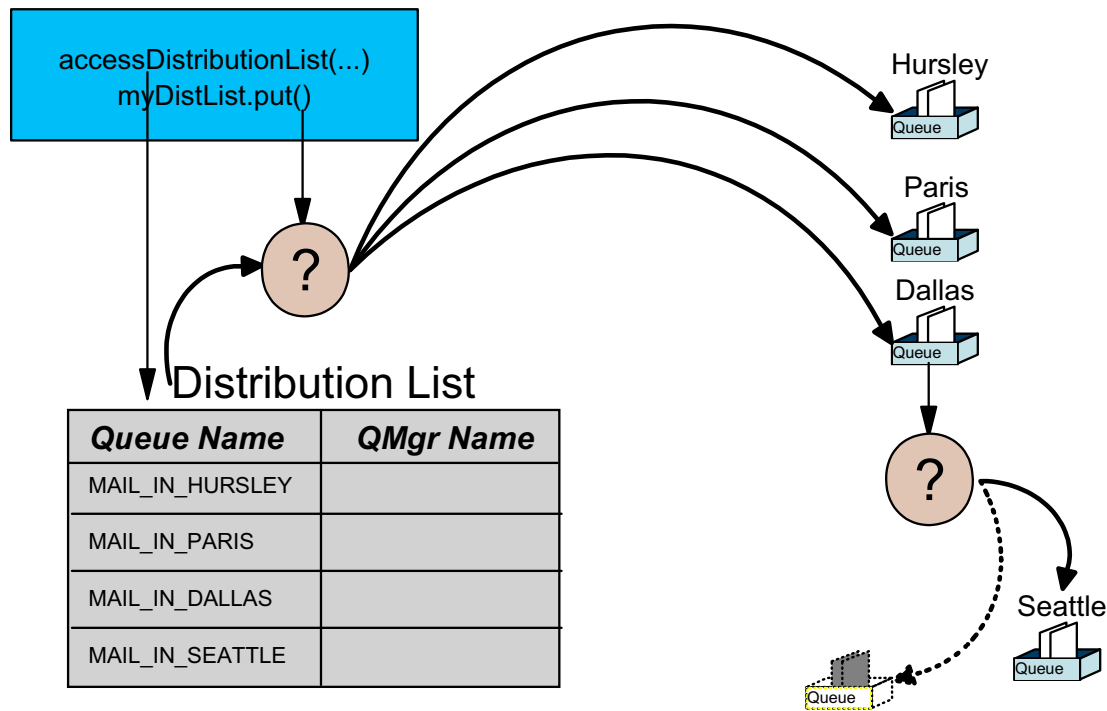


Figure 8-16. What are Distribution Lists?

MQ092.0

Notes:

Prior to availability of the Version 5 products, there was no true distribution list support in MQSeries. Distribution lists allow you to put a message to multiple destinations in a single MQPUT call. Multiple queues can be opened using a single MQOPEN and then a single message can be put to each of those queues using the single MQPUT.

The distribution list feature is implemented by the following MQSeries base classes for Java interface:

- MQDistributionList
- MQDistributionListItem
- MQMessageTracker

Also this feature utilizes the following four variables of the MQPutMessageOptions class:

- knownDestCount
- unknownDestCount
- invalidDestCount
- recordFields

Note: A distribution list can only be used to put a message, it may not be used with the get method.

The distribution list feature is not available on all implementations of MQSeries. Please refer to the documentation associated with the installed version before attempting to use this feature.

Creating a Distribution List

```
MQDistributionListItem[ ] myDLI = new MQDistributionListItem[6];
  for (int i = 0; i < 6; ++ i)
  {
    myDLI[i] = new MQDistributionListItem();
  }
myDLI[0].queueName = "Jim";
myDLI[1].queueName = "Kelly";
myDLI[2].queueName = "Kathy";
myDLI[3].queueName = "Gail";
myDLI[4].queueName = "Vanessa";
myDLI[5].queueName = "Ring";
```

Figure 8-17. Creating a Distribution List

MQ092.0

Notes:

Each queue that is to be opened is named by the value of the queueName variable of an MQDistributionListItem object.

The distribution list items are constructed as elements of an array, where the number of occurrences represents the number of queues that are to be opened.

Opening a Distribution List

```
int oOpts = MQC.MQOO_OUTPUT;
...
MQDistributionList myDL =
myQmgr.accessDistributionList(myDLI,oOpts);
```

Figure 8-18. Opening a Distribution List

MQ092.0

Notes:

The distribution list is created and an open request is issued to the named queues when the `accessDistributionList` request is issued.

The first parameter of this method is an array representing the `MQDistributionListItem` objects.

On completion of the `accessDistributionList`, each element will indicate the outcome of the request in regards to the named queue. An `MQException` will still be thrown on the occurrence of an exception, but as this request references potentially many queues, the `reasonCode` and `completionCode` of the `MQException` object can not reflect each and every queue. Therefore the `completionCode` and `reasonCode` relating to each queue is stored in variables of that name in the associated `MQDistributionListItem` object.

Putting a Message onto a Distribution List

```
MQPutMessageOptions myPMO = new MQPutMessageOptions();  
myMessage.writeChars("SME task completed");  
myDL.put(myMessage,myPMO);
```

Figure 8-19. Putting a Message onto a Distribution List

MQ092.0

Notes:

The put method of the associated distribution list object, will send the message to each of the queues that was successfully opened by the accessDistributionList method.

Error Handling

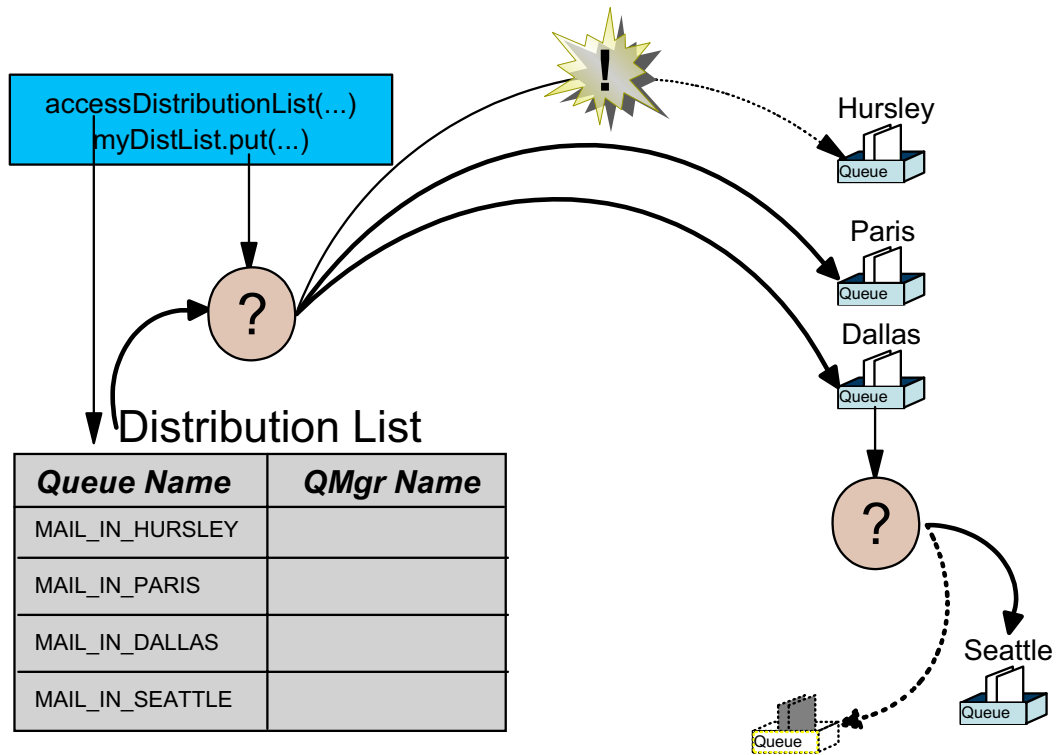


Figure 8-20. Error Handling

MQ092.0

Notes:

Each destination queue is opened separately, meaning some could succeed and others could fail if they are not valid. If all queues open successfully, the completion code returned from the MQOPEN will be MQCC_OK. If all the queues failed to open, it will be MQCC_FAILED. However, if some queues are successfully opened and some fail to open, the completion code will be MQCC_WARNING. In the last two cases, the reason code will be MQRC_MULTIPLE_REASONS.

Therefore, it is necessary to use other means to handle errors: on the unsuccessful completion of the accessDistributionList request, the following methods of the MQDistributionList class can assist in the problem analysis:

```

getFirstDistributionListItem
getNextDistributionListItem
getPreviousDistributionListItem
getValidDestinationCount
getInvalidDestinationCount

```

It is also worth checking the knownDestCount, unknownDestCount and invalidDestCount variables of the MQPutMessageOptions.

Problem Determination

```
myDLItem = myDL.getFirstDistributionListItem();
if (myDLItem.reasonCode == MQException.MQRC.UNKNOWN_Q_NAME) { ..}
```

```
int myCount = myDL.getValidDestinationCount();
if (myCount >= 0)
    myDL.put(myMessage, myPMO);
```

```
int myCount = myDL.getInvalidDestinationCount();
if (myCount >= 0)
    System.out.println(mycount + " queues failed to open. ");
```

```
myDLItem = myDL.getFirstDistributionListItem();
if (myDLItem.reasonCode == MQException.MQRC.UNKNOWN_Q_NAME) { ..}
loop
myDLItem = myDLItem.getNextDistributedItem();
if (myDLItem.reasonCode == MQException.MQRC.UNKNOWN_Q_NAME) { ..}
```

Figure 8-21. Problem Determination

MQ092.0

Notes:

The `getFirstDistributionListItem` method of the `MQDistributionList` class will return the first item associated with the list. Once returned, the variables of this item are available for analysis.

The `getNextDistributionListItem` method of the `MQDistributionListItem` class will return the next item associated with this list of items. Once returned, the variables of this item are available for analysis.

The `getPreviousDistributionListItem` method of the `MQDistributionListItem` class will return the previous item associated with this list of items. Once returned, the variables of this item are available for analysis.

The `getValidDestinationCount` method of the `MQDistributionList` class will return an integer, set to the number of items in the distribution list that were opened successfully.

The `getInvalidDestinationCount` method of the `MQDistributionList` class will return an integer, set to the number of items in the distribution list that failed the open request.

8.5 Checkpoint and Summary

Unit Checkpoint

1. Does the queue manager start a program?
2. Is the trigger event controlled by the message data?
3. Is the trigger event controlled by a message variable?
4. Is the message that results in the trigger event passed to the 'triggered' program?
5. List all the queue attributes that can be set?
6. In distribution list processing, what is the name of the class that holds the names of the queues to be opened?
7. What is the name of the class that represents the opened list of queues?

Notes:

Summary

You should now know how to:

- Use the triggering function of MQSeries
- Inquire and set attributes of MQSeries objects
- Handle data conversion
- Send messages to a distribution list

Notes:

Unit 9. Security

What This Unit is About

This unit describes the security related functions of the MQSeries base classes for Java.

What You Should Be Able to Do

After completing this unit, you should be able to

- Understand the message context variables.
- Manipulate the context fields.
- Use the alternate user authority function.

How You Will Check Your Progress

Accountability:

- Checkpoint
- Machine exercises

References

SC34-5456 *MQSeries Using Java*

SC33-1673 *MQSeries Application Programming Reference*

<http://www.ibm.com/software/ts/mqseries/messaging/>
WebSphere MQ

Objectives

- Understand the context variables of the message object
- Save the context fields in a program and then pass those fields on to another message
- Use alternate user authority to allow one user to send a message on behalf of another

Notes:

9.1 Local Security

User Identification

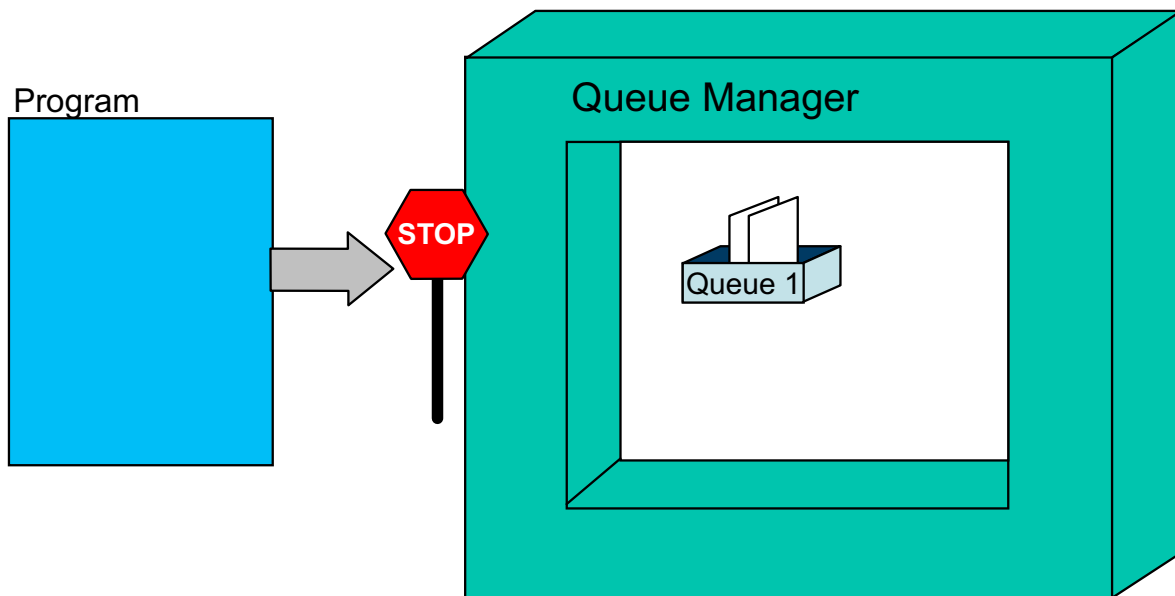


Figure 9-2. User Identification

MQ092.0

Notes:

When a program connects to a queue manager, the current user identity is checked for the authority to connect to this queue manager.

In the OS/390 environment the type of program is also checked for the authority to connect to this queue manager.

The queue manager does not include code for the checking of security or authority. Instead it calls upon an external environment specific authority or security manager. On OS/390 this manager is a SAF enabled security manager, for example the IBM RACF product. On the distributed platforms the authority manager is supplied, except on OS/2, as a component of the MQSeries product and is referred to as the Object Authority Manager.

When the program opens the queue, via the `accessQueue` method, the current user identification is checked by the object authority manager or external security manager as appropriate for the environment. The user must have the authority to open the queue for all the purposes as described by the open options.

For the Java program the options are values of the options integer as specified on the accessQueue request. If the user is not authorized for any of the open options, the accessQueue request will throw an MQException and the reasonCode variable will be set to MQC.MQRC_NOT_AUTHORIZED.

If the queue object being opened is an alias queue, the authority checking is performed on the name as specified on the accessQueue request and not the queue object that is the target of the alias definition. When the put method is issued, the messages are put on the target of the alias queue and not the alias queue.

9.2 Context Variables

The Context Variables

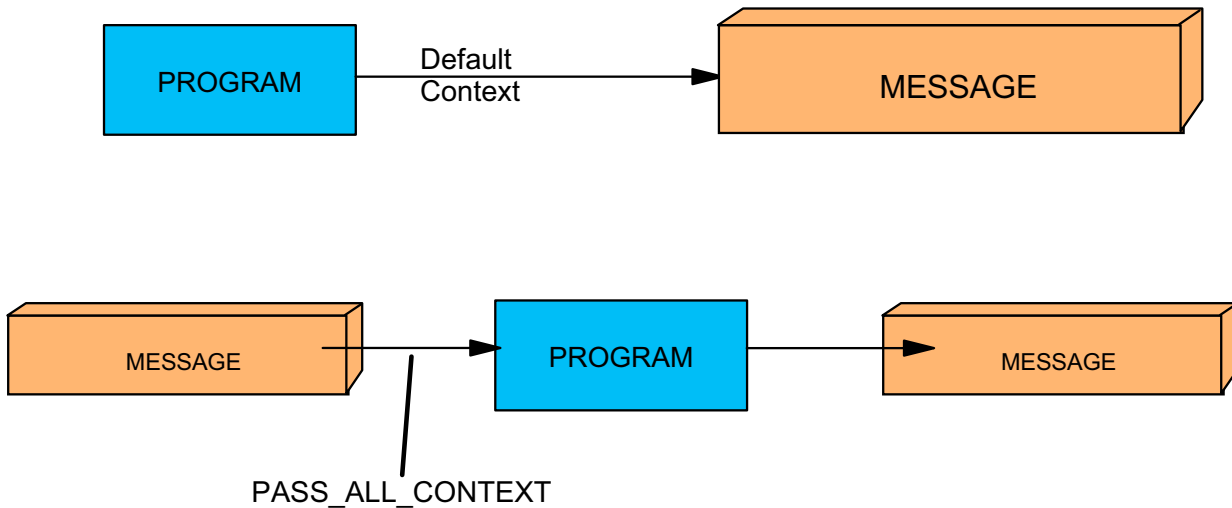


Figure 9-3. The Context Variables

MQ092.0

Notes:

The context variables are seven variables that collectively are managed as the context fields of the message descriptor. The actions performed on these variables is managed by options specified on the open and put requests.

After the put request has completed successfully by default these variables will reflect the environment of this program.

Typically, application programs will not explicitly update the context fields. The default behavior will be that the queue manager plugs in all of the fields. In fact, usually, an application will not be authorized to update the context fields.

In the case of an application that is simply passing a message along, it is possible (if authorized) to pass context information from an input message to an output message.

Context variables include userid, and information concerning the putting application.

Context Properties of a Message

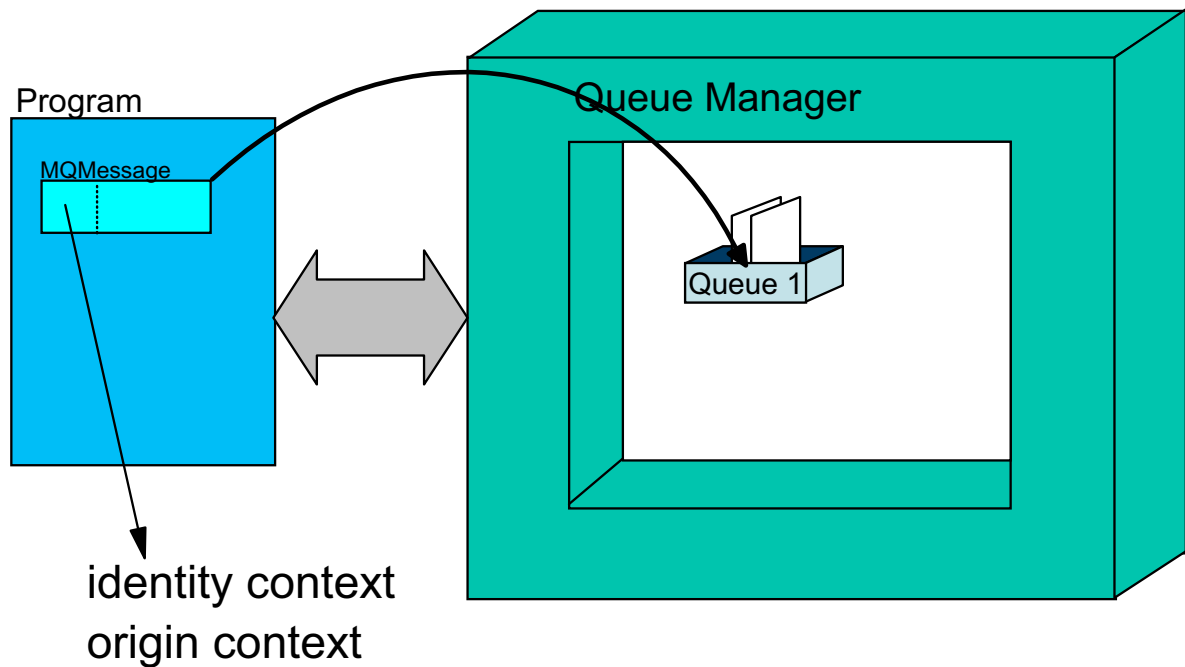


Figure 9-4. Context Properties of a Message

MQ092.0

Notes:

When a message is placed on the queue the setting of all the message variables has been completed. A group of the message variables are collectively known and managed as the context variables. The context variables indicate to the receiver of the message information pertaining to the originator of the message. For manageability the context variables are further categorized as either being associated with the identity or origin.

The identity context identifies the user, and consists of three variables:

`userId`

The identity of the user, as set at the time of the put request, is stored in this variable of the `MQMessage` object. The string value has a maximum length of 12 characters.

`accountingToken`

The `accountingToken` is a 32 byte variable that the queue manager treats as a string of bits.

The queue manager does not check the content of this variable on the get request. The value is set by the queue manager, at the time of the put request, as follows. The first byte of the field is set to the length of the accounting information present in the bytes that follow,

this length is in the range zero through thirty, and is stored in the first byte as a binary integer. The second and subsequent bytes (as specified by the length field) are set to the accounting information appropriate to the environment. The last byte of the accountingToken is identified as the accounting-token type.

applicationIdData

The applicationIdData variable is a 32 character value, the contents are defined by the program. This variable can be used to provide additional information about the message or its originator. The queue manager does not set the value to anything other than blank, unless context manipulation options are used on the put request.

The origin context identifies the program that created the message, and consists of four variables:

putApplicationType

This variable is an integer that can be set by the queue manager to indicate the type of program that has put the message. The value can also be set by a program when the correct put message options are coded.

putApplicationName

This variable is a 28 character string value that names the program that created the message.

putDateTime

The date and time that the message was put on the queue. The variable is stored as a GregorianCalendar type. The queue manager records all date and time values relative to Greenwich Mean Time. The Java classes provide the conversion between the putdate and puttime attributes of the MQMD object and the putDateTime variable of the MQMessage object.

applicationOriginData

The applicationOriginData variable is a 4 character string, the value of which is set by the program. The queue manager makes no use of this variable other than accepting the value set by the program when the put is issued with the appropriate put option.

Manipulating the Context Variables

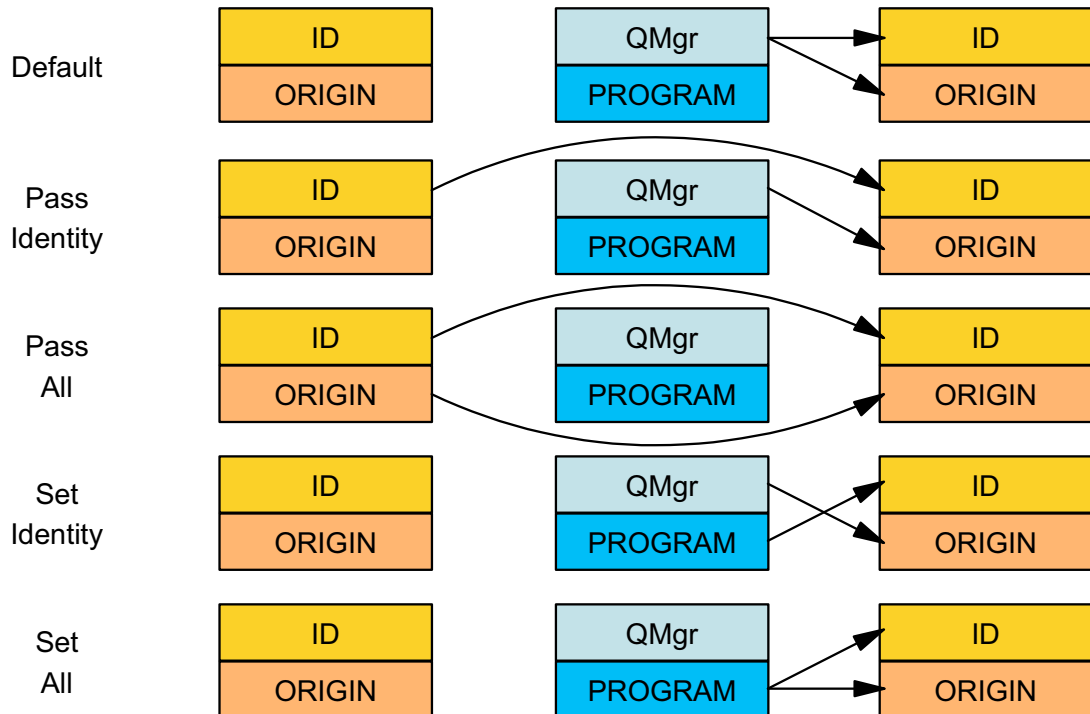


Figure 9-5. Manipulating the Context Variables

MQ092.0

Notes:

Depending on the Put Message Options, the context variables will contain different values after a put:

If the program sets the options variable of the MQPutMessageOptions object to include the option of MQC.MQPMO_DEFAULT_CONTEXT. The queue manager determines the values of the context variables from the environment and not from the MQMessage object. The values are returned to the program on completion of the put request. This context option is the default setting if no context option is specified.

If the program sets the options variable of the MQPutMessageOptions object to include the option of MQC.MQPMO_NO_CONTEXT. The queue manager sets both the origin and identity context variables to null. The values are returned to the program on completion of the put request.

If the program sets the options variable of the MQPutMessageOptions object to include the option of MQC.MQPMO_SET_IDENTITY_CONTEXT. The queue manager accepts the values of the identity context variables as coded in the MQMessage object, and determines

from the environment the values for the origin context variables. The values are returned to the program on completion of the put request.

(For the program to use the put option of MQC.MQPMO_SET_IDENTITY_CONTEXT, the option of MQC.MQOO_SET_IDENTITY_CONTEXT must have been included in the open options specified on the accessQueue request.)

If the program sets the options variable of the MQPutMessageOptions object to include the option of MQC.MQPMO_SET_ALL_CONTEXT. The queue manager accepts the values of all the identity and origin context variables from the MQMessage object.

(For the program to use the put option of MQC.MQPMO_SET_ALL_CONTEXT, the option of MQC.MQOO_SET_ALL_CONTEXT must have been included in the open options specified on the accessQueue request.)

If the program sets the options variable of the MQPutMessageOptions object to include the option of MQC.MQPMO_PASS_IDENTITY_CONTEXT. This request must also include the reference to another MQMessage object that had previously received a message. The queue manager copies from the specified MQMessage object the saved identity context variables. Then sets the values of the origin context variables from the current environment. The values are returned to the program on completion of the put request.

(For the program to use the put option of MQC.MQPMO_PASS_IDENTITY_CONTEXT, the option of MQC.MQOO_PASS_IDENTITY_CONTEXT must have been included in the open options specified on the accessQueue request.)

If the program sets the options variable of the MQPutMessageOptions object to include the option of MQC.MQPMO_PASS_ALL_CONTEXT. The queue manager copies all the values of the context variables from the referenced MQMessage object. No values are taken from the current environment or from the MQMessage object being put on the queue. The values are returned to the program on completion of the put request.

(For the program to use the put option of MQC.MQPMO_PASS_ALL_CONTEXT, the option of MQC.MQOO_PASS_ALL_CONTEXT must have been included in the open options specified on the accessQueue request.)

Note that you need special authorization to modify the context.

Pass Context Example Code

```
...
oOpts = MQC.MQOO_SAVE_ALL_CONTEXT |
        MQC.MQOO_INPUT_AS_Q_DEF;
myInputQueue = myQMGr.accessQueue("Q1", oOpts);

oOpts = MQC.MQOO_PASS_ALL_CONTEXT |
        MQC.MQOO_OUTPUT;
myOutputQueue = myQMGr.accessQueue("Q2", oOpts);

MQMessage retrievedMessage = new MQMessage();
MQGetMessageOptions gmo = new MQGetMessageOptions();
myInputQueue.get(retrievedMessage, gmo);

MQPutMessageOptions pmo = new MQPutMessageOptions();
pmo.options = MQC.MQPMO_PASS_ALL_CONTEXT;
pmo.contextReference(myInputQueue);
myOutputQueue.put(retrievedMessage, pmo);
...
```

Figure 9-6. Pass Context Example Code

MQ092.0

Notes:

9.3 Alternate User ID

Alternate User ID

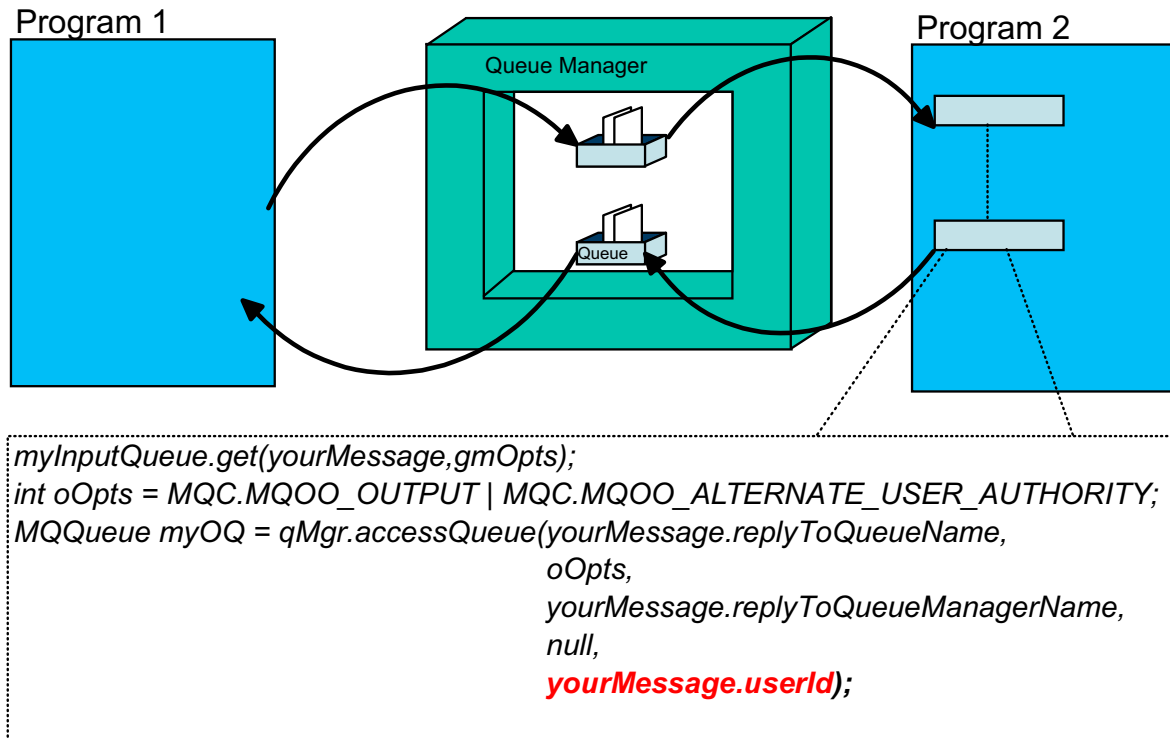


Figure 9-7. Alternate User ID

MQ092.0

Notes:

The server program gets a message from the input queue, services the message then responds to the client by putting a reply message on the output queue. The server is told the name of the output queue by the client program in the value of the `replyToQueueName` variable of the `MQMessage` object.

For the subsequent `accessQueue` request to be successful, the associated userid must be authorized to perform the specified options on the named queue. The question is: will the userid assigned to the server program need authority to open any queue for output processing?

MQSeries provides for a program to specify an alternate userid on the open request. This feature allows the server to specify the client's userid, as passed in the `MQMessage` variable `userId`, instead of its own userid. It is the client that has named the queue for the reply message to be put on, it should be therefore authorized to request this. MQSeries does not check that a userid is authorized to use the queue named as the value of the `replyToQueueName` variable. The authority over a queue is checked only during the `accessQueue` request.

The open option `MQC.MQOO_ALTERNATE_USER_AUTHORITY` must be included on the `accessQueue` request else the value specified as the alternate userid parameter will be ignored by the queue manager. The userid assigned to the program must be authorized to issue the `accessQueue` request with the open option of `MQC.MQOO_ALTERNATE_USER_AUTHORITY`. MQSeries will issue an authority request to the external security manager to verify that the current userid has this authority.

9.4 Checkpoint and Summary

Unit Checkpoint

1. Which methods can throw an MQException with the reasonCode variable set to MQC.MQRC_NOT_AUTHORIZED?
2. List all context variables.
3. Which Put Message Option will result in a message with null values in the context variables?

Notes:

Summary

- Context fields are contained within every message and can be used for security
- Alternate user authority can be used to allow one user to send messages on behalf of another user

Figure 9-9. Unit Summary

MQ092.0

Notes:

Unit 10. Units of Work

What This Unit is About

This unit describes how to handle units of work in MQSeries Java applications.

What You Should Be Able to Do

After completing this unit, you should be able to

- Work under syncpoint control.
- Implement local units of work.
- Implement global units of work.

How You Will Check Your Progress

Accountability:

- Checkpoint
- Machine exercises

References

SC34-5456 *MQSeries Using Java*

SC33-1673 *MQSeries Application Programming Reference*

<http://www.ibm.com/software/ts/mqseries/messaging/>
WebSphere MQ

Objectives

- Work under syncpoint control
- Implement local units of work
- Implement global units of work

Figure 10-1. Unit Objectives

MQ092.0

Notes:

10.1 Local Units of Work

Unit of Work

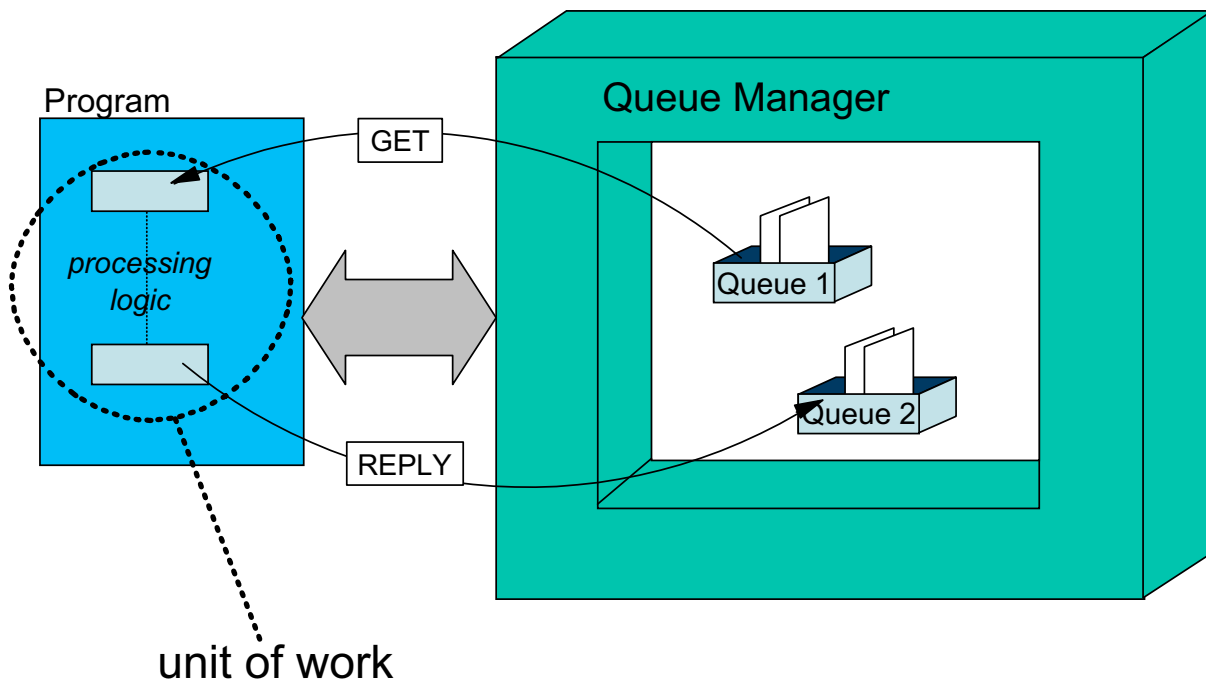


Figure 10-2. Unit of Work

MQ092.0

Notes:

In the above example, the program is designed to get a message, process the message and then respond by putting a reply message. This could then be termed to be our unit of work, as it encapsulates the total processing required by the program in servicing a message.

If the program is interrupted before it is able to complete the processing of the message, it would want the system to reverse the actions that have already been completed. The term 'backout' is used to describe the process of reversing the actions performed within the 'unit of work'. In this example the action that needs to be reversed is the getting of the message. The program issued the get request without the browse option, therefore the message was deleted.

To make this a system managed unit of work, the program would need to enable the system to reverse that part of the work that has been processed. The system that performs the management of the unit of work is referred to as the transaction manager or unit of work manager. The transaction manager treats all work as automatic encapsulations of unit of work.

The unit of work is started by the updating of a recoverable resource and is completed by the closing of the unit of work. For MQSeries the recoverable resource can only be a message. The closing of the unit of work is done by a method of the MQQueueManager class. The unit of work can not exceed the program disconnecting from the queue manager.

Implementing the Local UOW Processing

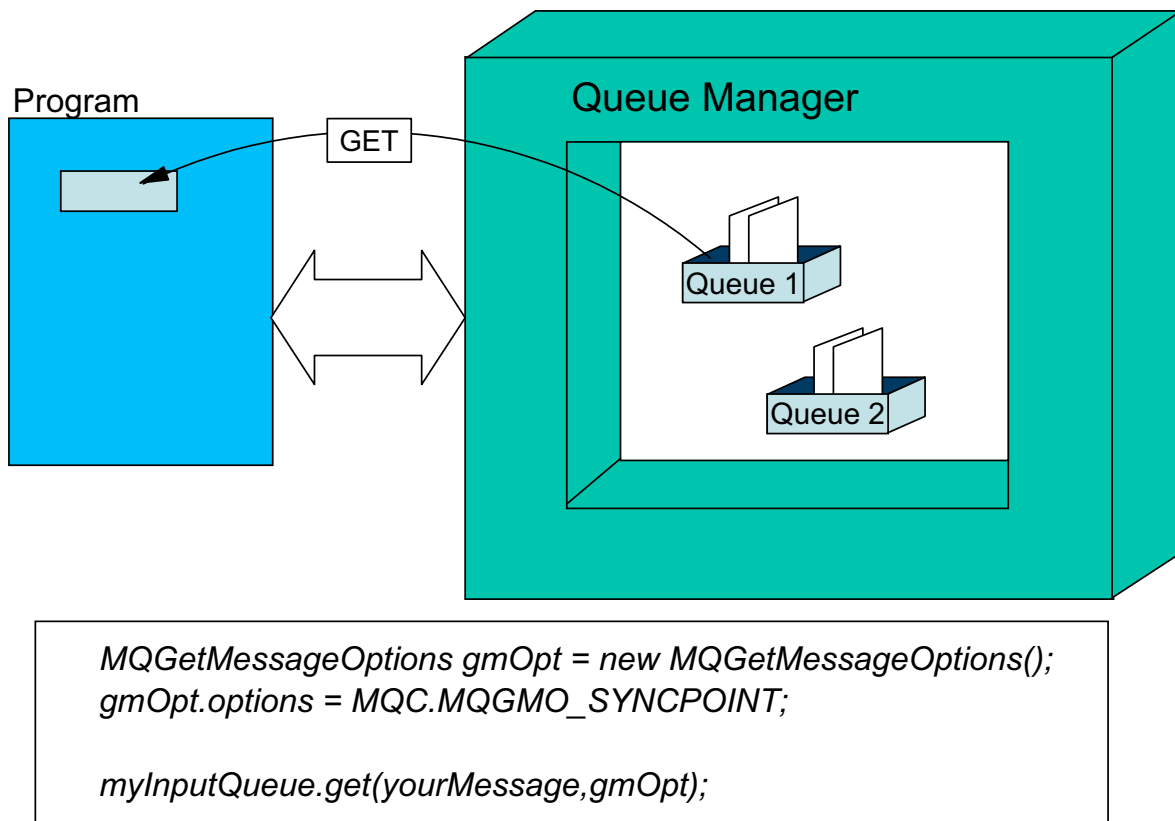


Figure 10-3. Implementing the Local UOW Processing

MQ092.0

Notes:

The get method, without the browse option, is a destructive request as the message will be deleted from the queue, upon successful completion of the get request. The automatic deletion of the message can be deferred by the inclusion of an option to include this message in the current unit of work:

MQC.MQGMO_SYNCPOINT

The options variable of the Get Message Options needs to be set to the syncpoint option to enable the message to be included in the unit of work. The message will be returned to the program but the deletion of the message from the queue will be deferred until the work unit is completed. If the program is interrupted before completing the unit of work, the effect of the get will be reversed and the message will become available for retrieval again.

MQC.MQGMO_SYNCPOINT_IF_PERSISTENT

The options variable of the get message options object can be set to this form of the syncpoint option. This enables the message to be included in the unit of work, but only if the message is also marked as persistent. If the message is not marked as persistent then the message will not be included in the unit of work.

MQC.MQGMO_NO_SYNCPOINT

If the options variable is set to the no_syncpoint option then the message will be deleted automatically by the queue manager on the successful completion of the get request. The message will not be included in the unit of work. The actions performed by the get request will not be reversible, regardless of the completion status of the program.

These options are also available in the Put Message Options:

MQC.MQPMO_SYNCPOINT

MQC.MQPMO_NO_SYNCPOINT

Commit

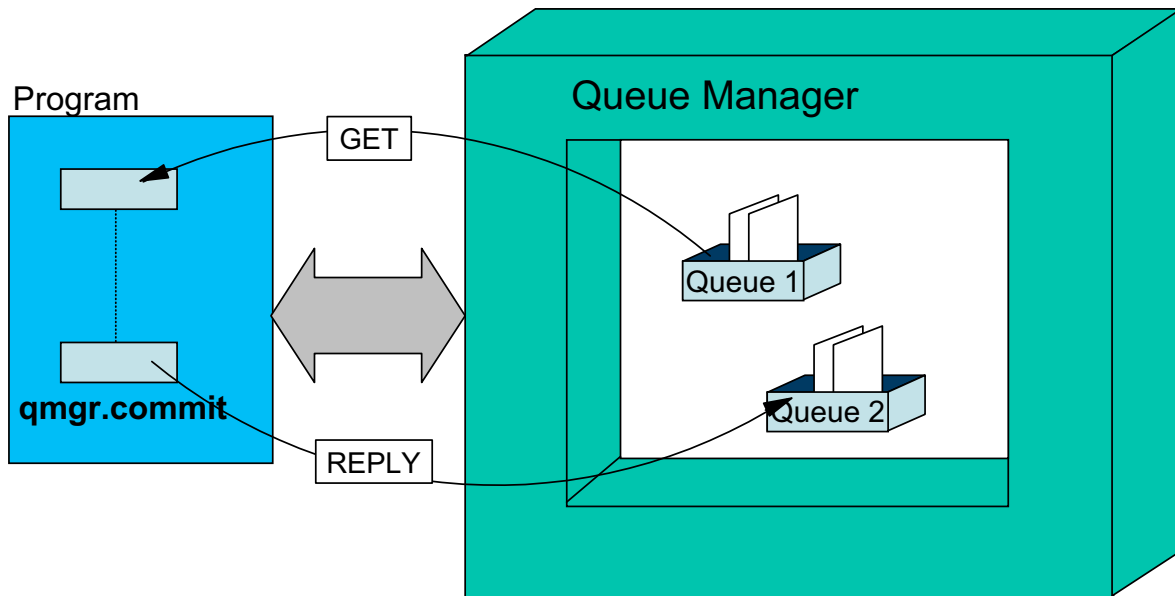


Figure 10-4. Commit

MQ092.0

Notes:

```
qMgr.commit();
```

The commit method of the queue manager class indicates that the program has completed the required processing and that the queue manager is to close the current unit of work held open by this program. The closing of the current unit of work will release the messages that have been held by the syncpoint option. The issuing of the commit method indicates to the queue manager that the actions of this program are not to be reversed after the completion of this request.

Backout

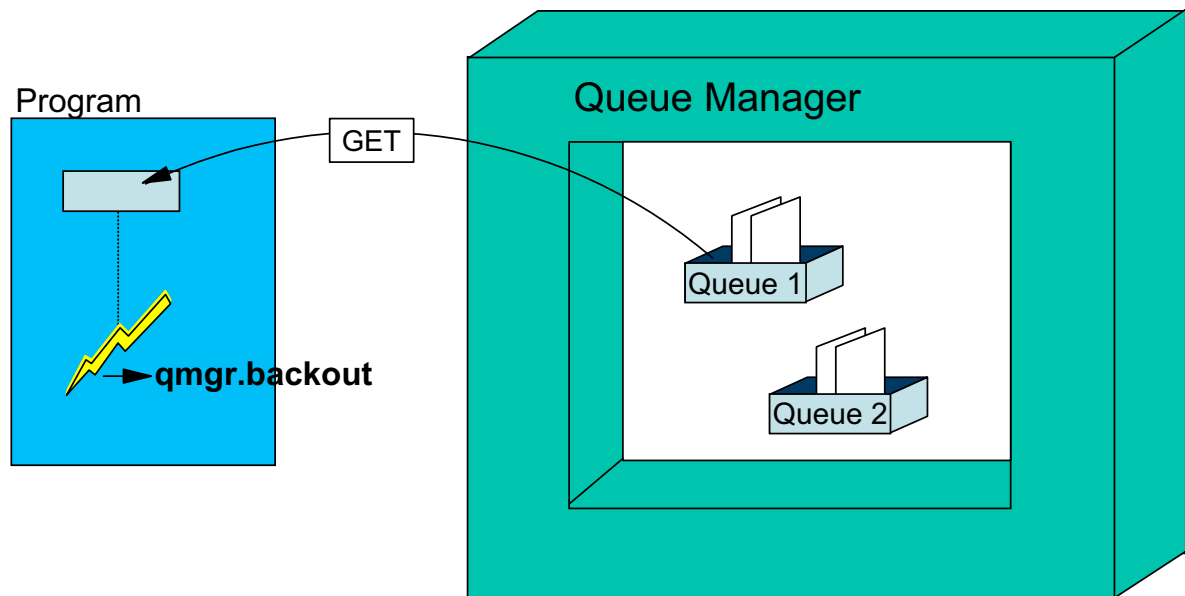


Figure 10-5. Backout

MQ092.0

Notes:

```
qMgr.backout();
```

The backout method of the queue manager class will reverse all the changes made by this program that are currently held with the syncpoint option. The affect of this method is to put messages back on the queues that have been retrieved, and delete from the queues those messages that have been put. But only if they were processed by this program within the current unit of work with the syncpoint option.

```
myMessage.backoutCount
```

Each time the actions of the get method are backed out and the deletion of a message is reversed, the backout counter variable of the message is incremented.

This action is dependent on an attribute of the queue definition, named the harden backout count. If it is set to yes the counter will be incremented automatically. If it is set to no, then the backout counter will always be zero regardless of the number of times it is backed out.

10.2 Global Units of Work

Implementing the Global UOW Processing

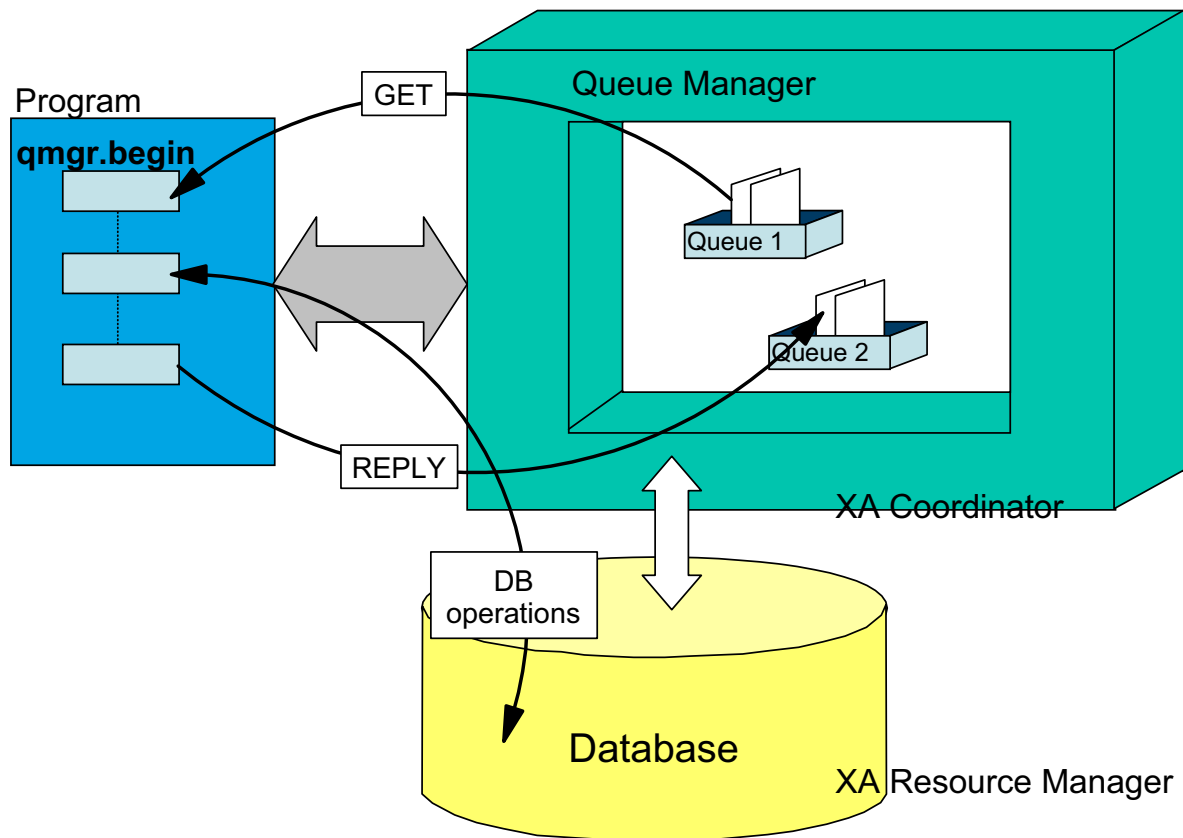


Figure 10-6. Implementing the Global UOW Processing

MQ092.0

Notes:

```
qMgr.begin();
```

The begin method of the MQQueueManager class begins a unit of work that is coordinated by the queue manager, and that may involve external resource managers. The begin method will establish the queue manager as a global unit of work coordinator with an external resource manager. This other manager will be an XA compliant client attached database environment.

This allows the SQL statements to be included within the same unit of work as the MQSeries requests without the need for a transactional manager. When the commit and backout requests are issued the effect of the begin method will be to also commit the database changes made by this program within this unit of work.

Global UOW Example Code

```
...  
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();  
...
```

Figure 10-7. Global UOW Example Code

MQ092.0

Notes:

Considerations for Global UOW Processing

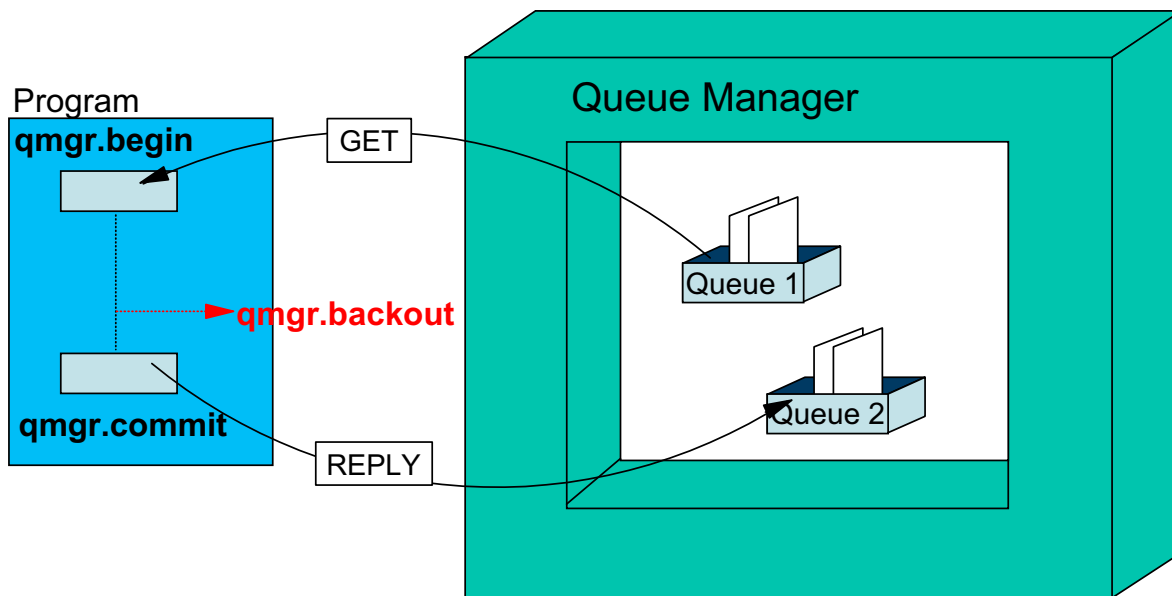


Figure 10-8. Considerations for Global UOW Processing

MQ092.0

Notes:

The begin method can not be requested while connected to the queue manager via the client bindings.

The begin method can not be requested while a unit of work is current.

The begin method request will fail if no external resource manager is identified to the queue manager.

The begin, commit and backout methods can not be requested while the program is managed by a transactional manager, example CICS Transaction Server for OS/390.

If a unit of work is still current when the connection to the queue manager is disconnected, the unit of work will be backed out.

When the Java program is run under the control of the CICS Transaction Server of OS/390 product the program should issue the `com.ibm.cics.server.Task.commit()` method to complete the unit of work or the `com.ibm.cics.server.Task.rollback()` method to backout the current unit of work. These requests must be issued before the program terminates, else the unit of work will be backed out.

10.3 Checkpoint and Summary

Unit Checkpoint

1. What is the get option that will defer the deletion of the message that has been retrieved by the get request?
2. What is the name of the method that will complete a unit of work?
3. What is the type of object that is associated with this method?

Notes:

Summary

You should now know how to:

- Code programs under syncpoint
- Implement lokal units of work
- Implement global units of work

Notes:

Unit 11. Exits

What This Unit is About

In this unit, you will learn how to code user channel exits in order to provide your own send, receive and security exits.

What You Should Be Able to Do

After completing this unit, you should be able to

- Work with send exits.
- Work with receive exits.
- Work with security exits.

How You Will Check Your Progress

Accountability:

- Checkpoint

References

SC34-5456 *MQSeries Using Java*

[http://www.ibm.com/software/ts/mqseries/messaging/
WebSphere MQ](http://www.ibm.com/software/ts/mqseries/messaging/WebSphere MQ)

Objectives

- Code user channel exits
- Provide own send, receive and security exits

Figure 11-1. Unit Objectives

MQ092.0

Notes:

11.1 Exits

Channel Exits

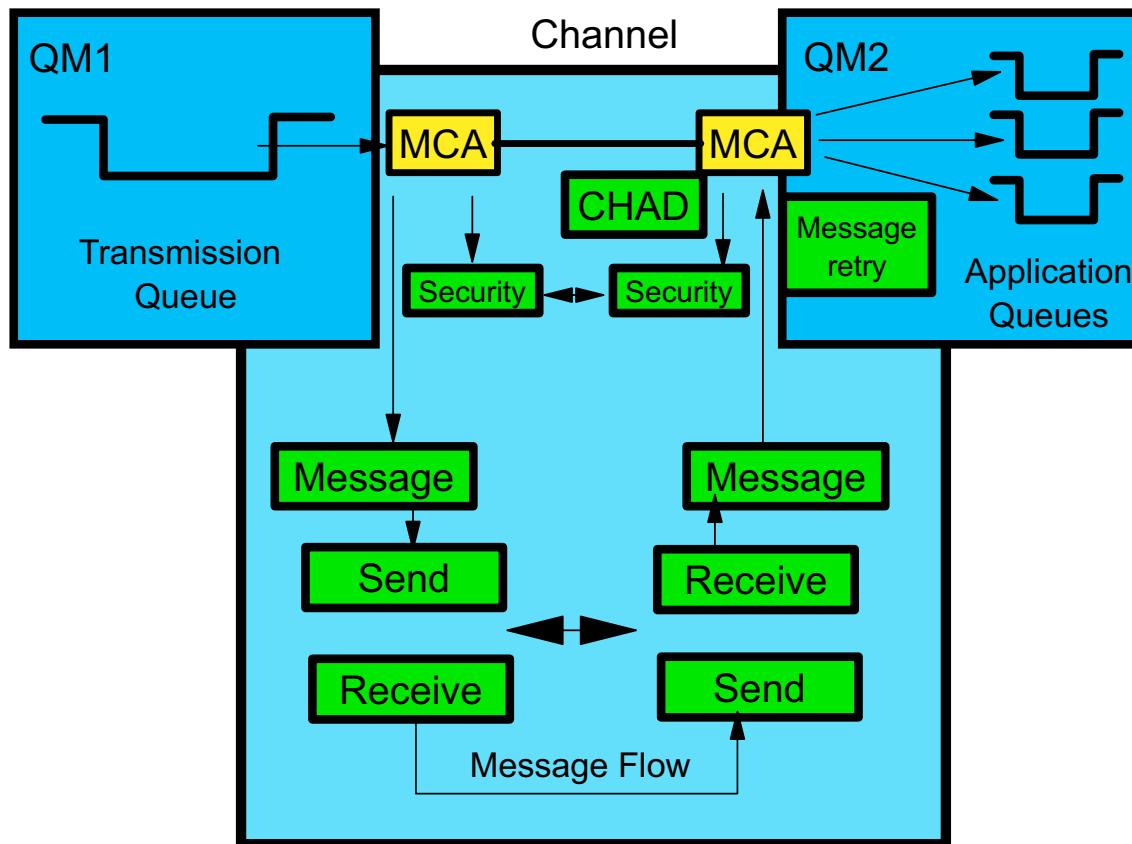


Figure 11-2. Channel Exits

MQ092.0

Notes:

MCAs (Message Channel Agents) are MQSeries applications that transmit messages from the transmission queue of the sending queue manager to one or more application input queues belonging to the target queue manager. A pair of MCAs is known as a channel.

MQSeries channels provide six exit points that enable a channel to be customized:

- Security Exit
- Message Exit
- Send Exit
- Receive Exit
- Message Retry Exit
- Channel Auto-definition Exit

Client Channel Exits

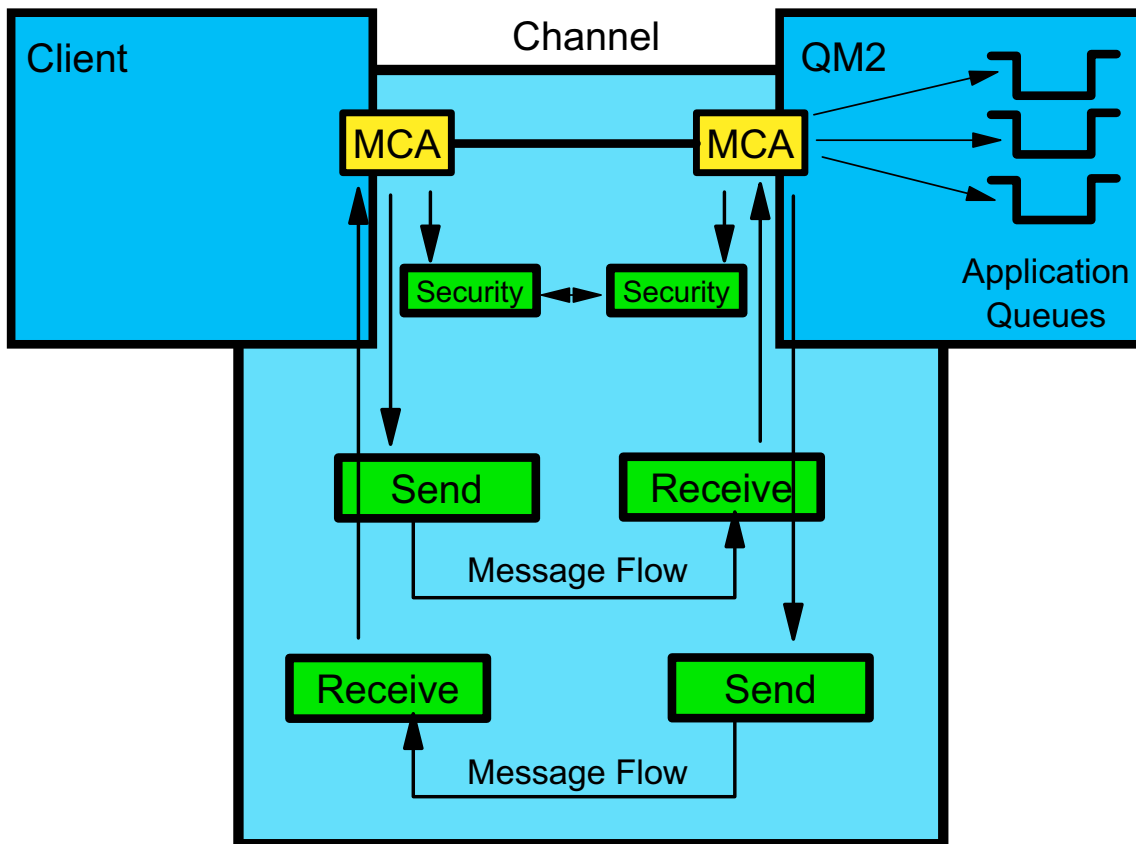


Figure 11-3. Client Channel Exits

MQ092.0

Notes:

MQI channels provide three exit points that enable a channel to be customized:

- Security Exit
- Send Exit
- Receive Exit

The MQSeries classes for Java allow you to write your own Send, Receive and Security exits. To implement an exit, you define a new Java class that implements the appropriate interface. There are three exit interfaces defined in the MQSeries package:

- MQSendExit
- MQReceiveExit
- MQSecurityExit

MQSendExit

```
// in MySendExit.java
class MySendExit implement MQSendExit {
    // you must provide an implementation of the sendExit method
    public byte[] sendExit (
        MQChannelExit      channelExitParms,
        MQChannelDefinition channelDefinition,
        byte[]              agentBuffer)
    {
        // your exit code goes here...
    }
}

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

Figure 11-4. MQSendExit

MQ092.0

Notes:

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

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

To provide your own send exit, define a class that implements this interface. Create a new instance of your class and assign the MQEnvironment.sendExit variable to it before constructing your MQQueueManager object.

In the example above, note the following parameters:

channelExitParms

Contains information regarding the context in which the exit is being invoked. The exitResponse member variable is an output parameter that you use to tell the MQSeries classes for Java what action to take next.

channelDefinition

Contains details of the channel through which all communications with the queue manager take place.

MQReceiveExit

```
// in MyReceiveExit.java
class MyReceiveExit implement MQReceiveExit {
    // you must provide an implementation
    //of the ReceiveExit method
    public byte[] receiveExit (
        MQChannelExit      channelExitParms,
        MQChannelDefinition channelDefinition,
        byte[]              agentBuffer)
    {
        // your exit code goes here...
    }
}

// in your main program...
MQEnvironment.receiveExit = new MyReceiveExit();
... // other initialization
MQQueueManager qMgr = new MQQueueManager("");
```

Figure 11-5. MQReceiveExit

MQ092.0

Notes:

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

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

To provide your own receive exit, define a class that implements this interface. Create a new instance of your class and assign the `MQEnvironment.receiveExit` variable to it before constructing your `MQQueueManager` object.

MQSecurityExit

```
// in MySecurityExit.java
class MySecurityExit implement MQSecurityExit {
    // you must provide an implementation
    //of the SecurityExit method
    public byte[] SecurityExit (
        MQChannelExit      channelExitParms,
        MQChannelDefinition channelDefinition,
        byte[]              agentBuffer)
    {
        // your exit code goes here...
    }
}
// in your main program...
MQEnvironment.SecurityExit = new MySecurityExit();
... // other initialization
MQQueueManager qMgr      = new MQQueueManager("");
```

Figure 11-6. MQSecurityExit

MQ092.0

Notes:

The security exit interface allows you to customize the security flows that occur when an attempt is made to connect to a queue manager.

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

To provide your own security exit, define a class that implements this interface. Create a new instance of your class and assign the `MQEnvironment.securityExit` variable to it before constructing your `MQQueueManager` object.

MQChannelExit and MQChannelDefinition

```
public class MQChannelExit
extends Object
```

```
public class MQChannelDefinition
extends Object
```

Figure 11-7. MQChannelExit and MQChannelDefinition

MQ092.0

Notes:

MQChannelExit

This class defines context information passed to the send, receive, and security exits when they are invoked. The `exitResponse` member variable should be set by the exit to indicate what action the MQSeries classes for Java should take next.

MQChannelDefinition

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

Note: These classes do not apply when connecting directly to MQSeries in bindings mode.

Example Code

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

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

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

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

Figure 11-8. Example Code

MQ092.0

Notes:

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

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

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

The simplest possible exit body is:

```
{
return agentBuffer;
}
```


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

11.2 Checkpoint and Summary

Unit Checkpoint

1. What interface allows you to define a user exit to alter the date received from the queue manager?
2. The MQChannelExit class is used to pass information concerning the connection to the queue manager to the send, receive and security exit. True/False?
3. The MQSecurityExit interface does not apply when connecting directly to MQSeries in bindings mode. True/False?

Notes:

Summary

You should now understand the use of user exits, in particular :

- Send exits
- Receive exits
- Security exits

You should also understand how to use the channel definition classes to implement these exits.

Notes:

Unit 12. Multithreading

What This Unit is About

In this unit, you will learn how to handle multiple threads when connecting to a queue manager.

What You Should Be Able to Do

After completing this unit, you should be able to

- Handle multiple threads.
- Understand thread synchronization.

How You Will Check Your Progress

Accountability:

- Checkpoint

References

SC34-5456 *MQSeries Using Java*

[http://www.ibm.com/software/ts/mqseries/messaging/
WebSphere MQ](http://www.ibm.com/software/ts/mqseries/messaging/WebSphereMQ)

Objectives

- Handle multiple threads
- Understand thread synchronization

Figure 12-1. Unit Objectives

MQ092.0

Notes:

12.1 Multithreading

Multithreaded Programs

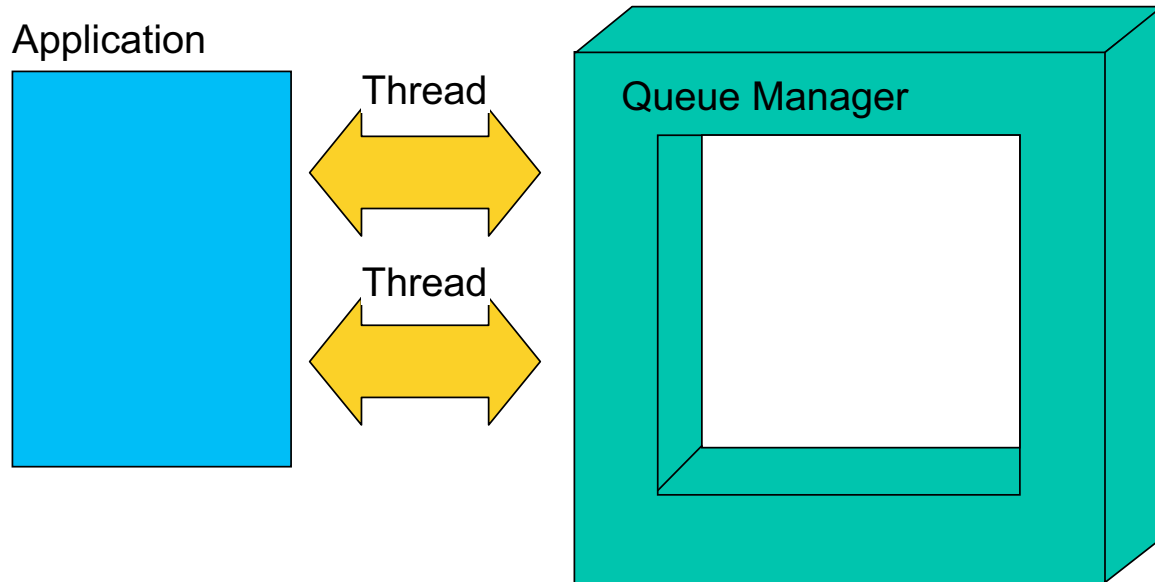


Figure 12-2. Multithreaded Programs

MQ092.0

Notes:

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

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

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

Thread Synchronization

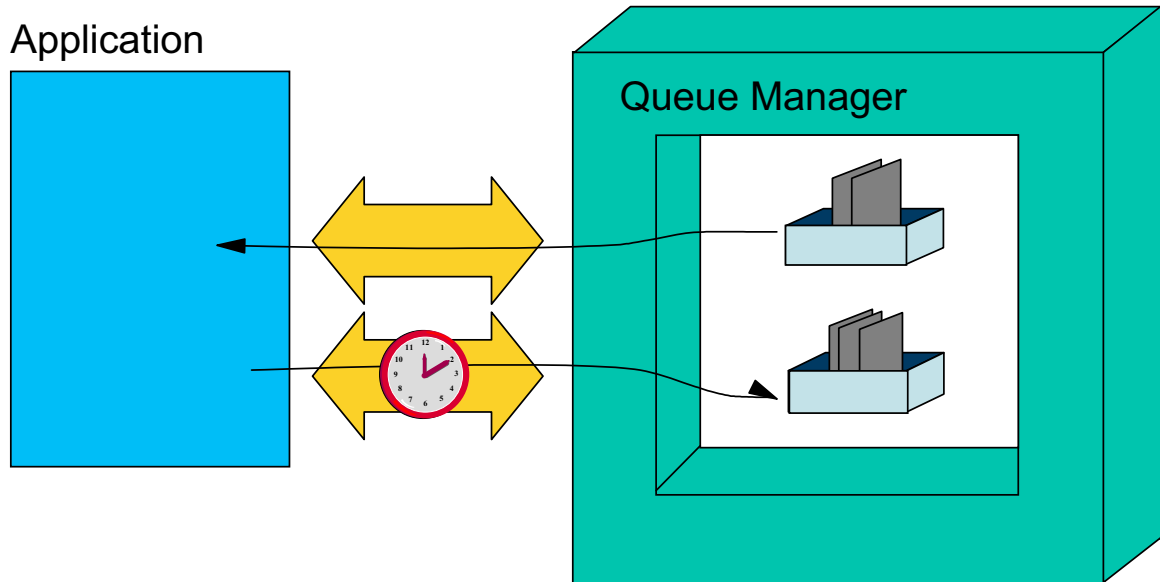


Figure 12-3. Thread Synchronization

MQ092.0

Notes:

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

Multithreading Example

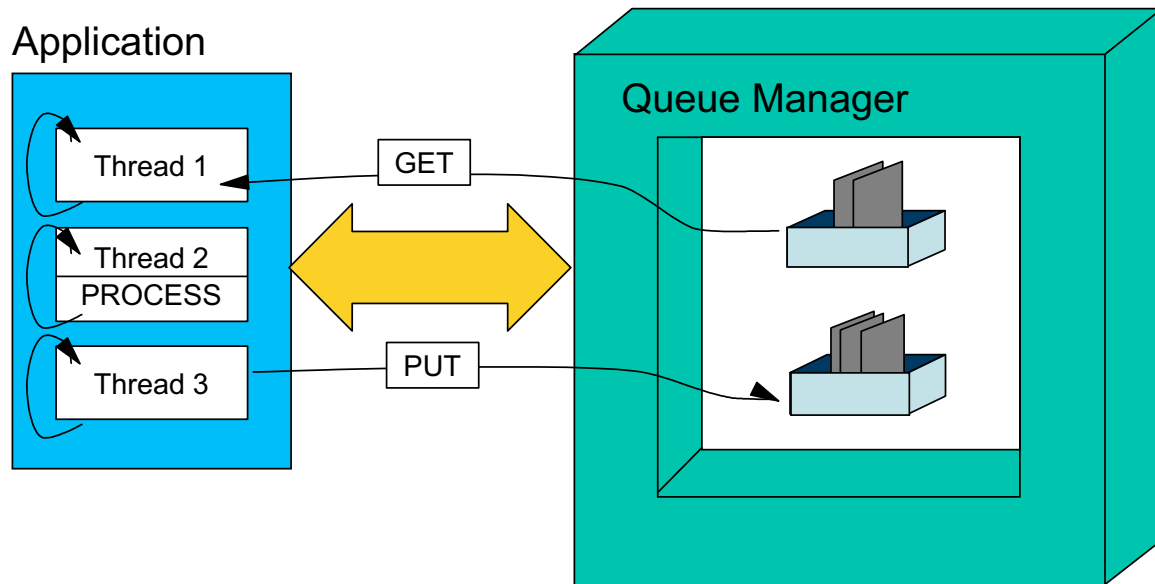


Figure 12-4. Multithreading Example

MQ092.0

Notes:

The above example shows you how you can use multiple threads to get a message, process this message and at the same time get another message.

- Thread #1 connects to a queue to get a message
- Thread #2 processes this message
- Thread #3 puts this message on another queue

When thread 1 has finished getting its message, it passes it on to thread 2 which processes the message. Thread 1 is now free to get the next message in the queue, without having to wait for thread 3 to put the message onto the other queue.

There is an important consideration in this case concerning the scope of the unit of work: you will not be able to do all this as a single UOW. The reason is that all calls to MQ must be done by the same thread that initiated the MQQueueManager object. The third thread can initiate another MQQueueManager object but the commit method will only commit work associated with the connection.

12.2 Checkpoint and Summary

Unit Checkpoint

1. One queue manager object can be shared by multiple threads. True/False?
2. Two threads can handle simultaneous accesses to the same queue manager. True/False?

Notes:

Summary

You should now know how to use threads to synchronize accesses, or to allow concurrent accesses to a queue manager.

Figure 12-6. Unit Summary

MQ092.0

Notes:

Appendix A. Checkpoint Solutions

Unit 2

1. MQQueueManager
2. b
3. b
4. a

Unit 3

1. b
2. True (using the bitwise OR operator)
3. True
4. False
5. a, d

Unit 4

1. reasonCode
2. exceptionSource

Unit 5

1. MQMessage
2. MQPutMessageOptions
3. myMessage
4. writeString
5. seek
6. MQC.MQOO_OUTPUT
7. myOutputQueue
8. MQC.MQOO_INPUT_SHARED
9. resizeBuffer
10. MQGetMessageOptions
11. myInputQueue

Unit 6

1. messageType
2. No
3. MQC.MQMT_REQUEST
4. replyToQueueName

Unit 7

1. MQMessage
2. correlationId
3. MQGetMessageOptions
4. MQC.MQGMO_WAIT
5. 7500
6. messageFlags
7. MQC.MQMF_MSG_IN_GROUP
8. MQC.MQMF_SEGMENTATION_ALLOWED
9. MQC.MQGMO_COMPLETE_MSG

Unit 8

1. No
2. No
3. Yes (priority)
4. No
5. InhibitGet, InhibitPut, TriggerControl, TriggerData, TriggerDepth, TriggerMessagePriority, TriggerType
6. MQDistributionListItem
7. MQDistributionList

Unit 9

1. accessQueue(), constructor MQQueue(), constructor MQQueueManager()
2. userId, accountingToken, applicationData, putApplicationType, putApplicationName, putDateTime, applicationOriginData
3. MQC.MQPMO_NO_CONTEXT

Unit 10

1. MQC.MQGMO_SYNCPOINT
2. commit
3. MQQueueManager

Unit 11

1. MQReceiveExit
2. False (that is MQChannelDefinition)
3. True

Unit 12

1. True
2. True (if you define a different queue manager object for each thread)

Appendix B. Bibliography

MQSeries cross-platform publications

MQSeries Brochure

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

MQSeries: An Introduction to Messaging and Queuing

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

MQSeries Planning Guide

The *MQSeries Planning Guide*, GC33-1349, describes some key MQSeries concepts, identifies items that need to be considered before MQSeries is installed, including storage requirements, backup and recovery, security, and migration from earlier releases, and specifies hardware and software requirements for every MQSeries platform.

MQSeries Release Guide V5.2

The *MQSeries Release Guide V5.2*, GC34-5761, introduces all the new functions in V5.2. These are additions to the cross-product books and should be used in conjunction with them.

This book is for users of any of the following products:

- AIX
- AS/400
- HP-UX
- Linux
- Sun Solaris
- Windows NT and 2000

MQSeries Intercommunication

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

MQSeries Clients

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

MQSeries System Administration

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

This book applies to the following MQSeries products only:

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

MQSeries Command Reference

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

MQSeries Programmable System Management

The *MQSeries Programmable System Management* book, SC33-1482, provides both reference and guidance information for users of MQSeries events, programmable command formats (PCFs), and installable services.

MQSeries Messages

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

- MQSeries for AIX V5.2
- MQSeries for HP-UX V5.2
- MQSeries for Linux V5.2
- MQSeries for Sun Solaris V5.2
- MQSeries for Windows NT and 2000 V5.2
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1

This book is available in softcopy only.

MQSeries Application Programming Guide

The *MQSeries Application Programming Guide*, SC33-0807, provides guidance information for users of the message queue interface (MQI). It describes how to design,

write, and build and MQSeries application. It also includes full descriptions of the sample programs supplied with MQSeries.

MQSeries Application Programming Reference

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

MQSeries Application Programming Reference Summary

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

MQSeries Using C++

MQSeries Using C++, SC33-1877, provides both guidance and reference information for users of the MQSeries C++ programming-language binding to the MQI. MQSeries C++ is supported by V5.1 of MQSeries for AIX, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT, and by MQSeries clients supplied with those products and installed in the following environments:

- AIX
- AS/400
- MQSeries for Compaq Tru64 UNIX
- HP-UX
- OS/2
- MQSeries for OS/390
- Sun Solaris
- Windows NT
- Windows 3.1
- Windows 95 & Windows 98

MQSeries Using Java

MQSeries Using Java, SC34-5456, provides both guidance and reference information for users of the MQSeries Base Classes for Java and the MQSeries classes for Java Message Service (JMS).

MQSeries Administration Interface Programming Guide and Reference

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

The book applies to the following MQSeries products only:

- AIX
- AS/400
- HP-UX
- OS/2
- Sun Solaris
- Windows NT

MQSeries Queue Manager Clusters

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

This book applies to the following MQSeries products only:

- AIX
- AS/400
- Compaq Tru64 UNIX
- AS/400
- HP-UX
- OS/2
- Sun Solaris
- Windows NT

MQSeries platform-specific publications

MQSeries for AIX

MQSeries for AIX V5.2 Quick Beginnings, GC33-1867

MQSeries for AT&T GIS UNIX

MQSeries for AT&T GIS UNIX Version 2.2 System Management Guide, SC33-1642

MQSeries for Digital OpenVMS

MQSeries for Digital OpenVMS Version 2.2 System Management Guide, GC33-1791

MQSeries for HP-UX

MQSeries for HP-UX V5.2 Quick Beginnings, GC33-1869

MQSeries for OS/390

MQSeries for OS/390 Version 5 Release 2 Licensed Program Specifications, GC34-5893

MQSeries for OS/390 Version 5 Release 2 Program Directory, GI10-2532

MQSeries for OS/390 Concepts and Planning V5.2, GC34-5650

MQSeries for OS/390 System Setup Guide V5.2, SC34-5651

MQSeries for OS/390 System Administration Guide V5.2, SC34-5652

MQSeries for OS/390 Problem Determination Guide V5.2, GC34-5892

MQSeries for OS/390 Messages and Codes V5.2, GC34-5891

MQSeries for OS/2 Warp

MQSeries for OS/2 Warp V5.1 Quick Beginnings, GC33-1868

MQSeries for AS/400

MQSeries for AS/400 V5.2 Quick Beginnings, GC34-5557

MQSeries for AS/400, V5.1 System Administration, SC34-5558

MQSeries for AS/400 V5.1 Application Programming Reference (RPG), SC34-5559

MQSeries link for R/3

MQSeries link for R/3 Version 1.2 User's Guide, GC33-1934

MQSeries for SINIX and DC/OSx

MQSeries for SINIX and DC/OSx Version 2.2 System Management Guide, GC33-1768

MQSeries for Sun Solaris

MQSeries for Sun Solaris V5.2 Quick Beginnings, GC33-1870

MQSeries for Tandem NonStop Kernel

MQSeries for Tandem NonStop Kernel Version 2.2 System Management Guide, GC33-1893

MQSeries for VSE/ESA

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

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

MQSeries for Windows

MQSeries for Windows Version 2.0 User's Guide, GC33-1822

MQSeries for Windows Version 2.1 User's Guide, GC33-1965

MQSeries for Windows NT and Windows 2000

MQSeries for Windows V5.2 Quick Beginnings, GC34-5389

MQSeries for Windows NT Using the Component Object Model Interface., SC34-5387

MQSeries LotusScript Extension., SC34-5404

MQSeries Level 1 product publications

MQSeries: Concepts and Architecture, GC33-1141

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

MQSeries for UnixWare Version 1.4.1 User's Guide, SC33-1379

Softcopy books

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

BookManager® format

The MQSeries library is supplied in IBM BookManager format on a variety of online library collection kits, including the *Transaction Processing and Data* collection kit, SK2T-0730.

You can view the softcopy books in IBM BookManager format using the following IBM licensed programs:

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

HTML format

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

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

The MQSeries books are also available from the MQSeries product family home page at

<http://www.ibm.com/software/ts/mqseries/>

Portable Document Format (PDF)

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

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

<http://www.adobe.com/>

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

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

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

<http://www.software.ibm.com/ts/mqseries/>

PostScript format

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

Windows Help format

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

MQSeries information available on the Internet

MQSeries URL

The URL of the MQSeries product family home page is:

<http://www.ibm.com/software/ts/mqseries/>

Appendix C. Glossary of terms and abbreviations

This glossary defines MQSeries® terms and abbreviations used in this book. If you do not find the term you are looking for, refer to the Index or the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.

A

administrator commands. MQSeries commands used to manage MQSeries objects, such as queues, processes, and namelist.

alias queue object. An MQSeries object, the name of which is an alias for a base queue defined to the local queue manager. When an applicator or a queue manager uses an alias queue, the alias name is resolved and the requested operation is performed on the associated base queue.

alternate user security. A security feature in which the authority of one user ID can be used by another user ID; for example, to open an MQSeries object.

application environment. The software facilities that are accessible by an application program. On the OS/390 platform, CICS and IMS are examples of application environments.

application log. In Windows NT, a log that records significant application events.

application queue. A queue used by an application.

asynchronous messaging. A method of communication between programs in which programs place messages on message queues. With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message. Contrast with *synchronous messaging*.

attribute. One of a set of properties that defines the characteristics of an MQSeries object.

authorization checks. Security checks that are performed when a user tries to issue administration commands against an object, for example to open a queue or connect to a queue manager.

authorization file. In MQSeries on UNIX systems, a file that provides security definitions for an object, a class of objects, or all classes of objects.

authorization service. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a service that provides authority checking of commands and MQI calls for the user identifier associated with the command or call.

B

backout. An operation that reverses all the changes made during the current unit of recovery or unit of work. After the operation is complete, a new unit of recovery or unit of work begins. Contrast with *commit*.

browse. In message queuing, to use the MQGET call to copy a message without removing it from the queue. See also *get*.

browse cursor. In message queuing, an indicator used when browsing a queue to identify the message that is next in sequence.

C

channel. See *message channel*.

client. A run-time component that provides access to queuing services on a server for local user applications. The queues used by the applications reside on the server. See also *MQSeries client*.

client application. An application, running on a workstation and linked to a client, that gives the application access to queuing services on a server.

client connection channel type. The type of MQI channel definition associated with an MQSeries client. See also *server connection channel type*.

coded character set identifier (CCSID). The name of a coded set of characters and their code point assignments.

command. In MQSeries, an administration instruction that can be carried out by the queue manager.

command processor. The MQSeries component that processes commands.

command server. The MQSeries component that reads commands from the system-command input queue, verifies them, and passes valid commands to the command processor.

commit. An operation that applies all the changes made during the current unit of recovery or unit of work. After the operation is complete, a new unit of recovery or unit of work begins. Contrast with *backout*.

completion code. A return code indicating how an MQI call has ended.

connect. To provide a queue manager connection handle, which an application uses on subsequent MQI calls. The connection is made either by the MQCONN call, or automatically by the MQOPEN call.

connection handle. The identifier or token by which a program accesses the queue manager to which it is connected.

context. Information about the origin of a message.

context security. In MQSeries, a method of allowing security to be handled such that messages are obliged to carry details of their origins in the message descriptor.

D

datagram. The simplest message that MQSeries Supports. This type of message does not require a reply.

dead-letter queue (DLQ). A queue to which a queue manager or application sends messages that it cannot deliver to their correct destination.

default object. A definition of an object (for example, a queue) with all attributes defined. If a user defines an object but does not specify all possible attributes for that object, the queue manager uses default attributes in place of any that were not specified.

distributed application. In message queuing, a set of application programs that can each be connected to a different queue manager, but that collectively constitute a single application.

DLQ. Dead-letter queue.

dynamic queue. A local queue created when a program opens a model queue object. See also *permanent dynamic queue* and *temporary dynamic queue*.

E

environment. See *application environment*.

F

FIFO. First-in-first-out.

first-in-first-out (FIFO). A queuing technique in which the next item to be retrieved is the item that has been in the queue for the longest time. (A)

G

get. In message queuing, to use the MQGET call to remove a message from a queue. See also *browse*.

H

handle. See *connection handle* and *object handle*.

hardened message. A message that is written to auxiliary (disk) storage so that the message will not be lost in the event of a system failure. See also *persistent message*.

I

in-doubt unit of recovery. In MQSeries, the status of a unit of recovery for which a syncpoint has been requested but not yet confirmed.

initiation queue. A local queue on which the queue manager puts trigger messages.

input/output parameter. A parameter of an MQI call in which you supply information when you make the call, and in which the queue manager changes the information when the call completes or fails.

input parameter. A parameter of an MQI call in which you supply information when you make the call.

L

listener. In MQSeries distributed queuing, a program that monitors for incoming network connections.

local definition. An MQSeries object belonging to a local queue manager.

local definition of a remote queue. An MQSeries object belonging to a local queue manager. This object defines the attributes of a queue that is owned by another queue manager. In addition, it is used for queue-manager aliasing and reply-to-queue aliasing.

locale. On UNIX systems, a subset of a user's environment that defines conventions for a specific culture (such as time, numeric, or monetary formatting and character classification, collation, or conversion). The queue manager CCSID is derived from the locale of the user ID that created the queue manager.

local queue. A queue that belongs to the local queue manager. A local queue can contain a list of messages waiting to be processed. Contrast with *remote queue*.

local queue manager. The queue manager to which a program is connected and that provides message queuing services to the program. Queue managers to which a program is not connected are called *remote queue managers*, even if they are running on the same system as the program.

logical unit of work (LUW). See *unit of work*.

M

MCA. Message channel agent.

message. (1) In message queuing applications, a communication sent between programs. See also *persistent message* and *nonpersistent message*. (2) In system programming, information intended for the terminal operator or system administrator.

message channel. In distributed message queuing, a mechanism for moving messages from one queue manager to another. A message channel comprises two message channel agents (a sender at one end and a receiver at the other end) and a communication link. Contrast with *MQI channel*.

message channel agent (MCA). A program that transmits prepared messages from a transmission queue to a communication link, or from a communication link to a destination queue. See also *message queue interface*.

message descriptor. Control information describing the message format and presentation that is carried as part of an MQSeries message. The format of the message descriptor is defined by the MQMD structure.

message priority. In MQSeries, an attribute of a message that can affect the order in which messages on a queue are retrieved, and whether a trigger event is generated.

message queue. Synonym for *queue*.

message queue interface (MQI). The programming interface provided by the MQSeries queue managers. This programming interface allows application programs to access message queuing services.

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

message sequence numbering. A programming technique in which messages are given unique numbers during transmission over a communication link. This enables the receiving process to check whether all messages are received, to place them in a queue in the original order, and to discard duplicate messages.

messaging. See *synchronous messaging* and *asynchronous messaging*.

model queue object. A set of queue attributes that act as a template when a program creates a dynamic queue.

MQI. Message queue interface.

MQI channel. Connects an MQSeries client to a queue manager on a server system, and transfers only MQI calls and responses in a bidirectional manner. Contrast with *message channel*.

MQSeries. A family of IBM licensed programs that provides message queuing services.

MQSeries client. Part of an MQSeries product that can be installed on a system without installing the full queue manager. The MQSeries client accepts MQI calls from applications and communicates with a queue manager on a server system.

MQSeries commands (MQSC). Human readable commands, uniform across all platforms, that are used to manipulate MQSeries objects. Contrast with *programmable command format (PCF)*.

N

namelist. An MQSeries object that contains a list of names, for example, queue names.

nonpersistent message. A message that does not survive a restart of the queue manager. Contrast with *persistent message*.

null character. The character that is represented by X'00'.

O

OAM. Object authority manager.

object. In MQSeries, an object is a queue manager, a queue, a process definition, a channel, a namelist, or a storage class (OS/390 only).

object authority manager (OAM). In MQSeries on UNIX systems and MQSeries for Windows NT, the default authorization service for command and object management. The OAM can be replaced by, or run in combination with, a customer-supplied security service.

object descriptor. A data structure that identifies a particular MQSeries object. Included in the descriptor are the name of the object and the object type.

object handle. The identifier or token by which a program accesses the MQSeries object with which it is working.

output parameter. A parameter of an MQI call in which the queue manager returns information when the call completes or fails.

P

PCF. Programmable command format.

PCF command. See *programmable command format*.

permanent dynamic queue. A dynamic queue that is deleted when it is closed only if deletion is explicitly requested. Permanent dynamic queues are recovered if the queue manager fails, so they can contain persistent messages. Contrast with *temporary dynamic queue*.

persistent message. A message that survives a restart of the queue manager. Contrast with *nonpersistent message*.

platform. In MQSeries, the operating system under which a queue manager is running.

principal. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a term used for a user identifier. Used by the object authority manager for checking authorizations to system resources.

process definition object. An MQSeries object that contains the definition of an MQSeries application. For example, a queue manager uses the definition when it works with trigger messages.

programmable command format (PCF). A type of MQSeries message used by:

- User administration applications, to put PCF commands onto the system command input queue of a specified queue manager
- User administration applications, to get the results of a PCF command from a specified queue manager
- A queue manager, as a notification that an event has occurred

Contrast with *MQSC*.

Q

queue. An MQSeries object. Message queuing applications can put messages on, and get messages from, a queue. A queue is owned and maintained by a queue manager. Local queues can contain a list of messages waiting to be processed. Queues of other types cannot contain messages—they point to other queues, or can be used as models for dynamic queues.

queue manager. (1) A system program that provides queuing services to applications. It provides an application programming interface so that programs can access messages on the queues that the queue manager owns. See also *local queue manager* and *remote queue manager*. (2) An MQSeries object that defines the attributes of a particular queue manager.

queuing. See *message queuing*.

quiescing. In MQSeries, the state of a queue manager prior to it being stopped. In this state, programs are allowed to finish processing, but no new programs are allowed to start.

R

reason code. A return code that describes the reason for the failure or partial success of an MQI call.

receiver channel. In message queuing, a channel that responds to a sender channel, takes messages from a communication link, and puts them on a local queue.

remote queue. A queue belonging to a remote queue manager. Programs can put messages on remote queues, but they cannot get messages from remote queues. Contrast with *local queue*.

remote queue manager. To a program, a queue manager that is not the one to which the program is connected.

remote queue object. See *local definition of a remote queue*.

remote queuing. In message queuing, the provision of services to enable applications to put messages on queues belonging to other queue managers.

reply message. A type of message used for replies to request messages. Contrast with *request message* and *report message*.

reply-to queue. The name of a queue to which the program that issued an MQPUT call wants a reply message or report message sent.

report message. A type of message that gives information about another message. A report message can indicate that a message has been delivered, has arrived at its destination, has expired, or could not be processed for some reason. Contrast with *reply message* and *request message*.

requester channel. In message queuing, a channel that may be started remotely by a sender channel. The requester channel accepts messages from the sender channel over a

communication link and puts the messages on the local queue designated in the message. See also *server channel*.

request message. A type of message used to request a reply from another program. Contrast with *reply message* and *report message*.

resolution path. The set of queues that are opened when an application specifies an alias or a remote queue on input to an MQOPEN call.

resource. Any facility of the computing system or operating system required by a job or task. In MQSeries for OS/390, examples of resources are buffer pools, page sets, log data sets, queues, and messages.

resource manager. An application, program, or transaction that manages and controls access to shared resources such as memory buffers and data sets. MQSeries, CICS, and IMS are resource managers.

responder. In distributed queuing, a program that replies to network connection requests from another system.

return codes. The collective name for completion codes and reason codes.

rollback. Synonym for *back out*.

S

sender channel. In message queuing, a channel that initiates transfers, removes messages from a transmission queue, and moves them over a communication link to a receiver or requester channel.

sequential delivery. In MQSeries, a method of transmitting messages with a sequence number so that the receiving channel can reestablish the message sequence when storing the messages. This is required where messages must be delivered only once, and in the correct order.

sequential number wrap value. In MQSeries, a method of ensuring that both ends of a communication link reset their current message sequence numbers at the same time. Transmitting messages with a sequence number ensures that the receiving channel can reestablish the message sequence when storing the messages.

server. (1) In MQSeries, a queue manager that provides queue services to client applications running on a remote workstation. (2) The program that responds to requests for information in the particular two-program, information-flow model of client/server. See also *client*.

server channel. In message queuing, a channel that responds to a requester channel, removes messages from a transmission queue, and moves them over a communication link to the requester channel.

server connection channel type. The type of MQI channel definition associated with the server that runs a queue manager. See also *client connection channel type*.

single-phase backout. A method in which an action in progress must not be allowed to finish, and all changes that are part of that action must be undone.

single-phase commit. A method in which a program can commit updates to a queue without coordinating those updates with updates the program has made to resources controlled by another resource manager. Contrast with *two-phase commit*.

store and forward. The temporary storing of packets, messages, or frames in a data network before they are retransmitted toward their destination.

synchronous messaging. A method of communication between programs in which programs place messages on message queues. With synchronous messaging, the sending program waits for a reply to its message before resuming its own processing. Contrast with *asynchronous messaging*.

syncpoint. An intermediate or end point during processing of a transaction at which the transaction's protected resources are consistent. At a syncpoint, changes to the resources can safely be committed, or they can be backed out to the previous syncpoint.

T

target queue manager. See *remote queue manager*.

temporary dynamic queue. A dynamic queue that is deleted when it is closed. Temporary dynamic queues are not recovered if the queue manager fails, so they can contain nonpersistent messages only. Contrast with *permanent dynamic queue*.

thread. In MQSeries, the lowest level of parallel execution available on an operating system platform.

time-independent messaging. See *asynchronous messaging*.

TMI. Trigger monitor interface.

transmission program. See *message channel agent*.

transmission queue. A local queue on which prepared messages destined for a remote queue manager are temporarily stored.

trigger event. An event (such as a message arriving on a queue) that causes a queue manager to create a trigger message on an initiation queue.

triggering. In MQSeries, a facility allowing a queue manager to start an application automatically when predetermined conditions on a queue are satisfied.

trigger message. A message containing information about the program that a trigger monitor is to start.

trigger monitor. A continuously-running application serving one or more initiation queues. When a trigger message arrives on an initiation queue, the trigger monitor retrieves the message. It uses the information in the trigger message to start a process that serves the queue on which a trigger event occurred.

trigger monitor interface (TMI). The MQSeries interface to which customer- or vendor-written trigger monitor programs must conform. A part of the MQSeries Framework.

two-phase commit. A protocol for the coordination of changes to recoverable resources when more than one resource manager is used by a single transaction. Contrast with *single-phase commit*.

U

undeliverable-message queue. See *dead-letter queue*.

undo/redo record. A log record used in recovery. The redo part of the record describes a change to be made to an MQSeries object. The undo part describes how to back out the change if the work is not committed.

unit of recovery. A recoverable sequence of operations within a single resource manager. Contrast with *unit of work*.

unit of work. A recoverable sequence of operations performed by an application between two points of consistency. A unit of work begins when a transaction starts or after a user-requested syncpoint. It ends either at a user-requested syncpoint or at the end of a transaction. Contrast with *unit of recovery*.

