

WebSphere Studio Application Developer 5.0: Practical J2EE Development

IGOR LIVSHIN



WebSphere Studio Application Developer 5.0: Practical J2EE Development
Copyright ©2003 by Igor Livshin

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-120-8

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Technical Reviewer: Jeff Miller

Editorial Directors: Dan Appleman, Gary Cornell, Martin Streicher, Jim Sumser,

Karen Watterson, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Sofia Marchant

Development Editor: Tracy Brown

Production Editor: Janet Vail

Proofreader: Lori Bring

Compositor: Diana Van Winkle, Van Winkle Design Group

Indexer: Ann Rogers

Artist and Cover Designer: Kurt Krames

Production Manager: Kari Brooks

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit <http://www.springer-ny.com>. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit <http://www.springer.de>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section.

J2EE Enterprise Messaging

THE ENTERPRISE JAVA BEAN (EJB) 2.0 and Java 2 Enterprise Edition (J2EE) 1.3 specifications now support Java Message Service (JMS). By including JMS in the J2EE specification, Sun Microsystems added extremely important functionality to the J2EE 1.3 environment—*asynchronous communication*. Before this addition, J2EE was a strictly *synchronous* environment of J2EE components communicating over the RMI-IIOP protocol.



NOTE *RMI-IIOP is a Common Object Request Broker Architecture (CORBA)–compliant version of the Java Remote Method Invocation (RMI) communication protocol that sends RMI messages via the CORBA platform and language-independent communication protocol.*

To support asynchronous communication, J2EE 1.3 also introduced a new type of EJB bean: the Message Driven Bean (MDB). An MDB is capable of receiving asynchronous messages in an otherwise synchronous J2EE environment. As already mentioned, before JMS, J2EE was a synchronous environment based on the Java RMI-IIOP communication protocol.

Asynchronous communication gives enterprise developers an alternative form of communication with the following important advantages:

- Asynchronously communicating programs do not exchange messages directly with each other. Subsequently, a user on each side of the communication session can continue working (sending messages) even if a program on the opposite (receiving side) is down.
- Asynchronous communication offers reliability. Middleware packages that support asynchronous communication provide guaranteed (assured) message delivery even if the entire environment is down. This is an attractive feature for applications involved in communication over the Internet (which is not a reliable medium).

- Programs sending messages asynchronously are not locked when waiting for a response (which is a substantial performance improvement).

JMS is not communication software but rather a set of standard Application Programming Interfaces (APIs) for vendor-independent asynchronous communication. In that respect, JMS is similar to Java Database Connectivity (JDBC) and Java Naming and Directory Interface (JNDI). As is the case with JDBC, which requires an underlying database provider, JMS requires an underlying asynchronous communication middleware provider that supports the JMS standard. This is typically called Message-Oriented Middleware (MOM).

MOM is a technology that allows programs to asynchronously communicate by exchanging messages. To some extent, this process is similar to people communicating via email. Making the same analogy, synchronously communicating programs are similar to people communicating over the phone. Programs involved in asynchronous communication are loosely coupled. In other words, it means they do not communicate directly but via virtual channels called *queues* or *topics*.

It also means they maintain a staged *store-and-forward* way of communication, which allows a program on the sending side to send messages even when a program on the opposite side of the communication is not running at that moment. When a program on the receiving side is up and running, the messages will be delivered to it. To some extent, this oversimplifies the real communication process that is subject to certain conditions, but it gives you a general idea of how this type of communication happens.

The main advantage of JMS communication is an environment where programs communicate using standard APIs that shield programmers from the complexities of different operating systems, data representation, and underlying network protocols.

This chapter discusses JMS and two JMS communication methods: Point-to-Point (P2P) and Publish/Subscribe (Pub/Sub). It also covers the structure of a JMS message, the main JMS objects, MDBs, JMS programming, message persistence, and JMS transaction support.

The chapter begins by discussing JMS.

Understanding JMS

Because JMS is a relatively new technology, this chapter discusses JMS programming in detail, followed by a discussion of WSAD 5.0 JMS development. As mentioned, JMS programs do not communicate directly. Messages are sent to destination objects: queues or topics. Both queues and topics are staging media objects capable of accumulating messages; however, queues and topics support different types of message delivery corresponding to two *domains* of communication: P2P and Pub/Sub.

Understanding the P2P Domain of Communication

The P2P domain of communication can operate as a “pull” or “push” type of message delivery. In the P2P pull type of communication, programs communicate using a virtual channel called a *queue*. On the sending side of the communication session, a sender program “puts” a message on a queue. On the receiving side, a receiver program periodically searches this queue looking for a message it expects to receive and process. The pull type is a less efficient message delivery method than the push type because it consumes resources during this repetitive checking for the arrival of the message. It is also important to understand that when the receiver program finds the message, it “gets” it, effectively removing it from the queue.

Therefore, even if multiple receiver programs process the same queue, only one receiver is capable of receiving a particular message. JMS programs can use multiple queues, and each queue can be processed by multiple programs, but only one program receives any specific message.

When the P2P domain of communication operates as the push type of message delivery, a sender program works in the same way, sending a message on a queue. However, the receiver program works differently. The receiver program implements a Listener Interface and includes the implementation of its `onMessage` callback method. In the J2EE environment, the task of listening on a specific queue for the arrival of messages is delegated to the container. Whenever a new message arrives on the queue, the container calls the `onMessage` method, passing the message as a parameter.

The most important point of a P2P communication domain (both types of message delivery) is that each message is received by a single program. Typically, P2P programs are more actively involved in the communication. A sender program can indicate to a receiver program the name of the queue to which it expects the reply messages to be sent. It also can request confirmation or report messages.

Understanding the Pub/Sub Domain of Communication

In the Pub/Sub domain of communication, programs communicate via a topic. Topics as a medium of communication require the support of a Pub/Sub *broker*. On the sending side, a producer program sends messages to a topic. On the receiving side, consumer programs subscribe to the topics of interest. When a message arrives at a topic, all consumer programs that are subscribed to the topic receive the message as a parameter of the `onMessage` method.

This is a push message delivery method. As you can see, multiple programs can receive a copy of the same message. Pub/Sub programs are less actively involved in communication. A producer program that sends messages to a particular topic does not know how many subscribers are receiving published messages (many or even none). Subscribing to a topic is a flexible scheme of communication. The number of subscribers to a topic changes dynamically without any change to the underlying communication infrastructure and is completely transparent to the overall communication process.

The Pub/Sub type of communication requires support from the Pub/Sub broker—a software package that coordinates messages to be delivered to subscribers. In contrast to the P2P domain where programs use a queue as a staging area for communication, programs involved in the Pub/Sub domain communicate directly with the special broker queues. This is why you later install the MAOC package on top of the regular WebSphere MQ installation. (This is necessary only if you are using WebSphere MQ as a JMS provider. MQ 5.3.1 or higher is required.) This package is a broker software package that supports the Pub/Sub domain of communication. (For more information, see the “Understanding JMS Pub/Sub Programming” section.)

The `QueueConnectionFactory` and `TopicConnectionFactory` JMS objects are factory classes that create the corresponding `QueueConnection` and `TopicConnection` objects used by JMS programs to connect to the underlying MOM technology.

Communicating with JMS Messages

JMS-based programs communicate by exchanging JMS messages. The JMS message consists of three parts: a header, the properties (optional), and a message body. The header consists of header fields that contain the delivery information and meta-data.

The properties part contains standard and application-specific fields that message selector can use to filter retrieved messages. JMS defines a standard and optional set of properties that is optional for MOM providers to support (see Table 9-1). The body part contains the application-specific data (the content to be delivered to a target destination).

Table 9-1. JMS Standard Message Properties

PROPERTY	TYPE	DESCRIPTION
<code>JMSXProducerTXID</code>	String	Transaction within which this message was produced
<code>JMSXConsumerTXID</code>	String	Transaction within which this message was consumed

Optional properties include JMSXUserID, JMSXAppID, JMSXProducerTXID, ConsumerTXID, JMSXRcvTimestamp, JMSXDeliveryCount, and JMSXState. The message headers provide information for the JMS messaging middleware that describes such system information as the intended message destination, the creator of the message, how long the message should be kept, and so on (see Table 9-2).

Table 9-2. JMS Header Fields

HEADER	FIELD	TYPE DESCRIPTION
JMSMessageID	String	This uniquely identifies a message and is set by the provider. This is undetermined until after the message is successfully sent.
JMSDeliveryMode	int	DeliveryMode.PERSISTENT or NON_PERSISTENT. This is a tradeoff between reliability and performance.
JMSDestination	Destination	This contains where the message is sent and is set by a message provider. The destination can be a queue or a topic.
JMSTimestamp	long	This is set by the provider during the send process.
JMSExpiration	long	This is the time the message should expire. This value is calculated during the send process. It can take a value of 0, meaning no expiration.
JMSPriority	int	This is the priority of the message. A priority of 0 is the lowest priority, and 9 is the highest priority.
JMSCorrelationID	String	This links a response message with a request message. The responding program typically copies JMSMessageID to this field.
JMSReplyTo	Destination	This is used by a requesting program to indicate a queue where a reply message should be returned.
JMSType	String	This indicates the type of message. The available types are as follows:* MapMessage contains a set of name-value pairs, where the name is a string and the value is a primitive Java type. TextMessage contains a serialized Java object. BytesMessage contains a byte stream in the message body.
JMSRedelivered	boolean	This indicates that the message was delivered, but the program did not acknowledge its receipt.

* Of all these types, TextMessage is the most widely used because of its simplicity and because of its ability to encapsulate Extensible Markup Language (XML) data.

JMS defines several types of body parts depending how this part is coded. You indicate the type of the body in the `JMSType` header field with the following possible values:

TextMessage: This contains the `java.lang.String` object. This is the simplest message format. You can set XML documents as `TextMessage`.

ObjectMessage: This contains a serializable Java object, which is built based on the serializable Java class.

MapMessage: This contains a set of name-value pairs of elements. It is typically used to transfer keyed data. To set the element of the message, you use setter methods such as `setInt`, `setFloat`, `setString`, and so on. On the receiving side, you use the corresponding getter methods such as `getInt`, `getFloat`, `getString`, and so on.

BytesMessage: This contains an array of primitive bytes. It is typically used when there is a need to send the message in the application's native format.

StreamMessage: This contains a stream of primitive Java types such as `int`, `char`, `double`, and so on. Primitive types are read from the message in the same order they are written. Similar getter and setter methods are provided to manipulate the message elements: `writeInt` and `readInt`, `writeString` and `readString`, and so on.

You create the JMS message object by using a JMS session object (discussed in the “Using the JMS QueueConnection Object” section). The following are examples of creating different message types:

```
TextMessage textMsg = session.createTextMessage();
MapMessage mapMsg = session.createMapMessage();
ObjectMessage objectMsg = session.createObjectMessage();
BytesMessage byteMsg = session.createBytesMessage();
```

The message object provides setter and getter methods for all message header fields. The following are several examples of getting and setting values of the JMS message header fields:

```
String messageId = testMsg.getJMSMessageID();
testMsg.setJMSCorrelationID(messageID);

int messagePriority = mapMsg.getJMSPriority();
mapMsg.setJMSPriority(1);
```

The message object also provides similar setter and getter methods for standard and application-specific property fields. The following are several examples of getting and setting values of the JMS message standard and application-specific property fields:

```
int groupSeq = objectMsg.getIntProperty("JMSGGroupSeq");
objectMsg.setStringProperty("FirstName", "Joe");
```

JMS provides a set of APIs for setting and getting the content of the message's body part. Listing 9-1 shows several examples of how to work with different types of message bodies.

Listing 9-1. Working with Different Types of Message Bodies

Text Message

```
TextMessage textMsg = session.createTextMessage();
textMsg.setText("This is the text type message");
```

Map Message

```
MapMessage mapMsg = session.createMapMessage();
mapMsg.setInt(BookCatalogNumber, 100);
mapMsg.setString(BookTitle, "WinSocks 2.0");
mapMsg.setLong(BookCost, 50.00);
```

```
String bookTitle = mapMsg.getString("BookTitle");
```

Object Message

```
ObjectMessage objectMsg = session.createObjectMessage();
Book book = new Book("WinSocks 2.0");
objectMsg.setObject(book);
```

BytesMessage



NOTE *The class of the object placed in the `ObjectMessage` type must implement the `Serializable` interface.*

The `BytesMessage` type contains a stream of uninterrupted bytes. The receiver of the message provides interpretation of the message bytes. You should use this message type for communication that requires the proprietary interpretation of message data.

Understanding JMS P2P Programming

In a JMS P2P domain, a sending application puts a message on a queue. Depending on the nature of communication, the sending application can expect a reply message (a *request-reply* pattern). In other situations, the sending application does not need an immediate reply (a *send-and-forget* pattern). If a reply message is necessary, the sending application indicates to the receiving application (in the message header field called `JMSReplyTo`) the name of a local queue where it expects to receive the reply message.

In the request-reply pattern, the sending application can function in two ways. In a pseudo-synchronous way, the application is blocked while waiting for the arrival of the reply message. In an asynchronous way, the application is not blocked and can perform other processing. Later, it can check the reply queue for the expected reply message. Listing 9-2 shows a fragment of the JMS code that sends a message.

Listing 9-2. Sending a Message

```
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueConnection;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.Queue;
import javax.jms.JMSException;
import javax.naming.InitialContext;
import javax.naming.Context;

public class MyJMSSender
{
    private String requestMessage;
    private String messageID;
    private int requestRetCode = 1;

    private QueueConnectionFactory queueConnectionFactory = null;
    private Queue requestQueue = null;
    private Queue responseQueue = null;
    private QueueConnection queueConnection = null;
    private QueueSession queueSession = null;
    private QueueSender queueSender = null;
    private TextMessage textMsg = null;
```

```

// Some code here
// some code here

public int processOutputMessages(String myMessage)
{
    // Lookup Administered Objects
    try {
        InitialContext initContext = new InitialContext();
        Context env = (Context) initContext.lookup("java:comp/env");
        queueConnectionFactory =
            (QueueConnectionFactory) env.lookup("tlQCF");

        requestQueue = (Queue) env.lookup("tlReqQueue");
        responseQueue = (Queue) env.lookup("tlResQueue");
        queueConnection = queueConnectionFactory.createQueueConnection();
        queueConnection.start();
        queueSession = queueConnection.
            createQueueSession(true, 0);
        queueSender = queueSession.createSender(requestQueue);
        textMsg = queueSession.createTextMessage();
        textMsg.setText(myMessage);
        textMsg.setJMSReplyTo(responseQueue);

        // Some processing here
        // Some processing here

        queueSender.send(textMsg);

        queueConnection.stop();
        queueConnection.close();
        queueConnection = null;
    }
    catch (Exception e)
    {
        // do something
    }
    return requestRetCode = 0;
}
}

```

Let's examine Listing 9-2. The first thing a JMS program needs to do is find the location of the JNDI naming context. If the program is developed under WSAD and is a part of a J2EE project, the location of the JNDI namespace is maintained

by the WSAD test server and is known to the runtime environment. In this case, it is sufficient to instantiate an instance of the `InitialContext` class by simply calling its default constructor:

```
InitialContext initContext = new InitialContext();
```

A program that runs outside WSAD or a program using a non-WSAD JNDI namespace—Lightweight Directory Access Protocol (LDAP), for example—has to provide some help in locating the JNDI namespace. This is done by specifying the `INITIAL_CONTEXT_FACTORY` class and `PROVIDER_URL` as parameters of the `Properties` or `Hashtable` object and then using this object as an input parameter to the `InitialContext` constructor method. You will now see several examples of creating the `InitialContext` object. Listing 9-3 is an example of locating the WSAD `InitialContext` object with a program running outside WSAD.

Listing 9-3. Locating the WSAD InitialContext Object with a Program Running Outside WSAD

```
Properties props = new Properties();
props.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.ibm.websphere.naming.WsnInitialContextFactory");
props.put(Context.PROVIDER_URL, "iiop://localhost/");

InitialContext initialContext = InitialContext(props);
```



NOTE Replace `localhost` with the hostname of your server where the JNDI server is located.

Listing 9-4 shows an example of locating the file-based JNDI `InitialContext`.

Listing 9-4. Locating the File-Based JNDI Context

```
Hashtable hashTab = new Hashtable ();
hashTab.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.fscontext.ReffFSContextFactory");
hashTab.put(Context.PROVIDER_URL, "file://c:/temp");

InitialContext initialContext = InitialContext(hashTab);
```

Listing 9-5 shows an example of locating the LDAP-based JNDI InitialContext.

Listing 9-5. Locating the LDAP JNDI Context

```
Hashtable hashTab = new Hashtable ();
hashTab.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
hashTab.put(Context.PROVIDER_URL,
            "file://server.company.com/o=provider_name, c=us");
InitialContext initialContext = InitialContext(hashTab);
```

The next step is to do a lookup for the subcontext `java:comp/env`. This is the J2EE-recommended JNDI naming subcontext where the environment variables are located. In this subcontext, the JMS program expects to find JMS-administered objects such as `QueueConnectionFactory` objects and `Queue` objects:

```
Context env = (Context) initContext.lookup("java:comp/env");
```

The following code fragment locates the JMS-administered object necessary for your program to operate:

```
queueConnectionFactory =
    (QueueConnectionFactory) env.lookup("QCF");
requestQueue = (Queue) env.lookup("requestQueue");
```

Next, you use the `QueueConnectionFactory` object to build the `QueueConnection` object:

```
queueConnection = queueConnectionFactory.createQueueConnection();
```

Using the JMS QueueConnection Object

The JMS `QueueConnection` object provides a connection to the underlying MOM (in this case, to the WebSphere MQ queue manager). A connection created this way uses the default Java binding transport to connect to the local queue manager. For an MQ client (an MQ program running on a machine without the local queue manager), the `QueueConnectionFactory` object needs to be adjusted to use the client transport:

```
QueueConn.setTransportType(JMSC.MQJMS_TP_CLIENT_MQ_TCPIP);
```

The `QueueConnection` object is always created in the stop mode and needs to be started:

```
queueConnection.start();
```

Once a connection is built, you use the `createQueueSession` method of the `QueueConnection` object to obtain a session. The `QueueSession` object has a single-threaded context and is not thread-safe. Therefore, the session object and objects created based on the session are not thread-safe and must be protected in a multi-threaded environment. The method takes two parameters. This is the statement that builds the `QueueSession` object:

```
queueSession =  
    queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
```

The first is a boolean parameter that specifies the JMS transaction type—in other words, whether the queue session is transacted (`true`) or nontransacted (`false`). The JMS transaction type is primarily used to regulate the message delivery mechanism and should not be confused with the EJB transaction type (`NotSupported`, `Required`, and so on), which determines the transaction context of the EJB module itself. The second parameter is an integer that determines the acknowledge mode. It determines how the message delivery is confirmed to the JMS server.

If the queue session is specified as transacted (`true`), the second parameter is ignored because the acknowledgment in this case happens automatically when the transaction is committed. If the transaction is rolled back, no acknowledgment will be performed, the message is considered undelivered, and the JMS server will attempt to redeliver the message. If multiple messages participate in the same session context, they are all processed as a group. Acknowledgment of the last message automatically acknowledges all previously unacknowledged messages. Similarly with rollback, the entire message group is considered undelivered with the JMS server attempting to redeliver them again.

This is how it works beneath the surface: When the sender sends a message, the JMS server receives the message. If the message is persistent, it writes the message to a disk first and then it acknowledges the message. Starting from this point, the JMS server is responsible for delivering the message to the destination. It will not remove the message from the staging area until it gets a client acknowledgment. For non-persistent messages, acknowledgment is sent as soon as the message is received and kept in memory.

If the queue session is specified as nontransacted (`false`), then the second parameter defines the acknowledgment mode. The available values are `AUTO_ACKNOWLEDGE`, `DUPS_OK_ACKNOWLEDGE`, and `CLIENT_ACKNOWLEDGE`:

- It is typical to use the `AUTO_ACKNOWLEDGE` mode for nontransacted sessions. For transacted sessions, `AUTO_ACKNOWLEDGE` is always assumed.
- The `DUPS_OK_ACKNOWLEDGE` mode is a “lazy acknowledgment” message delivery. It reduces the network overhead by minimizing work done to prevent message duplicates. You should only use it if duplicate messages are expected and there is logic in place to handle them.
- With the `CLIENT_ACKNOWLEDGE` mode, message delivery is explicitly acknowledged by calling the `acknowledge` method on the message object.

With the `AUTO_ACKNOWLEDGE` mode, acknowledgment is typically done at the end of the transaction. The `CLIENT_ACKNOWLEDGE` mode allows the application to speed up this process and make an acknowledgment as soon as the processing is done, well before the end of the transaction. This type of acknowledgment is also useful when a program is processing multiple messages. In this case, the program can issue an acknowledgment when all required messages are received.

For a nontransacted session, once a message is successfully put on a queue, it immediately becomes visible to the receiving program and it cannot be rolled back. For a transacted session, the JMS transacted context ensures that messages are sent or received as a group in one unit of work. The transacted context stores all messages produced during the transaction but does not send them to the destination.

Only when the transacted queue session is committed are the stored messages sent to the destination as one block and become visible to the receiving program. If an error occurs during the transacted queue session, messages that have been successfully processed before the error occurred will be undone. A queue session defined as transacted always has a current transaction; no `begin` statement is explicitly coded. As always, there is a tradeoff—transacted sessions are slower than nontransacted.



NOTE *You should not confuse the transacted and nontransacted JMS QueueSession modes with the corresponding property of the Java method that implements the JMS logic. The method property `TX_REQUIRED` indicates that the method runs within a transaction context. This ensures that a database update and a message placement on the queue would be executed as a unit of work (both actions committed or rolled back). By the way, when a container-managed transaction is selected, a global two-phase commit transaction context will be activated. (See the “Understanding Two-Phase Commit Transactions” section for more information.)*

In this case, the Datasource participating in the global transaction should be built based on the XA database driver. You will see an example of this type of processing in Chapter 10.

On the other hand, indicating true as the first parameter of the `createQueueSession` method establishes the JMS transaction context; multiple messages are treated as a unit of work. On the receiving side, multiple received messages are not confirmed until `queueSession.commit()` is issued, and when it is issued, it confirms receiving all messages uncommitted up to this point. On the sending side, all messages put on the destination queue are invisible until the sending program issues the `session.commit` statement.

Handling a Rollback

As already mentioned, for an abended transaction, a received message is sent back to the original queue. The next time the receiving program processes the queue, it will get the same message again, and the most likely scenario is that the transaction will again abend, sending the message back to the input queue. That creates a condition for an infinite loop.

To prevent this, you can set the `Max_retry` count on the listening port. After exhausting `Max_retry`, the message will no longer become available for selection by the receiving program or for a delivery in a push mode session. In addition, in a push mode, redelivered transactions will have a `JMSRedelivered` flag set. A program can check this flag by executing the `getJMSRedelivered` method on the message object. Messages are sent using a `QueueSender` JMS object. `QueueSender` is created by using the `createSender` method on the `QueueSession` object. A separate `QueueSender` is built for each queue:

```
queueSender = queueSession.createSender(requestQueue);
```

Next, you create a message (in this case, a `TextMessage` type) and set its content based on the string `myMessage` (passed to the method as an input parameter):

```
textMsg = queueSession.createTextMessage(myMessage);
```

You also specify the receiving queue where the receiving program should send a reply message:

```
textMsg.setJMSReplyTo(responseQueue);
```

Finally, with the `Sender` object, you send a message and then stop and close the connection:

```
queueSender.send(textMsg);
queueConnection.stop();
queueConnection.close();
```

After the message is sent, you can recover the message ID assigned by JMS to a message (by getting the value of the `JMSMessageID` header field). Later, you can use this value to find a reply message that matches your request messageID:

```
String messageID = message.getJMSMessageID();
```

With the JMS connection pooling in place, the closed session is not discarded but simply returned to the pool of available connections for reuse.

Closing JMS Objects

Garbage collection does not close a JMS object in a timely manner, which could lead to a problem when a program tries to create many short-lived JMS objects. It also consumes valuable resources. Therefore, it is important to explicitly close all JMS objects when they are no longer needed:

```
if (queueConn != null)
{
    queueConn.stop();
    queueConn.close();
    queueConn = null;
}
```

Closing a queue connection should automatically close all dependent objects created based on the connection. If this is not the case with your JMS provider, explicitly close all the open JMS objects in the order shown in Listing 9-6.

Listing 9-6. Closing JMS Objects

```
if (queueReceiver != null)
{
    queueReceiver.close();
    queueReceiver = null;
}

if (queueSender != null)
{
    queueSender.close();
    queueSender = null;
}

if (queueSession != null)
{
    queueSession.close();
    queueSession = null;
}

if (queueConn != null)
{
    queueConn.stop();
    queueConn.close();
    queueConn = null;
}
```

Receiving Messages

On the receiving side, the processing logic is similar to the sending side. Messages are received by the JMS `QueueReceiver` object, which is built based on the `QueueSession` object created for a specific queue. The important difference is how `QueueReceiver` receives messages. The `QueueReceiver` object can receive messages in a pseudo-synchronous or in an asynchronous way. Listing 9-7 shows the code fragment of the pseudo-synchronous way of receiving messages.

Listing 9-7. A Pseudo-Synchronous Way of Receiving Messages

```

import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueConnection;
import javax.jms.QueueSender;
import javax.jms.Queue;
import javax.jms.Exception;
import javax.naming.InitialContext;

public class MyJMSReceiver
{
    private String responseMessage;
    private String messageID;
    private int replyRetCode = 1;

    private QueueConnectionFactory queueConnectionFactory = null;
    private Queue inputQueue = null;
    private QueueConnection queueConnection = null;
    private QueueSession queueSession = null;
    private QueueReceiver queueReceiver = null;
    private TextMessage textMsg = null;

    public void processIncomingMessages()
    {
        // Lookup Administered Objects
        InitialContext initContext = new InitialContext();
        Context env = (Context) initContext.lookup("java:comp/env");
        queueConnectionFactory =
            (QueueConnectionFactory) env.lookup("tlQCF");
        inputQueue = (Queue) env.lookup("tlQueue");
        queueConnection = queueConnectionFactory.createQueueConnection();
        queueConnection.start();
        queueSession = queueConnection.createQueueSession(true, 0);
        queueReceiver = queueSession.createReceiver(inputQueue);

        // Wait one second for the arrival of a message
        TextMessage inputMessage = queueReceiver.receive(1000);

        // Some processing here
        // Some processing here
    }
}

```

```

        queueConnection.stop();
        queueConnection.close();
    }

}

```

Let's examine Listing 9-7. The message is received by the `QueueReceiver` object executing the `receive` method. This method has one parameter that indicates the wait interval (in milliseconds). In Listing 9-7, the `QueueReceiver` object waits for one second before it expires, gets unblocked, and returns control to the program. If the `wait-interval` parameter is specified, the `QueueReceiver` object is blocked for the specified interval, waiting for the arrival of a message. If no message arrives during the wait interval, the `QueueReceiver` object times out without getting a message and returns control to the program.

There is a “no wait” version of this method where the `QueueReceiver` checks for the message and immediately returns control to the program if no message is available. The following is the example:

```

TextMessage message = queueReceiver.receiveNowait();

```

If the `wait-interval` parameter is not specified, the `QueueReceiver` waits indefinitely for the message. This version of the `receive` method should be used with great care because the program can be locked indefinitely:

```

TextMessage message = queueReceiver.receive();

```

Regardless of the variations in the `wait-interval` parameter, this is a pull method of message delivery, which (as mentioned) is quite inefficient. In addition to just being inefficient, it is inappropriate for the J2EE EJB layer and cannot be used inside EJB components for reasons discussed shortly. However, this type of processing is suitable for processing inside servlets, JavaServer Pages (JSP), and stand-alone Java JMS applications.

The second way of receiving messages is asynchronous. To do this, the `QueueReceiver` object must register a `MessageListener` class by using the `setMessageListener(class_name)` method of the `QueueReceiver` object. The `class_name` parameter can point to any class that implements the `onMessage` interface method. In this example, it is the same class (indicated by the `this` parameter). Listing 9-8 shows a code example where the `onMessage` method is implemented in the same class (the `try/catch` blocks are not shown here for simplicity).



NOTE *The upcoming receiving message listings are not suitable for the EJB components. These code fragments are suitable for processing inside servlets, JSPs, and stand-alone Java JMS applications.*

Listing 9-8. Example of the Listener Class

```
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueConnection;
import javax.jms.QueueReceiver;
import javax.jms.Queue;
import javax.jms.Exception;
import javax.naming.InitialContext;

public class MyListenerClass implements javax.jms.MessageListener
{
    private int responseRetCode = 1;
    private boolean loopFlag = true;

    private QueueConnectionFactory queueConnectionFactory = null;
    private Queue responseQueue = null;
    private QueueConnection queueConnection = null;
    private QueueSession queueSession = null;
    private QueueSender queueSender = null;

    public void prepareEnvironment(String myMessage)
    {
        // Lookup Administered Objects
        InitialContext initContext = new InitialContext();
        Context env = (Context) initContext.lookup("java:comp/env");
        queueConnectionFactory =
            (QueueConnectionFactory) env.lookup("tlQCF");
        responseQueue = (Queue) env.lookup("tlResQueue");
        queueConnection = queueConnectionFactory.createQueueConnection();
        queueSession = queueConnection.createQueueSession(true, 0);
        queueReceiver = queueSession.createReceiver(responseQueue);

        queueReceiver.setMessageListener(this)
        queueConnection.start();
    }
}
```

```

public void onMessage(Message message)
{
    // We expect the text message type
    if (message instanceof TextMessage)
    {
        String responseText = "Confirmed. " +
                               ((TextMessage) message).getText();

        // When a message that starts from the @ character arrives, it stop the loop
        // and the MessageListener terminates.

        if (responseText.charAt(0) == '@')
        {
            loopFlag = 1; // Terminate processing;
        }
        else
        {
            // Continue processing message
            // We know the Reply Queue here and don't need this field.
            // It is used here to show how a queue to send the reply
            // message to can be obtained
            Destination replyToQueue = message.getJMSReplyTo();

            // Set the reply message
            TextMessage responseMessage =
                responseSession.createTextMessage(responseText);

            // Form a CorrelationID equal to the MessageID, so the client
            // can map the response record to his/her original request.
            messageID = message.getJMSMessageID();
            responseMessage.setJMSCorrelationID(messageID);
            //Set the message destination
            responseMessage.setJMSDestination(replyToQueue)
            queueSender.send(responseMessage);
        }
    }
}

// Keep the listener alive
while(loopFlag)
{
    // Yield control to other tasks (sleep for 2 seconds)
    System.out.println("Inside the listener loop");
    Thread.currentThread().sleep(2000);
}

```

```
// Terminate processing when the loopFlag field is set to false.
queueConn.stop();
queueConnection.close();
} // End of the MyListenerClass
```

When a `MessageListener` object is registered, a new thread is created that implements the `MessageListener` logic. You need to keep this thread alive, so you run a while loop that sleeps for a specified number of seconds (two seconds, in this case) to yield the processor control to other tasks. When it wakes up, it checks the monitored queue and goes back to sleep. Whenever a message arrives on a queue that is monitored by the registered `MessageListener` object, JMS invokes the `MessageListener` object and calls its `onMessage(message)` method, passing the message as a parameter.

This is a push method of message receiving. It is more efficient but still inappropriate inside EJB components. The following section discusses why both methods of receiving messages are inappropriate to use inside the EJB components and then shows the solution. Although this is discussed in the P2P domain, the same considerations apply to the Pub/Sub domain as well.

Using JMS Message Driven Beans (MDBs)

Earlier in this chapter (when discussing the JMS receive processing logic), you learned that the code listings are not suitable for EJBs, but they are suitable for servlets, JSPs, and stand-alone Java applications. This is because there are several technical issues involved with the receiving side of the JMS processing. Typically, JMS programs are developed using two interaction patterns:

Send-and-forget: The JMS client program sends a message but does not need a reply. From a performance point of view, this is the best pattern because the sender does not need to wait for the request to be processed and can continue processing.

Synchronous request-reply: The JMS client program sends a message and waits for a reply. Such interaction under JMS is done by executing a pseudo-synchronous receive method (already discussed). There is an issue here, however. If your EJB module operates under a transaction context (which is typically the case), you cannot perform request-reply processing in one transaction. The reason is that when the sender submits a message, the receiver can get it only when the sender commits the transaction. Therefore, within a single transaction, it is impossible to get a reply because within the uncommitted transaction context the receiver will never get the message and will not be able to reply. The solution is that request-reply must always be performed as two separate transactions.

There is an additional problem on the receiving side of the communication that is specific to EJBs. With asynchronous communication, you never know when to expect the reply. The main idea of asynchronous communication is that you can continue working on the sending side even if the receiving side is not currently active. The request-reply mode presumes that an EJB component (say, a session bean) should wait for a response after sending a message to a particular destination. J2EE is actually a component-based transaction-processing environment designed for processing a large number of short-lived tasks. It is not intended for tasks being blocked for a substantial period waiting for a response.

To solve this problem, Sun Microsystems developed and added to the EJB 2.0 specification a new type of EJB bean, the MDB. The MDB was specifically designed to handle the problems of processing JMS asynchronous messages (on the receiving side) with the EJB components. The solution was to remove the task of listening for a message's arrival from an EJB component and delegate it to a container. Thus, MDB components run under the control of the container. The container works as a listener on a particular queue or topic on behalf of an MDB. When a message arrives on that queue or topic, the container activates this MDB and calls its `onMessage` method (passing the arrived message as a parameter).

MDBs are asynchronous components, and they work differently than the rest of EJB components (session and entity beans) that are synchronous. MDBs do not have Remote and Home Interfaces because they cannot be activated by clients. MDBs are activated only by the arrival of messages. Another important aspect of MDBs is the way in which they run under the transaction and security contexts. Because MDB components are completely decoupled from their clients, they do not use the client's transaction and security contexts.

Remote clients that send JMS messages can potentially run in different environments that are not J2EE environments (they can be just Java programs). They might not have any security or transaction context. Therefore, the transaction and security contexts of the sender are never propagated to the receiver MDB components. Because MDBs are never activated by clients, they can never execute under the client's transaction context. Therefore, the following transaction attributes have no meaning for MDBs: `Supports`, `RequiresNew`, `Mandatory`, and `None`. These transactional attributes imply propagation of the client transaction context. Only two transactional attributes, `NotSupported` and `Required`, can be used with MDBs. MDB components with the `NotSupported` attribute process messages without any transaction context.

MDBs (as EJB beans) may participate in two types of transactions: bean-managed or container-managed transactions. Of all the MDB methods, only the `onMessage` method can participate in the transaction context. If a developer selects an MDB to participate in the bean-managed transaction context, then the MDB is allowed to begin and end a transaction inside the `onMessage` method. The problem with this assignment is that the received message stays outside of the transaction

context that always starts inside the `onMessage` method (too late for the message to be a part of it). In this case, you should handle messages in a rollback situation manually.

If a developer selects an MDB to participate in the container-managed transaction context, the entire scenario works differently. With the `Required` transactional attribute selected, the container starts a transaction at the time it receives a message; therefore, the message becomes part of the transaction, and it can be acknowledged whether the transaction is committed or returned to the sending queue if the transaction is rolled back.

Transaction rollback could happen in two situations: The program can explicitly call the `setRollbackOnly` method or throw a system exception within the `onMessage` method (remember that throwing the application exception does not trigger the rollback). In the case of transaction rollback, the message will be returned to the original queue, and the listener will send the message for processing again. Typically, this will create an indefinite loop and cripple the application. The `Max_retries` attribute, set during configuration of the listener port, controls the number of times the listener is allowed to retrieve the re-sent message. After that, the listener will stop processing messages (not a good solution).

WebSphere MQ, when used as the JMS provider, has a better solution. You can configure it to stop delivering the message after the specified number of attempts and send it to a specified error queue or `Dead.Letter.Queue`. Remember that MDBs are stateless components, meaning they do not maintain any state between different method invocations. They also have no identity of the client because they are never called by clients. MDBs execute anonymously. All MDB components must implement the `javax.ejb.MessageDrivenBean` and `javax.jms.MessageListener` interfaces.

In addition to the `onMessage` method, MDBs have several callback methods—methods called by a container:

- The `ejbCreate` method is called by a container to create the MDB instance. Some initialization logic can happen here.
- The `setMessageDrivenContext` method is called by the container when the bean is first added to the pool of MDB beans. This is typically used to capture the `MessageDrivenContext` and save it in a class variable, for example:

```
public void setMessageDrivenContext
    (javax.ejb.MessageDrivenContext mdbContext)
{
    messageDrivenContext = mdbContext;
}
```

- The `ejbRemove` method is called when the container moves the bean from the pool to no state. Typically, cleanup processing happens here.

Typically, it is not recommended to perform the business logic inside the `onMessage` method. It is a best practice to delegate other EJB components to perform these tasks. You will see an example of this type of delegation in Chapter 10.

The MDB container automatically handles the concurrency of processing multiple messages. Each MDB instance processes one message and is never called for processing another message until the `onMessage` method returns control to the container. When multiple messages need to be concurrently processed, the container activates multiple instances of the same MDB.

Starting from WebSphere 5.0, full support of MDBs is provided by the development environment (WSAD 5.0) and the runtime environment (WAS 5.0). Listing 9-9 shows a conceptual fragment of the MDB code example.

Listing 9-9. Conceptual Fragment of the MDB Code

```
package some-package
import javax.jms.Message;
import javax.jms.MapMessage;
import javax.naming.InitialContext;
import java.util.*;

public class LibraryNotificationBean implements javax.ejb.MessageDrivenBean,
javax.jms.MessageListener
{
    MessageDrivenContext messageDrivenContext;
    Context jndiContext;

    public void setMessageDrivenContext(MessageDrivenContext msgDrivenContext)
    {
        messageDrivenContext = msgDrivenContext;
        try
        {
            jndiContext = new InitialContext();
        }
        catch(NamingException ne )
        {
            throw new EJBException(ne);
        }
    }
}
```

```

public void ejbCreate()
{
}

public void onMessage(Message notifyMsg)
{
    try
    {
        MapMessage notifyMessage = (MapMessage) notifyMsg;
        String bookTitle = (String) notifyMessage.getString("BookTitle");
        String bookAuthor = (String) notifyMessage.getString("BookAuthor");
        String bookCatalogNumber = (String)
            notifyMessage.getString("bookCatalogNumber");

        Integer bookQuantity = (Integer)
            notifyMessage.getInteger("BookQuantity");

        // Do some processing (call EJB components)

    }
    catch (Exception e)
    {
        throw new EJBException(e);
    }
}

public void ejbRemove()
{
    try
    {
        jndiContext.close();
        jndiContext = null;
    }
    catch(NamingException ne)
    {
        // Do nothing
    }
}
}

```

Regulating Message Persistence

Messages can be persistent and nonpersistent. A single queue can hold both persistent and nonpersistent messages. Persistent messages are written to a disk and are recoverable if the system goes down. As usual, there is a performance cost for persistence. Persistent messages are about seven percent slower. One way of controlling the persistence of messages is to use the queue property when defining a queue: `DEFINE TYPE (name) [property]`. If the persistent property is not set explicitly, the system will use a default. The JMS application can also define persistence:

- `PERSISTENCE(QDEF)`: Persistence is inherited from the queue default.
- `PERSISTENCE(PERS)`: Messages are persistent.
- `PERSISTENCE(NON)`: Messages are nonpersistent.

You can also regulate message persistence by setting the value of the message attribute header `JMSDeliveryMode` to `DeliveryMode.PERSISTENT` or `DeliveryMode.NON_PERSISTENT`. Messages processed under the transacted session must always be persistent.

Using Message Selectors

JMS provides a mechanism for selecting a subset of messages on a queue by filtering out all messages that do not meet the selection condition. The selector can refer to message header fields as well as message property fields. The following are examples of using this facility:

```
QueueReceiver queueReceiver =
    queueSession.createReceiver(requestQueue, "BookTitle = 'Windows 2000'");
QueueBrowser queueBrowser =
    queueSession.createBrowser(requestQueue,
    "BookTitle = 'Windows 2000' AND
    BookAuthor = 'Robert Lee'");
```

Notice that the strings (such as `Windows 2000`) are surrounded by single quotes inside double quotes.

Understanding JMS Pub/Sub Programming

Programming for the Pub/Sub domain is similar to the P2P domain. The main difference is in the destination objects. Messages in the Pub/Sub domain are published to (similar to *sent*) and consumed from (similar to *received*) JMS objects called *topics*. Topics function as virtual channels and encapsulate a Pub/Sub destination object.

In the P2P mode, a message (sent to a queue) can be received by only one message consumer. In the Pub/Sub domain, a message producer can publish a single message on a topic that can be distributed and consumed by many message consumers. More than that, a message producer and its message consumer are so loosely coupled that a producer does not need to know anything about the message consumers. Both message producers and message consumers only need to know a common destination (which is a topic of conversation).

A message producer is called a *publisher*, and a message consumer is called a *subscriber*. All messages published by a publisher for a specific topic are distributed to all subscribers of that topic. A subscriber receives all messages for which it has subscribed. Each subscriber receives its copy of the message. Subscriptions can be durable or nondurable. Nondurable subscribers receive only messages that have been published after they have subscribed.

Durable subscribers are able to disconnect and later reconnect and still receive messages that have been published while they were disconnected. Durable subscribers in the Pub/Sub domain (with some level of approximation) are similar to persistent messages/queues in the P2P domain. Publishers and subscribers never communicate directly. The Pub/Sub broker functions as a message cop, delivering all published messages to their subscribers.



NOTE Starting from WebSphere MQ 5.2, WebSphere MQ with the MA88 and MA0C extensions can function as JMS Pub/Sub brokers. In addition, WebSphere MQ Integrator can function as a Pub/Sub broker. Starting from MQ 5.3, MA88 became part of the base package, so you need to install only MA0C on top of the MQ 5.3 installation. For MQ JMS, for Pub/Sub to work correctly, you must create a number of system queues on the queue manager that runs the Pub/Sub broker.

The MQ JMS MA0C extension provides a procedure that builds all the necessary Pub/Sub system queues. This procedure is called `MQJMS_PSQ.mqsc`, and it resides in the `<MQ-Install-Directory>\java\bin` directory. To build the system queues required by the Pub/Sub domain, enter the following command from this directory:

```
runmqsc < MQJMS_PSQ.mqsc
```

and press Enter.

You can arrange topic names in a tree-like hierarchy. Each topic name in the tree is separated by a slash (/)—for example, `Books/UnixBooks/SolarisBooks`. You can use wildcards within topics to facilitate subscribing to more than one topic. This is an example of a wildcard used within the topic hierarchy: `Books/#`.

Listing 9-10 shows a code fragment of JMS Pub/Sub coding (the try/catch blocks are not shown for simplicity). In this example, subscribers of the `Books/UnixBooks/SolarisBooks` topic will receive all messages sent to the `SolarisBooks` topic, and subscribers of the `Books/#` topic will receive all Books messages (including messages sent to the `UnixBooks` and `SolarisBooks` topics).

Listing 9-10. Seeing JMS Pub/Sub in Action

```
import javax.jms.*;
import javax.naming.*;
import javax.ejb.*;

public class PublisherExample implements javax.ejb.SessionBean
{
    private TopicConnectionFactory topicConnFactory = null;
    private TopicConnection topicConnection = null;
    private TopicPublisher topicPublisher = null;
    private TopicSession topicSession = null;
    private SessionContext sessionContext = null;

    public void setSessionContext(SessionContext ctx)
    {
        sessionContext = ctx;
    }
    public void ejbCreate() throws CreateException
    {
        InitialContext initContext = new InitialContext();

        // Look up the topic connection factory from JNDI
```

```

topicConnFactory =
    (TopicConnectionFactory)
        initContext.lookup("java:comp/env/TCF");

// Look up the topics from JNDI
Topic unixBooksTopic = (Topic)
    initContext.lookup("java:comp/env/UnixBooks");
Topic javaBooksTopic = (Topic)
    initContext.lookup("java:comp/env/JavaBooks");
Topic linuxBooksTopic = (Topic)
    initContext.lookup("java:comp/env/LinuxBooks");
Topic windowsBooksTopic = (Topic)
    initContext.lookup("java:comp/env/WindowsBooks");
Topic allBooksTopic = (Topic)
    initContext.lookup("java:comp/env/AllBooks");

// Create a connection
topicConnection = topicConnFactory.createTopicConnection();
topicConn.start();

// Create a session
topicSession =
    topicConn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
}

public void publishMessage(String workMessage, String topicToPublish)
{

    // Create a message
    TextMessage message = topicSession.createTextMessage(workMessage);

    // Create topic publishers and send messages
    if ((topicToPublish.toLowerCase()).equals("java"))
    {
        TopicPublisher javaBooksPublisher =
            topicSession.createPublisher(javaBooksTopic);
        javaBooksPublisher.publish(message);
    }

    if ((topicToPublish.toLowerCase()).equals("unix"))
    {
        TopicPublisher unixBooksPublisher =
            topicSession.createPublisher(unixBooksTopic);
    }
}

```

```

        unixBooksPublisher.publish(message);
    }

    if ((topicToPublish.toLowerCase()).equals("linux"))
    {
        TopicPublisher linuxBooksPublisher =
            topicSession.createPublisher(linuxBooksTopic);
        linuxBooksPublisher.publish(message);
    }

    if ((topicToPublish.toLowerCase()).equals("windows"))
    {
        TopicPublisher windowsBooksPublisher =
            topicSession.createPublisher(windowsBooksTopic);
        windowsBooksPublisher.publish(message);
    }

    TopicPublisher allBooksPublisher =
        topicSession.createPublisher(allBooksTopic);
    allBooksPublisher.publish(message);
}

public void ejbActivate()
{
}

public void ejbPassivate()
{
}

public void ejbRemove()
{
    // Clean up code fragment

    if (javaBooksPublisher != null)
    {
        javaBooksPublisher.close();
        javaBooksPublisher = null;
    }

    if (unixBooksPublisher != null)
    {
        unixBooksPublisher.close();
    }
}

```

```

        unixBooksPublisher = null;
    }

    if (linuxBooksPublisher != null)
    {
        linuxBooksPublisher.close();
        linuxBooksPublisher = null;
    }

    if (windowsBooksPublisher != null)
    {
        windowsBooksPublisher.close();
        windowsBooksPublisher = null;
    }

    if (allBooksPublisher != null)
    {
        allBooksPublisher.close();
        allBooksPublisher = null;
    }

    if (topicSession != null)
    {
        topicSession.close();
        topicSession = null;
    }

    if (topicConnection != null)
    {
        topicConnection.stop();
        topicConnection.close();
        topicConnection = null;
    }
}

```

This code is straightforward and does not need additional explanation. The only original part is how you publish a message for different topics. For each specific topic, you create a corresponding publisher and use it to publish a message on this topic.

If an MDB only receives messages and delegates future message processing to business components (meaning that there is no message sending or publishing logic inside the MDB), the code for the MDB is identical to the P2P domain of processing (see Listing 9-9). The only change for using the same MDB is that you must change the listener port from listening on a queue to listening on a topic. You will see the example of this dual usage in Chapter 10.

Understanding Two-Phase Commit Transactions

For enterprise-level processing, you typically operate under a transaction context to control the integrity of the JMS and non-JMS processing of the business logic (committing or rolling back all steps as a unit of work). The transaction context is especially important when (in addition to placing a message on a queue) you also need to place a record on a database (two-phase commit—all or nothing).

To support two-phase commit, the JMS specification provides an XA version of the following JMS objects: `XAConnectionFactory`, `XAQueueConnectionFactory`, `XASession`, `XAQueueSession`, `XATopicConnectionFactory`, `XATopicConnection`, and `XATopicSession`. In addition, you must use XA versions of other resources involved in the global transaction. Specifically, for JDBC resource, you must use the JDBC `XADatasource`. Finally, the JTA `TransactionManager` coordinates the global transaction. Listing 9-11 shows the steps necessary to establish a global transaction.

Listing 9-11. Setting a Global Transaction

```
// Obtain the JTA TransactionManager from the JNDI namespace.
TransactionManager globalTxnManager =
    jndiContext.lookup("java:comp/env/txt/txnmgr");

// Start the global transaction
globalTxnManager.begin();

// Get the transaction object
Transaction globalTxn = globalTxnManager.getTransaction();

// Obtain the SA Datasource
XADatasource xaDatasource =
    jndiContext.lookup("java:comp/env/jdbc/datasource");
```

```

// Obtain the connection
XAConnection jdbcXAConn = xaDatasource.getConnection();

// Obtain the XAResource from the XA connection
XAResource jdbcXAResource = jdbcXAConn.getXAResource();

// Enlist the XAResource in the global transaction
globalTxn .enlist(jdbcXAResource);

// Obtain XA Queue Connection Factory
XAQueueConnectionFactory xaQueueConnectionFactory =
    JndiContext.lookup("java:comp/env/jms/xaQCF")

// Obtain XA Queue Connection
XAQueueConnection xaQueueConnection =
    XaQueueConnectionFactory.createXAQueueConnection();

// Obtain XA Queue Session
XAQueueSession xaQueueSession = xaQueueConnection.createXAQueueSession();

// Obtain XA Resource from session
XAResource jmsXAResource = xaQueueSession.getXAResource();

// Enlist the XAResource in the global transaction
globalTxn .enlist(jmsXAResource);

// --- some work ---

// Commit global transaction
globalTxn.commit();

```

Summary

This chapter introduced you to JMS, the new J2EE 1.3 asynchronous messaging standard. It discussed the advantages of the asynchronous communication, the two JMS domains (P2P and Pub/Sub), MDBs, JMS transactions, and two-phase commit global transactions. In the next two chapters, you will see examples of JMS programming with WSAD 5.0.



<http://www.springer.com/978-1-59059-120-8>

WebSphere Studio Application Developer 5.0

Practical J2EE Development

Livshin, I.

2003, XXI, 656 p. 397 illus., Softcover

ISBN: 978-1-59059-120-8

A product of Apress