Prepare for the IBM Certification Test 996, WebSphere MQ Solution Designer. This tutorial addresses the major calls in the Message Queuing Interface (MQI), the application programming interface (API) used to develop WebSphere MQ applications. It is the fourth tutorial in a series of five tutorials on WebSphere MQ Solution Designer.

Section 1. Before you start

About this series

WebSphere® MQ Version 6.0 connects applications in a consistent, reliable, and easy-to-manage way, providing a trustworthy foundation for cross department, enterprise wide integration. Providing reliable once-and-only-once delivery of important messages and transactions, WebSphere MQ handles the complexities of communication protocols and dynamically distributes messaging workload across available resources. This series of five tutorials helps you prepare to take the IBM certification Test 996, IBM WebSphere MQ V6.0, Solution Design. This certification targets intermediate level designers who understand the concepts of asynchronous messaging and can plan, architect, and design solutions based on WebSphere MQ.

About this tutorial
This tutorial is the fourth in the series designed to help you prepare for the IBM Certification Test 996, WebSphere MQ V6.0, Solution Design. This tutorial covers the major calls in the Message Queuing Interface (MQI), including the housekeeping calls and calls to put and get messages. It also discusses opening queues, the message descriptor, and techniques for controlling message retrieval. After you complete this tutorial, continue with the fifth tutorial, which covers additional MQI topics.

Objectives

After completing this tutorial, you will be familiar with:

- The housekeeping calls of the MQI: MQCONN, MQOPEN, MQCLOSE, and MQDISC.
- The MQPUT verb.
- Opening queues.
- The message descriptor.
- The MQGET verb.
- The MQPUT1 verb.
- Controlling message retrieval.

Prerequisites

This tutorial is written for developers and architects with intermediate experience in application and solution design and implementation. It assumes intermediate knowledge and skills in the following:

- Transaction management and database products
- Systems management
- Basic programming concepts
- Data communications and networking
- Information technology security concepts

System requirements

The examples in this tutorial were developed with WebSphere MQ V6.0 for
Windows® and Rational® Application Developer V6.0 for Windows.

The system requirements for the products used in the tutorial can be found through the following links:

- WebSphere MQ
- Rational Application Developer

Section 2. Introduction

This section introduces the Message Queue Interface (MQI). It lists the MQI calls, and discusses data types, structures, and constants used by the MQI. This section also describes the programming languages supported and how applications are built, and looks at the Java APIs available for messaging applications.

The MQI calls

The MQI is a simple, high-level programming interface that consists of thirteen calls. These calls can be split into major and minor calls. The major calls are those most commonly used in business applications. The minor calls are used less frequently. This distinction is only informal.

The major calls and their functions are:

<table>
<thead>
<tr>
<th>Call</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQCONN</td>
<td>Connect to a queue manager.</td>
</tr>
<tr>
<td>MQOPEN</td>
<td>Open a WebSphere MQ object, such as a queue.</td>
</tr>
<tr>
<td>MQPUT</td>
<td>Place a message on a queue.</td>
</tr>
<tr>
<td>MQGET</td>
<td>Retrieve a message from a queue.</td>
</tr>
<tr>
<td>MQCLOSE</td>
<td>Close a WebSphere MQ object.</td>
</tr>
<tr>
<td>MQPUT1</td>
<td>Put a single message on a queue: performs MQOPEN, MQPUT, and MQCLOSE in one operation.</td>
</tr>
<tr>
<td>MQDISC</td>
<td>Disconnect from a queue manager.</td>
</tr>
</tbody>
</table>
The minor calls and their functions are:

<table>
<thead>
<tr>
<th>Call</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQINQ</td>
<td>Inquire about attributes of a WebSphere MQ object.</td>
</tr>
<tr>
<td>MQSET</td>
<td>Set certain specific queue attributes.</td>
</tr>
<tr>
<td>MQCONNX</td>
<td>Connect to a queue manager using options.</td>
</tr>
<tr>
<td>MQBEGIN</td>
<td>Begin a unit of work coordinated by the queue manager.</td>
</tr>
<tr>
<td>MQCMIT</td>
<td>Syncpoint notification for all syncpointed PUTs and GETs since the last syncpoint.</td>
</tr>
<tr>
<td>MQBACK</td>
<td>Backout notification for all syncpointed PUTs and GETs since the last syncpoint.</td>
</tr>
</tbody>
</table>

Elementary data types

WebSphere MQ defines elementary data types for the parameters used in MQI calls. These types are defined non-specific to a programming language, with corresponding programming language-specific definitions provided for each supported programming language.

Some of the more common elementary data types are:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQBYTE</td>
<td>A single byte of data.</td>
</tr>
<tr>
<td>MQBYTEn</td>
<td>A string of bytes of n length.</td>
</tr>
<tr>
<td>MQCHAR</td>
<td>A single-byte character.</td>
</tr>
<tr>
<td>MQCHARn</td>
<td>A string of n-byte characters.</td>
</tr>
<tr>
<td>MQHCONN</td>
<td>A connection handle (32 bytes long).</td>
</tr>
<tr>
<td>MQHOBJ</td>
<td>An object handle (32 bytes long).</td>
</tr>
<tr>
<td>MQLONG</td>
<td>A 32-bit signed binary integer.</td>
</tr>
<tr>
<td>PMQLONG</td>
<td>A pointer to data of type MQLONG.</td>
</tr>
</tbody>
</table>

More information on the elementary data types, along with mappings from MQI types to programming language-specific types, is in the *WebSphere MQ Application*.
**Programming Reference** in the WebSphere MQ library (see Resources).

## Structures used in the MQI

Some parameters of MQI calls are data structures containing multiple fields of data. All structures are defined in terms of elementary data types.

The most commonly used structures are:

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQMD</td>
<td>The message descriptor, which follows the message around the system and describes the attributes of the message itself. The data held in the message descriptor can be interrogated or changed by a program.</td>
</tr>
<tr>
<td>MQOD</td>
<td>The object descriptor is used when any WebSphere MQ object is opened. This means it is set up for use before every MQOPEN and before the MQPUT1 call.</td>
</tr>
<tr>
<td>MQGMO</td>
<td>Get message options, which contains information that controls the behavior of the MQGET call.</td>
</tr>
<tr>
<td>MQPMO</td>
<td>Put message options, which contains information that controls the behavior of the MQPUT call.</td>
</tr>
</tbody>
</table>

More information on the data structures is in the *WebSphere MQ Application Programming Reference* in the WebSphere MQ library (see Resources).

## MQI constants

WebSphere MQ also defines constants for use in MQI programming. These define valid values for the input and output parameters used in MQI calls. These symbolic constants are used extensively in MQI programs and significantly improve their readability. Most constants names begin with a prefix that identifies where they are used. For example, constants beginning with MQRC_ define reason codes, and constants beginning with MQOO_ define open options.

## Supported programming languages

WebSphere MQ provides support for a large number of programming languages, including C, C++, COBOL, .Net, PL/I, Assembler, and RPG. Some languages are supported on several platforms; some, such as Assembler and RPG, on only one (System z and System i, respectively). Additional programming language support,
such as REXX, is provided through SupportPacs.

Building a WebSphere MQ application

Because so many languages and compilers are supported across the many WebSphere MQ platforms, the process for a successful compile and link differs not only between platforms, but between languages and even between different compilers using the same language.

It is important that programmers know which compiler is being used, since the compile and link may be unique. Details on compiling and linking WebSphere MQ applications are in the WebSphere MQ Application Programming Guide found in the WebSphere MQ library (see Resources).

Java APIs

WebSphere MQ provides two Java APIs for building applications.

The first is WebSphere MQ classes for Java, sometimes called WebSphere MQ base Java. This API closely follows the MQI, but there are subtle differences because of the object-oriented nature of Java. For example, connecting to a queue manager does not involve a call that is easily recognizable as an MQCONN call. Instead, an instance of MQQueueManager is created. Subsequent MQI calls, such as MQOPEN or MQCLOSE, which would normally pass the connection handle to be returned by MQCONN are instead made using method calls on the MQQueueManager instance.

The sample programs provided in this tutorial use the WebSphere MQ classes for Java, and you'll see in them how MQI calls and parameters are mapped to Java methods and classes.

The second Java API is the WebSphere MQ classes for Java Message Service, also called WebSphere MQ JMS. This API provides an implementation of the Java Message Service (JMS) specification. JMS provides a standard API for accessing any enterprise messaging system, allowing programmers to focus on the business logic of their messaging applications, and hides the underlying implementation details.

When using WebSphere MQ JMS to communicate with programs not using JMS, note that WebSphere MQ JMS messages, by default, carry an additional header that the non-JMS programs may not be capable of handling. You can omit this additional header when you know you are sending to a non-JMS program.
Section 3. MQCONN

Overview

Before issuing any other MQI calls, an application must first connect to a queue manager using an MQCONN call. The queue manager to which an application connects is the *local* queue manager. In general, you can connect to a specific queue manager or to the default queue manager. An application will normally need authority to connect to a queue manager.

The syntax of the MQCONN call is:

```
MQCONN (QMgrName, Hconn, CompCode, Reason)
```

The MQCONN call has one input parameter and three output parameters, as follows:

- **Queue manager name** - `QMgrName` (input)
  The name of the queue manager to which the application wishes to connect.

- **Connection handle** - `Hconn` (output)
  The connection handle represents the connection to the queue manager and must be specified as a parameter on all subsequent MQI calls to the queue manager.

- **Completion code** - `CompCode` (output)
  Every MQI call returns a completion code to enable the application to determine quickly whether the call completed successfully, completed partially, or failed.

- **Reason code** - `Reason` (output)
  Every MQI call returns a reason code to provide more information to the application when a call completes partially or fails.

Connection handle
The connection handle that is returned to the application at connect time is to be used on all subsequent MQI calls. It must not be changed by the application program. A valid handle is an indication that the connect was successful and the application is authorized to work with the queue manager.

MQCONN completion and reason codes

All calls will result in one of three completion codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQCC_OK</td>
<td>A successful call completed.</td>
</tr>
<tr>
<td>MQCC_WARNING</td>
<td>The call completed but the reason code indicates certain special or unusual conditions.</td>
</tr>
<tr>
<td>MQCC_FAILED</td>
<td>The call did not complete successfully.</td>
</tr>
</tbody>
</table>

The program needs to check the completion code after each call, and, if a warning or failure occurred, the program needs to examine the reason code to determine whether the program should continue or not.

Some of the most common reason codes that might occur when an MQCONN is attempted are:

<table>
<thead>
<tr>
<th>Reason code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQRC_NONE</td>
<td>Set if the completion code is MQCC_OK.</td>
</tr>
<tr>
<td>MQRC_ALREADY_CONNECTED</td>
<td>Set if an attempt is made to connect to a queue manager to which the program already has a connection. Completion code is MQCC_WARNING.</td>
</tr>
<tr>
<td>MQRC_ANOTHER_Q_MGR_CONNECTED</td>
<td>Set if an attempt is made to connect to a second queue manager while still connected to another queue manager. Completion code is MQCC_FAILED.</td>
</tr>
<tr>
<td>MQRC_Q_MGR_QUIESCING</td>
<td>Set to indicate that the queue manager is shutting down. Completion code is MQCC_FAILED.</td>
</tr>
<tr>
<td>MQRC_NOT_AUTHORIZED</td>
<td>Set if the program is not authorized to access the queue manager specified. Completion code is MQCC_FAILED.</td>
</tr>
<tr>
<td>QRC_Q_MGR_NOTAVAILABLE</td>
<td>Set if the queue manager has not been started or has stopped running. Completion code is MQCC_FAILED.</td>
</tr>
</tbody>
</table>
There are many other possible reason codes that may be returned from an unsuccessful MQCONN attempt. A complete list is included with the description of the MQCONN call in the *WebSphere MQ Application Programming Reference* (see Resources).

On all calls, the programmer should test for three types of status:

- Everything OK, no special conditions reported (MQCC_OK and MQRC_NONE).
- Expected special conditions that can happen on various calls, which are application specific; for example, MQRC_NO_MSG_AVAILABLE on an MQGET.
- Unexpected conditions, such as MQRC_NOT_AUTHORIZED on an MQCONNECT or MQOPEN.

**CICS connection**

It is not required to issue an MQCONN for a CICS (z/OS) program. CICS performs the connects, and there is no choice of queue manager. There can only be one queue manager within a CICS region.

Inclusion of an MQCONN in a CICS program, although not required, might help you port the application to a non-CICS environment with a minimum of re-coding. Be aware that MQCONN always completes successfully in a CICS environment.

**Batch connection**

Batch programs in a mainframe environment *must* issue an MQCONN command. However, unlike CICS, a batch program can select which queue manager to connect to (if multiple queue managers have been set up on the system). It is important to note that a batch program can generally only be connected to one queue manager at a time. This is the type of behavior that is common to most other queue managers. It is possible, by attaching additional *task control blocks* (TCBs) to a batch application, to connect to additional queue managers concurrently. However, WebSphere MQ will not coordinate syncpointing or resource sharing between the TCBs.

**IMS connection**

Information Management System (IMS) provides a special case to connecting to multiple queue managers. An IMS program *can* connect to more than one queue manager.
manager simultaneously. As long as the queue manager is known to the IMS control region, a program running in the IMS-independent regions can connect to it, regardless of whether the program is already connected to other queue managers.

**Scope of connection handle**

The connection handle represents the connection to the queue manager. Used on all subsequent calls issued by the application when interacting with the queue manager, it ceases to be valid when an MQDISC call is made. This is also true when the unit of processing that defines the scope of the connection handle terminates.

The scope of the connection handle is restricted to the smallest unit of parallel processing within the environment concerned. For example, a thread under UNIX, Linux, or Windows; a CICS task, an IMS task (up to the syncpoint), a batch task, or a TSO task under z/OS. The connection handle is not valid for use outside of this scope.

**Connecting to more than one queue manager**

As a general rule, an application can only be connected to one queue manager at a time. There are some exceptions:

- On z/OS, batch, TSO, and IMS allow multiple concurrent connections.
- On Windows, each thread can connect to a different queue manager.
- Client applications can connect to more than one queue manager within a thread.

The rules for connecting to multiple queue managers differ among the supported platforms and environments, so it is an important consideration when writing applications. If there is a possibility that you will want to port your application, you need to consider the most restricted environment in which you will be running and design for that situation.

**Access control**

An application running under a specific userID may not be allowed to connect to a queue manager. There is little an application can do if access is denied (MQCC_FAILED and MQRC_NOT_AUTHORIZED) except report the problem to the WebSphere MQ administrator. It may be possible that a change to a profile or access control list is required. In any case, no connection handle will be returned, and it is pointless for the application to continue.
MQCONNX

Though the topic of this tutorial is the MQI major calls, and the MQCONNX call is considered a minor call, we'll discuss it briefly since it is really just a variation of MQCONN.

The MQCONNX call is another means of connecting to a queue manager on some platforms. It allows for a parameter to be passed in addition to those passed in an MQCONN call. This additional parameter is called connection options.

The syntax of the MQCONNX call is:

```
MQCONNX (QMgrName, ConnectOpts, Hconn, CompCode, Reason)
```

One purpose of the connection options is to control how a program connects. When an application connects to a queue manager using the MQCONN call, the queue manager code that is executed to service each subsequent MCI call runs as a separate unit of execution from that of the application. But, when an application connects to a queue manager using the MQCONNX call, it may specify the fastpath binding option on the call. By using this option, the queue manager code that is executed to service each subsequent MQI call runs within the same unit of execution as that of the application. The advantage of such an arrangement is that fewer system resources are required to run the application. The disadvantage is that the integrity of the queue manager is compromised, as there is no protection from overwriting its storage. Such an application is called a trusted application.

Another use for the connection options is for a client to use it to specify a specific server it wishes to connect to, overriding any definitions or other settings that control the server connection.

The MQCONNX call also allows you to create a shared (thread-independent) connection that can be used by all threads in a process.

MQCONN pseudocode

Listing 1 below is a pseudocode example of using MQCONN. We'll use pseudocode in this tutorial and the next to provide language-independent illustrations of code usage and techniques. While obviously not a real programming language, pseudocode lets us see the structure and sequence of code without being tied to a specific language.

Listing 1. MQCONN example
The code begins by defining a constant for the default queue manager name as an empty string. Passing an empty string for the queue manager name in MQCONN will result in connecting to the default queue manager on the system.

Variables are then defined to hold the queue manager name, connection handle, completion code, and reason code.

The default queue manager name constant is assigned to the queue manager name variable, and the call to MQCONN is made.

After the call is completed, the code checks the completion code, writing an error message and terminating the program if the completion code is not MQCC_OK.

---

Section 4. MQOPEN

Overview

Before an application can put messages on a queue, or get messages from a queue, it must first open the queue by issuing an MQOPEN call.

The syntax of the MQOPEN call is:

```
MQOPEN (HConn, ObjDesc, Options, Hobj, CompCode, Reason)
```
Those parameters of the MQOPEN call that have not been described previously are:

- **Object descriptor - ObjDesc (input/output)**
  One of the structures defined in the MQI. Its purpose is to identify the object being opened. It contains a number of fields specifying, among other things, the name of the object being opened and the type of object.

- **Options - Options (input)**
  Used by the application to specify which operations it wishes to perform on the object being opened, or to control the action of MQOPEN. Examples include opening a queue for putting messages on it, opening a queue for browsing messages only, and opening a queue to inquire on the values of some or all of its attributes. An application can specify more than one option by adding the values together or by combining them using the bitwise OR operation.

- **Object handle - Hobj (output)**
  Represents the access that has been established to the object, and it must be specified as a parameter on all subsequent MQI calls that operate on that object.

### Object handle

As with the connection handle returned from MQCONN, an object handle is returned by MQOPEN, representing the access that has been established to the queue. The object handle is used in all subsequent GETs and PUTs to the queue.

Once an MQCLOSE occurs, the object handle will be invalidated. This frees the programmer to re-use that object handle on a subsequent MQOPEN; the object handle will then only be associated with the object opened by the most recent MQOPEN and only for the options requested during that MQOPEN.

In most cases, however, it is easier to maintain a separate object handle for each queue that is accessed. It is common for an application to have more than one queue open at the same time. In this case, the programmer must set up separate handles for each one, and use the correct handle when doing PUTs or GETs to the queue.

### The object descriptor

The object descriptor (MQOD) is the structure an application uses to identify the type of object (most often a queue) being opened, and its name. The queue name is
never used on subsequent PUTs and GETs, just the object handle.

Some of the fields in the object descriptor may be initialized by the application (defaults can be used in many cases), and some are updated and returned to the application as a result of the MQOPEN call. The most commonly used fields in the object descriptor are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectType</td>
<td>What kind of WebSphere MQ object is being opened.</td>
</tr>
<tr>
<td>ObjectName</td>
<td>The actual name of the object being opened.</td>
</tr>
<tr>
<td>ObjectQMName</td>
<td>Can contain the name of the queue manager where the object is defined.</td>
</tr>
<tr>
<td>DynamicQName</td>
<td>Used when dynamically creating a queue (dynamic and model queues will be covered later).</td>
</tr>
</tbody>
</table>

The *WebSphere MQ Application Programming Reference* contains a complete list and description of all the fields in the object descriptor (see Resources).

Open options

Open options are the means by which the application declares its intent on the object being opened. Some common options and their meanings are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQOO_INPUT_SHARED</td>
<td>Open the queue to get messages with shared access.</td>
</tr>
<tr>
<td>MQOO_INPUT_EXCLUSIVE</td>
<td>Open the queue to get messages with exclusive access. The call fails if the queue is currently open by this or another application for input of any type (MQOO_INPUT_SHARED, MQOO_INPUT_EXCLUSIVE).</td>
</tr>
<tr>
<td>MQOO_OUTPUT</td>
<td>Open the queue to put messages.</td>
</tr>
<tr>
<td>MQOO_BROWSE</td>
<td>Open queue to browse messages.</td>
</tr>
<tr>
<td>MQOO_FAIL_IF_QUIESCING</td>
<td>The MQOPEN calls fails if the queue manager is shutting down.</td>
</tr>
</tbody>
</table>

MQOPEN reason codes

Some reason codes returned by MQOPEN and their meanings are:
<table>
<thead>
<tr>
<th>Reason code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQRC_NONE</td>
<td>The MQOPEN call succeeded.</td>
</tr>
<tr>
<td>MQRC_HANDLE_NOTAVAILABLE</td>
<td>When a queue manager is defined, one of its attributes is MAXHANDS, which controls the maximum number of handles that any one task can have open at any one time. This reason code is returned when that value is exceeded.</td>
</tr>
<tr>
<td>MQRC_HCONN_ERROR</td>
<td>Something is wrong with the HCONN value passed. You should check that:</td>
</tr>
<tr>
<td></td>
<td>• The MQCONN call completed successfully.</td>
</tr>
<tr>
<td></td>
<td>• The variable that represents the connection handle is in scope within the program structure.</td>
</tr>
<tr>
<td></td>
<td>• An MQDISC call has not been issued. Return of this reason code does not guarantee that all uses of invalid connection handles will be detected. Generally, an invalid HCONN can cause unpredictable results.</td>
</tr>
<tr>
<td>MQRC_OBJECT_IN_USE</td>
<td>You are trying to open the queue with MQOO_INPUT_EXCLUSIVE, but the queue is already opened for input elsewhere, or you are trying to open the queue as shared, but the queue is already opened for exclusive input elsewhere.</td>
</tr>
<tr>
<td>MQRC_OPTION_NOT_VALID_FOR_TYPE</td>
<td>The type of options specified do not match allowable options for the named object. For example, the only object than can have messages put to it are queues. If a different type of object were opened by the MQOPEN call specifying the MQOO_OUTPUT option, then this reason code would be returned.</td>
</tr>
<tr>
<td>MQRC_OPTIONS_ERROR</td>
<td>Whereas the previous reason code meant that the options were in themselves correct, just not applicable to the type of object being opened, this reason code means that the options are not recognized. This could mean that the options are nonsensical or that the specific combination of options is incompatible.</td>
</tr>
</tbody>
</table>

The *WebSphere MQ Application Programming Reference* describes all reason codes that can be returned by MQOPEN, and provides recommended actions (see Resources).
MQOPEN security

When an MQOPEN is issued, it is possible that the user or application is not authorized to open the object, or is not authorized for one or more of the options requested. If not, the MQOPEN fails with an MQRC_NOT_AUTHORIZED reason code. The WebSphere MQ system administrator will have to ensure that access is permitted for the object and all applicable options.

MQOPEN pseudocode

Listing 2 below provides pseudocode of an MQOPEN call.

Listing 2. MQOPEN example

```plaintext
DEFINE DEFAULT_QMGR AS CONSTANT ' '  
DEFINE QMGR AS MQCHAR48  
DEFINE CONN_HANDLE AS MQCONN  
DEFINE COMP_CODE AS MQLONG  
DEFINE REASON_CODE AS MQLONG  
DEFINE OBJ_DESC AS MQOD  
DEFINE MYQUEUE_HANDLE AS MQOBJ  
DEFINE OPEN_OPTIONS AS MQLONG  
QMGR = DEFAULT_QMGR  
CALL MQCONN(QMGR,  
           CONN_HANDLE,  
           COMP_CODE,  
           REASON_CODE)  
IF COMP_CODE NOT = MQCC_OK  
   write an error message detailing the REASON_CODE  
   terminate the program  
STOP  
END-IF

OBJ_DESC.ObjectType = MQOT_Q  
OBJ_DESC.ObjectName = 'MYQUEUE'  
OBJ_DESC.ObjectQMgrName = DEFAULT_QMGR  
OPEN_OPTIONS = MQOO_OUTPUT + MQ00_FAIL_IF_QUIESCING  
CALL MQOPEN(CONN_HANDLE,  
            OBJ_DESC,  
            OPEN_OPTIONS,  
            MYQUEUE_HANDLE,  
            COMP_CODE,  
            REASON_CODE)  
IF COMP_CODE NOT = MQCC_OK  
   write an error message detailing the REASON_CODE  
   terminate the program  
STOP  
END-IF
```
This code builds upon the MQCONN example in Listing 1. Notice the additional variables defined for use with the MQOPEN call (OBJ_DESC, MYQUEUE_HANDLE, and OPEN_OPTIONS). Also note the initialization of the object descriptor and the open options variables. Multiple open options can be specified by adding the appropriate symbolic constants together.

Similar to the MQCONN call, the completion code is checked after the MQOPEN call and appropriate action is taken if not successful.

Section 5. MQCLOSE and MQDISC

MQCLOSE

The MQCLOSE call relinquishes access to an object.

The syntax of the MQCLOSE call is:

MQCLOSE (HConn, Hobj, Options, CompCode, Reason)

The connection handle and object handle in the MQOPEN call are those that were issued by the MQOPEN call. The options that can be set are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQCO_NONE</td>
<td>No optional close processing required.</td>
</tr>
<tr>
<td>MQCO_DELETE</td>
<td>The queue is deleted if either of the following is true:</td>
</tr>
<tr>
<td></td>
<td>• It is a permanent dynamic queue, and there are no messages on the queue</td>
</tr>
<tr>
<td></td>
<td>and no uncommitted get or put requests outstanding for the queue.</td>
</tr>
<tr>
<td></td>
<td>• It is the temporary dynamic queue that was created by the MQOPEN call</td>
</tr>
<tr>
<td></td>
<td>that returned Hobj. In this case, all the messages on the queue are</td>
</tr>
</tbody>
</table>
The queue is deleted, and any messages on it purged, if either of the following is true:

- It is a permanent dynamic queue and there are no uncommitted get or put requests outstanding for the queue.
- It is the temporary dynamic queue that was created by the MQOPEN call that returned Hobj.

**MQCLOSE reason codes**

Although the program may have finished with the object, a failure to close properly could be an indication of a wider spread problem. Therefore, all reason codes should be reported and properly handled. Some reason codes that can be returned from MQCLOSE are:

<table>
<thead>
<tr>
<th>Reason code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQRC_NONE</td>
<td>The close succeeded.</td>
</tr>
<tr>
<td>MQRC_HCONN_ERROR</td>
<td>The connection handle is not valid.</td>
</tr>
<tr>
<td>MQRC_HOBJ_ERROR</td>
<td>The object handle is not valid.</td>
</tr>
<tr>
<td>MQRC_OPTION_NOT_VALID_FOR_TYPE</td>
<td>You are attempting to use MQCO_DELETE or MQCO_DELETE_PURGE and the object is not a dynamic queue.</td>
</tr>
<tr>
<td>MQRC_OPTIONS_ERROR</td>
<td>You attempted to supply invalid options or to combine options that are mutually exclusive.</td>
</tr>
</tbody>
</table>

**MQCLOSE pseudocode**

Listing 3 below provides pseudocode of an MQCLOSE call. As before, this code builds on top of the code from the previous listing to provide context.

**Listing 3. MQCLOSE example**

```c
DEFINE DEFAULT_QMGR AS CONSTANT ' '
DEFINE QMGR AS MQCHAR48
DEFINE CONN_HANDLE AS MQCONN
DEFINE COMP_CODE AS MQLONG
```
DEFINE REASON_CODE AS MQLONG
DEFINE OBJ_DESC AS MQOD
DEFINE MYQUEUE_HANDLE AS MQOBJ
DEFINE OPEN_OPTIONS AS MQLONG
DEFINE CLOSE_OPTIONS AS MQLONG

QMGR = DEFAULT_QMGR

CALL MQCONN(QMGR,
            CONN_HANDLE,
            COMP_CODE,
            REASON_CODE)

IF COMP_CODE NOT = MQCC_OK
    write an error message detailing the REASON_CODE
    terminate the program
    STOP
END-IF

OBJ_DESC.ObjectType = MQOT_Q
OBJ_DESC.ObjectName = 'MYQUEUE'
OBJ_DESC.ObjectQMgrName = DEFAULT_QMGR

OPEN_OPTIONS = MQOO_OUTPUT + MQOO_FAIL_IF_QUIESCING

CALL MQOPEN(CONN_HANDLE,
             OBJ_DESC,
             OPEN_OPTIONS,
             MYQUEUE_HANDLE,
             COMP_CODE,
             REASON_CODE)

IF COMP_CODE NOT = MQCC_OK
    write an error message detailing the REASON_CODE
    terminate the program
    STOP
END-IF

CLOSE_OPTIONS = MQCO_NONE

CALL MQCLOSE(CONN_HANDLE,
              MYQUEUE_HANDLE,
              CLOSE_OPTIONS,
              COMP_CODE,
              REASON_CODE)

IF COMP_CODE NOT = MQCC_OK
    write an error message detailing the REASON_CODE
    terminate the program
    STOP
END-IF

Note the variable defined to hold close options and the setting of that variable just before the call to MQCLOSE.

MQDISC

Once a program is finished with a queue manager and its resources, it should disconnect from the queue manager using MQDISC.
The syntax of the MQDISC call is:

```
MQDISC (HConn, CompCode, Reason)
```

MQDISC is a simple call and has very few parameters; there are no options. The result of a successful MQDISC call, in most cases, is the invalidation of the connection handle.

**MQDISC reason codes**

Although a failed MQDISC may not seem important to the disconnecting application, it is important to check for and report any unexpected results. Some reason codes that might be returned by MQDISC are:

<table>
<thead>
<tr>
<th>Reason code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQRC_NONE</td>
<td>The MQDISC call completed successfully.</td>
</tr>
<tr>
<td>MQRC_HCONN_ERROR</td>
<td>The connection handle is not valid.</td>
</tr>
<tr>
<td>MQRC_CONNECTION_BROKEN</td>
<td>The connection to the queue manager has been lost. This can occur because the queue manager has ended.</td>
</tr>
</tbody>
</table>

**MQDISC pseudocode**

Listing 4 below provides pseudocode of an MQDISC call, building on the previous listing.

**Listing 4. MQDISC example**

```
DEFINE DEFAULT_QMGR AS CONSTANT ' '
DEFINE QMGR AS MQCHAR48
DEFINE CONN_HANDLE AS MQHCONN
DEFINE COMP_CODE AS MQLONG
DEFINE REASON_CODE AS MQLONG
DEFINE OBJ_DESC AS MQOD
DEFINE MYQUEUE_HANDLE AS MQOBJ
DEFINE OPEN_OPTIONS AS MQLONG
DEFINE CLOSE_OPTIONS AS MQLONG
QMGR = DEFAULT_QMGR
CALL MQCONN(QMGR,
    CONN_HANDLE,
    COMP_CODE,
    REASON_CODE)
IF COMP_CODE NOT = MQCC_OK
    write an error message detailing the REASON_CODE
    terminate the program
STOP
END-IF
OBJ_DESC.ObjectType = MQOT_Q
OBJ_DESC.ObjectName = 'MYQUEUE'
```
**Section 6. MQPUT**

**Overview**

MQPUT is used to place messages onto a queue that has been opened for output.

The syntax of the MQPUT call is:

```c
MQPUT (Hconn, Hobj, MsgDesc, PutMsgOpts,
        BufferLength, Buffer, CompCode, Reason)
```
As you can see, the number of parameters passed on this call is greater than those for the housekeeping calls (MQCONN, MQOPEN, MQCLOSE, MQDISC). Some of the parameters, such as Hconn, Hobj, CompCode, and Reason, should be familiar since they were used in the housekeeping calls.

The Buffer and BufferLength parameters contain the application data (the message) and the length of the application data, respectively.

The message descriptor (MsgDesc) and put message options (PutMsgOpts) are discussed below.

The message descriptor

The message descriptor (MQMD structure) encapsulates the properties of the message. It can be updated by the application or by the queue manager. In many cases, an application will simply allow default values (taken from the queue definitions, for instance) to be supplied by the queue manager when the MQPUT is issued.

Some of the more important fields in the message descriptor are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Type</td>
<td>Used to help WebSphere MQ and applications know the intent of a message. There are four types that are defined by WebSphere MQ:</td>
</tr>
<tr>
<td></td>
<td>• Datagram (the default)</td>
</tr>
<tr>
<td></td>
<td>• Request</td>
</tr>
<tr>
<td></td>
<td>• Reply</td>
</tr>
<tr>
<td></td>
<td>• Report</td>
</tr>
<tr>
<td></td>
<td>It is possible to have application-defined message types as well.</td>
</tr>
<tr>
<td>Encoding</td>
<td>The representation of numeric data in the message.</td>
</tr>
<tr>
<td>CodedCharSetId</td>
<td>The International Standards Organization (ISO) code page number associated with the message.</td>
</tr>
<tr>
<td>Format</td>
<td>Used to help the queue manager know what to do with data that requires conversion. This field is blank unless filled in by the application.</td>
</tr>
<tr>
<td>Priority</td>
<td>Can be used to control the sequence in which messages will be retrieved from a queue.</td>
</tr>
<tr>
<td>Persistence</td>
<td>Determines whether the message will be logged to enable its recovery in the case of a restart of the queue manager or a rebuilding of a damaged</td>
</tr>
</tbody>
</table>
MsgId and CorrelId

Used to retrieve specific messages from a queue.

Put message options

The put message options (MQPMO structure) controls how the message is put on a queue. Some of the fields of the put message options are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options</td>
<td>Contains a value that is a combination of options that have been added together. Examples of options are:</td>
</tr>
<tr>
<td></td>
<td>• MQPMO_SYNCPOINT or MQPMO_NO_SYNCPOINT</td>
</tr>
<tr>
<td></td>
<td>Indicates if the MQPUT is part of a unit of work or not.</td>
</tr>
<tr>
<td></td>
<td>• MQPMO_FAIL_IFQUIESCING</td>
</tr>
<tr>
<td></td>
<td>The MQPUT call fails if the queue manager is shutting down.</td>
</tr>
<tr>
<td></td>
<td>• MQPMO_PASSIDENTITYCONTEXT</td>
</tr>
<tr>
<td></td>
<td>The message is to have context information associated with it.</td>
</tr>
<tr>
<td>Context</td>
<td>Contains the object handle of an input queue. Use of this field will be discussed in the security section of the next tutorial.</td>
</tr>
<tr>
<td>KnownDestCount, UnknownDestCount, InvalidDestCount</td>
<td>Used for distribution list support.</td>
</tr>
<tr>
<td>ResolvedQName and ResolvedQMgrName</td>
<td>Contain the resolved names of the destination queue and queue manager, respectively.</td>
</tr>
</tbody>
</table>

MQPUT reason codes

Some reason codes returned by MQPUT are:

<table>
<thead>
<tr>
<th>Reason code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQRC_NONE</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>MQRC_HCONN_ERROR</td>
<td>Connection handle not valid.</td>
</tr>
<tr>
<td>MQRC_HOBJ_ERROR</td>
<td>Object handle not valid.</td>
</tr>
<tr>
<td>MQRC_MD_ERROR</td>
<td>Message descriptor not valid.</td>
</tr>
<tr>
<td>MQRC_MSG_TOO_BIG_FOR_Q</td>
<td>Message length greater than the maximum for the queue.</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------</td>
</tr>
<tr>
<td>MQRC_NOT_OPEN_FOR_OUTPUT</td>
<td>The queue was not opened for output.</td>
</tr>
<tr>
<td>MQRC_OPTIONS_ERROR</td>
<td>Options are invalid or inconsistent.</td>
</tr>
<tr>
<td>MQRC_PMO_ERROR</td>
<td>The MQPMO structure is not valid.</td>
</tr>
<tr>
<td>MQRC_PUT_INHIBITED</td>
<td>Put calls have been inhibited for the queue.</td>
</tr>
<tr>
<td>MQRC_Q_FULL</td>
<td>The queue already contains the maximum number of messages.</td>
</tr>
</tbody>
</table>

MQPUT pseudocode

Listing 5 below contains pseudocode of an MQPUT call.

**Listing 5. MQPUT example**

```c
DEFINE DEFAULT_QMGR AS CONSTANT ' '
DEFINE QMGR AS MQCHAR48
DEFINE CONN_HANDLE AS MQCONN
DEFINE COMP_CODE AS MQLONG
DEFINE REASON_CODE AS MQLONG
DEFINE OBJ_DESC AS MQOD
DEFINE MYQUEUE_HANDLE AS MQHOBJ
DEFINE OPEN_OPTIONS AS MQLONG
DEFINE CLOSE_OPTIONS AS MQLONG
DEFINE MESSAGE_DESCRIPTOR AS MQMD
DEFINE PUT_OPTIONS AS MQPMO
DEFINE MESSAGE_LENGTH AS MQLONG
DEFINE MESSAGE AS CHAR100

QMGR = DEFAULT_QMGR
CALL MQCONN(QMGR,
            CONN_HANDLE,
            COMP_CODE,
            REASON_CODE)

IF COMP_CODE NOT = MQCC_OK
    write an error message detailing the REASON_CODE
    terminate the program
STOP
END-IF

OBJ_DESC.ObjectType = MQOT_Q
OBJ_DESC.ObjectName = 'MYQUEUE'
OBJ_DESC.ObjectQMgrName = DEFAULT_QMGR

OPEN_OPTIONS = MQOO_OUTPUT + MQ00_FAIL_IF_QUIESCING

CALL MQOPEN(CONN_HANDLE,
            OBJ_DESC,
            OPEN_OPTIONS,
            MYQUEUE_HANDLE,
            COMP_CODE,
            REASON_CODE)

IF COMP_CODE NOT = MQCC_OK
    write an error message detailing the REASON_CODE
```
Note the variables defined for the MQPUT call (MESSAGE_DESCRIPTOR, PUT_OPTIONS, MESSAGE_LENGTH, and MESSAGE). The pseudocode did not set any of the fields for the message descriptor nor the put message options, but instead accepted the default values for those structures.

Section 7. Message creation sample code

Listing 5 provides pseudocode for a complete, though not quite useful, program. It
connects to a queue manager, opens a queue, puts a message on that queue, then closes the queue and disconnects from the queue manager. In this section, you will look at some actual code, written in Java, and run that code to see it in action.

Before proceeding, you will need to install IBM Rational Application Developer v6.0 (Application Developer). See Resources for a link to a trial version of Application Developer. When downloading the trial version, you only need to download the files marked as required; you don't need any of the files marked optional. Accept all defaults when installing Application Developer.

Download the sample code

Download the .zip file, wes-cert9964samples.zip, containing the sample code and configuration files to your system (see Download). Extract the contents of the file to a folder on your system. After extracting, you will have four files: MQMajor.zip, setup01.txt, setup02.txt, and setup03.txt.

Copy the three setup files to your user directory, which is the default directory when you open a Command Prompt. Typically, this is \Documents and Settings\Administrator or \Documents and Settings\<your_user_name>.

Import the sample code

To import the sample code into Application Developer:

1. From the Start menu, launch Application Developer.
2. Accept the default when asked to select a workspace, and click OK.
3. From the main menu, select Window >Open Perspective > Java.
4. From the main menu, select File > Import....
5. Select Project Interchange and click Next. Your screen should look like Figure 1. The Project location root: field might be different on your system; that's OK. Figure 1. Import Project Interchange Contents
6. Click **Browse...** next to the From zip file: field to navigate to the folder into which you unzipped the downloaded sample code, and select **MQMajor.zip**. Your screen should look similar to Figure 2. (On my system, I unzipped the file to a folder named wes-cert9964samples; if you used a different folder that's OK).

**Figure 2. Selecting MQMajor.zip to open for import**
7. Click **Open**.

8. Select **Select All**, then **Finish**.

9. Expand **MQMajor** in the Package Explorer view, and expand **com.ibm.cert996** to display the sample code classes. Your screen should look like Figure 3.

**Figure 3. MQMajor project expanded**
10. Double-click **MQPUT.java**.

Review the sample code

Let's take a look at the sample code to see what it's doing. We'll go through the code in small pieces and discuss what each piece is doing. The class declaration, along with some declared constants and the **main()** method, are shown in Listing 6 below.

**Listing 6. MQPut.java class declaration**

```java
public class MQPut {
    private static final String qManager = "QMC1";
    private static final String qName = "CERT.TARGET";
    public static void main(String args[]) {
        new MQPut().run();
    }
}
```

Two constant strings are defined to name the queue manager and the queue. The **main()** method merely creates an instance of the class and calls its **run()** method. The **run()** method is the only other method in the class, and we'll spend the rest of
this review looking at it.

Look at the first several lines of the run() method in Listing 7.

Listing 7. Start of run() method

```java
public void run() {
    System.out.println("Start MQPut");
    try {
        MQQueueManager qMgr = new MQQueueManager(qManager);
        int openOptions = MQC.MQOO_OUTPUT + MQC.MQOO_FAIL_IF_QUIESCING;
        MQQueue queue = qMgr.accessQueue(qName, openOptions);
        MQPutMessageOptions putOptions = new MQPutMessageOptions();
        putOptions.options = MQC.MQPMO_NO_SYNCPOINT + MQC.MQPMO_FAIL_IF_QUIESCING;
    }
```

First, a message indicating that the program has started is printed to the console.

An MQQueueManager object is created, passing in the name of the queue manager. This provides the connection to the queue manager, and provides the same functions as a call to MQOPEN.

Then an integer is assigned the value MQC.MQOO_OUTPUT + MQC.MQOO_FAIL_IF_QUIESCING. This integer will be used as open options when the queue is opened. WebSphere MQ base Java provides a class, MQC, which contains the symbolic constants for use in WebSphere MQ programming. These constants have the same names as documented in the WebSphere MQ manuals.

The accessQueue() method is then called on the qMgr object, with the queue name and open options passed as parameters. accessQueue() provides the same functions as MQOPEN.

The program does not check completion code and reason code after connecting to the queue manager, nor after opening the queue. WebSphere MQ base Java does not return these codes after each call. Instead, an exception (MQException) will be thrown if an error occurs. Programmers must catch this exception, which contains the completion code and reason code, if an error is encountered.

Next, an MQPutMessageOptions object is created and initialized so that messages will not be under syncpoint control when put, and so that putting a message will fail if the queue manager is shutting down.

The next several lines, shown in Listing 8, are the heart of the program.

Listing 8. Reading a file and putting messages
Here, a file, testdata.txt, is opened for reading. A variable is declared (line) to contain the contents of each line read from the file, and a loop is entered that will read each line of the file and terminate when there are no more lines to be read.

Inside the loop, an MQMessage object is created and initialized: message type is set to datagram, and the message ID and correlation ID are set to none. Also, the contents of the line read from the file are "written" into the message as the application data. (WebSphere MQ base Java uses "write" and "read" to assign the application data to the message and to get the application data from the message, respectively.) Finally, the message is put to the queue. As before, this doesn't look exactly like what you might have expected from a straightforward procedural translation of the MQI. Nevertheless, it is following the same basic steps as the pseudocode in Listing 5, albeit in the object-oriented manner that best fits the Java language.

Once all of the lines have been read from the file and put to the queue, the queue is closed and the application disconnects from the queue manager, as shown in Listing 9.

**Listing 9. Closing the queue and disconnecting from the queue manager**

```java
queue.closeOptions = MQC.MQCO_NONE;
queue.close();
qMgr.disconnect();
```

Listing 10 below shows the catch block where MQException is caught and displayed. Finally, once the program has completed, a message to that effect is printed to the console.

**Listing 10. Exception handling and end of program**

```java
} catch (MQException mqex) {
    System.out.println("A WebSphere MQ Error occured : Completion Code "+ mqex.completionCode + " Reason Code ")
```

```
Run the sample code

Now let's run the code! The instructions here assume that you have followed the instructions in Part 2 of this series.

1. Open a Windows Command Prompt.
2. Type `strmqm` and press Enter.
3. Once the queue manager starts, type `runmqsc < setup01.txt` and press Enter. Your screen should look like Figure 4.

Figure 4. Queue manager started and configured

4. As you can see, setup01.txt contains the MQSC command to create a queue matching the name used in MQPUT.java.
5. Close the Windows Command Prompt.
6. In Application Developer, from the main menu select Window > Show View > Console.
7. Right-click MQPut.java in the Package Explorer view, and select Run > Java Application.

8. You should see messages in the Console indicating the program has started and ended.

9. Now let’s use a tool you haven’t used before to look at the messages on the queue. Start WebSphere MQ Explorer from the Windows Start menu.

10. In the WebSphere MQ Explorer - Navigator view, expand Queue Managers, then expand QMC1, and select Queues. Your screen should look like Figure 5. **Figure 5. WebSphere MQ Explorer displaying queues**

11. Right-click CERT.TARGET in the WebSphere MQ Explorer - Content view, and select Browse Messages.... You should see the messages on the queue.

12. Double-click testdata.txt in Application Developer and compare its contents with the messages in the queue.

13. Leave WebSphere MQ Explorer and Application Developer running.
Section 8. Opening queues

This section discusses some aspects of opening queues of different queue types.

Queue independence

When a program issues MQOPEN, it always considers the object that is being opened as if it is local. The queue manager will resolve the name of the object being opened.

If the name of the queue being opened has been defined as a remote definition of a queue on another system, it is the responsibility of the queue manager, using the definitions that the administrator created, to ensure that the message is put on the proper transmission queue and that it contains the necessary routing information to enable delivery on the remote side. This discussion pertains to putting messages only; you cannot get messages from a remote queue.

If the name of the queue being opened is an alias, it is again the responsibility of the queue manager to resolve that name to a definition of either a local or remote queue definition and to properly deliver the message.

The program truly does not know whether the queue that is being opened is a local queue, an alias queue, or a remote definition of a queue on another system.

Alias queues

An alias queue is simply a definition. It allows a local or remote queue to be referred to by another name. An alias queue can have different properties from the underlying queue to which it is pointing. For instance:
DEFINE QLOCAL(REALQ) GET(ENABLED) PUT(ENABLED)
DEFINE QALIAS(MYNAME) TARGQ(REALQ) GET(DISABLED)
allows the programs accessing the queue called REALQ to GET and PUT messages. However, if a program opens the queue called MYNAME, only PUTs will be permitted. GETs are disabled, even though the same queue (REALQ) is actually being used by both programs.

It is important to understand that the program will act as if the queue called MYNAME is a real queue, not simply a pointer to another queue.

Queue name resolution
When a program opens a queue, the object descriptor contains the object name and, optionally, the name of the queue manager. In most cases, the ObjectQueueManagerName field in the object descriptor will contain blanks. If it does, or if it contains the name of the queue manager to which the application is connected, local definitions are searched to resolve the queue name.

If the queue manager name field contains the name of another queue manager, it is assumed that this queue manager is remote, and a transmission queue with the same name is sought in which MQPUT messages may be placed.

Model queues

When an administrator defines a model queue, the definition is simply a template. When the name of a model queue is specified in the object descriptor for an MQOPEN, a queue with the attributes of the model is dynamically created. The model itself has no other purpose. If the characteristics of the new queue were displayed, it would appear as a local queue.

Temporary dynamic queues last only until the execution of the program that created it ends (normally or abnormally) or until the creating program closes it. There is no way to keep a temporary dynamic queue beyond that point.

Temporary dynamic queues may not hold persistent messages.

Permanent dynamic queues are created in exactly the same way but they are not automatically deleted. They must be specifically deleted, using a close option of delete or by the administrator with a delete command. Once created, there is nothing specific that WebSphere MQ does to keep track of dynamically created permanent dynamic queues.

The type of dynamic queue chosen is a matter of application design.

Dynamic queue names

The DynamicQueueName field in the object descriptor is used to control the name of the created dynamic queue. When an asterisk appears in the last position, it is replaced in the generated name by a string that is guaranteed unique for the local queue manager. There are several choices for the DynamicQueueName field:

- The default value (CSQ.* for z/OS; AMQ.* for other environments).
- A name, such as MYQUEUE, with no trailing asterisk. Such a name may not be unique.
- A name, such as MYQUEUE.*. Queues whose names start with the
common character string can be grouped for administrative and security purposes.

- An asterisk alone (not recommended).

After the MQOPEN completes successfully, the name of the new queue is available in the ObjectName field.

**Dynamic reply-to queues pseudocode**

Listing 11 below shows pseudocode for opening a dynamic queue and saving that name as a reply-to queue in a message descriptor for later use. The details of the MQCONN call and some variable definitions have been left out to simplify the example, and to concentrate on the opening and use of the dynamic queue.

**Listing 11. Dynamic reply-to queues example**

```plaintext
DEFINE DYN_OBJ AS MQOD
DEFINE A_MQMD AS MQMD
CALL MQCONN(......)
DYN_OBJ.DynamicQName = 'MYQUEUE.*'
DYN_OBJ.ObjectName = 'MODEL1'
CALL MQOPEN(CONN_HANDLE,
             DYN_OBJ,
             OPEN_OPTIONS,
             MYQUEUE_HANDLE,
             COMP_CODE,
             REASON_CODE)
IF COMP_CODE NOT = MQCC_OK
   /* error handling*/
ELSE
   /* save name assigned to created dynamic queue for later use */
   /* typically in the MQMD reply-to queue name field */
   A_MQMD.ReplyToQ = DYN_OBJ.ObjectName
END-IF
```

---

**Section 9. The message descriptor**

This section takes a more detailed look at the fields in the message descriptor (MQMD), some of which have been described before.
Persistence

The persistence attribute of a message determines whether a message is recoverable after queue manager restarts and rebuilds damaged queues. *Persistent messages* are logged to enable recovery, and this logging has a performance impact.

*Non-persistent messages* are not logged, and will not survive queue manager restarts or rebuilds of damaged queues.

The MQMD persistence field can be set by the application program to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQPER_NOT_PERSISTENT</td>
<td>The message is non-persistent.</td>
</tr>
<tr>
<td>MQPER_PERSISTENT</td>
<td>The message is persistent.</td>
</tr>
<tr>
<td>MQPER_PERSISTENCE_AS_Q_DEF</td>
<td>The persistence of the message is determined by the default persistence characteristic of the queue (DEFPSIST), defined by the administrator.</td>
</tr>
</tbody>
</table>

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.

If a message is critical and there is no simple way to recreate it, either the programmer should explicitly set the message descriptor persistence to MQPER_PERSISTENT or, if the queue has been defined with DEFPSIST(YES), the program can safely allow the default persistence as defined for the queue to be used.

Priority

The priority field in the message descriptor is used to set the priority of a message when it is put to a queue. This value may be in the range 0 - 9, or -1, which causes the default priority of the queue to be inherited.

The message delivery sequence attribute of the queue, set by the administrator, determines how the message is actually stored on the queue by the queue manager. If the value of this attribute is MQMDS_FIFO, messages are enqueued with the default priority of the queue, irrespective of the priority set by the application. If this attribute is MQMDS_PRIORITY, the priority value in the message descriptor is
honored when enqueuing the message.

A convention when constructing replies is to use the priority value of the incoming request.

**Message type**

WebSphere MQ provides four predefined messages types, identified by the message type field in the message descriptor:

**Requests**
When an application creates a message expecting a response from a target application, the message type should be set to request.

**Replies**
Application specific responses from a target application.

**Reports**
Most frequently generated by a target queue manager as a result of a specific condition, such as non-delivery, occurring.

**Datagrams**
The default message type, expecting no response.

Applications can also define their own messages within the range MQMT_APPL_FIRST and MQMT_APPL_LAST.

**Reply-to queue**

When a program sends a request message (by setting the message type field to MQMT_REQUEST) it is telling the program that receives its request that a reply is expected. The requesting application must also place the name of a queue in the reply-to queue name field of the message descriptor. MQPUT will fail if the message type is set to request and no reply-to queue name is supplied.

Using a reply-to queue in the message descriptor is useful when the name of the reply queue may differ because multiple requesters are sending messages, or if, each time a single requester program starts up, a new dynamic reply queue is created.

Because of the nature of asynchronous processing, there is no implied timing regarding when the reply will be sent back. We'll look at ways to set a time limit to wait for replies as you continue.
Report

The report field of the message descriptor is set by an application to indicate that it wishes to receive reports based on certain events that might occur while the message is being processed. Some of the most commonly used values for the report field are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQRO_EXCEPTION</td>
<td>This report type will be covered in the next topic below.</td>
</tr>
<tr>
<td>MQRO_EXPIRATION</td>
<td>Requests an expiry report that indicates an application attempted to retrieve a message that had reached its expiry threshold and that the message has been discarded.</td>
</tr>
<tr>
<td>MQRO_COA</td>
<td>Requests a confirmation of arrival report that indicates the message has reached its target queue.</td>
</tr>
<tr>
<td>MQRO_COD</td>
<td>Requests a confirmation of delivery report that indicates the message has been retrieved by a receiving application.</td>
</tr>
</tbody>
</table>

If the report field is set to any value other than MQRO_NONE, the reply-to queue name field must also be set.

Report options may be combined. For example, if you set the delivery confirmation report option and one of the exception report options, if the message fails to be delivered, you will receive an exception report message. However, if you select only the delivery confirmation report option and the message fails to be delivered, you will not get an exception report message.

Exception reports

When there is a problem with a local queue (queue full, does not exist, and so on), the queue manager can notify the application immediately and synchronously with a completion code and reason code.

With remote queues, a satisfactory completion code or reason code merely means that the message has been placed on the transmission queue. If the receiving queue manager subsequently encounters a problem, the completion code/reason code mechanism cannot be used: time independence means that the sending program may no longer be active. In this case, if the message descriptor report field has indicated that an exception report should be sent in the case of non-delivery, the target queue manager will construct an exception report and return it to the reply-to
queue specified in the message descriptor.

There are three exception report options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQRO_EXCEPTION</td>
<td>An exception report is sent with none of the application data from the original message.</td>
</tr>
<tr>
<td>MQRO_EXCEPTION_WITH_DATA</td>
<td>An exception report is sent with the first 100 bytes of the application data from the original message.</td>
</tr>
<tr>
<td>MQRO_EXCEPTION_WITH_FULL_DATA</td>
<td>An exception report is sent with all of the application data from the original message.</td>
</tr>
</tbody>
</table>

Feedback

The feedback field of a report message (other than an exception report) identifies the type of report that the message represents. For example, a confirmation delivery report message would contain MQFB_COD in the feedback field.

For exception reports, the feedback field contains a reason code describing the exception.

Encoding

When floating point, decimal, or integer numeric data is put into fields in the application data portion of a message, there is a problem because the data may be represented differently on different platforms. This problem is avoided by the encoding field in the message descriptor. Using the default value, MQENC_NATIVE, will enable the queue manager to put the correct value into the encoding field when the message is PUT, based on the platform where the program is running and the language in which the program is written. The destination queue manager can inspect the encoding field in the incoming message descriptor and, if requested, convert the numeric fields as appropriate.

CodedCharSetId

Data can come in many languages, and in ASCII or EBCDIC. So, when a message is put on a queue, it is important to identify what the coded character set is. Again, by using the default (MQCCSI_Q_MGR) for the coded character set ID field, the proper value for this field can be supplied by the queue manager at MQPUT time. The actual value will be the ISO code page number for that particular language or platform.
A set of tables for all of the supported code pages is in an appendix of the *WebSphere MQ Application Programming Reference* (see Resources).

**Format**

The encoding and coded character set ID fields on an incoming message enable the receiving queue manager to convert the message descriptor. The queue manager knows the location of the numeric fields, such as message type, and the character fields, such as reply-to queue name.

However, the receiving queue manager has no idea how the application data portion of the message has been designed; it sees only a stream of data.

If you require the application data portion of the message to be converted as well, you have three options:

- Create a message where the application data is all in character format, including numeric fields. If you set the format field of the message descriptor to MQFMT_STRING, application data will be converted on a character by character basis.
- Write a message conversion exit, and place its name in the format field.
- Use an additional product, such as WebSphere Message Broker, to perform data conversion.

---

**Section 10. MQGET**

**Overview**

MQGET is used to retrieve messages from a queue previously opened with one of the MQOO_INPUT options. Unless an MQOO_BROWSE option is used, messages are destructively removed from the queue and returned to the application buffer.

The syntax of the MQGET call is:

```
MQGET (HConn, Hobj, MsgDesc, GetMsgOpts, BufferLength, Buffer, DataLength, CompCode, Reason)
```

Again, the connection handle and object handle are the first two arguments passed.
The message descriptor is again both an input and output structure, as is the get message options structure.

In the case of MQGET, the buffer is an output field. Buffer length tells the queue manager how long the buffer is into which the application data portion of the message is to be returned.

The queue manager uses the data length parameter to return the actual length of the message data that was retrieved. In this way, the program can make use of a general purpose buffer that can contain small or large messages. It can know how big each message retrieved actually is by the data length value.

### Get message options

The get message options structure allows for the definition of options to be used when the MQGET is issued. Some of the fields of the put message options are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options</td>
<td>Contains the combination of options that an application specifies to control the behavior of the MQGET. Some of the more widely used options are:</td>
</tr>
<tr>
<td></td>
<td>- MQGMO_WAIT Wait until a message arrives on the queue.</td>
</tr>
<tr>
<td></td>
<td>- MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT Establishes whether the message should be part of a unit of work or immediately deleted from the input queue.</td>
</tr>
<tr>
<td></td>
<td>- MQGMO_ACCEPT_TRUNCATED_MESSAGE Fit as much of the message into the buffer as will fit and discard the remainder. If not specified and a message is larger than the buffer length as declared in the MQGET, the call will fail.</td>
</tr>
<tr>
<td></td>
<td>- MQGMO_CONVERT If the MQMD_ENCDODING or the MQMD_CCSID on the incoming message differs from those of the environment where the MQGET is issued, use the value in the</td>
</tr>
</tbody>
</table>
MQMD_FORMAT field to allow data to be converted to a usable state.

- MQGMO_BROWSE_FIRST,
  MQGMO_BROWSE_NEXT
Retrieve a copy of the requested message into the application buffer, leaving the message on the input queue.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WaitInterval</td>
<td>A value, in milliseconds, that specifies how long to wait for a message to arrive before issuing a unit of work. This field is valid only if MQGMO_WAIT is set in the options field.</td>
</tr>
<tr>
<td>ResolvedQName</td>
<td>If an application issues an MQGET on an alias queue, the name of the base queue will be returned in this field when the queue manager retrieves the message.</td>
</tr>
<tr>
<td>MatchOptions</td>
<td>Allows the application to choose which fields in the message descriptor parameter will be used to select the message returned by the MQGET call. The application sets the required options in this field, and then sets the corresponding fields in the message descriptor parameter to the values required for those fields. Only messages that have those values in the message descriptor for the message are candidates for retrieval using that message descriptor on the MQGET call.</td>
</tr>
</tbody>
</table>

**Buffer length**

It is important that the value used for buffer length *not* be larger than the actual buffer in your program. If it is, the MQGET can returning a message that may be larger than the work area you have established and non-buffer storage can be overwritten.

It is safest to ensure that the buffer length and buffer agree on the length. But, it is possible to set the length parameter to less than the actual size of the buffer. If the message is no larger than the stated buffer length, it will be placed into the buffer. It is important to remember that there may be data remaining from a previous get beyond the current buffer length.

If a message is larger than the buffer length specified, it is possible to retrieve that portion of the message that will fit in the buffer and discard any excess by using the MQGMO_ACCEPT_TRUNCATED_MESSAGE option. In this case, the message is removed from the queue. If this is not specified, the MQGET will fail if a message
larger than the buffer is encountered. However, the buffer will contain the portion of
the message that will fit, even though that message remains on the queue. In both
cases, the length of the complete application data portion of the message is returned
in data length.

MQGET reason codes

Some reason codes returned by MQGET are:

<table>
<thead>
<tr>
<th>Reason code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQRC_NONE</td>
<td>MQGET completed successfully.</td>
</tr>
<tr>
<td>MQRC_CONNECTION_BROKEN</td>
<td>The connection to the queue manager has been lost.</td>
</tr>
<tr>
<td>MQRC_GET_INHIBITED</td>
<td>Gets have been disabled for the queue.</td>
</tr>
<tr>
<td>MQRC_HCONN_ERROR</td>
<td>The connection handle is not valid.</td>
</tr>
<tr>
<td>MQRC_HOBJ_ERROR</td>
<td>The object handle is not valid.</td>
</tr>
<tr>
<td>MQRC_MD_ERROR</td>
<td>The message descriptor is not valid</td>
</tr>
<tr>
<td>MQRC_NO_MESSAGE_AVAILABLE</td>
<td>There are no messages available on the queue.</td>
</tr>
<tr>
<td>MQRC_NOT_CONVERTED</td>
<td>The application data was not converted, even though the MQGMO_CONVERT option was specified.</td>
</tr>
<tr>
<td>MQRC_NOT_OPEN_FOR_INPUT</td>
<td>The queue was not previously opened for input.</td>
</tr>
<tr>
<td>MQRC_OPTIONS_ERROR</td>
<td>The options are not valid or are inconsistent.</td>
</tr>
<tr>
<td>MQRC_TRUNCATED_MSG_ACCEPTED</td>
<td>A truncated message was returned (processing was completed).</td>
</tr>
<tr>
<td>MQRC_TRUNCATED_MSG_FAILED</td>
<td>A truncated message was returned (processing was not completed).</td>
</tr>
</tbody>
</table>

MQGET pseudocode

Listing 12 contains pseudocode of an MQGET call.

**Listing 12. MQGET example**

```plaintext
DEFINE DEFAULT_QMGR AS CONSTANT ' '
DEFINE QMGR AS MQCHAR48
DEFINE CONN_HANDLE AS MQHCONN
DEFINE COMP_CODE AS MQLONG
DEFINE REASON_CODE AS MQLONG
DEFINE OBJ_DESC AS MQOD
DEFINE MYQUEUE_HANDLE AS MQHOBJ
DEFINE OPEN_OPTIONS AS MQLONG
DEFINE CLOSE_OPTIONS AS MQLONG
```
DEFINE MESSAGE_DESCRIPTOR AS MQMD
DEFINE GET_OPTIONS AS MQGMO
DEFINE MESSAGE_LENGTH AS MQLONG
DEFINE MESSAGE AS CHAR100
DEFINE LENGTH_OF_MESSAGE AS MQLONG

QMGR = DEFAULT_QMGR

CALL MQCONN(QMGR,
             CONN_HANDLE,
             COMP_CODE,
             REASON_CODE)

IF COMP_CODE NOT = MQCC_OK
    write an error message detailing the REASON_CODE
    terminate the program
    STOP
END-IF

OBJ_DESC.ObjectType = MQOT_Q
OBJ_DESC.ObjectName = 'MYQUEUE'
OBJ_DESC.ObjectQMgrName = DEFAULT_QMGR

OPEN_OPTIONS = MQOO_INPUT_SHARED + MQ00_FAIL_IF_QUIESCING

CALL MQOPEN(CONN_HANDLE,
             OBJ_DESC,
             OPEN_OPTIONS,
             MYQUEUE_HANDLE,
             COMP_CODE,
             REASON_CODE)

IF COMP_CODE NOT = MQCC_OK
    write an error message detailing the REASON_CODE
    terminate the program
    STOP
END-IF

GET_OPTIONS.Options = MQGMO_NO_WAIT + MQGMO_NO_SYNCPOINT

MESSAGE_LENGTH = 100

DO
    MESSAGE_DESCRIPTOR.MsgId = MQMI_NONE
    MESSAGE_DESCRIPTOR.CorrelId = MQCI_NONE

    CALL MQGET(CONN_HANDLE,
                MYQUEUE_HANDLE,
                MESSAGE_DESCRIPTOR,
                GET_OPTIONS,
                MESSAGE_LENGTH,
                MESSAGE,
                LENGTH_OF_MESSAGE,
                COMP_CODE,
                REASON)

    UNTIL REASON_CODE NOT = MQRC_NO_MSG_AVAILABLE

    IF COMP_CODE NOT = MQCC_OK
        write an error message detailing the REASON_CODE
        terminate the program
        STOP
    END-IF

    CLOSE_OPTIONS = MQCO_NONE

    CALL MQCLOSE(CONN_HANDLE,
                  MYQUEUE_HANDLE,
In the pseudocode above, the program gets messages in a loop that terminates when no more messages are available. The message ID and correlation ID fields are set to MQMI_NONE and MQCI_NONE, respectively, before each MQGET. The reason for this is discussed in Controlling message retrieval.

Sending replies to the correct reply-to queue

When a program issues an MQGET, it might be designed to interrogate the message descriptor message type field to validate that the message is a request. If not, an error process might be invoked if the responding application always expects request type messages.

Once the message type is validated and business processing is completed, a reply message can be constructed to the reply-to queue on the reply-to queue manager.

If the queue that is receiving the responses is not on the local queue manager, the reply-to queue manager is significant. The queue manager looks for a transmission queue with the name of the reply-to queue manager, and that is where the message will be placed (with the proper transmission header containing routing information) for onward transmission by the channel.

Replying pseudocode

Listing 13 below contains pseudocode showing the steps in building and sending a reply message. Details outside of these steps have been left out to simplify the code and to focus on the reply processing.

Listing 13. Reply example
DEFINE IN_MSG_DESC AS MQMD
DEFINE OUT_MSG_DESC AS MQMD

CALL MQGET(...,
...,
IN_MSG_DESC,
...)

IF IN_MSG_DESC.MsgType = MQMT_REQUEST
formulate reply
OUT_MSG_DESC.MsgType = MQMT_REPLY
OBJ_DESC.ObjectName = IN_MSG_DESC.ReplyToQ
OBJ_DESC.ObjectQMgrName = IN_MSG_DESC.ReplyToQMgr

CALL MQPUT1(...,
OBJ_DESC,
OUT_MSG_DESC,
...)

END-IF

Section 11. MQPUT1

Overview

As with MQPUT, MQPUT1 is used to place messages on queues. With MQPUT1, however, no associated MQOPEN and MQCLOSE calls are required, as these two housekeeping calls are implied in the MQPUT1. MQPUT1 is commonly used when a server program is servicing requests from many different users and the message descriptor of each request specifies a different reply-to queue. Using MQPUT1 avoids the overhead of multiple MQOPEN and MQPUT calls.

The syntax of the MQPUT1 call is:

MQPUT1 (Hconn, ObjDesc, MsgDesc, PutMsgOpts,
BufferLength,
Buffer, CompCode, Reason)

The MQPUT1 parameters are a combination of those used for MQOPEN (object descriptor) and those used for MQPUT (message descriptor and put message options), but not including the object handle that is used with MQPUT. Since the open, put, and close are all done with one call, an object handle is not assigned.
MQPUT1 cannot be used with a model queue.

MQPUT1 security

To successfully execute an MQPUT1, the user must have proper access that will enable the queue to be opened for input. Since no explicit MQOPEN is executed, the access control checking that is available on the platform where the program is running must be done when the MQPUT1 is executed. So, where MQPUT would never return a reason code of MQRC_NOT_AUTHORIZED, MQPUT1 might do so.

MQPUT1 pseudocode

Listing 14 below contains pseudocode for an MQPUT1 call.

**Listing 14. MQPUT1 example**

```plaintext
DEFINE DEFAULT_QMGR AS CONSTANT ' '
DEFINE QMGR AS MQCHAR48
DEFINE CONN_HANDLE AS MQHCONN
DEFINE COMP_CODE AS MQLONG
DEFINE REASON_CODE AS MQLONG
DEFINE OBJ_DESC AS MQOD
DEFINE MESSAGE_DESCRIPTOR AS MQMD
DEFINE PUT_OPTIONS AS MQPMO
DEFINE MESSAGE_LENGTH AS MQLONG
DEFINE MESSAGE AS CHAR100

QMGR = DEFAULT_QMGR
CALL MQCONN(QMGR, CONN_HANDLE, COMP_CODE, REASON_CODE)

IF COMP_CODE NOT = MQCC_OK
   write an error message detailing the REASON_CODE
   terminate the program
   STOP
END-IF

OBJ_DESC.ObjectType = MQQT_Q
OBJ_DESC.ObjectName = 'MYQUEUE'
OBJ_DESC.ObjectQMgrName = DEFAULT_QMGR

MESSAGE_LENGTH = 11
MESSAGE = 'Hello World'

CALL MQPUT1(CONN_HANDLE, OBJ_DESC, MESSAGE_DESCRIPTOR, PUT_OPTIONS, MESSAGE_LENGTH, MESSAGE, COMP_CODE, REASON)
```
IF COMP_CODE NOT = MQCC_OK
   write an error message detailing the REASON_CODE
   terminate the program
   STOP
END-IF

CALL MQDISC(CONN_HANDLE,
              COMP_CODE,
              REASON_CODE)

IF COMP_CODE NOT = MQCC_OK
   write an error message detailing the REASON_CODE
   terminate the program
   STOP
END-IF

Compare this to Listing 5 showing the MQPUT call. Notice there are no variables for
the object handle, open options, or close options, since these are not required.

Section 12. Working with messages sample code

In this section you'll review and run a sample program that illustrates MQPUT and
MQPUT1 calls.

The Transfer program MQGETs messages from an input queue and attempts to
MQPUT them to an output queue. Not all of the messages can be MQPUT
successfully, and will fail because the messages are too long (the MAXMSGL
attribute of the output queue has been set for this to occur). For messages that
cannot be put to the output queue, Transfer uses MQPUT1 to send the messages to
a reply-to queue.

I have written another program, FeedTran, which reads lines from a file and puts
them to a queue, much like the first sample program, MQPut. The queue to which
FeedTran puts the messages is the input queue used by Transfer. FeedTran also
creates the reply-to queue to which Transfer returns the messages that cannot be
put to the output queue. FeedTran monitors the reply-to queue and prints the
returned messages to the console.

Because the focus in this sample is MQPUT and MQPUT1, I will not review the
FeedTran program. I encourage you to look it over and understand how it works.
We'll look at a sample program similar to it in Dynamic queues and managing
messages sample code.
Review the sample code

You have already downloaded and imported the sample code into Application Developer, so you can go straight to taking a look at the code. As before, you'll see the code in small pieces and what each piece is doing. Let's first look at the class declaration, along with some declared constants and the main() method, shown in Listing 15.

Listing 15. Transfer.java class declaration

```java
public class Transfer {
    private static final String qManager = "QMC1";
    private static final String inputQName = "CERT.INPUT";
    private static final String outputQName = "CERT.OUTPUT";
    public static void main(String[] args) {
        new Transfer().run();
    }
}
```

Three constant strings are defined to name the queue manager and the input and output queues. The main() method merely creates an instance of the class and calls its run() method. The run() method is the only other method in the class.

Now let's look at the first several lines of the run() method in Listing 16.

Listing 16. Start of run() method

```java
public void run() {
    try {
        MQQueueManager qMgr = new MQQueueManager(qManager);
        int openOptions = MQC.MQOO_INPUT_SHARED
            + MQC.MQOO_FAIL_IF_QUIESCING;
        MQQueue inputQueue = qMgr.accessQueue(inputQName, openOptions);
        openOptions = MQC.MQOO_OUTPUT + MQC.MQOO_FAIL_IF_QUIESCING;
        MQQueue outputQueue = qMgr.accessQueue(outputQName, openOptions);
        MQGetMessageOptions getOptions = new MQGetMessageOptions();
        getOptions.waitInterval = 10000;
    }
```

First, a connection is made to the queue manager, and the input queue and the output queue are opened. Then an MQGetMessageOptions object is created, and the wait interval field is set to 10000 (10 seconds).
Listing 17 shows the heart of the program.

**Listing 17. Getting and putting messages**

```java
boolean msgsAvailable = true;
while (msgsAvailable) {
    try {
        getOptions.options = MQC.MQGMO_WAIT
            + MQC.MQGMO_NO_SYNCPOINT
            + MQC.MQGMO_FAIL_IF_QUIESCING
            + MQC.MQGMO_ACCEPT_TRUNCATED_MSG;

        MQMessage msg = new MQMessage();
        msg.messageId = MQC.MQMI_NONE;
        msg.correlationId = MQC.MQCI_NONE;
        inputQueue.get(msg, getOptions);
        String replyQName = msg.replyToQueueName;

        MQPutMessageOptions putOptions = new MQPutMessageOptions();
        putOptions.options = MQC.MQPMO_NO_SYNCPOINT
            + MQC.MQPMO_FAIL_IF_QUIESCING;

        try {
            outputQueue.put(msg, putOptions);
        } catch (MQException mqex) {
            msg.messageType = MQC.MQMT_REPORT;
            msg.feedback = mqex.reasonCode;
            msg.messageId = MQC.MQMI_NONE;
            msg.correlationId = MQC.MQCI_NONE;
            msg.report = MQC.MQRO_NONE;
            qMgr.put(replyQName, msg, putOptions);
        }
    }
}
```

First a Boolean is declared and initialized to `true`, and a loop is entered that will terminate when the Boolean is `false`.

Then, the get options are set, a new message is created and initialized, and the get is performed.

Note the `try` block declared as the first line within the loop. This is the second `try` block declared in the program. The first `try` block, declared as the first line of the `run()` method, will catch errors during the connect to the queue manager and opening of the queues. If any of these fail, no more processing can be accomplished, and the program will end. This second `try` block is meant to catch errors that occur during the get. In particular, you want to determine if the reason code returned by any exception is MQRC_NO_MSG_AVAILABLE. You'll see that code a little further on.

Once the get is performed, the reply-to queue name is saved in a variable for later use. Then, a put is performed to the output queue.
Notice the third `try` block here. You want to only catch errors that occur during the put. If an error does occur, the `catch` will be invoked. It sets the message type to report, fills in the feedback field with the reason code from the exception, initializes the message ID and correlation ID fields, sets the report field to MQRO_NONE, and then does an MQPUT1 to the reply-to queue.

At this point, you're probably thinking "What MQPUT1? I don't see an MQPUT1." Look carefully at the put to the reply-to queue and the put to the output queue. The put to the output queue uses the `put()` method on the queue object. That corresponds to an MQPUT. The put to the reply-to queue uses the `put()` method on the queue manager object. That's the MQPUT1. If you think about it, it makes sense. You opened a queue by invoking a method on the queue manager (`accessQueue()`). Since MQPUT1 does an open, a put, and a close in one operation, it's the queue manager that would have to have the MQPUT1 functions.

One more point on Listing 17. If the put to the reply-to queue fails, since it is being performed in the `catch` associated with the third `try`, then that exception will be caught in the second `try` block that was declared.

Now you can look at the `catch` block of the second `try`, as shown in Listing 18.

Listing 18. Second catch block

```java
} catch (MQException mqex) {
    msgsAvailable = false;
    switch (mqex.reasonCode) {
    case MQException.MQRC_NO_MSG_AVAILABLE:
        System.out.println("No more input messages");
        break;
    default:
        System.out
            .println("A WebSphere MQ Error occurred: Completion Code "
            + mqex.completionCode
            + " Reason Code "
            + mqex.reasonCode);
        mqex.printStackTrace();
        break;
    }
}
```

Here, the Boolean that controls the loop is set to `false`. Then the reason code is checked. If no more messages are available, a message to that effect is printed to the console. If any other error occurred, the error is displayed.

Listing 19 below shows the end of the program: the queues are closed, the program disconnects from the queue manager, and a message indicating the end of the program is printed to the console. Listing 19 also shows the `catch` from the outermost `try` block.

Listing 19. Exception handling and end of program
Run the sample code

Now you're ready to run the code.

1. Create the input queue, the output queue, and a model queue from which FeedTran will create the reply-to queue. Open a Windows command prompt.

2. Type `runmqsc < setup02.txt` and press Enter.

3. Note that the MAXMSGL attribute of the output queue has been set to 25.


5. Run the FeedTrans program to place messages on the queue for Transfer. In Application Developer, right-click `FeedTran.java` in the Package Explorer view and select Run > Java Application.

6. Right-click `Transfer.java` in the Package Explorer view and select Run > Java Application.

7. Each program will display messages in its own console. Switch between the consoles by selecting Display Selected Console, outlined in red in Figure 6.
8. Once the programs have completed, the console for Transfer should appear similar to Figure 7. The lines in red are messages to System.err printed by WebSphere MQ base Java. Five messages were too long for the output queue (reason code 2030).

**Figure 7. Transfer console after program end**

9. The console for FeedTran should appear, as in Figure 8. Note the five messages that were displayed with feedback of 2030.

**Figure 8. FeedTran console after program end**
Section 13. Controlling message retrieval

This section explores techniques for controlling message retrieval.

MsgId and CorrelId

The MsgId and CorrelId fields of the message descriptor are used to provide a unique identity to a message (MsgId), and to correlate a reply message to a previously sent request (CorrelId). When an application sends a message, it can set the MsgId or have the queue manager generate a unique MsgId (by setting the MsgId to MQMI_NONE). In either case, the actual value of the MsgId field is returned to the application after the message is put to the queue. In most cases, the
CorrelId is left by the queue manager as assigned by the application.

An application that processes an incoming message and builds a reply message can copy the Msgld field from the original message to the CorrelId field of the reply message. Once the reply message is sent back to the requesting program, it can compare the CorrelId of the reply to the Msgld of the original request message to ensure that the reply received is for the request that was sent.

As you'll see later, it is also possible to specify that an MQGET only retrieve messages with a particular Msgld, CorrelId, or both.

Msgld and CorrelId are defined as MQBYTE24 and are not converted. These fields should be treated as a sequence of bytes and not as a string of characters.

**Generating Msgld and CorrelId**

As discussed earlier, an application can create its own Msgld for a message or have the queue manager generate a unique Msgld by setting Msgld to MQMI_NONE. And, usually, the CorrelId field is left alone by the queue manager.

On most systems, there is another way for Msgld to be set. MQPMO_NEW_MSG_ID can be set in the put message options to indicate that the queue manager should generate a unique identifier for Msgld, overriding any value that the application placed in that field.

Additionally, MQPMO_NEW_CORREL_ID can be set in the put message options to indicate that the queue manager should generate a unique identifier for CorrelId, again overriding any value that the application placed in that field. This is the only way to have the queue manager generate a CorrelId.

The use of MQPMO_NEW_MSG_ID and MQPMO_NEW_CORREL_ID can make the task of setting the Msgld and CorrelId easier for an application.

**Retrieval by Msgld and CorrelId**

An application can specify when executing an MQGET that only messages with a particular Msgld, or a particular CorrelId, or both, be returned. The application sets either or both of these fields in the message descriptor before issuing the MQGET. The queue manager then scans the queue for messages that match the Msgld or CorrelId as set by the application.

Using MQGET with Msgld or CorrelId can have a performance impact if there are many messages on the queue; this should be taken into consideration during application design.

**Retrieving every message**
To retrieve the first available message on a queue, regardless of MsgId and CorrelId, an application should set these fields to MQMI_NONE and MQCI_NONE, respectively. However, once an MQGET returns a message, these fields contain the values of the message just retrieved. Thus, an application must set these fields back to MQMI_NONE and MQCI_NONE before issuing another MQGET to ensure that the next available message, regardless of MsgId and CorrelId, is retrieved. Go back and look at Listing 12 to see that this is done in that program each time through the loop to MQGET messages.

**Request/reply queue consideration**

Having the CorrelId of a reply message set to the MsgId of a previously sent request message is most useful in a situation where multiple applications are using the same queue to receive replies. By only retrieving messages that have a CorrelId matching a previously sent request MsgId, the application does not remove messages from the reply queue belonging to other applications which are sharing that queue.

When designing a WebSphere MQ solution, it is important to consider if applications should share a reply queue or if each application should have its own. Using separate reply queues, which are often dynamic queues for easier administration, can limit the use of retrieving by CorrelId. Retrieving by CorrelId might still be necessary in this situation, however, if the requesting application sends multiple requests before it begins to receive any reply messages.

**MatchOptions**

On most platforms, there is a field available in the get message options structure called MatchOptions. This field can be set to MQMO_NONE to indicate that the next available message, regardless of MsgId and CorrelId, should be retrieved. This eliminates the need to reset MsgId and CorrelId before the next MQGET.

MatchOptions can also be set to MQMO_MATCH_MSG_ID or MQMO_MATCH_CORREL_ID to indicate retrieval by MsgId or CorrelId. The default is for MatchOptions to be set to both of these options. With these options set, the MQGET uses the values in MsgId or CorrelId to retrieve the appropriate message. Even with these options set, however, setting MsgId to MQMI_NONE and CorrelId to MQCI_NONE will return the next available message, regardless of MsgId or CorrelId.

**Indexes**

On z/OS, it is possible for the WebSphere MQ administrator to define either MsgId or CorrelId as an index. For example, if IndexType on a queue is MsgId, an index of all MsgIds on the queue is maintained and message retrieval by MsgId will be faster, even if there are many messages on the queue. In this instance, however, retrieving by CorrelId will result in slower processing, since it is not possible to specify an
IndexType of both.

Browse

If an application wants to look at a message without removing it from the queue, the MQGMO_Options field can include either MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT. The browse is unidirectional; there is no way to browse backwards in a queue.

In order to browse, a queue must be opened with the MQOO_BROWSE open option. Then, when the application uses the MQGMO_BROWSE_... option, instead of the destructive removal of messages, the application merely moves along through the messages in the queue, retrieving a copy of the next available message into its buffer.

When the open option for browsing is used, the queue manager creates a unique pointer that is associated with the object handle. This pointer is called the browse cursor. It is the browse cursor that monitors which message is currently being pointed to by the application. The browse cursor is not something that can be manipulated by the program except by issuing MQGET using one of the MQGMO_BROWSE_... options.

It is possible to combine the MQGMO_WAIT option with the browse. If planning to use this option, it is strongly advised that a wait interval be specified.

Since the browse is unidirectional, it is possible that an application can be browsing messages on a queue with a message delivery sequence of PRIORITY and reach the end of available messages (MQRC_NO_MSG_AVAILABLE) while new messages of a higher priority have been placed on the queue. In this situation, issue an MQGET with the MQGMO_BROWSE_FIRST option to reposition the browse cursor at the start of the queue.

When a message is found that the application wishes to remove from the queue, a further MQGET must be issued using the option MQGMO_MSG_UNDER_CURSOR. This will result in the destructive removal of the message pointed to by the browse cursor from the queue.

**Browsing the same message**

If an application has combined the MQOO_BROWSE and MQOO_INPUT_SHARED options when the queue is opened, it is possible that other programs might also be browsing and removing messages from the queue.

Two applications could browse the same message at the same time. Either one might then decide to use the MQGMO_MSG_UNDER_CURSOR option to remove
the message. If the second program then attempts the same, that call will fail with MQRC_NO_MSG_UNDER_CURSOR. In effect, the cursor is pointing to an empty slot in the queue.

You can use the MQGMO_LOCK option combined with MQGMO_BROWSE_... to avoid this problem. This will make the message being browsed invisible and unavailable to any other application that may also be browsing or getting messages from the queue. There is an MQGMO_UNLOCK option to allow for an explicit unlock of a previously locked message. The message will also be unlocked by a successful MQCLOSE on the queue.

**Browsing pseudocode**

Listing 20 shows pseudocode for browsing messages on a queue. Housekeeping calls have been abbreviated to simplify the code and focus on the browsing function.

**Listing 20. Browsing example**

```plaintext
DEFINE COMP_CODE AS MQLONG
DEFINE REASON_CODE AS MQLONG
DEFINE GET_OPTIONS AS MQGMO

CALL MQCONN(...)
CALL MQOPEN(...)
GET_OPTIONS.Options = MQGMO_BROWSE_FIRST + MQGMO_LOCK
CALL MQGET(...)
DO WHILE COMP_CODE = MQCC_OK
    IF you need to process this message
        GET_OPTIONS.Options = MQGMO_MSG_UNDER_CURSOR
        CALL MQGET(...)  
        IF COMP_CODE NOT = MQCC_OK
            IF REASON_CODE NOT = MQRC_NO_MSG_UNDER_CURSOR
                /* investigate what went wrong */
            END-IF
        ELSE
            /* process the message */
        END-IF
    END-IF
    GET_OPTIONS.Options = MQGMO_BROWSE_NEXT + MQGMO_LOCK
    CALL MQGET(...)  
END-DO
/* find out why we terminated the loop */
```
Wait

Because WebSphere MQ applications frequently communicate across a network or await a response from a triggered program, a reply is invariably not available immediately after the request message has been put. In these situations, if the application is designed to by synchronous, the MQGMO_WAIT option can be useful.

If an application uses MQGMO_WAIT, it is recommended that the MQGMO_FAIL_IF_QUIESCING option is used as well. This allows the call to be terminated if the queue manager is shutting down, returning MQRC_Q_MGR_QUIESCING.

The wait can also terminate if conditions regarding the queue’s ability to deliver messages changes. For instance, if the WebSphere MQ administrator changes the get attribute for the queue from allowed to inhibited, the call would then fail with MQRC_GET_INHIBITED.

WAIT with WaitInterval

Within the get message option structure is a field called WaitInterval to be used to set a time period (in milliseconds) during which the wait is in effect. 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 in the WaitInterval field, allows the program to establish a reasonable time to wait for a message. If none arrives during that time, the MQGET completes with an MQCC_FAILED completion code and a reason of MQRC_NO_MSG_AVAILABLE. During the wait interval, if a message arrives, the wait is immediately satisfied.

Note that the default wait interval is MQWI_UNLIMITED (forever). This should be used only with extreme discretion, and always with MQGMO_FAIL_IF_QUIESCING.

WAIT with WaitInterval pseudocode

Listing 21 below shows pseudocode for waiting for messages with a wait interval. Housekeeping calls have been abbreviated to simplify the code and focus on the wait function.

Listing 21. Waiting example

```c
DEFINE COMP_CODE AS MQLONG
DEFINE REASON_CODE AS MQLONG
DEFINE MESSAGE_DESCRIPTOR AS MQMD
DEFINE GET_OPTIONS AS MQGMO
```
CALL MQCONN(....)
CALL MQOPEN(....)
MESSAGE_DESCRIPTOR.MsgId = SomeValue
MESSAGE_DESCRIPTOR.CorrelId = SomeOtherValue
GET_OPTIONS.Options = MQGMO_WAIT + MQGMO_FAIL_IF_QUIESCING
GET_OPTIONS.WaitInterval = 300000
CALL MQGET(....)
IF COMP_CODE NOT = MQCC_OK
    IF REASON_CODE = MQRC_NO_MSG_AVAILABLE
        /* we were woken up but there was no */
        /* message available - timeout expired */
        END-IF
    IF REASON_CODE = MQRC_GET_INHIBITED
        /* we were woken up but we are not */
        /* now allowed to get a message */
        END-IF
ELSE
    /* we got the message OK, now process it */
END-IF

WAIT and multiple applications

If multiple programs are processing a queue for input, it is possible that a message that has satisfied a wait could be removed by another program that is, perhaps, just getting any available message from the queue. Which application will actually retrieve the message from the queue is unpredictable in this situation; it depends on the relative operating system priorities of the competing tasks.

Applications should therefore always be designed to allow for the possibility of receiving MQRC_NO_MSGAVAILABLE, even if the wait was set up with no wait interval.

MQGET with SET_SIGNAL

An option on the MQGET on some WebSphere MQ implementations allows the operating system to notify (or signal) the program when an expected message arrives on the queue. You cannot use the MQGMO_SET_SIGNAL option in conjunction with the MQGMO_WAIT option. However, the wait interval is permitted, and, in the case of IMS, is recommended to be used with the SET_SIGNAL option.

The advantage of MQGMO_SET_SIGNAL is that the application thread is freed to do other processing, relying on the operating system to notify the program when a message arrives.
A program can have outstanding signals set on multiple queue handles simultaneously. When the operating system notifies the application that a message has arrived for an outstanding signal request, the program must issue another MQGET to actually retrieve the message.

The notification of arrival is operating system specific. For example, on z/OS, a completion code is returned to the event control block (ECB) whose address is identified in the Signal1 field of the get message options structure. The completion code can notify the program that:

- The message has arrived on the queue.
- The wait interval expired and no message arrived.
- The signal has been cancelled (for example, gets have been inhibited on the queue).

**Expiry**

The value in the expiry field of the message descriptor is the message lifetime. This is a period expressed in tenths of a second, set by the application that puts the message. The message becomes eligible to be discarded if it has not been removed from the destination queue before this period of time elapses. The message is discarded when an MQGET call, browse or non-browse, occurs that would have returned the message had it not already expired.

The default value is MQEI_UNLIMITED for an unlimited lifetime.

The expiry value is usually used in conjunction with the wait interval. You should set the expiry value to that of the wait interval, as you are not willing to wait any longer than that amount of time.

---

**Section 14. Dynamic queue and managing messages sample code**

In this section you'll review and run a sample program that illustrates:

- Sending request messages
- Creating a dynamic queue on which to receive replies
- Waiting for replies to be sent
- Using MsgId and CorrelId to verify that retrieved messages are in fact
replies to the request message

The Request program reads lines from a file and puts them to a queue as request messages. After each request message is sent, it waits for reply messages to be received on a dynamic queue that it creates. If a reply message is received before the wait interval is exceeded, it retrieves the message and prints its contents to the console, indicating whether or not the reply message correlates to the request message previously sent.

I have written another program, EchoRequest, which gets the request messages sent by Request and sends reply messages to the dynamic queue created by Request. EchoRequest sends a number of replies to Request corresponding to the number found in the first byte of the request message. If there is a 1 in the first byte, it will send 1 reply; if there is a 5 in the first byte, it will send 5 replies; if there is a 9 in the first byte, it will send 9 replies, and so on. If there is a 0 (zero) in the first byte it will not send any replies.

Since the focus in this sample is on sending requests and waiting on replies, I won't review the EchoRequest program. I encourage you to look it over and understand how it works, though.

Review the sample code

As before, we'll review the code in small pieces and explain what each piece is doing. Let's first look at the class declaration, along with some declared constants and the main() method, shown in Listing 22.

**Listing 22. Request.java class declaration**

```java
public class Request {
    private static final String qManager = "QMC1";
    private static final String requestQName = "CERT.REQUEST";
    private static final String replyQName = "CERT.REPLY";
    private static final String dynamicQName = "CERT.REPLY.*";
    public static void main(String[] args) {
        new Request().run();
    }
}
```

Four constant strings are defined to name the queue manager, the request and reply queues, and the dynamic queue. The main() method merely creates an instance of the class and calls its run() method. The run() method is the only other method
Now let's look at the first several lines of the `run()` method, as shown in Listing 23.

**Listing 23. Start of run() method**

```java
public void run() {
    System.out.println("Start Request");
    try {
        MQQueueManager qMgr = new MQQueueManager(qManager);
        int openOptions = MQC.MQOO_OUTPUT + MQC.MQOO_FAIL_IF_QUIESCING;
        MQQueue requestQueue = qMgr.accessQueue(requestQName, openOptions);
        openOptions = MQC.MQOO_INPUT_SHARED + MQC.MQOO_FAIL_IF_QUIESCING;
        MQQueue replyQueue = qMgr.accessQueue(replyQName, openOptions,
                                             qManager, dynamicQName, null);
        System.out.println("Dynamic queue is " + replyQueue.name);
        MQPutMessageOptions putOptions = new MQPutMessageOptions();
        putOptions.options = MQC.MQPMO_NO_SYNCPOINT
                              + MQC.MQPMO_FAIL_IF_QUIESCING;

        First, a connection is made to the queue manager, and the request and reply queues
        are opened. The reply queue is a dynamic queue, and since the reply queue name
        supplied when opening the queue ended with an asterisk, the queue manager
        created the dynamic queue and generated a unique identifier to replace the asterisk.
        The application prints that name to the console after the queue is opened. Then, an
        MQPutMessageOptions object is created and initialized.

        Listing 24 shows reading lines from the file and putting messages to the request
        queue.

        **Listing 24. Reading a file and putting messages**
        ```
        BufferedReader br = new BufferedReader(new FileReader("testdata.txt"));
        String line = null;
        while ((line = br.readLine()) != null) {
            MQMessage msg = new MQMessage();
            msg.messageType = MQC.MQMT_REQUEST;
            msg.report = MQC.MQRO_EXCEPTION_WITH_DATA;
            msg.replyToQueueName = replyQueue.name;
            msg.format = MQC.MQFMT_STRING;
            msg.writeUTF(line);
            requestQueue.put(msg, putOptions);
            byte[] msgID = msg.messageId;
            MQGetMessageOptions getOptions = new MQGetMessageOptions();
            getOptions.waitInterval = 10000;
        ```
This code is similar to that in MQPut.java. Note that the message type is set to MQMT_REQUEST, a report is requested if an exception occurs, a reply-to queue is provided, and the format is set to MQFMT_STRING.

Once the message is put to the request queue, the message ID is saved to a variable for later referral. Then an MQGetMessageOptions object is created and the wait interval is set to 10 seconds.

Listing 25 shows the processing of reply messages.

Listing 25. Waiting for and receiving replies

```java
boolean msgsAvailable = true;
while (msgsAvailable) {
    try {
        getOptions.options = MQC.MQGMO_WAIT
            + MQC.MQGMO_NO_SYNCPOINT + MQC.MQGMO_CONVERT
            + MQC.MQGMO_FAIL_IF_QUIESCING
            + MQC.MQGMO_ACCEPT_TRUNCATED_MSG;
        MQMessage reply = new MQMessage();
        reply.encoding = MQC.MQENC_NATIVE;
        reply.characterSet = MQC.MQCCSI_Q_MGR;
        reply.messageId = MQC.MQMI_NONE;
        reply.correlationId = MQC.MQCI_NONE;
        replyQueue.get(reply, getOptions);
        getOptions.waitInterval = 1000;
        if (Arrays.equals(msgID, reply.correlationId)) {
            System.out.println("Correlated: " + reply.readUTF());
        } else {
            System.out.println("Non-correlated: " + reply.readUTF());
        }
        if (reply.messageType == MQC.MQMT_REPORT) {
            System.out.println("Report with feedback:" + reply.feedback);
        }
    }
}
```

A Boolean is declared and initialized to true, and a loop is entered that will terminate when the Boolean is false. Then, the get options are set, a new message is created and initialized, and the get is performed.

The code in the loop is within a try block. Like the second try block in Transfer.java, this try block is meant to catch errors that occur during the get, particularly checking for MQRC_NO_MSG_AVAILABLE to indicate that no more replies are available.

After the get, the wait interval is reduced to one second. Then, the message ID
saved earlier is compared to the correlation ID of the reply message and the message is printed to the console, indicating whether or not it is correlated to the request message. If the reply message is a report message, the feedback field is also printed to the console.

Once all replies have been received, the program will loop back to read a line from the file and send another request.

Listing 26 below shows the second catch block in the program, which is identical to the second catch block in Transfer.java.

Listing 26. Second catch block

```java
catch (MQException mqex) {
    msgsAvailable = false;
    switch (mqex.reasonCode) {
    case MQException.MQRC_NO_MSG_AVAILABLE:
        System.out.println("No more replies");
        break;
    default:
        System.out.
            println("A WebSphere MQ Error occured : Completion Code 
            + mqex.completionCode
            + " Reason Code " + mqex.reasonCode);
        mqex.printStackTrace();
        break;
    }
}
```

Once all lines of the file have been sent and all replies received, the program ends by closing the queues and disconnecting from the queue manager, as shown in Listing 27 below. The listing also shows the catch block from the outermost try block.

Listing 27. Exception handling and end of program

```java
requestQueue.closeOptions = MQC.MQCO_NONE;
requestQueue.close();
if (replyQueue.isOpen())
{
    replyQueue.closeOptions = MQC.MQCO_DELETE;
    replyQueue.close();
}
qMgr.disconnect();
System.out.println("End Request");
```
Run the sample code

First create the request queue. Use the model queue already created for the reply queue.

1. Open a Windows Command Prompt.
2. Type runmqsc < setup03.txt and press Enter.
3. Close the Windows Command Prompt.
4. Start the Request program.
5. In Application Developer, right-click Request.java in the Package Explorer view, and select Run > Java Application.
6. Right-click EchoRequest.java in the Package Explorer view, and select Run > Java Application.
7. Each program will display messages in its own console. You can switch between the consoles by selecting Display Selected Console.
8. During the execution of the programs, Request seems to pause from time to time. This pause is when a message is sent with 0 (zero) in the first
byte and no replies are sent. Request waits for 10 seconds for replies after sending a request.

9. Once the programs have completed, the console for Request should appear as shown in Figure 9. (Only a portion of the messages printed to the console are shown in Figure 9.)

**Figure 9. Request console after program end**

```
Start Request
Dynamic queue is CERT.REPLY.4534FBF902ED0B20
Correlated: 1 reply - message 1 ok
MQJE001: Completion Code 2, Reason 2033
No more replies
No more replies
MQJE001: Completion Code 2, Reason 2033
Correlated: 1 reply - message 3 1 too long
No more replies
MQJE001: Completion Code 2, Reason 2033
Correlated: 2 reply - message 4 ok
Correlated: 2 reply - message 4 ok
No more replies
MQJE001: Completion Code 2, Reason 2033
Correlated: 3 reply - message 5 ok
Correlated: 3 reply - message 5 ok
Correlated: 3 reply - message 5 ok
No more replies
MQJE001: Completion Code 2, Reason 2033
Correlated: 4 reply - message 6 ok
Correlated: 4 reply - message 6 ok
Correlated: 4 reply - message 6 ok
Correlated: 4 reply - message 6 ok
No more replies
MQJE001: Completion Code 2, Reason 2033
```
Section 15. Summary

This tutorial covered the major calls of the MQI, including the housekeeping calls and calls to put and get messages. It also discussed opening queues, the message descriptor, and techniques for controlling message retrieval. Completing the five tutorials in this series can help you gain the knowledge you need to prepare for Test 996, IBM WebSphere MQ V6.0, Solution Design, but nothing can replace the experience and knowledge you get from using the product and studying the documentation.

I hope you have found this tutorial helpful, and wish you luck as you prepare for your certification test.
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
<th>Download method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample code and configuration files</td>
<td>samples.zip</td>
<td>10KB</td>
<td>HTTP</td>
</tr>
</tbody>
</table>

*Information about download methods*
Resources

Learn

- **WebSphere MQ Solution Designer certification prep series**: Take this series of five tutorials to help prepare for the IBM certification Test 996, WebSphere MQ V6.0, Solution Design.

- Get certified as an IBM Certified Solution Designer - WebSphere MQ V6.0. Check out the objectives, sample assessment tests, and training resources for test 996, IBM Certified Solution Designer - WebSphere MQ V6.0.

- Read the IBM Redbook™ *WebSphere MQ V6 Fundamentals* to gain a broad technical introduction to WebSphere MQ.

- Use the WebSphere MQ library for detailed documentation on WebSphere MQ.

- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Download a free trial version of WebSphere MQ V6.0.

- Download a free trial version of Rational Application Developer V6.0.

- Build your next development project with IBM trial software, available for download directly from developerWorks.

Discuss

- Participate in the discussion forum for this content.

- Participate in developerWorks blogs and get involved in the developerWorks community.

About the author

Willy Farrell

Willy Farrell is a Senior Software Engineer in the IBM Developer Skills Program. As part of the developerWorks team, he provides relevant technical information and insight to developers on the latest e-business and industry trends and technologies through Web content, articles, speaking engagements, and consulting to faculty at IBM Academic Initiative member universities. He has been programming computers for a living since 1981, began using Java in 1996, and joined IBM in 1998. You can reach Willy at willyf@us.ibm.com.