8

Messaging

CERTIFICATION OBJECTIVES

This chapter is organized a little differently from other chapters in the book. Before we look at the certification objectives, we are going to cover material that will help you to understand the basics of messaging and the Java Message Service (JMS) API. This material provides a valuable and necessary context for the objectives, particularly if you are new to messaging or JMS.

Messaging Basics

The main subject areas in the messaging arena with which you need to be familiar are messages, message-oriented middleware, JMS, message types, and communication modes. We'll take a look at each of these subject areas before looking at the scenarios that are appropriate for messaging.

Messages

A *message* is a unit of data that is sent from one process to another processes running on either the same or a different machine. The data in the message can range from simple text to a more complex data structure (such as a Java `HashMap`, which can be used to store name value pairs). The only restriction is that the object must be *serializable* so that it can be easily transformed into a sequence of bytes, transmitted across a network, and then recreated into a copy of the original object.

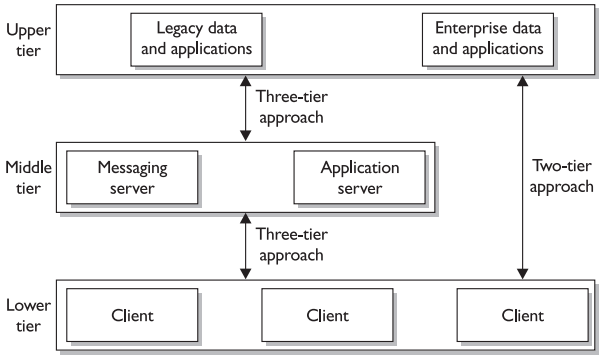
Middleware

Companies that create large transaction processing solutions to serve their customers, suppliers, or internal users swiftly become aware of the fact that a poorly designed system will not be able to keep pace with an ever-increasing transaction volume. They also quickly realize that just adding new hardware to the mix does not necessarily solve the problem.

These ever-increasing-in-volume types of applications are created using a three-tier instead of a two-tier approach. The existing two tiers, containing the presentation tier and the persistence tier, are supplemented with a middle tier that contains an application commonly known as *middleware*. Figure 8-1 depicts these three tiers.

FIGURE 8-1

Three tiers of an application



The middleware provides business solutions and services, such as these:

- **Database management** The ability to access a database server such as DB/2, Oracle, or SQL Server.
- **Messaging** The ability to send and receive data between applications.
- **Naming** The ability to find a resource by name instead of by location.
- **Security** The ability to authenticate and authorize a user (note that this is not solely a middle-tier responsibility).

The Java Platform Enterprise Edition (Java EE) APIs extend standard Java and provide access to these services. The middleware is loosely coupled with the parts of the application that are running in the presentation and persistence tiers. This loose coupling improves the reliability of middleware by isolating it from failures that may occur on either of the other tiers.

Message-Oriented Middleware

Message-oriented middleware, also known as MOM, is middleware that is used for messaging. This middleware is the infrastructure that provides dependable mechanisms that enable applications to create, send, and receive messages within an enterprise environment.

The advantage of message-based applications is that they are event driven. They exchange messages in a wide variety of formats and deliver messages quickly and reliably.

Here is the most recent list of the enterprise messaging vendors available at the time of writing:

- Adobe
- BEA Systems, Inc.
- IBM
- Oracle Corporation
- Sonic Software
- SpiritSoft, Inc.
- Sun
- TIBCO Software, Inc.

Communication Modes

Typically, applications use *synchronous* method calls for communication. In this type of communication, the requester is blocked from processing any further commands until the response (or a time-out) is received.

Synchronous communication is conducted between two active participants. The receiver has to acknowledge receipt of the message before the sender can proceed. From the sender's perspective, this is known as a *blocking* call. As the volume of traffic increases, more bandwidth is required, and the need for additional hardware becomes critical. These implementations are more easily directly affected by hardware, software, and network failures. When capacities are exceeded, the opportunity to process the information is typically lost.

An example of synchronous communication is credit card authorization. When your card is swiped through a card reader and the details of the purchase are entered, the machine dials the authorization computer and waits for a response (approval or denial of the purchase).

In *asynchronous* communication, the parties are peers and can send and receive messages at will. Asynchronous communication does not require real-time acknowledgment of a message; the requester can continue with other processing once it sends the message. From the sender's perspective, this is known as a *non-blocking* call.

An example of asynchronous communication is e-mail. Even if your computer is switched off or your e-mail client is not running, other people can still send e-mail messages to you. When you start your e-mail client, you will be able to view the e-mail messages that have accumulated in your inbox.

Message Models

JMS supports two basic message models known as *publish/subscribe* (pub/sub), in which messages are published on a one or more-to-many basis, and *point-to-point* (PTP), in which messages are sent on a one-to-one basis. The JMS specification requires that the messaging vendor product support at least one of these models in order to be compliant. An in-depth explanation of these two models appears later in this chapter in the sections, "How the Point-to-Point Message Model Works" and "How the Publish/Subscribe Message Model Works."

CERTIFICATION OBJECTIVE 8.01

Identify Scenarios That Are Appropriate to Implementation Using Messaging

The following table shows some example messaging implementations that can be used as solutions to the given scenarios.

SCENARIO & SOLUTION	
You need to call a validation application to approve a customer's credit card purchases. Which type of messaging model is best used for this type of communication, and what type of communication works best in such a scenario?	Point-to-point model messaging is best because the message only needs to be processed one time by the validation system. Synchronous communication works best because the results are required before the customer is allowed to use the merchandise.
You are using an e-mail application, and you want to send a message to several recipients and receive replies from them all. Which type of communication is best for this type of application?	Asynchronous communication is best suited to an e-mail application because the recipients are not required to be online for the sender to send the message.
You need to broadcast information to many recipients…	The publish/subscribe messaging model is best because the broadcast capability is part of its design.
What messaging technology is most appropriate for guaranteeing the delivery of a message to a single recipient and to multiple recipients?	For the single recipient, use the point-to-point messaging model with persistent delivery mode. For multiple recipients, use the publish/subscribe model with a persistent delivery mode and a durable subscriber.
What messaging technology is most appropriate for sending and receiving messages in a transactional way?	Use the point-to-point model. Create transacted sessions and process messages with commit and rollback methods.

CERTIFICATION OBJECTIVE 8.02

List Benefits of Synchronous and Asynchronous Messaging

The benefits of synchronous messaging follow:

- Because both parties must be active to participate in synchronous messaging, if either party is not active, the message transaction cannot be completed.
- A message must be acknowledged before proceeding to the next message. If it is not acknowledged, the message cannot be considered processed.

The benefits of asynchronous messaging are as follows:

- As the volume of traffic increases, asynchronous messaging is better able to handle the spike in demand by keeping a backlog of requests in its queue and then operating at maximum capacity over a period of time instead of needing to service the requests instantaneously.
- Asynchronous messaging is less affected by failures at the hardware, software, and network levels.
- When capacities are exceeded, information is not lost; instead, it is delayed.

CERTIFICATION OBJECTIVE 8.03

Identify Scenarios That Are More Appropriate to Implementation Using Asynchronous Messaging, Rather Than Synchronous

The following scenarios are more appropriate to implementation using asynchronous messaging:

SCENARIO & SOLUTION	
You need to implement a messaging system in which a response is not required or not immediately required. Which messaging system is most appropriate?	Asynchronous messaging
You need a high-volume transaction processing capability for sending messages. Which type of messaging is best suited for this use?	Asynchronous messaging
You want a messaging system that uses your system hardware in an efficient manner. Which type of messaging should be used?	Asynchronous messaging

CERTIFICATION OBJECTIVE 8.04

Identify Scenarios That Are More Appropriate to Implementation Using Synchronous Messaging, Rather Than Asynchronous

The following scenario is more appropriate to implementation using synchronous messaging:

SCENARIO & SOLUTION	
You are using a credit card authorization/user login authentication system to send a message in which the response to the message is required before the transaction can be completed. Which type of messaging is most appropriate?	Synchronous messaging

Java Message Service

Message-oriented middleware products allow a developer to couple applications loosely together. However, these products are proprietary and quite often complex and expensive. The JMS provides a standard Java interface to these messaging middleware products, freeing developers from having to write low-level infrastructure code, or "plumbing," and allowing solutions to be built quickly and easily. In short, the JMS API provides a convenient and easy way to create, send, receive, and read an enterprise messaging system's messages using Java.

JMS applications can use databases to provide the storage to support message persistence that is necessary for guaranteeing delivery and order of messages. With the arrival of the Enterprise JavaBeans (EJB) 2.0 specification, the EJB message-driven bean (MDB) has been able to receive and process messages asynchronously within the EJB container. These message-driven beans can be instantiated multiple times to provide concurrent processing (and therefore faster throughput) of a message queue.

JMS provides an interface from Java applications to messaging products. JMS enables clients (or peers) to exchange data in the form of messages.

Following are the major advantages of using messaging for this exchange:

- Easy integration of incompatible systems
- Asynchronous communications
- One-to-many communications
- Guaranteed messaging
- Transactional messaging

Table 8-1 shows the various components of a JMS messaging application.

Handling Exceptions

If a problem occurs, an application can be notified asynchronously via the *ExceptionListener* interface. This interface identifies the JMS provider problem details to the JMS client. To handle exceptions, the developer must create a listening object that implements the *ExceptionListener* interface and codes the `onException (JMSException exception)` method.

TABLE 8-1 Components of a JMS Messaging Application

Component	Function
JMS provider	The host application on which the JMS application runs. The JMS provider converses with JMS applications and supplies the underlying mechanisms required for a messaging application.
Administered objects	JMS objects that are created and maintained by an administrator to be used by JMS clients.
Clients	Applications that can send and/or receive messages.
Messages	Bundles of information that are passed between applications. Each application defines the types of information a message can contain.

The listening object must register itself with the JMS provider by calling the `setExceptionListener (listenerobject)` method on the session.

Managing Sessions

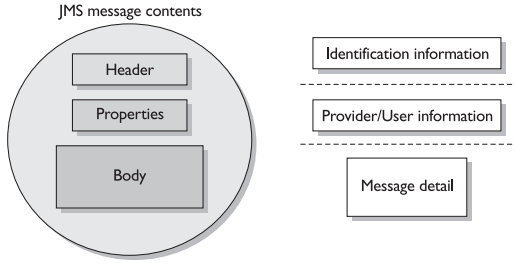
Table 8-2 describes the details of a JMS session. Unless noted otherwise, this information applies to both the publish/subscribe and point-to-point models.

TABLE 8-2 JMS Session Details

Session Detail	Description
Transacted session	A related group of messages that are treated as a single work unit. The transaction can be either committed or rolled back. When a message *sender* uses a transacted session and calls the `commit` method, the messages it produces are accepted for delivery. If it calls the `rollback` method, the messages it produces are destroyed. When a message *receiver* uses a transacted session and calls the `commit` method, the messages it consumes are acknowledged. If it calls the `rollback` method, the messages it consumes are recovered (not acknowledged).
Duplicate messages	Clients send messages knowing that JMS will deliver them only once. Therefore, the JMS provider must never deliver a message more than once or deliver a copy of a message that has already been acknowledged. When a copy of a message is delivered, the message header contains a redelivery flag field that will be set, telling the client that this message may have been received before but that, for whatever reason, the JMS server did not receive the client's acknowledgment of receipt. The redelivery flag is set by the JMS provider application, usually as the result of a recovery operation.
Message acknowledgment	If a JMS session is transacted, messages are acknowledged automatically by the commit mechanism and recovered by the rollback mechanism. If a session is not transacted, recovery must be handled manually, and messages are acknowledged in one of three ways: ***AUTO_ACKNOWLEDGE:*** For each message, the session automatically acknowledges that a client has received the message when the client returns from a call to receive a message or the *MessageListener* called by the session to process the message returns successfully. ***CLIENT_ACKNOWLEDGE:*** Client acknowledges the message by calling the `acknowledge` method on the message. This also acknowledges all messages that were processed during the session. ***DUPS_OK_ACKNOWLEDGE:*** Because the session lazily acknowledges the delivery of messages, duplication of messages may result if the JMS provider fails. This mode should be used only if consumers can tolerate duplicate messages. This mode reduces session overhead by minimizing the work the session does to prevent duplicate messages.

FIGURE 8-2

Structure of a JMS message



Components of a JMS Message

JMS messages have a simple and flexible format that allows the sender to create message formats used by non-JMS applications. The message can contain *simple* or *complex* data types. A JMS message is made up of one required component, called the *header*, and two optional components, called *properties* and a *body*. Figure 8-2 illustrates the structure of a JMS message.

Header Fields

The JMS message header includes a number of fields that contain information that can be used to identify and route messages. Each of these header fields includes the appropriate get/set methods. Most of the values are automatically set by the `send` or `publish` method, but the client can set some of them.

Table 8-3 describes a few header fields. For a complete list, refer to the JMS specification at the Sun web site *(http://java.sun.com/products/jms)*.

Properties

Properties are values that can add to the information contained in header fields. The JMS API provides some predefined property names that the JMS provider can support (these properties have a *JMS_* prefix).

The use of properties is optional. If you decide to use them, they can be of the type *boolean*, *byte*, *double*, *float*, *int*, *long*, *short*, or *String*. They can be set by the producer when the message is sent or by consumers upon receipt of the message. These properties, along with the header field, can be used in conjunction with a *MessageSelector* to filter and route messages based on the criteria specified.

Body

Five different message body formats, or types, allow a JMS client to send and receive data in many different forms and provide compatibility with existing messaging formats. Table 8-4 describes these message body formats.

TABLE 8-3

JMS Message Headers

Header Field	Description
JMSMessageID	Unique identifier for every message.
JMSDeliveryMode	*PERSISTENT* means that delivery of a message is guaranteed. It will continue to exist until all subscribers who requested it receive it. The message is delivered only once. *NON-PERSISTENT* delivery means that every reasonable attempt is made to deliver the message. But in the event of some kind of system failure, the message may be lost. These messages are delivered at most once.
JMSExpiration	The length of time, in milliseconds, that a message will exist before it is removed. Setting this to zero will prevent the message from being removed.
JMSPriority	Although it is not guaranteed, messages with a higher priority are generally delivered before messages with a lower priority. Priority 0 is the lowest and 9 is the highest. Priority 4 is the default. Priorities of 0–4 are grades of normal priority, and priorities of 5–9 are grades of higher priority.
JMSRedelivered	Notifies the client that it probably received this message at least once earlier, but for whatever reason, the client did not acknowledge its receipt. The JMS provider sets this flag, typically during a recovery operation after a failure.

TABLE 8-4

JMS Message Body Formats

Message Body Format	Content
ByteMessage	A stream of uninterpreted bytes. This type of message body should be used to match most legacy messages.
MapMessage	A set of name/value pairs, similar to a HashMap. The name part is a *String* object and the value is a Java primitive type.
ObjectMessage	A single serializable Java object or a collection of objects.
StreamMessage	A stream of Java primitive values that are entered and read sequentially.
TextMessage	Text formatted as a *String*. This form is well suited to exchanging XML data.

TABLE 8-5 JMS Interfaces

JMS Common Interfaces	Publish-Subscribe Interfaces	Point-To-Point Interfaces
Destination	Topic	Queue
ConnectionFactory	TopicConnectionFactory	QueueConnectionFactory
Connection	TopicConnection	QueueConnection
Session	TopicSession	QueueSession
MessageProducer	TopicPublisher	QueueSender
MessageConsumer	TopicSubscriber	QueueReceiver, QueueBrowser

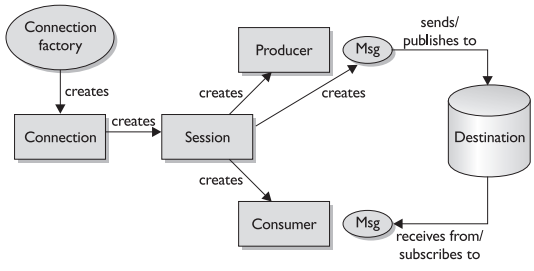
Required Components of a JMS Application

Several main components are required by an application that uses JMS. The first type is known as an *administered object*, because the administrator of the JMS provider application creates them. These objects are placed in a Java Naming and Directory Interface (JNDI) namespace and are administered from either a command-line program, a GUI, or an HTML-based management console. Table 8-5 contains a list of the main interfaces used by a JMS application.

JMS has two types of administered objects: *Destination* and *ConnectionFactory*. The *Destination* object contains configuration information supplied by the JMS provider. The client uses this object to specify a destination for messages that it wishes to send and/or a location from which to receive messages. Two types of *Destination* interfaces can be used: a *Queue* for the PTP model and a *Topic* for the pub/sub model. The *ConnectionFactory* is obtained via a JNDI lookup, and it contains the connection configuration information, or handle containing the IP address, enabling a JMS client application to create a connection with the JMS server.

The other main components required by an application that uses JMS are the *Connection* and the *Session*. The *Connection* component provides the physical connection to the JMS server, and the *Session* component is responsible for sending and receiving messages, managing transactions, and handling acknowledgments. Figure 8-3 shows the relationships between the JMS components.

FIGURE 8-3 JMS component relationships



Objects Used to Create and Receive Messages in a JMS Client Application

Four objects are used to create and receive messages in a JMS client application:

- *MessageProducer*
- *MessageConsumer*
- *MessageListener*
- *MessageSelector*

MessageProducer

A *MessageProducer* is created by a session and used to send messages to a destination. In the PTP model, the destination is called a *queue*. For the pub/sub model, the destination is called a *topic*.

When creating a *MessageProducer*, you can also specify the default delivery mode (*setDeliveryMode*). This can be either *NON_PERSISTENT*, which has a lower overhead because the message is not logged, or *PERSISTENT*, which requires the message to be logged, typically to a database. You can also specify the priority of the message (*setPriority*). Priority 0 is the lowest priority and 9 is the highest priority (4 is the default). Priorities of 0–4 are grades of normal priority, and priorities of 5–9 are grades of higher priority. You can also specify the expiration time *(setTimeToLive)*, which is the amount of time, in milliseconds, that a message should be available (set to 0 for unlimited time).

MessageConsumer

A *MessageConsumer* is created by a session and used to receive messages sent to the destination. The messages can be received in one of two ways: synchronously, where the client calls one of the receive methods (`receive` and `receiveNoWait`) after the consumer is started, or asynchronously, where the client registers a *MessageListener* and then starts the consumer.

The following code is an example of a synchronous connection. (Asynchronous connections are covered in the next section.)

```
// start the connection
queueConn.start();
// receive the first message (wait for a message)
Message message = queueReceiver.receive();
// receive the next message (wait for a minute only)
Message message = queueReceiver.receive(60000);
```

on the job

All messages in JMS are exchanged asynchronously between the clients, in that the producer does not receive acknowledgment from the consumer that it has processed the message. As soon as the message is sent or published, the producer is not blocked from sending or publishing another message immediately.

MessageListener

A *MessageListener* is an interface that needs to be implemented to process messages in an asynchronous fashion. To receive and process an asynchronous message, you must do the following:

- Create an object that implements the *MessageListener* interface. This includes coding the `onMessage()` method.
- Register the object with the session via the `setMessageListener()` method.
- Call the `Start()` method on the *Connection* object to begin receiving messages.

MessageSelector

A *MessageSelector* is a *java.lang.String* object specified by the client by the `createSubscriber()` method. The *MessageSelector* filters out messages that do not meet the criteria specified. The *MessageSelector* examines the message header and properties fields and compares them to an expression contained in a string. The syntax of this expression is based on a subset of SQL92 conditional expression syntax. SQL92 is a standard published by the SQL Standards committee formed by the American National Standards Institute and the International Standards Organization.

Table 8-6 shows some examples of these expressions, and the PTP example that is covered next shows a receiver that reads the queue with and without a *MessageSelector*.

exam watch

A message digest is a digital fingerprint value that is computed from a message, file, or byte stream.

How the Point-to Point Message Model Works

The PTP message model sends messages to a receiver on a one-to-one basis. Figure 8-4 is a diagram showing the PTP model.

TABLE 8-6

JMS Message Selector Examples

Value	Example
Arithmetic operators	`+, -, *, /`
Comparison operators	`<, >, <=, >=, IS NULL, IS NOT NULL, BETWEEN`

Expressions	`(Qty * Price) >= 12300`
	`Day = 'Tuesday'`
	`Month NOT IN ('June', 'July', 'August')`
	`Description LIKE 'UNITED%'`

Identifiers	`$name, JMSPriority, JMSXId, JMS_timeout`
Literals	`'string literal', 64, FALSE`
Logical operators	`AND, OR, NOT`

Examples of PTP implementation include the following:

- Instant messaging
- Receiving a transaction from another system
- Sending an order to another system
- Supply-chain processing

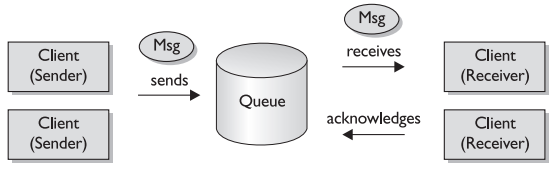
A message is delivered to a destination, known as a *queue*. Messages in a queue are processed on a first-in, first-out (FIFO) basis. In other words, the subscriber is guaranteed to get each message in the order in which it was sent. The first available receiver processes each message once. This differs from the pub/sub model, in which a single message can be published to one or more subscribers. In addition to processing the next message in a queue, the receiver is also able to browse through the messages in a queue (for example, to count them), but the receiver is unable to process the messages in any other order than FIFO.

The following is a list of the steps and interface classes required for the PTP model of communication:

1. Obtain the *QueueConnectionFactory* object via a JNDI lookup (the JNDI name will vary depending on the messaging vendor and site naming conventions).

FIGURE 8-4

Point-to-Point (PTP) model



2. Obtain a *QueueConnection* to the provider via the *QueueConnectionFactory*. (If security is enabled, pass a user ID and password to the `createQueue Connection` method.)
3. Obtain a *QueueSession* with the provider via the *QueueConnection*.
4. Obtain the queue via a JNDI lookup (the JNDI name will vary depending on the messaging vendor and site naming conventions).
5. Create either a *QueueSender* or a *QueueReceiver* via the *QueueSession* for the required queue.
6. Send and/or receive messages.
7. Close the *QueueConnection* (this will also close the *QueueSender* or *Queue Receiver*, and the *QueueSession*).

In the following example code, a JMS client sends a variety of *TextMessage*s and *ObjectMessage*s to a queue. Some of these messages are marked with a property called `Interesting` that is set to `true`. The sending client finishes up by sending text messages containing the text `finish` to indicate to the receiver that it can stop processing. The receiving JMS client is started with or without a command-line argument, and it will process the messages in the queue according to this argument setting.

The JNDI name for the connection factory and the queue will differ across messaging vendor implementations and also depend on your site's naming conventions.

Here is the code for the sending client:

```
package javaee.architect;
import java.util.*;
import javax.naming.*;
import javax.jms.*;
public class PTPSend {
  private static final String THIS = "PTPSend";
  public static final String JMS_FACTORY = "myQueueConnectionFactory";
  public static final String QUEUE = "myQueue1";
  public static void main(String[] args) throws Exception {
    // get the initial context
    InitialContext ctx = new InitialContext(System.getProperties());
    // lookup the queue connection factory
    QueueConnectionFactory qconnf =
      (QueueConnectionFactory) ctx.lookup(JMS_FACTORY);
```

```
// create a queue connection
QueueConnection qconn = qconnf.createQueueConnection();
// create a queue session
QueueSession qsess = qconn.createQueueSession(
  false, Session.AUTO_ACKNOWLEDGE);
// lookup the queue object
Queue queue = (Queue) ctx.lookup(QUEUE);
// create a queue sender
QueueSender qsend = qsess.createSender(queue);
qsend.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
// start the connection
qconn.start();
// get the number of messages to send
int numMsgs = (args.length > 0) ? new Integer(args[0]).intValue() : 10;
// create messages
Message msg = qsess.createMessage();
TextMessage tmsg = qsess.createTextMessage();
ObjectMessage omsg = qsess.createObjectMessage();
Hashtable hTable = new Hashtable();
log("Started.");
// send the messages
for (int i=0,j=0,k=0; i < numMsgs; i++) {
  j++;k++;    //Increment counters
  if (k == 1) {
    tmsg.setText("News item #"+(i+1));
    // randomly set the Interesting property to true or false
    tmsg.setBooleanProperty("Interesting",((j==3)?true:false));
    log(tmsg.getText()+((j==3)?" (Interesting)":""));
    qsend.send(tmsg);
  } else {
    hTable.clear();
    hTable.put("symbol","ucny");
    hTable.put("bid",new String(""+(100+i)));
    hTable.put("ask",new String(""+(100+i+1)));
    omsg.setObject(hTable);
    // randomly set the Interesting property to true or false
    omsg.setBooleanProperty("Interesting",((j==3)?true:false));
    log(hTable+((j==3)?" (Interesting)":""));
    qsend.send(omsg);
    k=0; //reset counter
  }
  if (j==3) j=0;
}
// create a couple of close messages
tmsg.setText("***CLOSE***");
```

```
    // set the Interesting property to true on this one
    tmsg.setBooleanProperty("Interesting", true);
    log(tmsg.getText()+" (Interesting)");
    qsend.send(tmsg);
    // set the Interesting property to false on this one
    tmsg.setBooleanProperty("Interesting", false);
    log(tmsg.getText());
    qsend.send(tmsg);
    // close up
    qsend.close();
    qsess.close();
    qconn.close();
    ctx.close();
    log("Finished.");

  }
  private static void log(String msg) {
    System.out.println(new java.util.Date()+" "+THIS+" "+msg);
  }
}
```

Here is the code for the receiving client. Note that the receiving client has been coded by implementing a message listener:

```
package javaee.architect;
import javax.naming.*;
import javax.jms.*;
public class PTPReceive implements MessageListener {
  private static final String THIS = "PTPReceive";
  public static final String JMS_FACTORY = "myQueueConnectionFactory";
  public static final String QUEUE = "myQueue1";
  private InitialContext ctx;
  protected QueueConnectionFactory qconnf;
  protected QueueConnection qconn;
  protected QueueSession qsess;
  protected Queue queue;
  protected QueueReceiver qrecv;
  private boolean quit = false;
  public boolean ready;
  public static void main(String[] args) throws Exception {
    // instantiate the receiver, if there is no argument
    // pass a null to the constructor.
    PTPReceive rec = new PTPReceive(args.length>0?args[0]:null);
    // if constructor code does not initialize, we exit.
    if (!rec.ready) {
      log("Not ready to receive messages.");
```

```
    System.exit(-1);
  }
  log("Started"+(args.length>0?
    " with filter ("+args[0]+").":" with no filter."));
  // Start the thread.
  rec.run();
  // We're done, so clean up.
  rec.close();
  log("Finished.");
}
public PTPReceive(String filter) {
  try {
  // get the initial context
  ctx = new InitialContext(System.getProperties());
  // lookup the queue connection factory
  qconnf = (QueueConnectionFactory) ctx.lookup(JMS_FACTORY);
  // create a queue connection
  qconn = qconnf.createQueueConnection();
  // create a queue session
  qsess = qconn.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
  // lookup the queue object
  queue = (Queue) ctx.lookup(QUEUE);
  // create a queue receiver
  if (filter == null)
    // this is a queue receiver WITHOUT a filter
    qrecv = qsess.createReceiver(queue);
  else
    // this is a queue receiver WITH a filter
    qrecv = qsess.createReceiver(queue, filter);
  // set the message listener
  qrecv.setMessageListener (this);
  // start the connection
  qconn.start();
  ready = true;
  } catch (Exception e) {
    ready = false;
    log("Initialization failed. "+e);
    close();
  }
}
/*
 * The following loop suspends the main thread
 * until the quit boolean is set to true by the
 * onMessage method. While the main thread is
 * suspended the JMS provider calls the onMessage
```

```
 * method with its thread.
*/
public void run() {
  if (!ready) return;
  synchronized (this) {
    while (!quit) {
      try { wait(); }
      catch (InterruptedException ie) { }
    }
  }
}
public void onMessage (Message msg) {
  String text;
  try
  {
    if(msg instanceof ObjectMessage) {
      // we received an object message
         ObjectMessage objmsg = (ObjectMessage)msg;
         log(objmsg.getObject().toString());
    } else if(msg instanceof TextMessage) {
      // we received a text message
         text = ((TextMessage) msg).getText();
      log(text);
      if (text.equals("***CLOSE***")) {
        // we've received the close down message
        // so set the quit boolean to true and
        // wake up all threads that are waiting on
        // this object's monitor (i.e. the main thread)
        synchronized(this) {
          quit = true;
          this.notifyAll();
        }
        }
    } else log("message type not supported");
  } catch (JMSException e) {
    e.printStackTrace ();
  }
}
public void close () {
  try { if (qrecv != null) qrecv.close();
  } catch (Exception e) { log("Can't close queue receiver. "+e); }
  try { if (qsess != null) qsess.close();
  } catch (Exception e) { log("Can't close session. "+e); }
  try { if (qconn != null) qconn.close();
  } catch (Exception e) { log("Can't close connection. "+e); }
  try { if (ctx != null) ctx.close();
```

```
    } catch (Exception e) { log("Can't close context. "+e); }
  }
  private static void log(String msg) {
    System.out.println(new java.util.Date()+" "+THIS+" "+msg);
  }
}
```

When executed, the following output is from the sending client:

```
Sun Oct 01 17:28:48 EDT 2006 PTPSend Started.
Sun Oct 01 17:28:48 EDT 2006 PTPSend News item #1
Sun Oct 01 17:28:48 EDT 2006 PTPSend {bid=101, symbol=ucny, ask=102}
Sun Oct 01 17:28:48 EDT 2006 PTPSend News item #3 (Interesting)
Sun Oct 01 17:28:48 EDT 2006 PTPSend {bid=103, symbol=ucny, ask=104}
Sun Oct 01 17:28:48 EDT 2006 PTPSend News item #5
Sun Oct 01 17:28:48 EDT 2006 PTPSend {bid=105, symbol=ucny, ask=106} (Interesting)
Sun Oct 01 17:28:48 EDT 2006 PTPSend News item #7
Sun Oct 01 17:28:48 EDT 2006 PTPSend {bid=107, symbol=ucny, ask=108}
Sun Oct 01 17:28:48 EDT 2006 PTPSend News item #9 (Interesting)
Sun Oct 01 17:28:48 EDT 2006 PTPSend {bid=109, symbol=ucny, ask=110}
Sun Oct 01 17:28:48 EDT 2006 PTPSend ***CLOSE*** (Interesting)
Sun Oct 01 17:28:48 EDT 2006 PTPSend ***CLOSE***
Sun Oct 01 17:28:48 EDT 2006 PTPSend Finished.
```

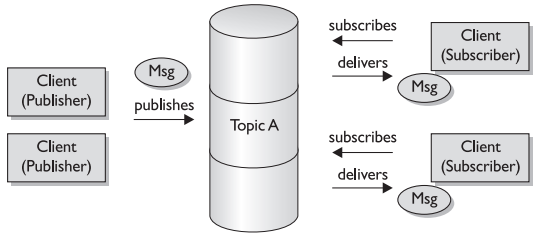
When executed, here is the output from the receiving client when it is executed with an argument `Interesting=true`, which becomes the message filter, and when it is executed with no argument, which means it must read all items in the queue:

```
Sun Oct 01 17:28:56 EDT 2006 PTPReceive Started with filter (Interesting=true).
Sun Oct 01 17:28:57 EDT 2006 PTPReceive News item #3
Sun Oct 01 17:28:57 EDT 2006 PTPReceive {ask=106, symbol=ucny, bid=105}
Sun Oct 01 17:28:57 EDT 2006 PTPReceive News item #9
Sun Oct 01 17:28:57 EDT 2006 PTPReceive ***CLOSE***
Sun Oct 01 17:28:57 EDT 2006 PTPReceive Finished.

Sun Oct 01 17:29:02 EDT 2006 PTPReceive Started with no filter.
Sun Oct 01 17:29:02 EDT 2006 PTPReceive News item #1
Sun Oct 01 17:29:02 EDT 2006 PTPReceive {ask=102, symbol=ucny, bid=101}
Sun Oct 01 17:29:02 EDT 2006 PTPReceive {ask=104, symbol=ucny, bid=103}
Sun Oct 01 17:29:02 EDT 2006 PTPReceive News item #5
Sun Oct 01 17:29:02 EDT 2006 PTPReceive News item #7
Sun Oct 01 17:29:02 EDT 2006 PTPReceive {ask=108, symbol=ucny, bid=107}
Sun Oct 01 17:29:02 EDT 2006 PTPReceive {ask=110, symbol=ucny, bid=109}
Sun Oct 01 17:29:02 EDT 2006 PTPReceive ***CLOSE***
Sun Oct 01 17:29:02 EDT 2006 PTPReceive Finished.
```

FIGURE 8-5

Publish/subscribe (pub/sub) model



How the Publish/Subscribe Message Model Works

The pub/sub message model allows an application to publish messages on a one-to-many or a many-to-many basis. Figure 8-5 is a diagram depicting the pub/sub model.

Following are some examples of a pub/sub implementation:

- Sending sales forecasts to various people in an organization
- Sending news items to interested parties
- Sending stock prices to traders on the trading floor

Messages are published to a *topic* (or subject). One or more publishers can publish messages to the same topic. Any client application that wants to receive messages on this topic must first subscribe to the topic. Multiple clients can subscribe to the topic and subsequently receive a copy of the message.

In the nondurable subscription model, the subscriber must be connected at the time a message is published to receive that message. If no subscribers are online, the messages will be published and destroyed soon thereafter. The subscriber can also use a durable subscription model, in which case the messages will be received when the subscriber is reconnected to the topic. Durable subscriptions come with greater overhead because they require additional resources to persist the messages until they can be delivered to all of the known durable subscribers.

The following is a list of the steps and interface classes required for the pub/sub model of communication:

1. Obtain the *TopicConnectionFactory* object via a JNDI lookup (the JNDI name will vary depending on the messaging vendor and site naming conventions).
2. Obtain a *TopicConnection* to the provider via the *TopicConnectionFactory*. (If security is enabled, pass a user ID and password to the `createTopicConnection` method.)

3. Obtain a *TopicSession* with the provider via the *TopicConnection*.
4. Obtain the topic via a JNDI lookup. (The JNDI name will vary depending on the messaging vendor and site naming conventions.)
5. Create either a *TopicPublisher* or a *TopicSubscriber* via the *TopicSession* for the required topic.
6. Publish and/or receive messages.
7. Close the *TopicPublisher* or *TopicSubscriber*, the session, and the connection.

In this example, the publishing client publishes text and object messages to the topic. The subscribing client receives the text and object messages from the topic. The text messages are displayed, and the object messages are executed. Note that the JNDI name for the connection factory and the queue will differ according to the messaging vendor and site naming convention.

Here is the code for the publishing client:

```
package javaee.architect;
import java.io.*;
import java.util.*;
import javax.naming.*;
import javax.jms.*;
public class PSPublish {
  private static final String THIS = "PSPublish";
  public static final String JMS_FACTORY = "myTopicConnectionFactory";
  public static final String TOPIC = "myTopic1";
  public static void main(String[] args) throws Exception {
    // get the initial context
    InitialContext ctx = new InitialContext(System.getProperties());
    // lookup the topic connection factory
    TopicConnectionFactory tconnf =
      (TopicConnectionFactory) ctx.lookup(JMS_FACTORY);
    // create a topic connection
    TopicConnection tconn = tconnf.createTopicConnection();
    //tconn.setClientID(THIS);
    // create a topic session
    TopicSession tsess = tconn.createTopicSession(false,
      Session.AUTO_ACKNOWLEDGE);
    // lookup the topic object
    Topic topic = (Topic) ctx.lookup(TOPIC);
    // create a topic publisher
    TopicPublisher tpublish = tsess.createPublisher(topic);
    tpublish.setDeliveryMode(DeliveryMode.PERSISTENT);
    // start the connection
```

```
    tconn.start();
    log("Started.");
    ObjectMessage omsg = tsess.createObjectMessage();
    TextMessage tmsg = tsess.createTextMessage();
    publishText(tpublish, tmsg, "Market open.");
    publishObject(tpublish, omsg, new PSOrder("BUY", "200", "UCNY"));
    publishObject(tpublish, omsg, new PSOrder("BUY", "100", "UCUS"));
    publishObject(tpublish, omsg, new PSOrder("SELL", "50", "UC"));
    publishText(tpublish, tmsg, "Market closed.");
    publishText(tpublish, tmsg, "After hours market open.");
    publishObject(tpublish, omsg, new PSOrder("SELL", "25", "UC"));
    publishObject(tpublish, omsg, new PSOrder("BUY", "150", "UCUS"));
    publishText(tpublish, tmsg, "After hours market closed.");
    publishText(tpublish, tmsg, "***CLOSE***");  // Close message
    // close up
    tpublish.close();
    tsess.close();
    tconn.close();
    ctx.close();
    log("Finished.");
  }
  public static void publishObject(TopicPublisher tpublish,
    ObjectMessage omsg, Serializable obj) {
    try {
      log(obj.toString());
      omsg.setObject(obj);
      tpublish.publish(omsg);
      sleep(1000);
    } catch (Exception e) { log("Can't publish message: " + e); }
  }
  public static void publishText(TopicPublisher tpublish,
    TextMessage tmsg, String s) {
    try {
      log(s.toString());
      tmsg.setText(s);
      tpublish.publish(tmsg);
      sleep(1000);
    } catch (Exception e) { log("Can't publish message: " + e); }
  }
  private static void log(String msg) {
    System.out.println(new java.util.Date()+" "+THIS+" "+msg);
  }
  private static void sleep(int m) {
    try { Thread.currentThread().sleep( m );
    } catch (Exception e) {}
  }
}
```

Here is the code for the object that is published:

```
package javaee.architect;
import java.io.Serializable;
// The object implements 'Runnable' so that the receiver
// can call the run() method. This is not a typical use
// of messaging nor of runnable objects.
public class PSOrder implements Runnable, Serializable {
  String side;
  String security;
  String quantity;
  /* constructor methods */
  public PSOrder() {
      setSide("<side not set>");
      setQuantity("<quantity not set>");
      setSecurity("<security not set>");
  }
  public PSOrder(String t, String q, String s) {
      setSide(t);
      setQuantity(q);
      setSecurity(s);
  }
  /* run method */
  public void run() {
    System.out.println(new java.util.Date()+" PSOrder "+this.toString());
  }
  /* toString method */
  public String toString() {
    return getSide()+" "+getQuantity()+" "+getSecurity();
  }
  /* get/set methods */
  public String getSide() {
    return side;
  }
  public void setSide(String t) {
    side = t;
  }
  public String getSecurity() {
    return security;
  }
  public void setSecurity(String s) {
    security = s;
  }
```

```
  public String getQuantity() {
    return quantity;
  }
  public void setQuantity(String s) {
    quantity = s;
  }
}
```

Here is the code for the subscribing client. Note that the subscribing client has been coded by implementing a message listener:

```
package javaee.architect;
import java.util.*;
import javax.naming.*;
import javax.jms.*;
public class PSSubscribe implements MessageListener {
  private static String THIS = "PSSubscribe";
  public static final String JMS_FACTORY = "myTopicConnectionFactory";
  public static final String TOPIC = "myTopic1";
  private InitialContext ctx;
  private TopicConnectionFactory tconnf;
  private TopicConnection tconn;
  private TopicSession tsess;
  private TopicSubscriber tsubscribe;
  private Topic topic;
  private boolean quit = false;
  public boolean ready;
  public static void main(String[] args) {
    PSSubscribe sub = new PSSubscribe(args.length>0?args[0]:"");
    if (!sub.ready) {
      log("Not ready to subscribe to messages.");
      System.exit(-1);
    }
    log("Started.");
    sub.run();
    sub.close();
    log("Finished.");
  }
  public PSSubscribe(String durable) {
    THIS = THIS+durable;
    try {
      ctx = new InitialContext(System.getProperties());
      tconnf = (TopicConnectionFactory) ctx.lookup(JMS_FACTORY);
      tconn = tconnf.createTopicConnection();
      tconn.setClientID(THIS);
```

```
    tsess = tconn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
    topic = (Topic) ctx.lookup(TOPIC);
    // Create a durable or non-durable based on argument
    if (durable.equalsIgnoreCase("durable"))
      tsubscribe = tsess.createDurableSubscriber(topic,
                     tconn.getClientID());
    else
      tsubscribe = tsess.createSubscriber(topic);
    tsubscribe.setMessageListener(this);
    // start the connection
    tconn.start();
    ready = true;
  } catch (Exception e) {
    ready = false;
    log("Initialization failed. "+e);
    close();
  }
}
public void run() {
  if (!ready) return;
  synchronized (this) {
    while (!quit) {
      try { wait(); }
      catch (InterruptedException ie) { }
    }
  }
}
public void onMessage(Message msg) {
  if (!ready) return;
  // Declare a reference for the 'Runnable' object messages
  Runnable obj = null;
  try {
    if (msg instanceof ObjectMessage) {
      try {
        obj = (Runnable) ((ObjectMessage) msg).getObject();
      } catch (Exception e) {
        log("Message is not an object!");
        return;
      }
      // The object messages implement 'Runnable'. This is not
      // a typical use of messaging nor of runnable objects.
      try { if (obj != null) { obj.run(); }
      } catch (Exception e) {
        log("Can't run the object.");
      }
```

```
      } else if (msg instanceof TextMessage) {
        String text = ((TextMessage) msg).getText();
        log(text);
        if (text.equals("***CLOSE***")) {
          synchronized (this) {
            quit = true;
            notifyAll();
          }
        }
      } else {
        log("Message must be ObjectMessage or TextMessage.");
        log(msg.toString());
      }
    } catch (Exception e) {
      log("Can't receive message:  " + e);
    }
  }
  public void close() {
    try {
      if (tsubscribe != null) tsubscribe.close();
      if (tsess != null) tsess.close();
      if (tconn != null) tconn.close();
      if (ctx != null) ctx.close();
    } catch (Exception e) {
      log("Can't close up. "+e);
    }
  }
  private static void log(String msg) {
    System.out.println(new java.util.Date()+" "+THIS+" "+msg);
  }
}
```

When executed, here is the output from the publishing client:

```
Sun Oct 01 17:25:25 EDT 2006 PSPublish Started.
Sun Oct 01 17:25:25 EDT 2006 PSPublish Market open.
Sun Oct 01 17:25:27 EDT 2006 PSPublish BUY 200 UCNY
Sun Oct 01 17:25:28 EDT 2006 PSPublish BUY 100 UCUS
Sun Oct 01 17:25:29 EDT 2006 PSPublish SELL 50 UC
Sun Oct 01 17:25:30 EDT 2006 PSPublish Market closed.
Sun Oct 01 17:25:31 EDT 2006 PSPublish After hours market open.
Sun Oct 01 17:25:32 EDT 2006 PSPublish SELL 25 UC
Sun Oct 01 17:25:34 EDT 2006 PSPublish BUY 150 UCUS
Sun Oct 01 17:25:35 EDT 2006 PSPublish After hours market closed.
Sun Oct 01 17:25:36 EDT 2006 PSPublish ***CLOSE***
Sun Oct 01 17:25:37 EDT 2006 PSPublish Finished.
```

When executed, here is the output from the subscribing client:

```
Sun Oct 01 17:25:18 EDT 2006 PSSubscribe Started.
Sun Oct 01 17:25:26 EDT 2006 PSSubscribe Market open.
Sun Oct 01 17:25:27 EDT 2006 PSOrder BUY 200 UCNY
Sun Oct 01 17:25:28 EDT 2006 PSOrder BUY 100 UCUS
Sun Oct 01 17:25:29 EDT 2006 PSOrder SELL 50 UC
Sun Oct 01 17:25:30 EDT 2006 PSSubscribe Market closed.
Sun Oct 01 17:25:31 EDT 2006 PSSubscribe After hours market open.
Sun Oct 01 17:25:33 EDT 2006 PSOrder SELL 25 UC
Sun Oct 01 17:25:34 EDT 2006 PSOrder BUY 150 UCUS
Sun Oct 01 17:25:35 EDT 2006 PSSubscribe After hours market closed.
Sun Oct 01 17:25:36 EDT 2006 PSSubscribe ***CLOSE***
Sun Oct 01 17:25:36 EDT 2006 PSSubscribe Finished.
```

Message-Driven Bean (MDB) Component

The message-driven bean (MDB) is a stateless component that is invoked by the EJB container when a JMS message arrives for the destination (topic or queue) for which the bean has registered. An MDB is a message consumer, and like other JMS message consumers, it receives messages from a destination because it implements the *javax.jms.MessageListener* interface. It is then able to perform business logic based on the message contents.

MDBs receive JMS messages from clients in the same manner as any other JMS servicing object. A client that writes to a destination has no knowledge of the fact that an MDB is acting as the message consumer. MDBs were created to have an EJB that can be asynchronously invoked to process messages, while receiving all of the same EJB container services that are provided to session and entity beans.

When a message is sent to a destination, the EJB container ensures that the MDB registered to process the destination exists. If the MDB needs to be instantiated, the container will do this. The `onMessage()` method of the bean is called, and the message is passed in as an argument. MDBs and stateless session EJBs are similar in the sense that they do not maintain state across invocations. MDBs differ from stateless session beans and entity beans in that they have no home or remote interface. Internal or external clients cannot directly access the MDBs methods. Clients can only indirectly interact with MDBs by sending a message to the destination.

The EJB *deployer* is the person responsible for assigning MDBs to a destination at deployment time. The EJB container provides the service of creating and removing MDB instances as necessary or as specified at deployment time.

EJB Container and Message-Driven Beans

The EJB container allows for the concurrent consumption of messages and provides support for distributed transactions. This means that database updates, message processing, and connections to Enterprise Information Systems using the Java EE Connector Architecture (JCA) can all participate in the same transaction context.

The EJB container or application server provides many services for MDBs, so the bean developer can concentrate on implementing business logic. Here are some of the services provided by the EJB container:

- Handles all communication for JMS messages
- Checks the pool of available bean instances to see which MDB is to be used
- Enables the propagation of a security context by associating the role specified in the deployment descriptor to the appropriate execution thread
- Creates and associates a transactional context if one is specified in the deployment descriptor
- Passes the message as an argument to the `onMessage()` method of the appropriate MDB instance
- Reallocates MDB resources to a pool of available instances

The EJB container also provides the following services based on the entries in the deployment descriptor file.

MDB Life cycle Management The life cycle of an MDB corresponds to the life span of the EJB container in which it is deployed. Since MDBs are stateless, bean instances are usually pooled by the EJB container and retrieved by the container when a message is written to the destination for which it is a message consumer.

The container creates a bean instance by invoking the `newInstance()` method of the bean instance class object. After the instance is created, the container creates an instance of *javax.ejb.MessageDrivenContext* and passes it to the bean instance via the `setMessageDrivenContext()` method. The `ejbCreate()` method is also called on the bean instance before it is placed in the pool and is then made available to process messages.

Exception Handling MDBs may not throw application exceptions while processing messages. This means that the only exceptions that may be thrown by a MDB are runtime exceptions indicating a system-level error. The container will

handle these exceptions by removing the bean instance and rolling back any transaction started by the bean instance or by the container.

Threading and Concurrency An MDB instance is assumed to execute in a single thread of control. The EJB container will guarantee this behavior. In addition, the EJB container may provide a mode of operation that allows multiple messages to be handled concurrently by separate bean instances. This deployment option utilizes expert level classes that are defined in the JMS specification. The JMS provider is not required to provide implementations for these classes, so the EJB container may not be able to take advantage of them with every JMS implementation. Using these classes involves a trade-off between performance and serialization of messages delivered to the server.

Message Acknowledgment The container always handles message acknowledgment for MDBs. It is prohibited for the bean to use any message acknowledgment methods—for example, `acknowledge()` or `rollback()`. The message acknowledgment can be set to either *AUTO_ACKNOWLEDGE*, allowing the message to be delivered once, or *DUPS_OK_ACKNOWLEDGE*, allowing the delivery of duplicate messages after a failure. Note that if a bean has the *Required* transaction attribute, it will process the `onMessage()` method inside a transaction.

Because the MDB has no client, no security principal is propagated to the EJB container on receipt of a message. The EJB framework provides facilities for a bean method to execute in a role specified in the deployment descriptor. As a result, the MDB can be configured to execute within a security context that can then be propagated to other EJBs that are called during the processing of a message.

Example MDB Code

In the following example code, the publishing client publishes simple messages to a topic. The subscribing MDB client receives the simple messages from the topic. Note that the JNDI name for the connection factory and the topic will differ per the messaging vendor and site naming convention.

Here is the code for the publishing client:

```
package javaee.architect;
import javax.naming.*;
import javax.jms.*;
public class PSMDBPublish {
  private static final String THIS = "PSMDBPublish";
  public static final String JMS_FACTORY = "myTopicConnectionFactory";
```

```
  public static final String TOPIC = "myTopic3";
  public static void main(String[] args) throws Exception {
    // get the initial context
    InitialContext ctx = new InitialContext(System.getProperties());
    // lookup the topic connection factory
    TopicConnectionFactory tconnf =
      (TopicConnectionFactory) ctx.lookup(JMS_FACTORY);
    // create a topic connection
    TopicConnection tconn = tconnf.createTopicConnection();
    // create a topic session
    TopicSession tsess = tconn.createTopicSession(false,
        Session.AUTO_ACKNOWLEDGE);
    // lookup the topic object
    Topic topic = (Topic) ctx.lookup(TOPIC);
    // create a topic publisher
    TopicPublisher tpublish = tsess.createPublisher(topic);
    tpublish.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
    // start the connection
    tconn.start();
    log("Started.");
    // create a simple message
    TextMessage tmsg = tsess.createTextMessage();
    // publish the messages
    tmsg.setText("Market open.");
    tpublish.publish(tmsg);
    log(tmsg.getText());
    tmsg.setText("Market closed.");
    tpublish.publish(tmsg);
    log(tmsg.getText());
    tmsg.setText("After hours market open.");
    tpublish.publish(tmsg);
    log(tmsg.getText());
    tmsg.setText("After hours market closed.");
    tpublish.publish(tmsg);
    log(tmsg.getText());
    // close up
    tpublish.close();
    tsess.close();
    tconn.close();
    ctx.close();
    log("Finished.");
  }
  private static void log(String msg) {
    System.out.println(new java.util.Date()+" "+THIS+" "+msg);
  }
}
```

Here is the code for the subscribing MDB client:

```
package javaee.architect;
import javax.ejb.*;
import javax.jms.*;
public class PSMDBSubscribe implements MessageDrivenBean, MessageListener {
  private static final String THIS = "PSMDBSubscribe";
  protected MessageDrivenContext ctx;
  // Associate bean instance with a particular context.
  public void setMessageDrivenContext(MessageDrivenContext ctx) {
    this.ctx = ctx;
  }
  // When the bean is initialized.
  public void ejbCreate() { log("ejbCreate()"); }
  // When the bean is destroyed.
  public void ejbRemove() { log("ejbRemove()"); }
  // main business method.
  public void onMessage(Message msg) {
    try {
      // This class processes TextMessages.
      if (msg instanceof TextMessage) {
        log(((TextMessage) msg).getText());
      }
    } catch (Exception e) { log("Can't receive message:  " + e); }
  }
  private void log(String msg) {
    System.out.println(new java.util.Date()+" "+THIS+" "+msg);
  }
}
```

Here is the deployment descriptor for the subscribing MDB client:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
 <enterprise-beans>
  <message-driven>
   <ejb-name>PSMDB</ejb-name>
   <ejb-class>javaee.architect.PSMDBSubscribe</ejb-class>
   <transaction-type>Container</transaction-type>
   <message-driven-destination>
    <destination-type>javax.jms.Topic</destination-type>
   </message-driven-destination>
  </message-driven>
 </enterprise-beans>
</ejb-jar>
```

Here is the WebLogic 8.1 deployment descriptor for the subscribing MDB client:

```
<?xml version="1.0"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC "-//BEA Systems, Inc.//DTD WebLogic 8.1.0
EJB//EN" "http://www.bea.com/servers/wls810/dtd/weblogic-ejb-jar.dtd">
<weblogic-ejb-jar>
 <weblogic-enterprise-bean>
  <ejb-name>PSMDB</ejb-name>
  <message-driven-descriptor>
   <pool>
    <max-beans-in-free-pool>10</max-beans-in-free-pool>
    <initial-beans-in-free-pool>2</initial-beans-in-free-pool>
   </pool>
   <destination-jndi-name>myTopic3</destination-jndi-name>
  </message-driven-descriptor>
  <jndi-name>PSMDB</jndi-name>
 </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

When executed, here is the output from the publishing client:

```
Sun Oct 01 17:19:23 EDT 2006 PSMDBPublish Started.
Sun Oct 01 17:19:23 EDT 2006 PSMDBPublish Market open.
Sun Oct 01 17:19:23 EDT 2006 PSMDBPublish Market closed.
Sun Oct 01 17:19:23 EDT 2006 PSMDBPublish After hours market open.
Sun Oct 01 17:19:23 EDT 2006 PSMDBPublish After hours market closed.
Sun Oct 01 17:19:23 EDT 2006 PSMDBPublish Finished.
```

When executed, here is the output from the subscribing MDB client:

```
Sun Oct 01 17:19:23 EDT 2006 PSMDBSubscribe Market open.
Sun Oct 01 17:19:23 EDT 2006 PSMDBSubscribe Market closed.
Sun Oct 01 17:19:23 EDT 2006 PSMDBSubscribe After hours market open.
Sun Oct 01 17:19:23 EDT 2006 PSMDBSubscribe After hours market closed.
```

EJB 2.1 Message Driven Beans

Everything described to this point for message-driven beans (MDB) is prior to the EJB 2.1 specification. With the EJB 2.1 specification, MDBs are no longer restricted to simply supporting JMS messages. In fact, they can be defined to handle any kind of messaging system from any vendor. As such, an MDB can now implement any interface, with the only requirements of the EJB 2.1 vendors being that new types of MDBs implement the *javax.ejb.MessageDrivenBean* interface and adhere to the message-driven bean's life cycle. In addition, EJB 2.1 vendors must also support any MDB type that is based on the Java EE Connector Architecture (JCA) 1.5 (See Chapter 6 for more information on JCA).

This flexibility means that an MDB is no longer limited to being passed a JMS message. The `onMessage` method now receives a Record object, which also means that something other than a JMS message arriving at a destination can trigger the invocation of the method. So, conceivably an MDB can now be driven as the result of a resource adapter receiving a prompt from a back-end system or an internal event—it could even be driven by a timer event.

So the first difference for non-JMS-driven MDBs lies in defining the type of messaging server interface being implemented. For example, in order to listen to JAXM messages, the MDB must implement `javax.xml.messaging.OneWayListener` or `javax.xml.messaging.ReqRespListener`. To listen to messages from the JCA Common Client Interface (CCI) connector, the MDB must implement `javax.resource.cci.MessageListener`, which is shown here:

```
public interface MessageListener {
    Record onMessage(Record inputData)
           throws ResourceException;
}
```

The second difference is in the configuration of the MDB. The EJB container needs to know which destination or endpoint to which it must connect. This is done via new tags within the deployment descriptor. The `<messaging-type>` tag defines the interface being implemented. The other configuration properties are defined with the `<activation-config>` tag, which contains arbitrary name/value pairs for properties that are specific to the messaging service being used. Here is an example deployment descriptor for an MDB using a JCA CCI connector:

```
<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>SCEA MDB</ejb-name>
      <ejb-class>com.ucny.sceaMdb </ejb-class>
      <messaging-type>com.ucny.SCEA_JCAListener</messaging-type>
      <transaction-type>Bean</transaction-type>
      <activation-config>
        <activation-config-property>
          <activation-config-property-name>
            HostID
          </activation-config-property-name>
          <activation-config-property-value>
            ZOSCICS07
          </activation-config-property-value>
        </activation-config-property>
        <activation-config-property>
```

```
            <activation-config-property-name>
              UserID
            </activation-config-property-name>
            <activation-config-property-value>
              LCPRA
            </activation-config-property-value>
          </activation-config-property>
        </activation-config>
      </message-driven>
    </enterprise-beans>
  </ejb-jar>
```

CERTIFICATION OBJECTIVE 8.05

Identify Scenarios That Are Appropriate to Implementation Using Messaging, Enterprise JavaBeans Technology, or Both

The following table shows messaging and EJB implementations that can be used as solutions for the given scenarios.

SCENARIO & SOLUTION	
You need to perform a transaction that is distributed across multiple applications and systems; which technology is most appropriate for maintaining this type of distributed transaction?	The EJB container provides support for database updates, message processing, and connections to EIS systems using the Java EE Connector Architecture (JCA). This will allow all to participate in the same transaction context. Messaging by itself is not a complete solution for this scenario.
You need to broadcast stock prices to applications executing on a trader's desktop…	A publish/subscribe messaging solution will be sufficient.
You need to send an order request to another system…	Possibly use a combination of EJB for retrieving order data and messaging for sending the data to the other system.
What technology is appropriate for easier integration of incompatible systems?	Use a messaging solution to provide the interface between systems that are not able to communicate directly.

CERTIFICATION SUMMARY

JMS provides a highly flexible and scalable solution for building loosely coupled applications in the enterprise environment. It brings all of the advantages of a messaging-based application into the Java language. JMS links messaging systems with all the benefits of Java technology for rapid application deployment and application maintenance.

This chapter should give you an understanding of the JMS and messaging in general and the appropriate scenarios for using messaging-in applications.

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in Chapter 8.

Identify Scenarios That Are Appropriate to Implementation Using Messaging

- ❑ Scenarios appropriate to implementation using message include asynchronous communication, one-to-many communication, guaranteed messaging, and transactional messaging.

List Benefits of Synchronous and Asynchronous Messaging

- ❑ Some benefits to synchronous messaging are that both parties must be active to participate and the message must be acknowledged before proceeding to the next.
- ❑ Asynchronous messaging benefits are that as the volume of traffic increases, more bandwidth or additional hardware is not required; it is less affected by failures at the hardware, software, and network levels; and when capacities are exceeded, information is not lost but is instead only delayed.

Identify Scenarios That Are More Appropriate to Implementation Using Asynchronous Messaging, Rather Than Synchronous

- ❑ Scenarios more appropriate to asynchronous messaging include those in which a response is not required or not immediately required.
- ❑ Asynchronous processing is also more appropriate for high-volume transaction processing.

Identify Scenarios That Are More Appropriate to Implementation Using Synchronous Messaging, Rather Than Asynchronous

- ❑ One scenario more appropriate to synchronous messaging includes that in which a response to the message is required before continuing, for example, for transactions requiring credit card or user login authentication.
- ❑ A second scenario includes a transaction where both parties must be active participants.

Identify Scenarios That Are Appropriate to Implementation Using Messaging, Enterprise JavaBeans Technology, or Both

- ❑ The scenarios appropriate for messaging technology include broadcasting stock prices to traders, instant messages, and in situations when integration of incompatible systems is necessary.
- ❑ The scenarios appropriate for EJB technology include those that perform business logic and those that maintain persistent data.
- ❑ The scenarios appropriate for messaging and EJB technology including those that require maintenance of distributed transactions and those that send an order to another system.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there may be more than one correct answer. Choose all correct answers for each question.

Identify Scenarios That Are Appropriate to Implementation Using Messaging

1. Which of the following are characteristics of publish/subscribe message model?
   A. Always use a URL to identify publishers.
   B. Subject-based addressing.
   C. Location-independent publishers.
   D. Only synchronous communication between publishers and subscribers is possible.
2. Which of the following are valid methods for a *TopicSubscriber*?
   A. `receive()`
   B. `receiveNoWait()`
   C. `receiveWait()`
   D. `receiveSync()`
3. What are the types of messaging models supported in JMS?
   A. Point-to-point
   B. Send/receive
   C. Transmit/receive
   D. Publish/subscribe
4. What is a message digest?
   A. A digital fingerprint value that is computed from a message, file, or byte stream
   B. A shortened summary of a message
   C. The subject line of a message
   D. A processing function of the mail server
5. Which of the following scenarios are suitable for publish/subscribe messaging model?
   A. It is used to receive news stories.
   B. It is used for receiving sales forecasts.
   C. It is used for sending stock prices to traders on the trading floor.
   D. It is used to authorize a user ID and password.

6. What deliver modes are available in JMS?
   A. *PERSISTENT*
   B. *NON_PERSISTENT*
   C. *PERMANENT*
   D. *DURABLE*
7. Which of the following are valid message acknowledgment types?
   A. *AUTO_ACKNOWLEDGE*
   B. *CLIENT_ACKNOWLEDGE*
   C. *DUPS_OK_ACKNOWLEDGE*
   D. *NO_ACKNOWLEDGE*
8. Which of the following are *not* valid message body formats?
   A. *MapMessage*
   B. *ObjectMessage*
   C. *TextMessage*
   D. *StringMessage*
9. Which of the following are *not* valid JMS objects?
   A. *MessageProducer*
   B. *MessageConsumer*
   C. *MessageViewer*
   D. *MessageSelector*
10. Which of the following would *not* be used in a client application performing point-to-point messaging?
   A. Topic
   B. *InitialContext*
   C. Queue
   D. Session

List Benefits of Synchronous and Asynchronous Messaging

11. Which of the following are advantages of asynchronous messaging architectures?
   A. Better use of bandwidth
   B. Supports load balancing
   C. Provides sender with instant response
   D. Scalability

12. Which of the following statements are true for asynchronous messaging?
- A. It decouples senders and receivers.
- B. It can increase performance.
- C. It is better suited to smaller message sizes.
- D. It only works with blocking calls.

Identify Scenarios That Are More Appropriate to Implementation Using Asynchronous Messaging, Rather Than Synchronous

13. Which method must be called to receive messages asynchronously?
- A. The `receive` method
- B. The `processMessage` method
- C. The `readMessage` method
- D. The `onMessage` method

14. Which of the following are *not* features of asynchronous messaging?
- A. As the volume of traffic increases, it is better able to handle the spike in demand.
- B. A message must be acknowledged before the producer can send another.
- C. It is less affected by failures at the hardware, software, and network levels.
- D. When capacities are exceeded, information is not lost; instead, it is delayed.

Identify Scenarios That Are More Appropriate to Implementation Using Synchronous Messaging, Rather Than Asynchronous

15. Which method must be called to receive messages synchronously?
- A. The `receive` method
- B. The `processMessage` method
- C. The `readMessage` method
- D. The `onMessage` method

16. Which of the following cases are better suited to synchronous messaging?
- A. Electronic mail
- B. Credit card authorization
- C. Electronic processing of tax returns
- D. Validation of data entered

17. Which of the following are features of synchronous messaging?

A. Both parties must be active to participate.
B. Messages must be acknowledged before proceeding.
C. It decouples senders and receivers.
D. It does not work with blocking calls.

18. Which of the following are *not* features of synchronous messaging?

A. Both parties must be active to participate.
B. It is unaffected by increases in traffic volume.
C. Message must be acknowledged before proceeding to the next.
D. Message is queued until it is ready for processing.

Identify Scenarios That Are Appropriate to Implementation Using Messaging, Enterprise JavaBeans Technology, or Both

19. Which of the following scenarios are not suitable for publish/subscribe messaging model?

A. Sending an instant message
B. Sending an order to another system
C. Sending news stories to interested parties
D. Sending a transaction to another system

20. What method must be implemented to receive messages in a message-driven bean (MDB)?

A. The `receive` method
B. The `onMessage` method
C. The `readMessage` method
D. The `processMessage` method

SELF TEST ANSWERS

Identify Scenarios That Are Appropriate to Implementation Using Messaging

1. ☑ **B** and **C** are correct. Publish/subscribe messages use subject-based addressing and provide location-independence for publishers.
   ☒ **A** and **D** are incorrect. URLs are not used to identify publishers. Publish/subscribe is not limited to synchronous communication.
2. ☑ **A** and **B** are correct. `receive()` and `receiveNoWait()` are valid methods for *TopicSubscriber*.
   ☒ **C** and **D** are incorrect. `receiveWait()` and `receiveSync()` are not valid methods.
3. ☑ **A** and **D** are correct. Point-to-point and publish/subscribe are the messaging models supported in JMS.
   ☒ **B** and **C** are incorrect. Send/receive and transmit/receive are not valid messaging models.
4. ☑ **A** is correct. A message digest is a digital fingerprint value that is computed from a message, file, or byte stream.
   ☒ **B, C,** and **D** are incorrect. These are not definitions of a message digest.
5. ☑ **D** is correct. Authorizing user IDs and passwords must use a synchronous process.
   ☒ **A, B,** and **C** are incorrect. Receiving news stories, sales forecasts, and sending stock prices are suitable for asynchronous messaging.
6. ☑ **A** and **B** are correct. *PERSISTENT* and *NON_PERSISTENT* are valid delivery modes.
   ☒ **C** and **D** are incorrect. *PERMANENT* and *DURABLE* are invalid delivery modes.
7. ☑ **A, B,** and **C** are correct. *AUTO_ACKNOWLEDGE*, *CLIENT_ACKNOWLEDGE*, and *DUPS_OK_ACKNOWLEDGE*, are valid.
   ☒ **D** is incorrect. *NO_ACKNOWLEDGE* is an invalid message acknowledgment type.
8. ☑ **D** is correct. *StringMessage* is not a valid message body format.
   ☒ **A, B,** and **C** are incorrect. *MapMessage*, *ObjectMessage*, *TextMessage* are valid.
9. ☑ **C** is correct. *MessageViewer* is not a valid JMS object.
   ☒ **A, B,** and **D** are incorrect. *MessageProducer*, *MessageConsumer*, *MessageSelector* are valid.
10. ☑ **A** is correct. Topics are used in publish/subscribe messaging.
    ☒ **B, C,** and **D** are incorrect. These are valid classes in point-to-point messaging.

List Benefits of Synchronous and Asynchronous Messaging

11. ☑ **A, B,** and **D** are correct. Asynchronous architectures make better use of bandwidth, support leveling of workloads, and are more scalable.
☒ **C** is incorrect. These architectures do not provide senders with instant response.

12. ☑ **A, B,** and **C** are correct. Asynchronous messaging decouples senders and receivers, can increase performance, and is better suited to smaller message sizes.
☒ **D** is incorrect. Asynchronous messaging does not work with blocking calls.

Identify Scenarios That Are More Appropriate to Implementation Using Asynchronous Messaging, Rather Than Synchronous

13. ☑ **D** is correct. The `onMessage` method must be implemented to receive messages asynchronously.
☒ **A, B,** and **C** are incorrect. The `processMessage` and `readMessage` methods do not exist. The `receive` method is used for synchronous messaging.

14. ☑ **B** is correct. A message is not acknowledged before a producer can send another in asynchronous messaging.
☒ **A, C,** and **D** are incorrect. These are valid features.

Identify Scenarios That Are More Appropriate to Implementation Using Synchronous Messaging, Rather Than Asynchronous

15. ☑ **A** is correct. The `receive` method must be implemented to receive messages synchronously.
☒ **B, C,** and **D** are incorrect. The `processMessage` and `readMessage` methods do not exist. The `onMessage` method is used for asynchronous messaging.

16. ☑ **B** and **D** are correct. Credit card authorization and validation of data entered are better suited to synchronous messaging because of the need for an instant response.
☒ **A** and **C** are incorrect. Electronic mail and electronic processing of a tax return do not need instant responses.

17. ☑ **A** and **B** are correct. Both parties must be active to participate, and messages must be acknowledged before proceeding.
☒ **C** and **D** are incorrect. Synchronous messaging does not decouple senders and receivers and only works with blocking calls.

18. ☑ **A** and **C** are correct. Synchronous messaging is affected by volume increase, and synchronous messages are not queued.
☒ **B** and **D** are incorrect. These are not valid features for synchronous messaging.

Identify Scenarios That Are Appropriate to Implementation Using Messaging, Enterprise JavaBeans Technology, or Both

19. ☑ **A, B,** and **D** are correct. Sending an instant message, an order to another system, or a transaction to another system is not suitable for the publish/subscribe message model.
☒ **C** is incorrect. Sending news stories to interested parties is suitable for the publish/subscribe message model.

20. ☑ **B** is correct. The `onMessage()` method is the correct method.
☒ **A, C,** and **D** are incorrect. These are incorrect methods to receive messages in a message-driven bean (MDB).