5

Concepts and Fundamentals of JMS Programming

In this chapter, you will learn the fundamentals and basic elements of Java Message Service (JMS) programming. In the next chapter, you will write three simple programs to help you understand how to develop a JMS application.

What Is a Messaging System or Service?

Messaging is a way or a mechanism that provides communication between software applications, programs, or objects on a distributed system. Remote Method Invocation (RMI) and socket programming are also types of messaging according to this definition. But, the focus of this book is on a message-based messaging system. As a simple definition, a message identifies the content transmitted between two or more applications or programs. One or more programs send a message, and the other one or more programs receive the message. You might think that a query from a SQL-based database using a graphical user interface (GUI) is a message. It is direct, one-to-one messaging, but a messaging system is more sophisticated than this simple example. It is more like using TCP/IP packets on a computer network. In a messaging system, there are clients that can send and receive messages. Each client connects to the messaging system, which provides a platform to create, send, and receive messages. A messaging system has three major features:

- A messaging system is loosely coupled. This is the most important feature of a messaging system and might be an advantage compared to other systems such as RMI. An application or program, called a *sender* or *publisher*, sends a message to a destination, not directly to another client. Another application or program, called a *receiver* or *subscriber*, receives the message from a destination. Senders and receivers do not have to be aware of each other.
- A messaging system isolates clients from each other. Neither sender nor receiver needs to know about each other. They only need to know the message format and destination.
- A messaging system allows decoupling. A sender and receiver use the system at different times. They do not have to be up and running at the same time. A sender sends the message to a destination, and the receiver takes the messages whenever it is ready. A sender does not need to wait for a response. It can process another task without being blocked. I refer to this feature as *asynchronous* messaging in the remainder of the book, which generally means that clients are able to use the system at different times, and they do not have to know whether other clients in the system are up and running.

Some developers consider email a part of a messaging system. Although email is a way of communication between people, and sometimes between people and software applications, a messaging system is different. It is used for communication between software applications or objects.

A messaging system is based on message-oriented middleware (MOM), which was explained in the previous chapter. MOM defines the rule of messaging as:

- How the message looks
- How a sender application sends the message
- How a receiver application receives the message
- How a receiver application reads the message

Advantages and Disadvantages of a Messaging System

In the messaging service (or system), there is a server, and clients connect to this server to communicate with each other (see Figure 5.1).

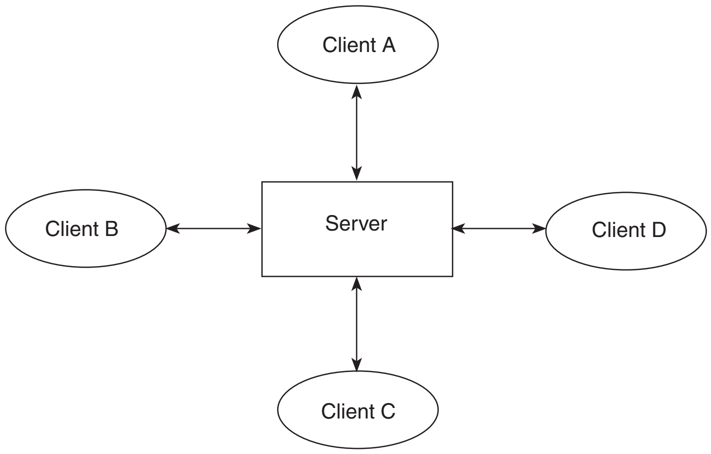

Figure 5.1 The messaging service architecture.

The server provides some essential services such as message persistence, load balancing, and security. The server provides asynchronous communication and guaranteed delivery from senders to receivers.

A messaging service is a great way for applications to communicate with each other, but it has some disadvantages:

- You have to send header and other information with the message content. Therefore, the total amount of information is larger than the message content itself, which might increase network traffic.
- Every message goes to the receivers through a server, which makes communication slower than a direct connection.
- Your messaging service provider (vendor) might not support all the JMS specifications defined by Sun Microsystems.

Before designing your JMS projects, you should compare the disadvantages such as network traffic, slower communication, and vendor-specific issues with the advantages such as loosely coupled or decoupled systems, portability, persistent messaging, and guaranteed delivery.

Briefly, if you only need to insert some new records into a local database and you have a very reliable network, you might not need to use a messaging system. Keep in mind that a messaging-based application consumes additional resources.

However, if you want to isolate networking problems in your client source code or if the database you want to access is in a different location, you might need to use a messaging service. A messaging service can ensure completion of transactions of an application properly such as manipulating data in a table of the database (for example, Insert, Delete, and Update statements), particularly if the database is unavailable (such as when using a laptop that is disconnected from the network). It will process the command (the message content) at a later time. Decoupling or loosely coupling is the best feature of a messaging service.

What Is the JMS API?

The JMS is a Java Application Programming Interface (API), which allows software applications, components, and objects to create, send, receive, and read messages. Sun Microsystems is a JMS vendor that markets an iPlanet product, but Sun designed and developed JMS specifications in collaboration with JMS vendors, not by itself. Sun also provides developers with reference implementations to test and apply specifications to your projects. In this book, I will use the JMS API Reference Implementation bundled with the Java 2 Platform Enterprise Edition (J2EE) version 1.3 or later to test sample applications instead of commercial JMS products. If you need more information about JMS products, refer to Appendix C, "Java Message Service (JMS) API Vendors." The JMS API enables communication that:

- Is loosely coupled
- Is asynchronous, which means that a JMS server delivers a message to the client, but the client does not have to read immediately
- Is reliable, which means that a JMS server ensures that a message is delivered once and only once

Point-to-Point and Publish-and-Subscribe Messaging

As mentioned in the previous chapter, there are two major messaging types: point-to-point (p2p or PTP) and publish-and-subscribe (pub/sub). They are the fundamentals of MOM and are supported by JMS specifications. (JMS vendors are not required to support both types of messaging, although many of them do.)

Recall that in p2p messaging, the domain (or destination) is called a queue (see Figure 5.2). The sender sends the message to the queue, and the receiver (recipient) takes (or reads) the message from the queue whenever it is ready. Although this seems like peer-to-peer, there can be two or more senders for the same queue.

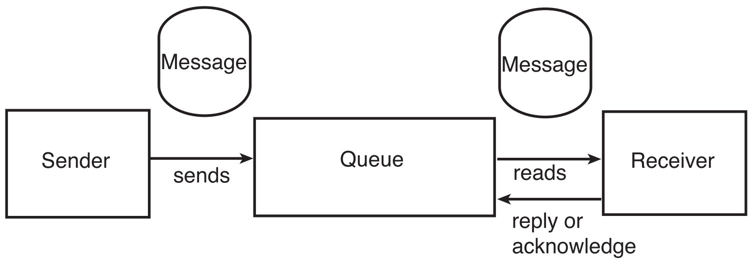

Figure 5.2 The p2p messaging service.

This queue is stored in the messaging server or in a relational database if data persistence is required. JMS does not ban using direct messaging, but it uses a queue for p2p messaging.

A human resource application that sends a message to the accounting application about annual salary increases for workers in a factory plant in Wisconsin is an example of p2p messaging.

In the pub/sub messaging type, a messaging domain (destination) is called a topic, a sender is called a publisher, and a receiver is called a subscriber (see Figure 5.3). Publishers send the message to a topic. Subscribers receive all of the messages sent to that topic as long as they subscribe to the topic. In this model, there are one or more publishers and receivers. If one publisher sends a message to the topic, all subscribers receive a copy of the same message. You might need to use this messaging model to notify a group of applications using the same message. An example of the pub/sub messaging model is when a production application sends a message to a *NewProduct* topic, and subscribers to the *NewProduct* topic, such as a sales application and a marketing application, receive this message.

This model supports multiple senders and receivers, and applications do not need to act together. Senders called publishers send (publish) their messages at different times, independently from other senders to the topic. Receivers called subscribers also read (subscribe) the messages from the topic, independently from other receivers.

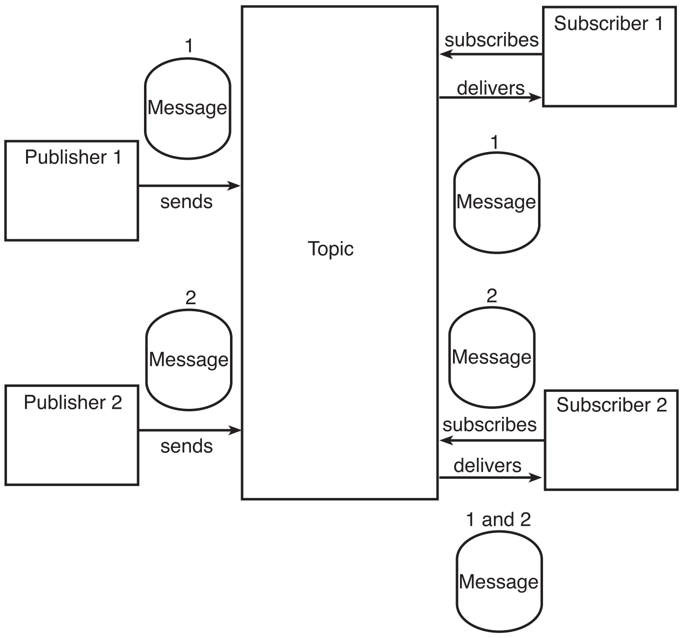

Figure 5.3 The pub/sub messaging service.

The JMS API and the J2EE Platform

Sun first released the JMS API in 1998. Its main purpose was to allow Java applications to work with MOM-based products. Because it was found very useful, many MOM vendors adopted and implemented the JMS API. In version 1.2 of the J2EE platform, vendors were not required to implement the JMS API. It was an add-on product, and vendors had to provide a JMS API interface. With version 1.3 of the J2EE platform, the JMS API is an integrated part of the platform. J2EE certified vendors, including Sun Microsystem's own application server product, "iPlanet," must support the JMS API. The JMS API in the J2EE platform version 1.3 has some valuable features:

- Enterprise JavaBeans (EJBs) or an enterprise Web component can create, send, and synchronously receive a message.
- Message-driven beans, which are new enterprise beans included with version 1.3 of the J2EE platform, allow asynchronous messaging.
- Messages that are sent and received can participate in Java Transaction API (JTA) transactions.

Additionally, EJB container architecture provides support for distributed transactions and allows for the concurrent consumption of messages.

The JMS API makes developing enterprise applications easier for developers and allows loosely coupled, synchronous and asynchronous, reliable communications and interactions between J2EE components and other applications capable of messaging. You can develop enterprise applications with new message-driven beans for specific business events in addition to the existing business events.

Another technology, the J2EE Connector, exists within the J2EE platform version 1.3, and it provides tight integration between Java enterprise applications and enterprise information systems (EIS). The JMS API is different from connector technology because it provides loosely coupled interaction between J2EE applications and database servers or information application servers (IAS).

Concepts of JMS Programming

In this section, I discuss some basic elements of JMS programming before providing you with some simple JMS examples and advanced applications.

JMS Architecture

Figure 5.4 shows the five main elements of the JMS architecture.

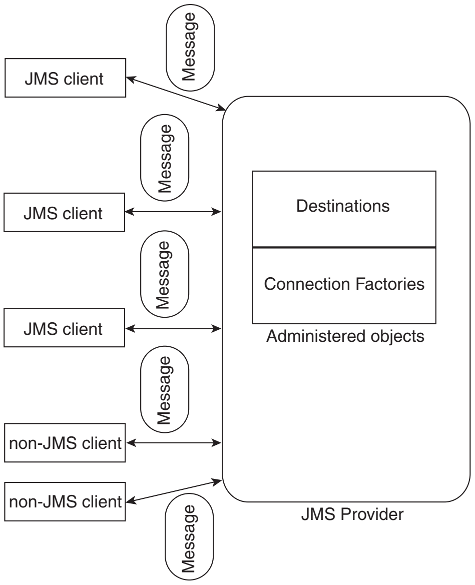

Figure 5.4 The five main elements of the JMS architecture.

Some brief information about each element follows, and more detailed information is provided in the remainder of the book.

- JMS provider—The JMS provider is like a container for the messaging system. It implements JMS interfaces, which are defined by the JMS specifications. It also provides some administrative features as well as some additional components, which are not required by the JMS specification, but are MOM-based technologies.
- JMS clients—The JMS clients are applications, components, or objects that produce and consume messages. They must be written in the Java language in order to be a JMS client.
- Non-JMS clients—Non-JMS clients are applications that use native client APIs instead of the JMS API.
- Administered objects—Administered objects are for client use, but are created by an administrator. There are two main administered objects—*destination* and *connection factories*—which are examined in subsequent sections.
- Message—The message is the heart of the messaging system. It is the object that provides information transfer between messaging system clients. If you do not have a message, there is no need to use a messaging system.

Message Consumption

One of the important concepts in the messaging system is message consumption, which is defined by timing properties of the system. Consuming a message means receiving the message, reading the message, or taking the message from the destination (queue or topic). A message is produced by a messaging client (sender or publisher), but the producing client is not interested in how the message will be consumed. This process is part of the receiving side, which is the target of the message.

The JMS specification defines two ways to consume a message:

- Synchronously—A receiver reads or takes the message from the destination by calling the `receive()` method. This `receive()` method blocks the application until a message arrives (blocking means that the receiver application waits for a message to arrive and does not perform any other transactions during this time). As a developer, you can specify a time limit to receive a message. If the message does not arrive within a specified time limit, it can time out, which releases the block.

- Asynchronously—A client can define a message listener (a mechanism similar to an event listener). Whenever a message arrives at its destination, a messaging server (JMS provider) delivers the message to the recipient (subscriber) by calling the `onMessage()` method. The client will not be blocked while waiting for the message.

Destinations

As one of two administered objects, a destination is the target of a message. The destination is where the message will be delivered. It is provider-independent. In the JMS specification, the Destination interface does not define a specific method. It is an administrative object, and its physical location on the server is chosen and handled by the provider. There are two types of destinations in the JMS specifications: queue and topic.

Recall from a previous section that the p2p messaging model uses a queue destination, and the pub/sub model uses a topic destination. The most important aspect and advantage of a destination is that its implementation is defined by the JMS provider. A sender sends the message to the destination by using the Destination interface, and the recipient receives the message from the Destination object. A recipient does not see the detail of the implementation. The messaging server, which in this case is the JMS provider, performs the implementation of the Destination object.

A destination is an administered object, and you can create it by using the Administrator tool in your application server, which is also included in the JMS server. Because J2EE Reference Implementation version 1.3 is used as the application server in this book, I will not use any vendor-specific features for this server. You will have to modify some commands or steps depending on your server.

If you want to create a queue for a p2p messaging model, you can type j2eeadmin at the command line as follows:

```
j2eeadmin -addJmsDestination <jndi_name_for_queue> queue
```

As an example, if you want to create a queue named levent_Boston, you can type it at the command line like this:

```
j2eeadmin -addJmsDestination levent_Boston queue
```

In JMS application client code, you usually need to look up a destination after you look up a connection factory. When you look up the levent_Boston queue, the line might look like the following in the code for lookup:

```
Queue erdQueue = (Queue) ctx.lookup("levent_Boston");
```

If you want to create a topic for a pub/sub messaging model, you can type j2eeadmin at the command line as follows:

```
j2eeadmin -addJmsDestination <jndi_name_for_topic> topic
```

As an example, if you want to create a topic named levent_Europe, you can type it at the command line like this:

```
j2eeadmin -addJmsDestination levent_Europe topic
```

In JMS application client code, you usually need to look up a destination after you look up a connection factory. When you look up the levent_Europe topic, the line might look like the following in the code for lookup:

```
Topic erdTopic = (Topic) ctx.lookup("levent_Europe");
```

A JMS application can use multiple queues and topics depending on your projects.

In addition to a permanent queue and topic, which are created by JMS administrators for the use of JMS clients, there are temporary queues and topics. These queues and topics are dynamic and are only created for the lifetime of the session. These temporary destinations:

- Are only used when a response is expected. A response can be specified by a `JMSReplyTo` message header when a message is sent.
- Are created by the current destination (queue or topic) and can only be accessed by this session and all other sessions that belong to the same connection.
- Are deleted automatically when the session is closed.

Connection Factory

A connection factory is another administered object. It is used to create a connection to the JMS provider by the client. It is similar to DriverManager in the Java Database Connectivity (JDBC) API, which hides the JDBC driver detail from the programmer. It encapsulates a series of connection configuration parameters and information. The host, which the JMS provider is running, or the port, which the JMS provider is listening to, is an example of the information that will be put into a connection factory. These types of configurations are the JMS administrator's responsibility, but they are needed by the client to create a proper connection to the JMS server.

A connection factory is defined in the JMS specification as an interface (javax.jms.ConnectionFactory) without a method. This is the root interface. In the client applications, you use two subtype interfaces—javax.jms. QueueConnectionFactory and javax.jms.TopicConnectionFactory—

depending on the messaging model (destination) used in your project. These subtype interfaces define the methods to create a connection to the server.

You can use two default connections, QueueConnectionFactory and TopicConnectionFactory, to create connections with the J2EE Reference Implementation version 1.3 if the JMS administrator did not create a connection factory. You can also create new connection factories as a JMS administrator.

The p2p Messaging Model for the Connection Factory

In the p2p messaging model, you can create a connection factory at the command line like this by typing the following:

```
j2eeadmin -addJmsFactory <jndi_name_for_conn_factory> queue
```

For example, if you name the connection factory conFactory_Montreal, the command will look like this:

```
j2eeadmin -addJmsFactory conFactory_Montreal queue
```

At the beginning of the JMS client application, you have to look up the connection factory after calling `InitialContext()`. For p2p messaging, the lines that contain calling the initial context and looking up the connection factory will look like this:

```
Context myContext = new InitialContext();
QueueConnectionFactory myQueueConnectionFactory =
   (QueueConnectionFactory) ctx.lookup("QueueConnectionFactory");
```

QueueConnectionFactory in the `ctx.lookup()` is the default connection factory preconfigured on the JMS server. If another is created and it needs to be used, you should substitute the default connection factory with the created one.

The pub/sub Messaging Model for the Connection Factory

In the pub/sub messaging model, you can create a connection factory at the command line like this by typing the following:

```
j2eeadmin -addJmsFactory <jndi_name_for_conn_factory> topic
```

For example, if you name the connection factory conFactory_SanDiego, the command will look like this:

```
j2eeadmin -addJmsFactory conFactory_SanDiego topic
```

At the beginning of the JMS client application, you have to look up the connection factory after calling `InitialContext()`. For pub/sub messaging, the lines that call the initial context and look up the connection factor, will look like this:

```
Context myContext = new InitialContext();
TopicConnectionFactory myTopicConnectionFactory =
    (TopicConnectionFactory) ctx.lookup("TopicConnectionFactory");
```

TopicConnectionFactory in the `ctx.lookup()` is the default connection factory preconfigured on the JMS server. If another is created and needs to be used, you should substitute the default connection factory with the created one.

In both messaging model examples, I used `InitialContext()` without parameters. This means that your code will search the jndi.properties file in the current CLASSPATH if it exists. This file contains vendor-specific information about the parameters to connect the JMS provider as well as other Java Naming Directory Interface (JNDI) parameters. If you need to, you can create a properties type variable and put these parameters into this properties variable. You can then provide this variable as a parameter in the `InitialContext()`, but you will lose portability of the client application. The parameters will specifically define the properties of the JMS vendor. If you want your application to be vendor-independent, use `InitialContext()` with no parameter.

Connections

A client application makes a connection after it completes configuring the administered objects, such as the connection factory object and the destination. It creates a virtual connection to the JMS provider, which is similar to an open TCP/IP socket between the client and the JMS provider.

Connections are created using factory methods from connection factories. If the client application has a connection factory, it can use this connection factory to create a connection. You can think of the connection as a communication channel between the application and the messaging server. You use a connection to create one or more sessions.

In the JMS specifications, methods of connections are defined in the javax.jms.Connection interface. Refer to Appendix D, "Overview of JMS Package Classes," for methods of the Connection interface.

Similar to connection factories, this interface has two subtype interfaces—javax.jms.QueueConnection and javax.jms.TopicConnection—depending on the messaging model. In the client code, you choose the methods from one of the interfaces depending the messaging model you are using.

The p2p Messaging Model for a Connection

In the client application of a p2p messaging model (queue type destination), you can create a connection like this:

```
QueueConnection myQueueConnection =
      myQueueConnectionFactory.createQueueConnection();
```

The myQueueConnectionFactory object should be created before creating the `QueueConnection` line by using the default or administrator-created ConnectionFactory object. Before the application consumes the message, you need to call this connection's `start()` method like this:

```
myQueueConnection.start();
```

When the application completes, you need to close any connection you created; otherwise, you will keep occupying the resource without using it. Closing a connection also closes its sessions, message producers, and message consumers. You can close the connection by calling the `close()` method like this:

```
myQueueConnection.close();
```

If you want to stop delivery of the messages without closing the connection, call the `stop()` method of the connection like this:

```
myQueueConnection.stop();
```

The pub/sub Messaging Model for a Connection

Similar steps are carried out when creating a connection using the topic (pub/sub) model as those carried out using the queue (p2p) model. Just substitute the word queue with the word topic. For consistency, I will explain them explicitly. You can create a connection for the pub/sub model like this:

```
TopicConnection myTopicConnection =
      myTopicConnectionFactory.createTopicConnection();
```

The myTopicConnectionFactory object should be created before creating the TopicConnection line by using the default or administrator created ConnectionFactory object. Before the application consumes the message, you need to call this connection's `start()` method like this:

```
myTopicConnection.start();
```

When the application completes, you need to close any connection you created; otherwise, you will keep occupying the resource without using it. Closing a connection also closes its sessions, message producers, and message consumers. You can close the connection by calling the `close()` method like this:

```
myTopicConnection.close();
```

If you want to stop delivery of the messages without closing the connection, call the `stop()` method of the connection like this:

```
myTopicConnection.stop();
```

When a connection is stopped by calling the `stop()` method temporarily, message delivery to this connection channel will be stopped. You must call the `start()` method to restart message delivery. Stopped mode prevents a client from receiving a message, but the client can still send a message.

Sessions

As defined in the JMS specification, a session is a single-threaded context used to produce and consume the messages in a messaging system. After creating a connection, you should create a session, which creates message producers, message consumers, and messages. Sessions allow the application to access the connection in order to send and receive messages. Sessions serialize the message, which is sent and received in a single-threaded model.

Let me explain the serialization in a single-threaded model. A messaging application (JMS client) acting as a sender produces *n* messages, but another application acting as a receiver will not receive these *n* messages at the same time. A JMS provider ensures that a receiver consumes the messages one by one.

As defined in the JMS specifications, a session provides a *transactional* context for the messaging. This means that the message is sent or received as a group in one unit. The context stores the messages for delivery until the messages are committed. For example, if the message has four parts, and if transacted messaging is chosen, these four messages are not delivered by the server (provider) until the transaction is committed. They are sent as a block.

Transactions are very important if you send a group of related information in different sessions. All of the transactions must be completed at once, such as in a banking transaction. For example, you might want to transfer an amount of money from one account to another. One of the sessions withdraws money from the first account, and the other transaction deposits the money into the second account. If you encounter a networking problem after you complete the first session, the money that you want to transfer from one account to another will disappear. Therefore, all related transactions should be put in one transaction unit and must be committed at the end of all successful transactions.

Another advantage of a transaction is that it gives you the chance to change your mind before completing the transaction by providing a *rollback* option. You can cancel all messages that are sent in one block using the rollback option.

A transaction is optional, and if it is *off-state* (meaning that no transaction is chosen), messages are delivered when they are sent. They are not stored for block delivery. If the session is without transactions, the recipient sends an acknowledgment when the message is received. If the sender client receives the acknowledgment, the message will not be sent to the client again.

There are three types of acknowledgment options:

- AUTO_ACKNOWLEDGE—An acknowledgment message is automatically sent to the sender whenever the delivery is complete.
- CLIENT_ ACKNOWLEDGE—The client must send the acknowledgment for each message programmatically.
- DUPS_OK_ ACKNOWLEDGE—The acknowledgment is not very strict and delivering the message again is possible if networking problems occur.

The basic methods for JMS sessions are specified in the javax.jms.Session interface. Refer to Appendix D for more information about the methods of a Session interface.

A Session interface, like a Connection interface, has two subtype interfaces—javax.jms.QueueSession and javax.jms.TopicSession—depending on the messaging model. I will explain how you can create a session in the client application in the following sections.

The p2p Messaging Model for a Session

In the p2p messaging model (queue type destination), you can create a session like this:

```
QueueSession myQueueSession =
      myQueueConnection.createQueueSession(false,
              Session.AUTO_ACKNOWLEDGE);
```

A queue session is created without a transaction (the first parameter is false in the `createQueueSession()` method) and the sender is acknowledged whenever the recipient receives the message (the second parameter is Session.AUTO_ACKNOWLEDGE in the `createQueueSession()` method). Do not forget to create the myQueueConnection object before this line.

The pub/sub Messaging Model for Session

Tasks in the pub/sub messaging model (topic type destination) are similar to the p2p model; you can create a session like this:

```
TopicSession myTopicSession =
      myTopicConnection.createTopicSession(true,
              Session.AUTO_ACKNOWLEDGE);
```

A topic session is created with a transaction (the first parameter is true in the `createTopicSession()` method) and the sender is acknowledged whenever the recipient receives the message (the second parameter is Session.AUTO_ACKNOWLEDGE in the `createTopicSession()` method). Do not forget to create the myTopicConnection object before this line.

A JMS connection can have one or more JMS sessions associated with this connection.

Message Producer

In the JMS specification, a message producer is defined as an object that is created by a session and is used to send a message to the destination. If you want to send a message in your code, you have to create a message producer through the session. Message-sending methods are implemented by the root javax.jms.MessageProducer interface. Refer to Appendix D for the methods of the MessageProducer interface. If the default values are valid for your application, you do not need to implement all the methods of MessageProducer. You most likely only need to create a message producer, send the created messages with the producer, and close it.

Like other interfaces, such as Session and Connection, this interface has two subtype interfaces: QueueSender and TopicPublisher. You should choose one of them in the client application depending on the messaging model you are using.

The p2p Messaging Model for a Message Producer

In the p2p messaging model (queue type destination), you can create a message producer object by using the QueueSender interface in the client code like this:

```
QueueSender myQueueSender =
      myQueueSession.createSender(myQueue);
```

The myQueueSender object is created for the myQueueSession session object to send a message to a queue type destination. Do not forget to create a queue session (myQueueSession object) before this line. myQueue is an administered object that specifies the queue name.

You can create an unidentified producer by specifying null as an argument in the `createSender()` method. In this case, you can wait until you send a message to specify the destination to send the message to.

After you create a message producer, your client application is ready to send the message if you created and prepared a message to send. In the p2p model, you can send the message by using the `send()` method like this:

```
myQueueSender.send(message);
```

If you created an unidentified producer, specify the destination as the first parameter in the `send()` method. The following lines show how to send a message to the myQueue queue:

```
QueueSender myQueueSender =
      myQueueSession.createSender(null);
       //other lines such as creating message
myQueueSender.send(myQueue, message);
```

The pub/sub Messaging Model for a Message Producer

Similarly, in the pub/sub messaging model (topic type destination), you can create a message producer object by using the TopicPublisher interface in the client code like this:

```
TopicPublisher myTopicPublisher =
      myTopicSession.createPublisher(myTopic);
```

The myTopicSender object is created for the myTopicSession session object to send a message to a topic type destination. Do not forget to create a topic session (myTopicSession object) before this line. myTopic is an administered object that specifies the topic name.

You can create an unidentified producer by specifying null as an argument in the `createPublisher()` method. In this case, you can wait until you send a message to specify the destination of the message.

After you create a message producer, your client application is ready to publish the message if you created and prepared a message to send. In the pub/sub model, you can send the message by using the `publish()` method like this:

```
myTopicPublisher.publish(message);
```

If you created an unidentified producer, specify the destination as the first parameter in the `publish()` method. The following lines show how to send a message to the myTopic topic:

```
TopicPublisher myTopicPublisher =
      myTopicSession.createPublisher(null);
       //other lines such as creating message
myTopicPublisher.publish(myTopic, message);
```

Message Consumer

If an application produces a message, it should be consumed by other applications. In the JMS specification, a message consumer is defined as an object that is created by a session and is used to receive a message that is sent to the destination. If you want to receive a message in your code, you have to create a message consumer through the session. Message-sending methods are implemented by the root javax.jms.MessageConsumer interface. Refer to Appendix D for the methods of the MessageConsumer interface.

Like other interfaces, such as Session and Connection, this interface has two subtype interfaces: QueueReceiver and TopicSubscriber. You should choose one of them in the client application depending on the messaging model you are using.

A message consumer allows the JMS client to register to a destination through a JMS provider, which handles and manages message delivery from the specified destination to the registered consumers for this destination.

Messages are consumed in two ways: synchronously and asynchronously for both messaging models. The following sections examine synchronous and asynchronous consuming for the p2p and pub/sub messaging models.

Synchronous Consuming for the p2p Messaging Model

In the synchronous p2p messaging model (queue type destination), you can create a message consumer object by using the QueueReceiver interface in the client code like this:

```
QueueReceiver myQueueReceiver =
      myQueueSession.createReciever(myQueue);
```

The myQueueReceiver object is created for the myQueueSession session object to receive a message that is sent to a queue type destination. Do not forget to create a queue session (myQueueSession object) before this line. myQueue is an administered object that specifies the queue name.

In the synchronous p2p messaging model, there is an additional receiving method: `receiveNoWait()`. This method is used when the message is immediately available. The difference between the `receive()` and `receiveNoWait()` methods is blocking. The `receive()` method blocks and waits until a message arrives if a timeout is not specified as a parameter. The `receiveNoWait()` method does not block a message consumer.

After you create a message consumer, your client application is ready to receive the message using the `receive()` method like this:

```
Message theMessageReceived = myQueueReceiver.receive();
```

If you want to set a timeout, specify the time in milliseconds as a parameter in the `receive()` method. You can use the `receive()` method at any time after calling the `start()` method of the queue connection. By adding the starting queue connection line, receiving lines listed in the preceding paragraph will look like this:

```
myQueueConnection.start();
Message theMessageReceived = myQueueReceiver.receive();
```

The myQueueConnection is the connection that was previously created.

After you receive the message, you can call the `close()` method to close the message consumer and release the resources dedicated to the message consumer.

Synchronous Consuming for the pub/sub Messaging Model

In the synchronous pub/sub messaging model (topic type destination), you can create a message consumer object by using the TopicSubscriber interface in the client code like this:

```
TopicSubscriber myTopicSubscriber =
      myTopicSession.createSubscriber(myTopic);
```

The myTopicSubscriber object is created for the myTopicSession session object to receive a message that is sent to a topic type destination. Do not forget to create a topic session (myTopicSession object) before this line. myTopic is an administered object that specifies the topic name.

In the synchronous pub/sub messaging model, there is an additional receiving method: `receiveNoWait()`. This method is used when the message is immediately available. The difference between the `receive()` and `receiveNoWait()` methods is blocking. The `receive()` method blocks and waits until a message arrives if a timeout is not specified as a parameter. The `receiveNoWait()` method does not block a message consumer.

After you create a message consumer, your client application is ready to receive the message using the `receive()` method like this:

```
Message theMessageReceived = myTopicReceiver.receive();
```

If you want to set a timeout, specify the time in milliseconds as a parameter in the `receive()` method. You can use the `receive()` method at any time after calling the `start()` method of a topic connection. By adding a starting topic connection line, the receiving lines listed in the preceding paragraph will look like this:

```
myTopicConnection.start();
Message theMessageReceived = myTopicReceiver.receive();
```

The myTopicConnection is the connection that was previously created.

After you receive the message, you can call the `close()` method to close the message consumer and release the resources dedicated to the message consumer.

A Message Listener for Asynchronous Messaging

Before providing you with information about asynchronous messaging, I need to explain the concept of a message listener. You will find more information about p2p and pub/sub messaging as well as sample code in the next two sections of this chapter.

As defined in the JMS specifications, a message listener is an object that acts as an asynchronous event handler for messages. There is one method in a message listener class that implements the MessageListener interface: `onMessage()`. In the `onMessage()` method, you define what should be done when a message arrives. From a developer's view, you develop a class that implements the MessageListener interface and overrides the `onMessage()` method in this class by putting proper codes in this method. The following sample code lines show you the structure of a developer created message listener class that can be used in an asynchronous receiver or publisher client application:

```
public class myListener implements MessageListener {
   // overriding onMessage method for your message type and format
   public void onMessage(Message parMessage) {
      // in the body of onMessage() method,
     // put the proper code lines what should
    //  be done when a message arrives
  } //end of onMessage method
}  //end of class
```

One of the most important aspects of the message listener is that it is not specific to a particular destination type. The same listener can obtain messages from either a queue or a topic, depending on the messaging model, for example, p2p (QueueReceiver) or pub/sub (TopicSubscriber).

Another important aspect of the message listener is that a message listener usually expects a specific message type and format. You should define this message type in the `onMessage()` method. If the client application does not receive the proper message type, it must warn the client.

A message listener can reply to messages in two ways. First, it assumes a particular destination type specified in the code explicitly. Second, it can obtain the destination type of the message and create a producer for that destination type by using a temporary destination.

A message listener is serialized. The session that created the message consumer serializes the execution of all message listeners registered with the session. This means that only one of the current session's message listeners is running.

J2EE version 1.3 has a special bean that acts as a message listener: the message-driven bean. Message-driven beans are discussed in Chapter 10, "JMS and Web."

Asynchronous Consuming for the p2p Messaging Model

Asynchronous messaging is preferred in the pub/sub messaging model, but you can use it for both types of messaging models by using the MessageListener interface.

You should first register the message listener with the current QueueReceiver through the `setMessageListener` method for asynchronous messaging. For example, let's assume that you have developed a LevQueueListener class as a message listener that implements the MessageListener interface and overrides the `onMessage()` method. You can register the message listener like this:

```
LevQueueListener myQueueListener = new LevQueueListener();
myQueueReceiver.setMessageListener(myQueueListener);
```

The myQueueReceiver object is created for the myQueueSession session object to receive a message that is sent to a queue type destination. The myQueueListener object is a message listener object that listens to the channel whether the message arrives or not.

After you register the message listener, do not forget to call the `start()` method on the QueueConnection to begin message delivery. If you call the `start()` method before you register the message listener, you will miss messages.

Once message delivery begins, the message consumer automatically calls the message listener's `onMessage()` method whenever a message is delivered. The `onMessage()` method has a message type parameter. This parameter, which is a delivered message from the destination, will be cast as a proper message for the client. I will provide information about the message types in the next section.

Asynchronous Consuming for the pub/sub Messaging Model

Asynchronous messaging for the pub/sub messaging model is implemented by using a MessageListener interface similar to the one in the p2p model.

You should first register the message listener with the current TopicSubscriber through the `setMessageListener` method for asynchronous messaging. For example, let's assume that you have developed an ErdTopicListener class as a message listener that implements the MessageListener interface and overrides the `onMessage()` method. You can register the message listener like this:

```
ErdTopicListener myTopicListener = new ErdTopicListener ();
myTopicSubscriber.setMessageListener(myTopicListener);
```

The myTopicSubscriber object is created for the myTopicSession session object to receive a message that is sent to a topic type destination. The myTopicListener object is a message listener object that listens to the channel whether the message arrives or not.

After you register the message listener, do not forget to call the `start()` method on the TopicConnection to begin message delivery. If you call the `start()` method before you register the message listener, you will miss messages.

Similar to the p2p messaging model, once message delivery begins, the message consumer automatically calls the message listener's `onMessage()` method whenever a message is delivered. The `onMessage()` method has a message type parameter. This parameter, which is a delivered message from the destination, will be cast as a proper message for the client.

Messages

Earlier, I explained many concepts and techniques for a messaging system based on the JMS specifications. So far, I've discussed transferring a message from one application (client) to another application (client) or applications. This discussion about messages is the most important part of the chapter.

Technically, by transferring a message from one point to another you are actually producing a message in one software application and consuming that message in another software application. Even though the JMS message format is very simple, it still allows you to create compatible messages with other non-JMS applications on different types of platforms.

In other systems, such as Java RMI, Distributed Component Object Model (DCOM), or Common Object Request Broker Architecture (CORBA), message definition and usage is different from the JMS specification. In these systems, a message is a command to execute a method or procedure. In a JMS system, a message is not a command. It is a pure message that is transferred from one point to another. The message does not force the recipient to do something. The sender does not wait for a response.

As shown in Figure 5.5, a message object has three main parts: a message header, message properties, and a message body. Detailed information about the parts of a message object is provided in the following sections.

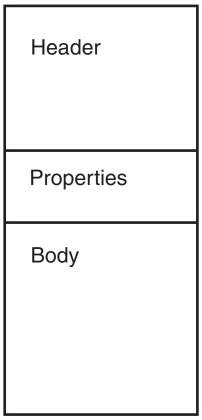

Figure 5.5 The three parts of a message structure.

The Header Part of the Message Object

A JMS message header contains some predefined fields. These fields are identifiers for a client and a provider. For example, every message has a unique id, time stamp, or priority. In addition to some optional fields, a message header must have certain fields such as a JMS destination. These accessor or mutator methods obey setJMS<HeaderName> and getJMS<HeaderName> syntax. A list of methods in a message header follows. If you need more information about properties methods, refer to Appendix D.

- `getJMSDestination()`
- `setJMSDestination(Destination paramDestination)`
- `getJMSDeliveryMode()`
- `setJMSDeliveryMode(int deliveryMode)`
- `getJMSMessageID()`
- `setJMSMessageID(String ID)`
- `getJMSTimestamp()`
- `setJMSTimestamp(long timeStamp)`
- `getJMSExpiration()`
- `setJMSExpiration(long expiration)`
- `getJMSRedelivered()`
- `setJMSRedelivered(boolean redelivery)`

- `getJMSPriority()`
- `setJMSPriority(int priority)`
- `getJMSReplyTo()`
- `setJMSReplyTo(Destination replyTo)`
- `getJMSCorrelationID()`
- `setJMSCorrelationID(String correlationValue)`
- `getJMSCorrelationIDAsBytes()`
- `setJMSCorrelationIDAsBytes(byte[] correlationValue)`
- `getJMSType()`
- `setJMSType(String type)`

Some of these fields are assigned by the client, some are assigned automatically when the message is delivered, and one field is assigned by a JMS provider (see Table 5.1).

Table 5.1 **Message Header Fields and How They Are Set**

Header Field	Set By
JMSDestination	Message Producer
JMSDeliveryMode	Message Producer
JMSExpiration	Message Producer
JMSPriority	Message Producer
JMSMessageID	Message Producer
JMSTimestamp	Message Producer
JMSRedelivered	JMS provider
JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client

The Properties Part of the Message Object

Properties of a JMS message are like additional headers that are assigned to a message to provide information to the developer. These properties can be used to provide compatibility with other messaging systems. Properties can be used for message selectors, which are discussed later in this section.

Properties can either be predefined or user defined. They can be divided into three classes: application-specific properties, JMS-defined properties, and provider-specific properties.

For example, if you want to define a String type hostuserID property, you can define it by using the `setStringProperty()` accessor like this:

```
myTextMessage.setStringProperty("hostUserID", hostuserID);
```

The myTextMessage is the TextMessage type message created by using the `createTextMessage()` method. This hostUserID property is valid and meaningful only in the application. This kind of property specified in the property part of the message object is useful to filter messages using message selectors.

Most properties methods (accessor or mutator methods) of a message obey the set<DataType>Property and get<DataType>Property syntax. The properties methods list follows. If you need more information about properties methods, refer to Appendix D.

- `clearProperties()`
- `propertyExists(String name)`
- `getBooleanProperty(String name)`
- `getByteProperty(String name)`
- `getShortProperty(String name)`
- `getIntProperty(String name)`
- `getLongProperty(String name)`
- `getFloatProperty(String name)`
- `getDoubleProperty(String name)`
- `getStringProperty(String name)`
- `getObjectProperty(String name)`
- `getPropertyNames()`
- `setBooleanProperty(String name, boolean value)`
- `setByteProperty(String name, byte value)`
- `setShortProperty(String name, short value)`
- `setIntProperty(String name, int value)`
- `setLongProperty(String name, long value)`
- `setFloatProperty(String name, float value)`
- `setDoubleProperty(String name, double value)`
- `setStringProperty(String name, String value)`
- `setObjectProperty(String name, Object value)`

Message Selectors

In this section, I provide some brief information about message selectors, which are used in message headers and properties.

Another feature of a message consumer is to filter the message using message selectors. This process is like a SQL type query, and a message selector is a string that contains a criteria expression. You can filter a message with criteria from the message headers and the message properties. The message consumer only receives messages whose headers and properties match the criteria specified in the selector. A message selector cannot select messages based on the content of the message body.

For example, assume that there is an identifier named cityName in the JMS property of a message. You want to filter some cities when the message arrives. You can write a selector like this:

```
String mySelector = "cityName IN ('New York', 'Boston', 'Newark');
```

You can use this selector in the message receiver like this:

```
TopicSubscriber mySubscriber =
session.createSubscriber(topic, mySelector, false);
```

The selector sentence can be as complex as necessary. The identifier used in the selector sentence must always refer to the JMS property name or the JMS header name in the message.

The Body of the Message Object

In the JMS specifications, there are five different message body formats, which are also called message types: TextMessage, ObjectMessage, MapMessage, BytesMessage, and StreamMessage. Actually, these five message types are subinterfaces of the Message interface. JMS defines the Message interface, but it does not define its implementation. Vendors are free to implement and transfer the message in their own way. JMS tries to maintain standard interfaces for JMS developers. Vendors might ignore one message type and support their own message type in the Message interface.

Information about these five subinterfaces is provided in the following sections. You can find more information about subinterfaces of the Message interface and their methods in Appendix D.

TextMessage Interface

The TextMessage interface contains the java.lang.String type object. It is used when you need to transfer simple text messages. You can transfer more complicated messages such as XML documents as long as they are text-based. Before you send the message, you need to use the `createTextMessage()` and `setText()` methods. The `setText()` method takes a String type value or variable as a parameter. The following sample lines send a text message to a topic destination (pub/sub):

```
TextMessage messageOnBoard = session.createTextMessage();
messageOnBoard.setText("This is a text message");
myTopicPublisher.publish(messageOnBoard);
```

If the destination is a queue type (p2p), you only need to change the sender method as follows:

```
TextMessage messageOnBoard = session.createTextMessage();
messageOnBoard.setText("This is a text message");
myQueueSender.send(messageOnBoard);
```

When the message arrives at the consumer, it is extracted by the `getText()` method and is assigned a String variable. As an example, you can extract a text type message from the preceding example like this:

```
TextMessage recTextMessage = (TextMessage)message;
String recOnBoard = recTextMessage.getText();
```

ObjectMessage Interface

The ObjectMessage interface contains a serializable Java object. It is used when you need to transfer Java objects. Before you send the message, you need to use the `createObjectMessage()` and `setObject()` methods. The `setObject()` method takes a serializable object as a parameter. The following sample lines send an object message to a topic destination (pub/sub):

```
ObjectMessage theObjMessage =
session.createObjectMessage();
theObjMessage.setObj("This is an object message");
myTopicPublisher.publish(theObjMessage);
```

If the destination is a queue type (p2p), you only need to change the sender method as follows:

```
ObjectMessage theObjMessage =
session.createObjectMessage();
theObjMessage.setObj("This is an object message");
myQueueSender.send(theObjMessage);
```

When the message arrives at the consumer, it is extracted by the `getObject()` method and is assigned an Object variable. As an example, you can extract an object type message from the preceding example like this:

```
ObjectMessage recObjectMessage = (ObjectMessage)message;
Object recObjMessage = recObjectMessage.getObject();
```

MapMessage Interface

The MapMessage interface contains a set of name-value pairs. It is used when you need to transfer keyed data. This method takes a key-value pair as a parameter. Before you send the message, you need to use the `createMapMessage()` method and certain `setXXX()` methods depending on the value data type. The `setXXX()` methods are:

- `setInt()` for integer type value
- `setFloat()` for float type
- `setString()` for String type value
- `setObject()` for object type value
- `setBoolean()` for boolean type value
- `setBytes()` for byte type value
- `setShort()` for short type value
- `setChar()` for char type value
- `setLong()` for long type value
- `setDouble()` for double type value

The following sample lines send a map message to a topic destination (pub/sub):

```
MapMessage theMapMessage = session.createMapMessage();
theMapMessage.setString("HostName", "Montreal");
theMapMessage.setFloat("RAM", 512);
theMapMessage.setFloat("Disk", 80);
myTopicPublisher.publish(theMapMessage);
```

If the destination is a queue type (p2p), you only need to change the sender method as follows:

```
MapMessage theMapMessage = session.createMapMessage();
theMapMessage.setString("HostName", "Montreal");
theMapMessage.setFloat("RAM", 512);
theMapMessage.setInt("Disk", 80);
myQueueSender.send(theMapMessage);
```

When the message arrives at the consumer, it is extracted by the `getXXX()` methods and assigned to a proper type variable. As an example, you can extract the map type message from the preceding example like this:

```
MapMessage recMapMessage = (MapMessage)message;
String recHostName = recMapMessage.getString("HostName");
float recRAM = recMapMessage.getFloat("RAM");
int recDisk = recMapMessage.getInt("Disk");
```

BytesMessage Interface

The BytesMessage interface contains an array of primitive bytes. It is used when you need to transfer data in the application's native format, which might not be suitable for existing message types in the JMS specifications. You can transfer data between two applications regardless of its JMS status. Before you send the message, you need to use the `createBytesMessage()` method and certain `writeXXX()` methods depending on the value data type. The `writeXXX()` methods are:

- `writeByte(byte parValue)` for byte type value
- `writeBytes(byte[] parValue)` for array of byte value
- `writeBoolean(boolean parValue)` for boolean type value
- `writeChar(char parValue)` for char type value
- `writeShort(short parValue)` for short type value
- `writeInt(int parValue)` for integer type value
- `writeLong(long parValue)` for long type value
- `writeFloat(float parValue)` for float type value
- `writeDouble(double parValue)` for double type value
- `writeUTF(String parValue)` for String type value
- `writeObject(Object parValue)` for object type value

The BytesMessage interface is very similar to java.io.DataInputStream and java.io.DataOutputStream. The following sample lines send byte messages to a topic destination (pub/sub):

```
BytesMessage theBytesMessage =
session.createBytesMessage();
theBytesMessage.writeUTF("San Fransisco");
theBytesMessage.writeInt(120);
myTopicPublisher.publish(theBytesMessage);
```

If the destination is a queue type (p2p), you only need to change the sender method as follows:

```
BytesMessage theBytesMessage =
session.createBytesMessage();
theBytesMessage.writeUTF("San Fransisco");
theBytesMessage.writeInt(120);
myQueueSender.send(theBytesMessage);
```

When the message arrives at the consumer, it is extracted by the `readXXX()` methods and is assigned to a proper type variable. As an example, you can extract the bytes type message from the preceding example like this:

```
BytesMessage recBytesMessage = (BytesMessage)message;
String cityName = recBytesMessage.readUTF();
int carParking = recBytesMessage. readInt();
```

StreamMessage Interface

The StreamMessage interface contains a stream of primitive Java types such as int, char, double, boolean, and so on. Primitive types are read from the message in the same order they are written. The StreamMessage interface and its methods look like the BytesMessage, but they are not the same. StreamMessage keeps track of the order of written messages, and the message is then converted to the primitive type by following formal conversion rules.

Before you send the message, you need to use the `createStreamMessage()` and certain `writeXXX()` methods depending on the value data type. The `writeXXX()` methods are:

- `writeByte(byte parValue)` for byte type value
- `writeBytes(byte[] parValue)` for array of byte value
- `writeBoolean(boolean parValue)` for boolean type value
- `writeChar(char parValue)` for char type value
- `writeShort(short parValue)` for short type value
- `writeInt(int parValue)` for integer type value
- `writeLong(long parValue)` for long type value
- `writeFloat(float parValue)` for float type value
- `writeDouble(double parValue)` for double type value
- `writeString(String parValue)` for String type value
- `writeObject(Object parValue)` for object type value

You have to be careful when converting a message stream from a written format to a reading format. For example, you can write the message in long data type, and you can read it in long or String data type. Data type conversions are listed in Table 5.2.

Table 5.2 **StreamMessage Conversion Rules**

Written Message Type	Read Message Type
boolean	boolean, String
byte	byte, short, int, long, String
char	char, String
short	short, int, long, String

Written Message Type	Read Message Type
int	int, long, String
long	long, String
float	float, double, String
double	double, String
String	String, boolean, byte, short, int, long, float, double
byte[ ]	byte[ ]

The following sample lines send a stream message to a topic destination (pub/sub):

```
StreamMessage theStreamMessage =
session.createStreamMessage();
theStreamMessage.writeString("Alaska");
theStreamMessage.writeShort(49);
myTopicPublisher.publish(theStreamMessage);
```

If the destination is a queue type (p2p), you only need to change the sender method as follows:

```
StreamMessage theStreamMessage =
session.createStreamMessage();
theStreamMessage.writeString("Alaska");
theStreamMessage.writeShort(49);
myQueueSender.send(theStreamMessage);
```

When the message arrives at the consumer, it is extracted by the `readXXX()` methods and assigned to a proper type variable. As an example, you can extract the stream type message from the preceding example like this:

```
StreamMessage recStreamMessage = (StreamMessage)message;
String stateName = recStreamMessage.readString();
int stateNo = recStreamMessage. readShort();
```

Because some data types of the message written can be read in another data type by obeying formal conversion rules, the last line of the reading example can be written like this:

```
int stateNo = recStreamMessage. readInt();
```

or

```
int stateNo = recStreamMessage. readLong();
```

Summary

In this chapter, you learned the concepts of JMS programming and the basics of JMS API programming techniques. I started this chapter by explaining the definition of a messaging service and its major features. The messaging system is based on message-oriented middleware (MOM), which was explained in the previous chapter. Advantages and disadvantages of using a messaging system were then discussed prior to providing information about the JMS API.

The JMS API, which is built onto the J2EE 1.3 platform specifications, covers point-to-point and publish-and-subscriber messaging models used in MOM providers.

In this chapter, you learned the concepts of JMS programming including its architecture, message consumption, destination, connection factory, connection, session, message producer, message consumer, and message listener.

I also showed you how to create two administered objects, ConnectionFactory and Destination, and how to use them in JMS messaging applications.

I explained synchronous messaging and asynchronous messaging and compared the two models. You also learned about message structure and different message types in the JMS specifications.

Although I provided some information about JMS programming concepts and techniques, more details such as methods of some interfaces are discussed in Appendix D. Many concepts were explained using a few sample lines of code, but you can find complete and more valuable examples in the next chapter.

Questions and Answers

1. If there is no ConnectionFactory object created by the JMS provider administrator, what can I do?

A ConnectionFactory object is an administered object and must be created by the JMS provider administrator. If there is no ConnectionFactory object created, you can use the default ConnectionFactory object. You will learn more detailed information in the section, "Basic Steps to Write a JMS Application," in the next chapter.

2. If there is no Destination object created by the JMS provider administrator, what can I do?

A Destination object is an administered object like the ConnectionFactory object and must be created by the JMS provider administrator. If there is no Destination object created, you cannot use the JMS API for messaging. There is no default Destination object.

3. If I do not specify a value for the time-to-live method of the Message interface, what will happen to the message if the receiver is inactive?

If you do not specify a time for time-to-live, it means that the message will never expire. The message will be delivered whenever the receiver client is available, depending on the messaging model you are using.

4. Can I create two or more connections to send messages?

Yes, you can. Some advanced applications might use several connections. But a connection is a relatively heavyweight object. Therefore, only one connection is preferred. If you need to, you can create two or more sessions—which are lightweight JMS objects—and a number of message producers and consumers to send messages on different channels.

5. If I use transacted messaging and if the connection or session is closed in the middle of transmitting without an acknowledgment, what will happen to the message that is not sent to the destination?

In this case, the JMS provider will call the `rollback()` method, the messages that are not delivered will be deleted, and the messages already delivered to the destination will be removed from the destination by a JMS provider. The recipient will not receive any messages.

6. I want to filter some messages that should not be delivered to the consumer. How can I define a field in a message selector for some values in the message body?

A message selector only works with header and property fields. You cannot filter messages based on values in the message body.

7. In the message received, there is a ReplyTo field. If this destination is not created, how can I send a message to the destination specified in a ReplyTo field?

You can create a temporary destination dynamically. During the session, the connection is open, so you can send the message to this temporary destination. Whenever the connection is closed, this destination is removed along with the messages that have not yet been delivered.