Prepare for the IBM Certification Test 996, WebSphere MQ Solution Designer. This tutorial completes the series by continuing the discussion of the Message Queue Interface (MQI) begun in the previous tutorial. It is the fifth tutorial in a series of five tutorials.

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 covers additional topics of the MQI not covered in the previous tutorial,
including MQINQ, MQSET, transaction support, triggering, MQI security, message groups and segmentation, and distribution lists. This tutorial concludes the series.

Objectives

After completing this tutorial, you will be familiar with:

- MQINQ verb
- MQSET verb
- Transaction support
- Triggering
- Message groups and segmentation
- Distribution lists

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. MQINQ and MQSET

Overview

The MQINQ verb lets you interrogate all of the attributes of any queue, process, queue manager, or namelist. From within a program, you can use this call to discover such things as a queue’s maximum message length, the application pointed to by a process, the name of the queue manager’s dead letter queue, or the list of attributes for a namelist.

The MQSET verb lets you change attributes, but only the attributes of a queue. The attributes that you can set are primarily associated with triggering.

Both the MQINQ and MQSET calls use arrays of selectors to identity those attributes you want to interrogate or set. There is a selector for each attribute you can work with. The selector name has a prefix, determined by the nature of the attribute:

<table>
<thead>
<tr>
<th>Selector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQCA_</td>
<td>These selectors refer to attributes that contain character data, for example the name of a queue.</td>
</tr>
<tr>
<td>MQIA_</td>
<td>These selectors refer to the attributes that contain either numeric values (such as CurrentQueueDepth, the number of messages on a queue), or a constant value (such as Syncpoint, which indicates whether or not the queue manager supports syncpoints).</td>
</tr>
</tbody>
</table>

Before using the MQINQ or MQSET calls, your application must be connected to the queue manager, and you must use the MQOPEN call to open the object for setting or inquiring about attributes.

MQINQ

The MQINQ call can be used to find out the settings of attributes of WebSphere MQ objects. This includes queues, queue managers, processes, and namelists. The call
returns an array of integers and a set of character strings that contain the requested attributes. The object needs to be opened for inquiry before the MQINQ call is made.

The syntax of the MQINQ call is:

MQINQ (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount, IntAttrs, CharAttrLength, CharAttrs, CompCode, Reason)

MQINQ (and MQSET) require more parameters than any of the other MQI calls. The common parameters used on other calls include the connection handle (Hconn) and the object handle (Hobj) as the first two parameters. The last two, as with all MQI calls, are the completion code (CompCode) and reason code (Reason).

By breaking the other parameters into groups, it is fairly easy to understand them:

- SelectorCount and Selectors
- IntAttrCount and IntAttrs
- CharAttrLength and CharAttrs

**SelectorCount**
The total number of attributes that are included in the MQINQ. It is simply a count of how many attributes you are asking about. Selectors is an array (or list) of those attributes. Each attribute has a symbolic name assigned that can be used in place of the numeric value assigned to that attribute. All of the attributes are listed by object in the discussion of MQINQ in the *WebSphere MQ Application Programming Reference* (see Resources).

**IntAttrCount**
The total number of the selectors that are represented as integers. These include things like MQIA_CURRENT_Q_DEPTH, MQIA_Q_TYPE, or MQIA_MAX_Q_DEPTH. IntAttrs is an array (or list) that will be used to return the requested integer attributes.

**CharAttrLength**
The total length of all the character attributes that have been requested. CharAttrs is the name of a buffer area where all of the requested character attributes will be returned as one long string up to the length specified by CharAttrLength.

The character attributes require a bit more work to set up than the integer attributes. If the character attributes MQCA_Q_NAME and MQCA_Q_DESC were desired, the lengths of these two attributes (48 and 64, respectively) would be added together to set the value in CharAttrLength (112).
It is not necessary to know the length of each of these fields since the lengths also have symbolic names. In the example just given, the value of CharAttrLength can be obtained by adding MQ_Q_NAME_LENGTH and MQ_Q_DESC_LENGTH. Of course, you must ensure that you have a buffer that will accommodate the longest string of characters you would expect to have returned.

Responses are returned in the exact order requested. Assume that the Selectors are:

1. MQIA_CURRENT_DEPTH
2. MQCA_Q_NAME
3. MQIA_MAX_Q_DEPTH
4. MQCA_Q_DESC

A buffer of at least 112 characters has been set up for characters and an array has been set up for integer attributes. The program supplies the value of 2 in IntAttrCount and 112 in CharAttrLength. On return from the call, the value in the first position of the integer array will be that of the current depth (MQIA_CURRENT_Q_DEPTH), followed by the maximum depth (MQIA_MAX_Q_DEPTH). The buffer of character strings will contain the queue name (MQCA_Q_NAME) in the first 48 characters, immediately followed by the 64-character queue description (MQCA_Q_DESC). The program would need to break up the character string into the various pieces.

**MQINQ pseudo code**

Listing 1 shows pseudo code for an MQINQ call. Housekeeping calls have been abbreviated to simplify the code and focus on the MQINQ.

**Listing 1. MQINQ example**

```c
DEFINE CONN_HANDLE AS MQHCONN
DEFINE OBJ_HANDLE AS MQHOBJ
DEFINE COMP_CODE AS MQLONG
DEFINE REASON_CODE AS MQLONG
DEFINE SELECTORS AS ARRAY 5 MQLONG
DEFINE SELECTOR_COUNT AS MQLONG
DEFINE INT_COUNT AS MQLONG
DEFINE INTS AS ARRAY 3 MQLONG
DEFINE CHAR_COUNT AS MQLONG
DEFINE CHARS AS CHAR2000

CALL MQCONN(....)
CALL MQOPEN(....)
SELECTOR_COUNT = 4
```
SELECTORS is defined as an array with 5 entries, and SELECTOR_COUNT is a 4-byte field that will contain the number of SELECTORS. INT_COUNT will contain the number of SELECTORS that are integers, and INTS is the array where the integer values will be returned. The field named CHAR_COUNT will contain the total length of all the character attributes to be returned, and CHARS is the buffer where the string made up of all those character attributes will be returned.

Assume that the MQOPEN includes the MQOO_INQUIRE option.

Four selectors will be requested for this MQINQ. The four desired attributes are placed in the SELECTOR array. The INT_COUNT is set to 2 and the lengths of the character attributes are added together to establish CHAR_COUNT.

Finally, the MQINQ call is performed. If the call is successful, the application can retrieve the requested information from INTS and CHARS.

**MQSET**

The syntax of the MQSET call is:

```
MQSET (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount, IntAttrs, CharAttrLength, CharAttrs, CompCode, Reason)
```

As you can see, the parameters are identical in type and order to MQINQ. The difference is in the direction of the flow of some of the data. Fields that were output fields on MQINQ (IntAttrs and CharAttrs) are input to MQSET.

A more important difference is that MQSET is not permitted for any object except a queue. And, only a small subset of attributes can be changed. The attribute that indicates if PUTs are permitted (MQIA_INHIBIT_PUT) is settable for all queue types.
The MQIA_GET_INHIBITED attribute can only be set for local and alias queues. The following attributes for a local queue that are associated with triggering are also permitted to be set:

- MQCA_TRIGGER_DATA
- MQIA_TRIGGER_CONTROL
- MQIA_TRIGGER_DEPTH
- MQIA_TRIGGER_MSG_PRIORITY
- MQIA_TRIGGER_TYPE

MQIA_DIST_LISTS is also permitted to be set for those platforms that support distribution lists.

The queue must be opened using the MQOO_SET option. If any attribute fails to be set (perhaps one of the requested attributes is not permitted with MQSET), then none of the requested attributes are set and the call fails. If the call is successful, the changes take effect immediately.

**MQSET pseudo code**

Listing 2 shows pseudo code for an MQSET call. Housekeeping calls have been abbreviated to simplify the code and focus on the MQSET.

### Listing 2. MQSET example

```plaintext
DEFINE CONN_HANDLE AS MQHCONN
DEFINE OBJ_HANDLE AS MQHOBJ
DEFINE COMP_CODE AS MQLONG
DEFINE REASON_CODE AS MQLONG
DEFINE SELECTORS AS ARRAY 5 MQLONG
DEFINE SELECTOR_COUNT AS MQLONG
DEFINE INT_COUNT AS MQLONG
DEFINE INTS AS ARRAY 3 MQLONG
DEFINE CHAR_COUNT AS MQLONG
DEFINE CHARS AS CHAR2000

CALL MQCONN(....)
CALL MQOPEN(....)

SELECTOR_COUNT = 2
SELECTORS(1) = MQIA_TRIGGER_TYPE
SELECTORS(2) = MQIA_TRIGGER_DEPTH

INTS(1) = MQTT_DEPTH
INTS(2) = 10

INT_COUNT = 2
CHAR_COUNT = 0
```
The code that defines the various fields is exactly the same as for MQINQ.

The call intends to change the trigger type for the queue to allow a trigger on depth (MQIA_TRIGGER_TYPE). Since triggering on depth implies something other than one (that would be MQTT_FIRST), the number of messages desired to cause the trigger must be specified in the MQIA_TRIGGER_DEPTH attribute for the queue (in this case, 10).

Of the selectors being set up, both are integers, so INT_COUNT is set to 2 and CHAR_COUNT is set to 0.

Finally, the MQSET call is issued using the parameters that have been set up.

Section 3. Working with queue attributes sample code

In this section, you review and run sample code that inquires on and sets attributes on a queue.

Download the sample code

Download the .zip file containing the sample code and configuration files (samples.zip) to your system. Extract the contents of the file to a folder on your system. After extracting, you will have three files: MQAdditional.zip, setup04.txt, and setup05.txt.

Copy the two 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
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, then click Next.
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 MQAdditional.zip.
7. Click Open.
8. Select Select All, then click Finish.
9. Expand MQAdditional in the Package Explorer view, and expand com.ibm.cert996 to display the sample code classes.
10. Double-click MQInq.java.
11. Double-click MQSet.java.

Review the sample code

There are actually four programs you will run to exercise the MQINQ and MQSET functions. Two of those, MQInq.java and MQSet.java, I will review here. The other two programs, PutToRequest.java and MQUnset.java, are utility programs to assist with this exercise so we won't review them. I encourage you to look them over, however, and understand how they work.

MQInq.java

In this review of MQInq.java, we’ll look at 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, as shown in Listing 3.

Listing 3. MQInq.java class declaration

```java
public class MQInq {
    private static final String qManager = "QMCl";
```
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.

Listing 4 shows the first several lines of the run() method.

**Listing 4. Start of run() method**

```java
public void run() {
    System.out.println("Start MQInq");
    try {
        MQQueueManager qMgr = new MQQueueManager(qManager);
        int openOptions = MQC.MQOO_INQUIRE +
        MQC.MQOO_FAIL_IF_QUIESCING;
        MQQueue queue = qMgr.accessQueue(qName, openOptions);
        System.out.println("Inquiring on attributes of queue: "
            + queue.name);
```

A message is printed to the console indicating that MQInq has started, a connection is made to the queue manager, and the queue is opened for inquiry. Then a message containing the queue name is printed.

Listing 5 shows "inquiring" on and displaying the queue type of the queue.

**Listing 5. Inquiring on queue type**

```java
int queueTypeValue = queue.getQueueType();
String queueTypeName = "Unknown";
switch(queueTypeValue) {
    case MQC.MQQT_ALIAS:
        queueTypeName = "Alias";
        break;
    case MQC.MQQT_LOCAL:
        queueTypeName = "Local";
        break;
    case MQC.MQQT_MODEL:
        queueTypeName = "Model";
        break;
    case MQC.MQQT_REMOTE:
        queueTypeName = "Remote";
        break;
    default:  
```
If you are thinking that Listing 5 doesn't look like MQINQ, you're right. In the WebSphere MQ base Java classes, several of the queue attributes are defined as properties of the queue object. It's a simple matter of calling a `get<AttributeName>()` method to retrieve those attributes. Queue type is one of those attributes that is a property of the queue object. There are several others, as shown in Listing 6.

**Listing 6. Other queue properties**

```java
int maximumDepth = queue.getMaximumDepth();
System.out.println("Maximum depth: " + maximumDepth);

int currentDepth = queue.getCurrentDepth();
System.out.println("Current depth: " + currentDepth);

int maximumMsgLength = queue.getMaximumMessageLength();
System.out.println("Maximum message length: " + maximumMsgLength);

int inhibitGetValue = queue.getInhibitGet();
String inhibitGetName = "Unknown";
switch (inhibitGetValue) {
    case MQC.MQQA_GET_INHIBITED:
        inhibitGetName = "Inhibited";
        break;
    case MQC.MQQA_GET_ALLOWED:
        inhibitGetName = "Allowed";
        break;
    default:
        break;
}
System.out.println("Get: " + inhibitGetName);

int inhibitPutValue = queue.getInhibitPut();
String inhibitPutName = "Unknown";
switch (inhibitPutValue) {
    case MQC.MQQA_PUT_INHIBITED:
        inhibitPutName = "Inhibited";
        break;
    case MQC.MQQA_PUT_ALLOWED:
        inhibitPutName = "Allowed";
        break;
    default:
        break;
}
System.out.println("Put: " + inhibitPutName);
```

You can see that, for these attributes, retrieving their values is much simpler than setting up integer and character counts and arrays. However, not all queue attributes are available as properties. For those that are not, the WebSphere MQ base Java classes provide for a mechanism that should look familiar to you as MQINQ. The actual method corresponding to MQINQ is the `inquire()` method on the queue object. Listing 7 shows this method being used.
Listing 7. Using the inquire() method

```java
int[] selectors = { MQC.MQCA_BACKOUT_REQ_Q_NAME,
                   MQC.MQIA_BACKOUT_THRESHOLD }; int[] intAttrs = new int[1];
byte[] charAttrs = new byte[MQC.MQ_Q_NAME_LENGTH];
queue.inquire(selectors, intAttrs, charAttrs);
System.out.println("Backout queue name: " +
                   new String(charAttrs).trim());
System.out.println("Backout threshold: " + intAttrs[0]);
```

Here you see the selectors, integer attribute array, and character attribute array that you learned about in the discussion of MQINQ. No count fields are required, since Java allows the size of arrays to be discovered, and the assumption is that arrays only large enough to hold the provided input are supplied.

Listing 8 shows closing the queue, disconnecting from the queue manager, and the exception handling code.

Listing 8. End of program and exception handling

```java
queue.closeOptions = MQC.MQCO_NONE;
queue.close();
qMgr.disconnect();
System.out.println("End MQInq");
}
```

MQSet.java

MQSet.java is really quite simple, so I'll show its `run()` method all in one listing below. All of the sample programs in this tutorial have the same structure, of constant strings for queue manager name and queue name(s), then a `main()` method that instantiates the class and calls `run()`, so we won’t review those parts of the sample programs from here forward. I'll just show the `run()` methods.

Listing 9. MQSet.java run() method

```java
public void run() {
```
System.out.println("Start MQSet");

try {
    MQQueueManager qMgr = new MQQueueManager(qManager);
    int openOptions = MQC.MQOO_SET + MQC.MQOO_FAIL_IF_QUIESCING;
    MQQueue queue = qMgr.accessQueue(qName, openOptions);
    System.out.println("Setting attributes of queue: " + queue.name);
    System.out.println("Inhibiting get");
    queue.setInhibitGet(MQC.MQQA_GET_INHIBITED);
    System.out.println("Inhibiting put");
    queue.setInhibitPut(MQC.MQQA_PUT_INHIBITED);
    queue.closeOptions = MQC.MQCO_NONE;
    queue.close();
    qMgr.disconnect();
    System.out.println("End MQSet");
} catch (MQException mqex) {
    System.out.println("A WebSphere MQ Error occured : Completion Code " + mqex.completionCode + " Reason Code " + mqex.reasonCode);
    mqex.printStackTrace();
}

You can set properties on a queue using set<AttributeName>() methods that correspond to the get<AttributeName>() methods we saw in MQInq.java. There is also a set() method on the queue object that uses selectors, an integer attribute array, and a character array as parameters just as in the inquire() method, though that method is not shown here.

Run the sample code

1. Open a Windows command prompt.
2. Type runmqsc < setup04.txt and press Enter.
3. The setup04.txt command file redefines the CERT.REQUEST queue used in the previous tutorial, and clears all existing messages from the queue. There are some additional definitions and attributes in this command file that will be explained in Transaction support sample code.
5. In Application Developer, from the main menu select Window > Show
View > Console.

6. Right-click **MQInq.java** in the Package Explorer view, and select **Run > Java Application**.

7. The Console should look like Figure 1.

**Figure 1. Output from MQInq.java**

Start MQInq
Inquiring on attributes of queue: CERT.REQUEST
Queue type: Local
Maximum depth: 5000
Current depth: 0
Maximum message length: 4194304
Get: Allowed
Put: Allowed
Backout queue name: CERT.BACKOUT
Backout threshold: 5
End MQInq

8. Now you can run MQSet.java to change some attributes.

9. Right-click **MQSet.java** in the Package Explorer view, and select **Run > Java Application**.

10. Now run MQInq.java again. The output should look like Figure 2.

**Figure 2. Output from MQInq.java after running MQSet.java**
11. Run MQUnset.java to reverse the effects of MQSet.java.

12. Now run PutToRequest.java to put some messages on CERT.REQUEST, then run MQInq.java again. The output should look like Figure 3.

**Figure 3. Output from MQInq.java after running PutToRequest.java**

```
Start MQInq
Inquiring on attributes of queue: CERT.REQUEST
Queue type: Local
Maximum depth: 5000
Current depth: 21
Maximum message length: 4194304
Get: Allowed
Put: Allowed
Backout queue name: CERT.BACKOUT
Backout threshold: 5
End MQInq
```

13. Finally, you should clear the messages from CERT.REQUEST.

14. Open a Windows command prompt.
15. Type `runmqsc` and press **Enter**.
16. Type `clear ql(CERT.REQUEST)` and press **Enter**.
17. Type `end` and press **Enter**.
18. Close the Windows command prompt.

---

**Section 4. Transaction support**

**Transactions**

In the situation shown in Figure 4 below, the application would like to have both queue updates and the database update occur, or none of the updates if there should be an error in any one of them. This is sometimes called the *transactionality* of an application. In WebSphere MQ, the transactionality of MQ resources is ensured by getting and putting in syncpoint.

Figure 4. Transactional application

When a program gets a message from a queue in syncpoint, that message remains on the queue until the program commits the unit of work. However, the message is not available to be retrieved by other programs.
Similarly, when a program puts a message on a queue within a unit of work, the message is made visible to other programs only when the program commits the unit of work. This enables a program to ensure that the entire unit of work (perhaps consisting of an MQGET of a message, some validation of the input data and some processing, and the MQPUT of an output message) is successfully completed. During the processing, if a problem is encountered, all the activities associated with the unit of work can be backed out. Essentially, everything is put back as it was before any processing of that unit of work started. This is called the last point of consistency.

In addition to messages on queues, other resource managers, such as database managers, can use unit of work processing to ensure that their resources are coordinated.

In some cases, these different resource managers and their activities can be combined into a single unit of work. The unit of work is then generally coordinated by a transaction manager. CICS, Encina, and Tuxedo are examples of transaction managers.

Local and global transactions

A local unit of work involves only updates to WebSphere MQ resources (queues). A global unit of work includes the updates to resources belonging to other resource managers. Most implementations of WebSphere MQ support queue manager coordination of global units of work, in certain cases.

To start the global unit of work, a special call, MQBEGIN, is used. The only parameters supplied to MQBEGIN are the queue manager as input and the completion and reason codes as output. The updates to resources are handled as they would normally be when within a unit of work. When ready to commit, the MQCMIT call is used. MQBACK is used to back out all updates. These calls will be discussed later in more detail. Most of the widely available database managers can have their updates coordinated by WebSphere MQ.

External syncpoint coordination occurs when a syncpoint coordinator other than WebSphere MQ has been selected. In this situation, WebSphere MQ registers its interest in the outcome of the unit of work with the syncpoint coordinator so that it can commit or roll back any uncommitted get or put operations, as required. When an external coordinator is used, MQCMIT, MQBACK, and MQBEGIN may not be issued. Calls to these functions fail with the reason code MQRC_ENVIRONMENT_ERROR.

Syncpoint
Not all MQGETs and MQPUTs within a program need to be part of a unit of work. It is possible to change the MQGMO_SYNCPOINT or MQPMO_SYNCPOINT option to MQGMO_NO_SYNCPOINT or MQPMO_NO_SYNCPOINT, respectively.

With no syncpoint, messages retrieved from a queue are removed from the queue immediately (a destructive read), and messages put on an output queue are immediately available to be retrieved by other programs.

Using syncpoint delays deletion of messages that are retrieved from input queues, and delays delivery of messages that have been placed on output queues until the unit of work is committed.

Finally, remember that it is best to explicitly specify the desired syncpoint option since the default varies depending on the platform where the program executes.

MQPUT within syncpoint control

Assuming MQPMO_SYNCPOINT is specified, the message is physically placed on the queue immediately and the completion code indicates that result. However, the message, although physically on the queue, is not visible or available to other applications.

Messages can continue to be placed on the queue using syncpoint. They will count towards the number of messages on the queue (they are part of the physical count) and the current depth will reflect these messages, but any attempt to use MQGET to retrieve them will fail with MQRC_NO_MSG_AVAILABLE.

Only after the commit is issued will the messages be available to other applications.

MQGET within syncpoint control

The MQGET with syncpoint means that the application is given a copy of the message and the actual message is left on the input queue but marked for deletion and not available to other MQGETs. Once again, the physical number of messages on a queue include those that have been gotten in syncpoint. The messages are not physically removed until the commit is completed (delayed deletion).

If a rollback occurs, the message is simply made available again and the delete flag is turned off.

MQBEGIN

The MQBEGIN call begins a unit or work that is coordinated by the queue manager,
and that may involve external resource managers.

The syntax of the MQBEGIN call is:

```
MQBEGIN (Hconn, CompCode, Reason)
```

The parameters, again, are very simple. The connection handle is passed as input and the completion and reason codes are returned as output. Be sure to check the completion and reason codes after each call.

**MQCMIT**

Once a program is satisfied with all processing that has been done within a unit of work, the MQCMIT call can be used to process the completion of the unit of work, physically deleting any messages that were retrieved from input queues, and making messages that were put to output queues available to other applications.

The syntax of the MQCMIT call is:

```
MQCMIT (Hconn, CompCode, Reason)
```

MQCMIT cannot be used if the queue manager is running in an environment where an external coordinator is used. Any attempt to use MQCMIT, MQBACK, or MQBEGIN will result in a failure with MQRC_ENVIRONMENT_ERROR.

All MQPUTs and MQGETs that have been done in syncpoint, and any participating updates for other resource managers that occurred after the MQBEGIN call, will be considered part of the unit of work and will be committed. If a global unit of work is being committed, including non-WebSphere MQ resources, then a two-phase commit process is used. This means that all the participating resource managers are asked first to prepare for commit. Only if all the resource managers acknowledge that they are prepared to commit will the second phase (the actual commit) occur. Otherwise, all involved resources will be rolled back.

**MQBACK**

The MQBACK call works in the reverse of the MQCMIT call. All resources involved in the unit of work are put back to the way things were before the unit of work began. Messages retrieved are no longer marked for deletion, and they are available to be gotten again. Messages that have been put are removed from the output queues. In the case of a global unit of work, identified by MQBEGIN, the MQBACK results in all updates done by those other resource managers being undone as well.
The syntax of the MQBACK call is:

```
MQBACK (Hconn, CompCode, Reason)
```

The parameters, again, are very simple. The connection handle is passed as input and the completion and reason codes are returned as output. Be sure to check the completion and reason codes after each call.

After rollback, any messages that had been retrieved using an MQGET in syncpoint will have the BackoutCount incremented by one.

### Syncpoint pseudo code

Listing 10 shows pseudo code for using MQGET and MQPUT under syncpoint control. Housekeeping calls have been abbreviated to simplify the code and focus on the use of syncpoint.

#### Listing 10. Syncpoint example

```
DEFINE GET_OPTIONS AS MQGMO
DEFINT PUT_OPTIONS AS MQPMO
CALL MQCONN(....)
CALL MQOPEN(....)
GET_OPTIONS.Options = MQGMO_NO_WAIT + MQGMO_SYNCPOINT + MQGMO_FAIL_IF_QUIESCING
CALL MQGET(....)
PUT_OPTIONS.Options = MQPMO_SYNCPOINT + MQPMO_FAIL_IF_QUIESCING
CALL MQPUT1(....)
CALL MQCMIT(....)
```

The inclusion of MQGMO_SYNCPOINT in the get message options will result in the message being left on the queue, invisible and marked for deletion. A copy will be returned to the program when the MQGET completes.

The setting of MQPMO_SYNCPOINT in the put message options will result in the message being placed on the output queue, but not available to other options when the MQPUT1 completes.

The MQCMIT will cause the message on the input queue to be deleted and the message on the output queue to be made visible to other applications.
Poisoned messages

If a message is retrieved from a queue in syncpoint and the data in that message contains some invalid data that causes a backout, the message that contains the bad data is now, once again, available on the input queue. This could result in a loop where the message is continually rolled back and retrieved.

The message descriptor has a field that might be useful in this situation: the BackoutCount. When a message participates in a rollback, the counter that is maintained in this field is incremented by one. It is now possible to check the BackoutCount when the next MQGET is issued, and to perform some type of error handling routine if it is greater than zero.

There are also some backout attributes associated with the queue definition. The BOTHRESH (BackoutThreshold) attribute can be set to a value. There is also a BOQNAME (Backout Requeue Queue Name) attribute. The application could inquire of these attributes (using MQINQ) and then compare the value in the BackoutCount of the message descriptor to the threshold. If greater, the application could re-route the message to the queue named in BOQNAME. The queue manager takes no action on its own when the threshold is met during processing.

Rollback pseudo code

Listing 11 below shows pseudo code for rollback processing using BackoutCount and Backout Requeue Queue Name. Housekeeping calls have been abbreviated to simplify the code and focus on the use of syncpoint.

Listing 11. Rollback example

```assembly
DEFINE GET_OPTIONS AS MQGMO
DEFINT PUT_OPTIONS AS MQPMO
DEFINE ERROR_OBJDESC AS MQOD

CALL MQCONN(....)

CALL MQOPEN(....)
CALL MQINQ(...) /* BACKOUT_THRESHOLD BACKOUT_REQ_Q_NAME */
GET_OPTIONS.Options = MQGMO_NO_WAIT +
MQGMO_SYNCPOINT +
MQGMO_FAIL_IF_QUIESCING

CALL MQGET(....)
IF MQMD.BackoutCount > SAVED_BACKOUT_THRESHOLD
/* handle message as error and commit */
PUT_OPTIONS.Options = MQPMO_SYNCPOINT + MQPMO_FAIL_IF_QUIESCING
ERROR_OBJDESC.OBJECTNAME = SAVED_BACKOUT_REQ_Q_NAME
CALL MQPUT1(....)
CALL MQCMIT(....)

ELSE
CONTINUE
```
After connecting to the queue manager, the backout threshold and backout queue name are retrieved using MQINQ and saved. Once the MQGET is performed, the backout count is checked and, if the count is greater than the threshold, the message is put to the backout queue.

Remote updates

When WebSphere MQ applications are distributed across more than one platform, time independence means that there is no such thing as a traditional distributed unit of work. The message channel agents also use unit of work processing for assured delivery. If that message is placed on a transmission queue and no commit occurs to release it, the MCA cannot get the message off the queue to send it across the channel. Figure 5 below shows an illustration.

Figure 5. Asynchronous distributed unit of work

The philosophy must be one of an optimist: the message will be successfully delivered and the expected processing will occur. In cases where this does not happen, the application will need to include some explicit exception handling, called compensating transactions. In the example in Figure 5, if the database update in unit of work 3 occurs with an error, a report should be sent to the application performing unit of work 1 so that it can perform a database update to undo the write that occurred before the original message was sent.

Clients and transactions
A standard client application can participate in a unit of work that is managed by a queue manager to which it is connected. Within the unit of work, the client application can put messages to, and get messages from, the queues that are owned by that queue manager. The client application can then use the MQCMIT call to commit the unit of work or the MQBACK call to back out the unit of work. However, within the same unit of work, the client application cannot update the resources of another resource manager (for example, tables in a database). Using a WebSphere MQ extended transactional client removes this restriction.

A WebSphere MQ extended transactional client allows a client application, within the same unit of work:

- To put messages to, and get messages from, queues that are owned by the queue manager to which it is connected.
- To update the resources of a resource manager other than a WebSphere MQ queue manager.

This unit of work must be managed by an external transaction manager that is running on the same system as the client application. The unit of work cannot be managed by the queue manager to which the client application is connected. This means that the queue manager can act only as a resource manager, not as a transaction manager. It also means that the client application can commit or back out the unit of work using only the API provided by the external transaction manager. The client application cannot, therefore, use the MQI calls MQBEGIN, MQCMIT, and MQBACK.

Section 5. Transaction sample code

In this section you'll review and run some sample code that shows how to use transactions in WebSphere MQ.

Review the sample code

Let's start by looking at the MQSC command file you executed before running the last set of sample code, as shown in Listing 12.

Listing 12. setup04.txt

```
DEFINE QLOCAL('CERT.REQUEST') REPLACE + BOTHRESH(5) BOQNAME('CERT.BACKOUT')
```
This command file, as noted earlier, redefines the CERT.REQUEST queue used in the previous tutorial by adding a backout threshold (5) and backout queue name. It then clears any messages from CERT.REQUEST. Next, it defines a local queue, CERT.BACKOUT, to serve as the backout queue for CERT.REQUEST. Finally, it defines a remote queue, CERT.REMOTE, but this queue definition is incorrect, since the remote queue, remote queue manager, and transmission queue that are named do not exist. This will cause an error to any message that is MQPUT to this queue definition.

The Respond.java sample program reads messages from CERT.REQUEST under syncpoint and, if the message is a request message, sends the message back as a reply to the reply-to queue specified in the message, also under syncpoint. If no errors are encountered in sending the reply, Respond.java commits the unit of work. If an error is encountered, the backout count is compared to the backout threshold of CERT.REQUEST. If the backout count is less than the threshold, the unit of work is backed out. If it is equal to the threshold, then the message is put on the backout queue under syncpoint control, and the unit of work is committed. If the message retrieved from CERT.REQUEST is a datagram, the message is discarded, and the unit of work is committed.

I have written another program, ErrRequest.java, to put messages on CERT.REQUEST for Respond.java to process and to receive and display replies sent by Respond.java. It is almost identical to the sample program Request.java in the previous tutorial. Every fourth message that ErrRequest.java puts on CERT.REQUEST is a datagram message, and every fifth message it puts to the queue contains CERT.REMOTE as the reply-to queue. Since CERT.REMOTE is an incorrectly defined remote queue, this will cause an error in Respond.java.

Let's start by looking at the first few lines of the run() method in Respond.java, as shown in Listing 13.

Listing 13. Start of run() method

```
CLEAR QLOCAL('CERT.REQUEST')
DEFINE QLOCAL('CERT.BACKOUT') REPLACE
DEFINE QREMOTE('CERT.REMOTE') REPLACE +
  RNAME('QUEUE.ELSEWHERE') +
  RQMNAME('QM.SOMEWHERE') +
  XMITQ('SOME.XMITQ')
```
public void run() {
    System.out.println("Start Respond\n");
    try {
        MQQueueManager qMgr = new MQQueueManager(qManager);
        int openOptions = MQC.MQOO_INPUT_SHARED + MQC.MQOO_INQUIRE
                             + MQC.MQOO_FAIL_IF_QUIESCING;
        MQQueue requestQueue = qMgr.accessQueue(requestQName, openOptions);
        int[] selectors = {MQC.MQCA_BACKOUT_REQ_Q_NAME,
                           MQC.MQIA_BACKOUT_THRESHOLD};
        int[] intAttrs = new int[1];
        byte[] charAttrs = new byte[MQC.MQ_Q_NAME_LENGTH];
        requestQueue.inquire(selectors, intAttrs, charAttrs);
        String backoutQueueName = new String(charAttrs).trim();
        int backoutThreshold = intAttrs[0];
        MQGetMessageOptions getOptions = new MQGetMessageOptions();
        getOptions.waitInterval = 15000;

The method begins by displaying a message to the console that the program has
started, then connects to the queue manager, and opens the queue for input and
inquiry. Next, it inquires for the name of the backout queue and the backout
threshold (exactly as in MQInq.java) and saves those values. It then creates an
MQGetMessageOptions object and sets the wait interval to 15 seconds.

With that accomplished, the main processing loop of the program begins, as shown
below.

**Listing 14. Waiting for and receiving requests**

    boolean msgsAvailable = true;
    while (msgsAvailable) {
        try {
            getOptions.options = MQC.MQGMO_WAIT + MQC.MQGMO_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;
            requestQueue.get(msg, getOptions);
            String line = msg.readUTF();
            if (msg.messageType != MQC.MQMT_REQUEST) {
                System.out.println("Not a request: " + line + "\n");
                qMgr.commit();
                continue;
            }
        } catch (Exception e) { /* handle MQ errors */ }
    }

The method begins by displaying a message to the console that the program has
started, then connects to the queue manager, and opens the queue for input and
inquiry. Next, it inquires for the name of the backout queue and the backout
threshold (exactly as in MQInq.java) and saves those values. It then creates an
MQGetMessageOptions object and sets the wait interval to 15 seconds.

With that accomplished, the main processing loop of the program begins, as shown
below.

**Listing 14. Waiting for and receiving requests**

    boolean msgsAvailable = true;
    while (msgsAvailable) {
        try {
            getOptions.options = MQC.MQGMO_WAIT + MQC.MQGMO_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;
            requestQueue.get(msg, getOptions);
            String line = msg.readUTF();
            if (msg.messageType != MQC.MQMT_REQUEST) {
                System.out.println("Not a request: " + line + "\n");
                qMgr.commit();
                continue;
            }
        } catch (Exception e) { /* handle MQ errors */ }
    }
A Boolean is set to `true` and a loop is entered that will terminate when the Boolean is false. Note the second `try` block, which is meant to catch MQRC_NO_MSG_AVAILABLE errors during processing.

The get options are then set, including use of MQGMO_SYNCPOINT, an MQMessage object is created and initialized, and the get is issued. Once a message is received, it is checked to see if it is a request message. If not, this fact, along with the message contents, is displayed to the console.

Listing 15 shows the sending of reply messages if the message is a request.

**Listing 15. Sending the reply message**

```java
String replyQName = msg.replyToQueueName;
int msgBackoutCount = msg.backoutCount;
msg.correlationId = msg.messageId;
msg.messageId = MQC.MQMI_NONE;
msg.report = MQC.MQRO_NONE;
MQPutMessageOptions putOptions = new MQPutMessageOptions();
putOptions.options = MQC.MQPMO_SYNCPOINT + MQC.MQPMO_FAIL_IF_QUIESCING;
try {
    qMgr.put(replyQName, msg, putOptions);
    qMgr.commit();
} catch (MQException mqex) {
    if (msgBackoutCount < backoutThreshold) {
        qMgr.backout();
        System.out.println("Backing out for message: "+ line);
    } else {
        qMgr.put(backoutQueueName, msg, putOptions);
        qMgr.commit();
        System.out.println("Sending to backout queue for message: "+ line + "\n");
    }
}
```

First, the reply-to queue name and backout count are saved. Then the correlation ID of the message is set to the message ID, and message ID is initialized to MQMI_NONE. An MQPutMessageOptions object is created and initialized, including MQPMO_SYNCPOINT. Then the message is put to the reply queue. Notice this put is in a third `try` block, to catch errors specific to this put.

If the message is put successfully to the reply-to queue, then `commit()` is called on the queue manager object (this is the equivalent of MQCMIT) to commit the unit of work.

If an error is encountered, then the `catch` is invoked. Here, the backout count is compared to the backout threshold, and if less, the `backout` method is called on the...
queue manager (this is the equivalent of MQBACK). If the backout count is not less than the backout threshold, then the message is put to the backout queue and the unit of work is committed. The program displays messages to the console for each of these actions.

Listing 16 below shows the catch for the second try block, closing the input queue, disconnecting from the queue manager, and the catch for the first try block, which should all look very familiar to you by now.

**Listing 16. Exception handling and program end**

```java
Listing 16. Exception handling and program end

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

requestQueue.closeOptions = MQC.MQCO_NONE;
requestQueue.close();
qMgr.disconnect();
System.out.println("End Respond");
} catch (MQException mqex) {
    System.out.println("A WebSphere MQ Error occurred : Completion Code "
                        + mqex.completionCode + " Reason Code "
                        + mqex.reasonCode);
    mqex.printStackTrace();
}
```

**Run the sample code**

Now you can run the sample code. You'll need to start ErrRequest.java first, to put messages on the queue for Respond.java to process.

1. In Application Developer, right-click **ErrRequest.java** in the Package
Explorer view, and select **Run > Java Application.**

2. Right-click **Respond.java** in the Package Explorer view, and select **Run > Java Application.**

3. When both programs have finished running, the console for Respond.java should look like Figure 6.

*Figure 6. Console for Respond.java*
Start Respond

Not a request: message 4

Backing out for message: message 5
Backing out for message: message 5
Backing out for message: message 5
Backing out for message: message 5
Backing out for message: message 5
Sending to backout queue for message: message 5

Not a request: message 8

Backing out for message: message 10
Backing out for message: message 10
Backing out for message: message 10
Backing out for message: message 10
Backing out for message: message 10
Sending to backout queue for message: message 10

Not a request: message 12

Backing out for message: message 15
Backing out for message: message 15
Backing out for message: message 15
Backing out for message: message 15
Backing out for message: message 15
Sending to backout queue for message: message 15

Not a request: message 16

Backing out for message: message 20
Backing out for message: message 20
Backing out for message: message 20
Backing out for message: message 20
Backing out for message: message 20
Sending to backout queue for message: message 20

No more request messages

End Respond
Note that every fourth message was noted as not being a request message, and that every fifth message was backed out five times (the backout threshold) and then sent to the backout queue.

4. The console for ErrRequest.java should look like Figure 7 (though the entire console cannot be shown in a single figure).

**Figure 7. Console for ErrRequest.java**
5. Use WebSphere MQ Explorer to browse the messages in CERT.BACKOUT.
Section 6. Triggering

Overview

Triggering is a function in WebSphere MQ that allows applications to be started only when messages they are to process are delivered to the appropriate queue.

Several definitions must be created by the WebSphere MQ administrator in order for triggering to work properly. First, the queue must be defined as a triggered queue, with attributes that point to a process definition and an initiation queue.

Second, the process must be defined, with the APPLICID attribute containing the name of the program that is to be started when a message arrives at the triggered queue. An initiation queue must then be defined, as a simple local queue with no special attributes. When the triggered queue meets its conditions for triggering, the queue manager will create a specially formatted trigger message, with information from the triggered queue and process definitions, and will place that trigger message in the initiation queue.

Finally, the administrator must start a WebSphere MQ supplied application called a trigger monitor. This program is designed to start other applications based upon the contents of the trigger message that it retrieves from the initiation queue. Although trigger monitors are supplied with all MQ implementations, some customers write their own. A trigger monitor is merely an application that understands the layout of the trigger message and how to invoke applications in the environment in question.

Trigger types

The TRIGTYPE parameter of the triggered queue defines the circumstances in which the triggered program will be started.

- **TRIGTYPE=FIRST**
  Is the default. When the number of messages on the triggered queue goes from zero to one, the queue is triggered, the queue manager creates a trigger message and places it on the initiation queue, and the trigger monitor reads that message and starts the program.

- **TRIGTYPE=DEPTH**
  When the depth of the queue goes from one less than TRIGDPTH to equal to TRIGDPTH, the queue is triggered and the process continues as in
TRIGTYPE=FIRST.

TRIGTYPE=EVERY
A trigger message will be created for each of the messages that arrives on the triggered queue, and therefore a separate instance of the triggered program will be started for each of those trigger messages.

TRIGTYPE of NONE
Is the same as defining the queue as NOTRIGGER.

As an application designer and programmer, you should be aware of the TRIGTYPE and its effect on your program. A program that will be triggered by a queue that is TRIGTYPE=FIRST or DEPTH should probably be designed to continue to MQGET messages from the triggered queue until the queue is empty.

Trigger type depth

For a TRIGTYPE of DEPTH, when the trigger conditions are satisfied on the triggered queue, the queue manager creates a trigger message and disables triggering on that queue using the TriggerControl attribute. The triggered application must re-enable triggering itself by using the MQSET call.

To avoid a triggering yo-yo effect, MQGET should have a wait interval to see if more messages arrive before turning triggering on and exiting.

Trigger message

When a trigger monitor retrieves a trigger message (MQTM structure) from an initiation queue, the trigger monitor may need to pass some or all of the information in the trigger message to the application that is started by the trigger monitor. Information that may be needed by the started application includes queue name, trigger data, and user data. The trigger monitor application can pass the MQTM structure directly to the started application.

By passing the queue name to the triggered program, you can design a generic program capable of being triggered by different queues. The triggered program retrieves the name of the queue from the trigger message and uses that in the object name field in the object descriptor for the MQOPEN call. The trigger data and user data fields give you different ways of passing information from the queue to the triggered program.

Triggered application pseudo code
Listing 17 shows pseudo code for a triggered application. Housekeeping calls have been abbreviated to simplify the code and focus on the use of syncpoint.

**Listing 17. Triggered application example**

```plaintext
DEFINE OBJ_DESC AS MQOD
DEFINE TRIG AS MQTM
DEFINE CONN_HANDLE AS MQHCONN
DEFINE OBJ_HANDLE AS MQHOBJ
COPY INPUTPARAMETER TO TRIG
CALL MQCONN(QMGR_NAME, CONN_HANDLE, ...)
OBJ_DESC.ObjectName = TRIG.QName
CALL MQOPEN(CONN_HANDLE, OBJ_DESC, ..., OBJ_HANDLE, ...) 
CALL MQGET(CONN_HANDLE, OBJ_HANDLE, ...) 
```

The trigger monitor passes the MQTM structure to the application as a parameter. The structure can be used to get the queue name, so the application is generic. The queue name is set in the ObjectName field of the MQOD structure to the queue name passed in by the trigger message before the MQOPEN.

**Triggering and syncpoint**

If a message is physically placed on a queue, it has the potential to satisfy trigger conditions associated with the queue. For example, if a message is put in syncpoint and is the first message placed on the queue, if TRIGTYPE=FIRST has been specified and triggering is enabled for the queue, a trigger message will be built by the queue manager and placed on the appropriate initiation queue. However, the trigger message will also be put in syncpoint.

This can delay the start of an application to process the messages, which may be what is desired. But, consider a queue that has a mix of messages put in syncpoint and without syncpoint. The subsequent messages put without syncpoint will not be processed until the trigger message is made available to be received by the trigger monitor.

When any syncpoint occurs, whether a commit or a rollback, the message in the initiation queue will be committed and made available to the trigger monitor. This may result in an application starting and finding no messages available (if they were rolled back).
Section 7. MQI security

Message context

Message context allows an application that retrieves a message to find out about the originator of the message. The retrieving application may:

- Check the authority of the sending application.
- Perform some accounting functions (that is, charge back) using the initiator's information.
- Keep an audit trail of messages that have been processed.

The information about the originating user and application is contained in eight fields in the message descriptor. As a group, these fields are called the context fields. The information is further subdivided into two parts: the identity data and the origin data.

The setting and use of these fields should be carefully controlled.

Context

Typically, application programs will not explicitly update the context fields. The default behavior will be that the queue manager sets all of the fields. Usually an application will not be authorized to update the context fields.

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

In some cases, only the identity data portion of the context information is passed or updated. Again, an application and user must be authorized to do this.

Normally, business applications will not concern themselves with the context fields; the queue manager will update them as you shall see.

Default context

When an application issues an MQPUT, if no actions have been taken to update the context fields, the queue manager, by default, will update each of the fields as follows:
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserIdentifier (user identifier)</td>
<td>The user identifier under which the putting application is running</td>
</tr>
</tbody>
</table>
| AccountingToken (accounting token) | CICS - The unit of work identifier  
IMS - The PSB name + the IMS recover token  
OS/400 - A 1 byte length + the job accounting code  
UNIX - The user number |
| ApplIdentityData (application identity data) | Blank |
| PutApplType (put application type) | CICS, UNIX, 32-bit Windows, etc. |
| PutApplName (put application name) | The name of the application that put the message |
| PutDate (put date) | The date the message was put (YYYYMMDD) |
| PutTime (put time) | The time the message was put (HHMMSSSTH) |
| ApplOriginData (application origin data) | Blank |

It is important to note that UserIdentifier is 12 characters. In some environments, the user ID can be larger (for example, the Administrator user ID in Windows). Only the first 12 characters of a large user ID will be stored in the UserIdentifier field.

A Windows security identifier (SID) is stored in the AccountingToken field when a message is created under WebSphere MQ for Windows. The SID can be used to supplement the UserIdentifier field and to establish the credentials of a user.

The PutApplName field is 28 characters long. If the path and application name are longer, the last 28 characters are used.

The PutTime is in GMT.

**No context**

If the put message option of MQPMO_NO_CONTEXT is specified, any data in any of the context fields is cleared by the queue manager at MQPUT time. Since the queue manager does not have to determine values to be filled in, this may be slightly faster.

No special authority is required to use No Context.

The only time the PutApplType will be blank is if the message retrieved was put with a No Context specification.
Context handling

Putting programs can stipulate how the context fields should be filled in. The possible options are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQPMO_DEFAULT_CONTEXT</td>
<td>The queue manager is to supply both the identity context and the origin context data.</td>
</tr>
<tr>
<td>MQPMO_PASS_IDENTITY_CONTEXT</td>
<td>The queue manager is to supply the origin context, but the identity context is to be copied from the input message. For this to work correctly, the input queue must be opened with an option of MQOO_SAVE_ALL_CONTEXT and the output queue must be opened with the option MQOO_PASS_IDENTITY_CONTEXT.</td>
</tr>
<tr>
<td>MQPMO_PASS_ALL_CONTEXT</td>
<td>All eight of the context fields are to be passed from the input message to the output message. The input queue must be opened with MQOO_SAVE_ALL_CONTEXT and the output queue must be opened with the option MQOO_PASS_ALL_CONTEXT.</td>
</tr>
<tr>
<td>MQPMO_SET_IDENTITY_CONTEXT</td>
<td>The queue manager is to supply the origin context but the application is to set the identity context. The output queue must be opened using the option MQOO_SET_IDENTITY_CONTEXT.</td>
</tr>
<tr>
<td>MQPMO_SET_ALL_CONTEXT</td>
<td>The program will control the content of all context fields. The output queue must be opened with the option MQOO_SET_ALL_CONTEXT.</td>
</tr>
</tbody>
</table>

Except for the first, all of these options require that the user and application attempting them have the proper authority.

Programs that set context

The MQPMO options to set context should only be for privileged programs. If an application is permitted to use the set capabilities, it might be able to use a more highly authorized user ID and application name. Then, as the message is passed along, any ability to trace back to the actual user and application will be lost.

As noted earlier, the setting and use of context fields should be carefully controlled.

Pass context pseudo code

Listing 18 shows pseudo code for passing context from an input message to an output message.
Listing 18. Passing context example

```
DEFINE OPEN_OPTIONS AS MQLONG
DEFINE MY_IN_HANDLE AS MQHOBJ
DEFINE MY_OUT_HANDLE AS MQHOBJ
DEFINE MY_INPUT_QUEUE AS MQOD
DEFINE MY_OUTPUT_QUEUE AS MQOD
DEFINE MESSAGE_DESCRIPTOR AS MQMD
DEFINE PUT_OPTIONS AS MQPMO

CALL MQCONN(QMGR,
            CONN_HANDLE,
            COMP_CODE,
            REASON_CODE)

OPEN_OPTIONS = MQOO_SAVE_ALL_CONTEXT +
               MQOO_INPUT_AS_Q_DEF

MY_INPUT_QUEUE.ObjectName = 'INP1'

CALL MQOPEN(CONN_HANDLE,
            MY_INPUT_QUEUE,
            OPEN_OPTIONS,
            MY_IN_HANDLE,
            COMP_CODE,
            REASON_CODE)

OPEN_OPTIONS = MQOO_PASS_ALL_CONTEXT +
               MQOO_OUTPUT

MY_OUTPUT_QUEUE.ObjectName = 'OUT1'

CALL MQOPEN(CONN_HANDLE,
            MY_OUTPUT_QUEUE,
            OPEN_OPTIONS,
            MY_OUT_HANDLE,
            COMP_CODE,
            REASON_CODE)

MQGET(....)

PUT_OPTIONS.Options = MQPMO_PASS_ALL_CONTEXT
PUT_OPTIONS.Context = MY_IN_HANDLE

CALL MQPUT(....)
```

The input queue is opened with MQOO_SAVE_ALL_CONTEXT and the output queue is opened with MQOO_PASS_ALL_CONTEXT. Then, before the message is put, the put option is set to MQPMO_PASS_ALL_CONTEXT, and the Context field is set to the object descriptor for the input queue. This results in all of the context information from the message retrieved from the input queue to be passed to the context fields of the output message descriptor.

Alternate user authority

Another security option in the MQI is alternate user authority. Alternate user authority enables a program to get a message from an input queue and move the UserIdentifier from the context of the input message to a field in the object descriptor.
called MQOD_AlternateUserId. To use this field with an MQOPEN, the option MQOO_ALTERNATE_USER_AUTHORITY is set. Or, if an MQPUT1 is to be used, the MQPMO_ALTERNATE_USER_AUTHORITY option is used.

When the MQOPEN or MQPUT1 is executed, the authority of the issuing program to request alternate user authority is checked and the authority of the user ID specified in the MQOD. AlternateUserId is checked to see if it is authorized to open the queue for output.

Section 8. Message groups and segmentation

Introduction

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

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

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

A message group is made up of one or more logical messages, consisting of one or more physical messages that have some relationship. The possible relationships are explored as you proceed.

By using the capabilities associated with message groups, a program can make sure that it retrieves the messages in the proper order without having to use some combination of MsgId and CorrelId. Also, it provides an alternative means to allow grouping of related messages, again without using MsgId and CorrelId.

As for segments, they are simply a means to handle messages that are too large for a program or queue manager (including an intermediate queue manager through which a message passes).

Message groups
There are two fields in the message descriptor structure that are used when working with message groups:

**GroupId**
- A 24-byte field that can contain a unique value to identify a message group. Each logical message that is a member of a particular group will have the same GroupId.

**MsgSeqNumber**
- The sequence of the message within the group; although the GroupId will be the same, each logical message will have a different MsgSeqNumber (the numbering starts at 1).

If a message is not part of a group, the GroupId will be all nulls while the MsgSeqNumber will be set to 1.

### Grouping logical messages

The recommended approach to creating a message group is to allow the queue manager to create a unique GroupId and use it for each of the messages within a particular group. This is accomplished with the put message option MQPMO_LOGICAL_ORDER and a field in the message descriptor called MsgFlags.

By specifying MQPMO_LOGICAL_ORDER and setting MsgFlags to either MQMF_MSG_IN_GROUP or MQMG_LAST_MSG_IN_GROUP, the queue manager will generate a new GroupId for the starting message of each new group. It will then keep that GroupId and assign a new MsgSeqNumber for each new logical message within the group.

The queue manager will know to create a new GroupId if the last message that it placed contained the MsgFlag set to MQMF_LAST_MSG_IN_GROUP. Since messaging groups imply more than one message will be placed on an output queue, this automatic handling by the queue manager is only supported for MQPUT, not MQPUT1.

If the application wishes to use message groups with MQPUT1 or simply wishes to control the setting up of the message group, it would not use the MQPMO_LOGICAL_ORDER option. If the application sets the GroupId to MQGI_NONE, a unique GroupId will be generated. It would then be the application's responsibility to ensure that unique MsgSeqNumbers are assigned for each logical message in the group, and to ensure that the GroupId continues to be the same until a new GroupId is desired.

When MQPMO_LOGICAL_ORDER is specified, the queue manager requires that messages in a group must be either all persistent or all non-persistent. If not, the call will fail with MQRC_INCONSISTENT_PERSISTENCE. If MQPMO_LOGICAL_ORDER is not specified, it will be the responsibility of the
application to determine that persistence be kept consistent; no check will be done by the queue manager. This means that some messages in a group could be lost if a queue manager stops for any reason and the messages within a group are a mixture of persistent and non-persistent.

**Retrieving logical messages**

There are two get message options that can be used to allow the proper retrieval of logical messages within a group. By specifying MQPMO_ALL_MSGS_AVAILABLE, the program can prevent retrieval of messages belonging to an incomplete group. By specifying MQGMO_LOGICAL_ORDER, messages in groups are returned in order. This order may be different from the physical order of messages on a queue.

As with MQPUT, the MQGET of messages in groups can be controlled by the application. However, this is not the recommended approach except to restart a group after a system or queue manager failure. The queue manager will then retain the group and segment information; subsequent calls, using the same queue handle, can revert to using MQGMO_LOGICAL_ORDER.

Likewise, messages that are simply forwarding physical messages should not use the MQGMO_LOGICAL_ORDER option or the GroupId of the original message can be corrupted.

WebSphere MQ for z/OS supports message grouping for shared and non-shared queues. This function is enabled by the use of INDXTYPE of GROUPID for local queues. It uses message priority internally to provide an efficient method of finding candidate groups and checking them for completeness.

The MatchOptions field of the get message options structure allows a program to control which group it retrieves. If an application wishes to retrieve a particular group, it can use the MQGMO_LOGICAL_ORDER option (as long as no current logical message is being processed) in conjunction with the MatchOption MQMO_MATCH_GROUP_ID. It is also possible to retrieve a message with a specific GroupId by not specifying MQGMO_LOGICAL_ORDER and specifying the MatchOption MQMO_MATCH_GROUP_ID.

The MQMO_MATCH_MSG_SEQ_NUM allows the retrieval of a specific message with a matching MsgSeqNumber. As with the other MatchOptions, this is in addition to any other matches that may apply. The MQMO_MATCH_MSG_SEQ_NUM is not valid if combined with MQGMO_LOGICAL_ORDER.

Finally, an application can determine if it has processed the final message in a group by checking the GroupStatus field in the get message options structure after a message is retrieved. If the message is the last message in a group, this field will contain MQGS_LAST_MSG_IN_GROUP. If not, the field will contain MQGS_MSG_IN_GROUP.
### Spanning units of work

The MQPMO_LOGICAL_ORDER option affects units of work as follows:

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

- Conversely, if the first message of a group has specified MQPMO_NO_SYNCPOINT, then none of the subsequent messages in the group can do otherwise.

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

Because it is possible to split a group over multiple units of work, it is possible to have some messages committed and some uncommitted when a failure occurs. It is the responsibility of the application to keep track of the status information associated with a group that spans units of work. By keeping track of the GroupId and MsgSeqNumber on a special status queue (using syncpoint options for the MQPUT and MQGET), an accurate picture of what has been completed can be kept.

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

### Message segmentation

Message segments were defined at a high level previously, and now we'll explore them in more detail. This discussion assumes that all MQPUT and MQGET work will operate within a unit of work. This is strongly recommended as a standard practice in order to avoid any possibility of incomplete segmented messages being present in a network.

Segmented messages use a field in the message descriptor called Offset to depict where a particular segment’s data would begin in a complete logical message.

Putting and getting segmented messages can be done under control of the queue manager or the application.

**Segmentation by the queue manager**

If the MsgFlag field in the message descriptor includes the value...
MQMF_SEGMENTATION_ALLOWED, the queue manager will handle building a segmented message. If the queue manager recognizes that the message will be too big for its MaxMsgLength or for a queue's MaxMsgLength, segmentation will be performed.

If the message descriptor included the MQRO_..._WITH_DATA option, it will be modified to eliminate the request for data on any segments that do not include the first 100 bytes.

Since the queue manager handles the splitting into segments, no assumptions can be made about how the data is split.

Persistent message segmentation can only be performed within a unit of work. If the queue manager determines that there is no application-defined unit of work active, the queue manager will create a unit of work that will only be active for the duration of the call.

**Reassembly by the queue manager**

To retrieve a complete segmented message, an application sets the option MQGMO_COMPLETE_MESSAGE. If the queue manager determines that the message is segmented, it will retrieve all the segments and return the complete message in the program's buffer.

Data conversion is done after the message is placed in the program's buffer, so this is no different with segmented messages. However, data conversion requested by a sender channel will fail for a message segment because the exit will not have all of the data from the message at one time.

If the queue manager determines that the message is a persistent message and there is no user-defined unit of work, the queue manager will create a unit of work for the duration of the call. It will not automatically create a unit of work for non-persistent messages.

**Segmentation by the application**

Application segmentation is used for two reasons:

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

The application should use the put message option MQPMO_LOGICAL_ORDER and set the MsgFlags field in the message descriptor to MQMF_SEGMENT or
MQMG_SEGMENT_LAST. The queue manager will assign and maintain the GroupId, MsgSeqNumber, and Offset fields.

If the application does not specify the MQPMO_LOGICAL_ORDER option, then the program is responsible for ensuring a new GroupId is assigned, as discussed in the coverage of message groups, as well as assigning proper MsgSeqNumbers and Offsets.

Putting and getting messages that span units of work is permitted, as discussed with message groups.

**Reassembly by the application**

It is possible to specify the MQGMO_COMPLETE_MESSAGE_OPTION and the MQMF_SEGMENTATION_ALLOWED MsgFlag to enable queue manager retrieval of a message that was segmented by an application.

If the application chooses to retrieve the segments individually, most likely because of a buffer size constraint, there is the capability to reassemble the message under application control.

If the program specifies MQGMO_ALL_MSGS_AVAILABLE or MQGMO_ALL_SEGMENTS_AVAILABLE as one of the get message options, processing will not occur if all message segments are not available. It makes sense to include the MQGMO_WAIT (with a wait interval) to allow time for all segments to arrive.

Once the first message is retrieved, use the MQGMO_LOGICAL_ORDER option to ensure all remaining segments for the message are processed.

Not specifying MQGMO_LOGICAL_ORDER can be used for recovery purposes, as was previously discussed with message groups. The only additional requirement is the need to keep track of the Offset of the segment as well as the GroupId and MsgSeqNumber.

Any intermediate applications that are simply passing the data should not use the MQGMO_LOGICAL_ORDER option to ensure the Offset field is not corrupted.

The MatchOptions field is also used to retrieve message segments. In addition to the MQMO_MATCH_GROUP_ID and MQMO_MATCH_MSG_SEQ_NUM options, MatchOptions can also be set to MQMO_MATCH_OFFSET.

An application can determine if it has processed the final segment of a message by checking the SegmentStatus of the get message options structure after retrieving a message segment. If the message is the last message in a segmented message, this field will contain MQSS_LAST_SEGMENT. If not, the field will contain MQSS SEGMENT.
Section 9. Distribution lists

Overview

Distribution lists allow you to put a message to multiple destinations in a single MQPUT or MQPUT1 call. Multiple queues can be opened using a single MQOPEN, and a message can then be put to each of those queues using a single MQPUT. Some generic information from the MQI structures used for this process can be superseded by specific information relating to the individual destinations in the distribution list.

Opening a distribution list

The object record (MQOR) structure is used to provide the ObjectName and ObjecQMgrName of each destination in the distribution list. A field in the object descriptor (MQOD), RecsPresent, indicates that a distribution list is being used. If RecsPresent is greater than zero, then it contains the number of object records in the distribution list.

There are two fields in the object descriptor that point to the object records: ObjectRecOffset and ObjectRecPtr. These fields are mutually exclusive.

ObjectRecOffset is the offset of the first MQOR from the start of the object descriptor. This approach lends itself to programming languages that do not support the pointer data type, or implement the pointer data type in a way that is not portable between environments (COBOL, for example).

ObjectRecPtr is a pointer to the start of the array of MQORs. This method is recommended for programming languages that support the pointer data type in a portable manner (C, for example).

ObjectName in each MQOR is the name of a queue and must not be a model queue. ObjectQMgrName is the name of the queue manager on which that queue resides. Note that the MQODs are built by the application; a distribution list is not a WebSphere MQ object to be defined by the administrator.

When the MQORs are built for all the queues in the distribution list, and the object descriptor is set up with RecsPresent and either ObjectRecOffset or ObjectRecPtr, the MQOPEN is issued.
Each destination queue is opened separately, meaning some could succeed and others could fail if they are not valid. If all queues open successfully, the completion code returned from the MQOPEN will be MQCC_OK. If all the queues failed to open, the completion code will be MQCC_FAILED. However, if some queues are successfully opened and some fail to open, the completion code will be MQCC_WARNING. For both failures and warnings, the reason code will be MQRC_MULTIPLE_REASONS.

Three *_DEST_COUNT fields in the object descriptor will be updated after the MQOPEN:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KnownDestCount</td>
<td>The number of local queues that were opened successfully.</td>
</tr>
<tr>
<td>UnknownDestCount</td>
<td>The number of remote queues that were opened successfully.</td>
</tr>
<tr>
<td>InvalidDestCount</td>
<td>The number of queues that failed to open successfully.</td>
</tr>
</tbody>
</table>

The completion code and reason code for each queue that was opened are returned in a structure called the response record (MQRR). As before with the object record, the RecsPresent field of the object descriptor gives the number of response records. And, as before there are two ways to address the MQRR array: ResponseRecOffset to define the offset, or ResponseRecPtr to point to the address. The MQRR records and the offset or pointer must be set up before the call to MQOPEN.

Adding a message to a distribution list

There are several fields in the put message options (MQPMO) structure related to distribution lists.

PutmsgRecFields includes the flags indicating which fields from a structure, the put message record (MQPMR), are present. One or more of the flags can be specified:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQPMRF_MSG_ID</td>
<td>Message identifier</td>
</tr>
<tr>
<td>MQPMRF_CORREL_ID</td>
<td>Correlation identifier</td>
</tr>
<tr>
<td>MQPMRF_GROUP_ID</td>
<td>Group identifier</td>
</tr>
<tr>
<td>MQPMRF_FEEDBACK</td>
<td>Feedback</td>
</tr>
<tr>
<td>MQPMRF_ACCOUNTING_TOKEN</td>
<td>Accounting token</td>
</tr>
<tr>
<td>MQPMRF_NONE</td>
<td>No put message records are present</td>
</tr>
</tbody>
</table>

The RecsPresent field provides a count of the number of put message records. PutMsgRecOffset and PutMsgRecPtr provide the offset of the first put message.
record or the address of the first put message record, respectively, and are mutually exclusive. If no put message records are present, both fields should contain zero.

The put message record (MQPMR) structure can contain up to five fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MsgId</strong></td>
<td>The message identifier to be used for the message sent to the queue whose name was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. For those MQPMRs that do not contain a MsgId, the value from the message descriptor MsgId field is used. If that value is MQMI_NONE, a unique MsgId is generated for each of those destinations (no two destinations will have the same MsgId). If MQPMO_NEW_MSG_ID is specified, a new MsgId will be generated for each of the destinations on the distribution list, regardless of whether they have MQPMR records.</td>
</tr>
<tr>
<td><strong>CorrelId</strong></td>
<td>The correlation identifier to be used for the message sent to the queue whose name was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. The CorrelId field of the message descriptor is used if this field is not present or there are fewer MQPMRs than destinations. If MQPMO_NEW_CORREL_ID is specified, a single new correlation identifier is generated and used for all of the messages in the distribution list.</td>
</tr>
<tr>
<td><strong>GroupId</strong></td>
<td>The group identifier to be used for the message sent to the queue whose name was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. The GroupId of the message descriptor will be used in the same manner as for the conditions described in MsgId and CorrelId.</td>
</tr>
<tr>
<td><strong>Feedback</strong></td>
<td>Used for the message sent to the queue whose name was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. If not present, the Feedback of the message descriptor will be used.</td>
</tr>
<tr>
<td><strong>AccountingToken</strong></td>
<td>Used for the message sent to the queue whose name was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. If not present, the AccountingToken of the message descriptor will be used.</td>
</tr>
</tbody>
</table>
As described in the MQOPEN for distribution lists, the same applies for the MQPUT. MQCC_OK will only be returned if all the puts succeeded, MQCC_FAILED will only be returned if all the puts failed, and MQCC_WARNING will be returned if some succeeded and some failed. If MQCC_WARNING or MQCC_FAILED is returned, the reason will be MQRC_MULTIPLE_REASONS.

The response record for each queue will contain the completion code and reason for the put to that queue. Since a separate MQPUT is issued to each queue by the queue manager, it is possible to keep track of each one separately.

The KnownDestCount, UnknownDestCount, and InvalidDestCount fields of the MQPMO structure provide counts of messages sent successfully to local queues, messages sent successfully to remote queues, and messages that could not be sent successfully, respectively.

The completion code and reason code for each put that was attempted are returned in a structure called the response record (MQRR). The RecsPresent field of the MQPMO structure gives the number of response records. And, as before there are two ways to address the MQRR array: ResponseRecOffset to define the offset, or ResponseRecPtr to point to the address. The MQRR records and the offset or pointer must be set up before the call to MQOPEN.

Closing a distribution list

When closing a distribution list, any close options other than MQCO_NONE will result in MQRC_OPTION_NOT_VALID_FOR_TYPE or MQRC_OPTIONS_ERROR.

Only the completion code and reason code for the MQCLOSE are available for diagnostic purposes. No individual completion codes and reason codes are returned, and any queue that encounters an error is not identified. Even though a failure may occur, the queue manager will continue processing and will attempt to close the remaining queues in the distribution list.

Section 10. Distribution list sample code

In this section we'll review and run some sample code to demonstrate use of a distribution list.

Review the sample code
First let’s look at the configuration file you will run to set up queues for the distribution list, shown in Listing 19.

**Listing 19. setup05.txt configuration file**

```
DEFINE QLOCAL('CERT.LIST1') REPLACE
DEFINE QLOCAL('CERT.LIST2') REPLACE
DEFINE QLOCAL('CERT.LIST3') REPLACE
DEFINE QLOCAL('CERT.LIST4') REPLACE +
  PUT(DISABLED)
```

This configuration field merely creates four queues, CERT.LIST1-4, but disables putting messages on CERT.LIST4.

DistLists.java creates a distribution list, opens it, and puts a single message to it. It then reports on the completion and reason codes returned for each queue. As always, the action takes place in the `run()` method, so take a look at the first several lines of that method, shown below.

**Listing 20. Start of run() method**

```
public void run() {
    System.out.println("Start DistList");
    try {
        MQQueueManager qMgr = new MQQueueManager(qManager);
        MQDistributionListItem item1 = new MQDistributionListItem();
        item1.queueManagerName = qManager;
        item1.queueName = item1Name;
        MQDistributionListItem item2 = new MQDistributionListItem();
        item2.queueManagerName = qManager;
        item2.queueName = item2Name;
        MQDistributionListItem item3 = new MQDistributionListItem();
        item3.queueManagerName = qManager;
        item3.queueName = item3Name;
        MQDistributionListItem item4 = new MQDistributionListItem();
        item4.queueManagerName = qManager;
        item4.queueName = item4Name;
        MQDistributionListItem[] items = { item1, item2, item3, item4 };
        int openOptions = MQC.MQOO_OUTPUT + MQC.MQOO_FAIL_IF_QUIESCING;
        MQDistributionList distList = new MQDistributionList(qMgr, items,
                                                          openOptions, null);
    }
```

Instead of MQOR and MQRR structures, the WebSphere MQ base Java classes provide `MQDistributionListItem` and `MQDistributionList` classes that
provide similar functions to those structures. MQDistributionListItems are created and initialized, then placed into an array that is a parameter to the constructor for MQDistributionList. Constructing an MQDistributionList is the same as opening a distribution list.

So, the code here shows connecting to the queue manager, creating four distribution list items, then placing them in an array. The open options are initialized and the distribution list is created.

Listing 21 shows creating a message and putting it to the distribution list.

**Listing 21. Putting the message to the distribution list**

```java
String message = "A message sent to a distribution list";
MQPutMessageOptions putOptions = new MQPutMessageOptions();
putOptions.options = MQC.MQPMO_NO_SYNCPOINT
    + MQC.MQPMO_FAIL_IF_QUIESCING;
MQMessage msg = new MQMessage();
msg.messageType = MQC.MQMT_DATAGRAM;
msg.writeUTF(message);
msg.format = MQC.MQFMT_STRING;
msg.messageId = MQC.MQMI_NONE;
msg.correlationId = MQC.MQCI_NONE;
try {
    distList.put(msg, putOptions);
} catch (MQException mqex) {
    if (mqex.completionCode == MQException.MQCC_WARNING) {
        MQDistributionListItem item = distList.getFirstDistributionListItem();
        while (item != null) {
            System.out.println("Error for queue: " + item.queueName);
            System.out.println("Completion code: " + item.completionCode);
            System.out.println("Reason code: " + item.reasonCode);
            item = item.getNextDistributedItem();
        }
    } else {
        System.out.println("A WebSphere MQ Error occured : Completion Code "
```

This code should look familiar. The only difference between it and the puts in other sample programs is the use of the `put()` method on the distribution list object. Also notice that the put is in its own `try` block, so you can catch just the errors from the put.

Listing 22 shows the error handling code for the put.

**Listing 22. Error handling for the put**

```java
    } catch (MQException mqex) {
        if (mqex.completionCode == MQException.MQCC_WARNING) {
            MQDistributionListItem item = distList.getFirstDistributionListItem();
            while (item != null) {
                System.out.println("Error for queue: " + item.queueName);
                System.out.println("Completion code: " + item.completionCode);
                System.out.println("Reason code: " + item.reasonCode);
                item = item.getNextDistributedItem();
            }
        } else {
            System.out.println("A WebSphere MQ Error occured : Completion Code ")
```
```
The error handling code first checks for MQCC_WARNING. This tells it that some puts worked and some didn't. It then steps through the distribution list items, retrieving and displaying the queue names, completion codes, and reason codes. In place of Offset or Ptr variables, the WebSphere MQ base Java classes provide two methods on the distribution list to facilitate traversing the list: `getFirstDistributionListItem()` and `getNextDistributionListItem()`.

Listing 23 below shows the end of the program: the distribution list is closed, the program disconnects from the queue manager, and a message indicating the end of the program is printed to the console. Listing 23 also shows the `catch` from the outermost `try` block.

Listing 23. End of program and exception handling

```java
    distList.close();
    qMgr.disconnect();
    System.out.println("End DistList");
} catch (MQException mqex) {
    System.out
        .println("A WebSphere MQ Error occured : Completion Code "
        + mqex.completionCode + " Reason Code "
        + mqex.reasonCode);
    mqex.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Run the sample code

Now you can set up the distribution list queues and run the sample code.

1. Open a Windows command prompt.
2. Type `runmqsc < setup05.txt`
3. Close the Windows command prompt.
4. In Application Developer, right-click **DistList.java** in the Package Explorer view, and select **Run > Java Application**.

5. When the program is finished running, the console should look like Figure 8.

**Figure 8. Console for DistList.java**

```
Start DistList
Error for queue: CERT.LIST1
 Completion code: 0
  Reason code: 0
Error for queue: CERT.LIST2
 Completion code: 0
  Reason code: 0
Error for queue: CERT.LIST3
 Completion code: 0
  Reason code: 0
Error for queue: CERT.LIST4
 Completion code: 2
  Reason code: 2051
End DistList
```

6. A completion code of 0 is MQCC_OK and a reason code of 0 is MQRC_NONE. Those values were returned for the first three queues in the list. A completion code of 2 is MQCC_FAILED and a reason code of 2051 is MQCC_PUT_INHIBITED. Which is exactly what you got for CERT.LIST4, and exactly what you expected, since the queue was created with put disabled.

7. Use WebSphere Explorer to browse the messages in CERT.LIST1, CERT.LIST2, CERT.LIST3, and CERT.LIST4 (which should have no messages).

---

**Section 11. Conclusion**
This tutorial finishes the discussion of the MQI begun in the previous tutorial, covering MQINQ, MQSET, transactions, triggering, security, message groups and segments, and distribution lists. 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 obtained 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>13KB</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 version 6.0.

• 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. Willy is an IBM Certified WebSphere MQ V6.0 Solution Designer.