Java™ Message Service API Tutorial

by Kim Haase

Sun Microsystems, Inc.

Contents

CHAPTER 1

Overview

THIS overview of the Java™ Message Service Application Programming Interface (the JMS API) answers the following questions.

- What is messaging?
- What is the JMS API?
- How can you use the JMS API?
- How does the JMS API work with the Java 2 Platform, Enterprise Edition (J2EE™ platform)?

1.1 What Is Messaging?

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

Messaging enables distributed communication that is *loosely coupled*. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only what message format and what destination to use. In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods.

Messaging also differs from electronic mail (e-mail), which is a method of communication between people or between software applications and people. Messaging is used for communication between software applications or software components.

1.2 What Is the JMS API?

The Java Message Service is a Java API that allows applications to create, send, receive, and read messages. Designed by Sun and several partner companies, the JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations.

The JMS API minimizes the set of concepts a programmer must learn to use messaging products but provides enough features to support sophisticated messaging applications. It also strives to maximize the portability of JMS applications across JMS providers in the same messaging domain.

The JMS API enables communication that is not only loosely coupled but also

- **Asynchronous.** A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.
- **Reliable.** The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

The JMS Specification was first published in August 1998. The latest version of the JMS Specification is Version 1.0.2b, which was released in August 2001. You can download a copy of the Specification from the JMS Web site, `http://java.sun.com/products/jms/`.

1.3 When Can You Use the JMS API?

An enterprise application provider is likely to choose a messaging API over a tightly coupled API, such as Remote Procedure Call (RPC), under the following circumstances.

- The provider wants the components not to depend on information about other components' interfaces, so that components can be easily replaced.
- The provider wants the application to run whether or not all components are up and running simultaneously.
- The application business model allows a component to send information to another and to continue to operate without receiving an immediate response.

For example, components of an enterprise application for an automobile manufacturer can use the JMS API in situations like these.

- The inventory component can send a message to the factory component when the inventory level for a product goes below a certain level, so the factory can make more cars.
- The factory component can send a message to the parts components so that the factory can assemble the parts it needs.
- The parts components in turn can send messages to their own inventory and order components to update their inventories and to order new parts from suppliers.
- Both the factory and the parts components can send messages to the accounting component to update their budget numbers.
- The business can publish updated catalog items to its sales force.

Using messaging for these tasks allows the various components to interact with one another efficiently, without tying up network or other resources. Figure 1.1 illustrates how this simple example might work.

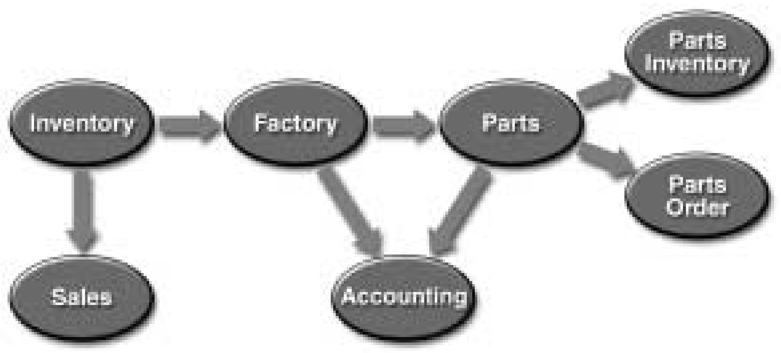

Figure 1.1 Messaging in an Enterprise Application

Manufacturing is only one example of how an enterprise can use the JMS API. Retail applications, financial services applications, health services applications, and many others can make use of messaging.

1.4 How Does the JMS API Work with the J2EE™ Platform?

When the JMS API was introduced in 1998, its most important purpose was to allow Java applications to access existing messaging-oriented middleware (MOM) systems, such as MQSeries from IBM. Since that time, many vendors have adopted and implemented the JMS API, so that a JMS product can now provide a complete messaging capability for an enterprise.

At the 1.2 release of the J2EE platform, a service provider based on J2EE technology ("J2EE provider") was required to provide the JMS API interfaces but was not required to implement them. Now, with the 1.3 release of the J2EE platform ("the J2EE 1.3 platform"), the JMS API is an integral part of the platform, and application developers can use messaging with components using J2EE APIs ("J2EE components").

The JMS API in the J2EE 1.3 platform has the following features.

- Application clients, Enterprise JavaBeans (EJB™) components, and Web components can send or synchronously receive a JMS message. Application clients can in addition receive JMS messages asynchronously. (Applets, however, are not required to support the JMS API.)

- A new kind of enterprise bean, the message-driven bean, enables the asynchronous consumption of messages. A JMS provider may optionally implement concurrent processing of messages by message-driven beans.
- Message sends and receives can participate in distributed transactions.

The addition of the JMS API enhances the J2EE platform by simplifying enterprise development, allowing loosely coupled, reliable, asynchronous interactions among J2EE components and legacy systems capable of messaging. A developer can easily add new behavior to a J2EE application with existing business events by adding a new message-driven bean to operate on specific business events. The J2EE platform's EJB container architecture, moreover, enhances the JMS API by providing support for distributed transactions and allowing for the concurrent consumption of messages.

Another technology new to the J2EE 1.3 platform, the J2EE Connector Architecture, provides tight integration between J2EE applications and existing Enterprise Information (EIS) systems. The JMS API, on the other hand, allows for a very loosely coupled interaction between J2EE applications and existing EIS systems.

CHAPTER 2

Basic JMS API Concepts

THIS chapter introduces the most basic JMS API concepts, the ones you must know to get started writing simple JMS client applications:

- JMS API architecture
- Messaging domains
- Message consumption

The next chapter introduces the JMS API programming model. Later chapters cover more advanced concepts, including the ones you need to write J2EE applications that use message-driven beans.

2.1 JMS API Architecture

A JMS application is composed of the following parts.

- A *JMS provider* is a messaging system that implements the JMS interfaces and provides administrative and control features. An implementation of the J2EE platform at release 1.3 includes a JMS provider.
- *JMS clients* are the programs or components, written in the Java programming language, that produce and consume messages.
- *Messages* are the objects that communicate information between JMS clients.
- *Administered objects* are preconfigured JMS objects created by an administrator for the use of clients. The two kinds of administered objects are destinations and connection factories, which are described in Section 3.1 on page 22.

- *Native clients* are programs that use a messaging product's native client API instead of the JMS API. An application first created before the JMS API became available and subsequently modified is likely to include both JMS and native clients.

Figure 2.1 illustrates the way these parts interact. Administrative tools allow you to bind destinations and connection factories into a Java Naming and Directory Interface™ (JNDI) API namespace. A JMS client can then look up the administered objects in the namespace and then establish a logical connection to the same objects through the JMS provider.

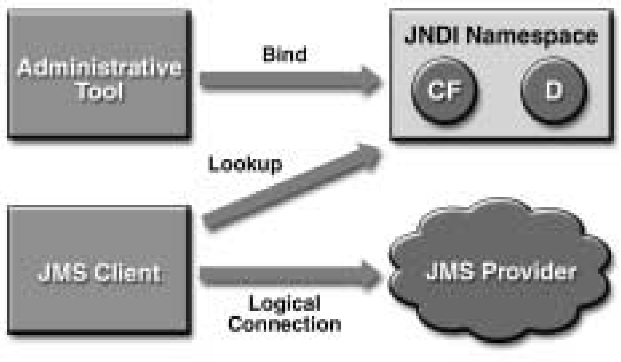

Figure 2.1 JMS API Architecture

2.2 Messaging Domains

Before the JMS API existed, most messaging products supported either the *point-to-point* or the *publish/subscribe* approach to messaging. The JMS Specification provides a separate domain for each approach and defines compliance for each domain. A standalone JMS provider may implement one or both domains. A J2EE provider must implement both domains.

In fact, most current implementations of the JMS API provide support for both the point-to-point and the publish/subscribe domains, and some JMS clients combine the use of both domains in a single application. In this way, the JMS API has extended the power and flexibility of messaging products.

2.2.1 Point-to-Point Messaging Domain

A point-to-point (PTP) product or application is built around the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queue(s) established to hold their messages. Queues retain all messages sent to them until the messages are consumed or until the messages expire.

PTP messaging has the following characteristics and is illustrated in Figure 2.2.

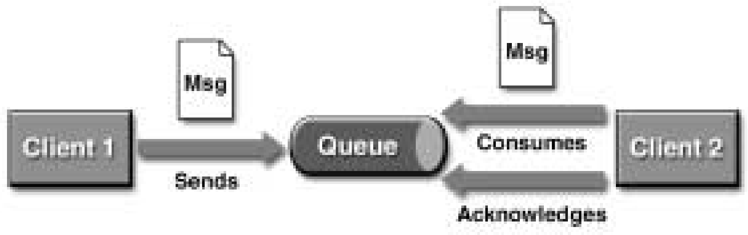

Figure 2.2 Point-to-Point Messaging

- Each message has only one consumer.
- A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.
- The receiver acknowledges the successful processing of a message.

Use PTP messaging when every message you send must be processed successfully by one consumer.

2.2.2 Publish/Subscribe Messaging Domain

In a publish/subscribe (pub/sub) product or application, clients address messages to a topic. Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to current subscribers.

Pub/sub messaging has the following characteristics.

- Each message may have multiple consumers.
- Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

The JMS API relaxes this timing dependency to some extent by allowing clients to create *durable subscriptions*. Durable subscriptions can receive messages sent while the subscribers are not active. Durable subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients. For more information about durable subscriptions, see Section 5.2.1 on page 67.

Use pub/sub messaging when each message can be processed by zero, one, or many consumers. Figure 2.3 illustrates pub/sub messaging.

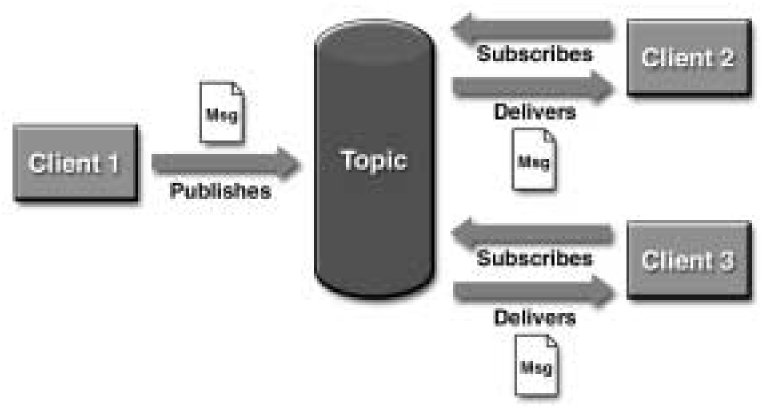

Figure 2.3 Publish/Subscribe Messaging

2.3 Message Consumption

Messaging products are inherently asynchronous in that no fundamental timing dependency exists between the production and the consumption of a message. However, the JMS Specification uses this term in a more precise sense. Messages can be consumed in either of two ways:

- **Synchronously.** A subscriber or a receiver explicitly fetches the message from the destination by calling the `receive` method. The `receive` method can block until a message arrives or can time out if a message does not arrive within a specified time limit.
- **Asynchronously.** A client can register a *message listener* with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's `onMessage` method, which acts on the contents of the message.

CHAPTER 3

The JMS API Programming Model

THE basic building blocks of a JMS application consist of

- Administered objects: connection factories and destinations
- Connections
- Sessions
- Message producers
- Message consumers
- Messages

Figure 3.1 shows how all these objects fit together in a JMS client application.

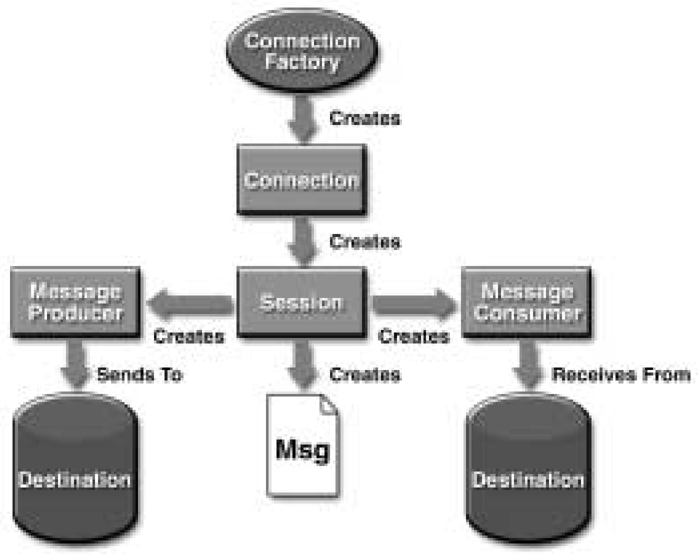

Figure 3.1 The JMS API Programming Model

This chapter describes all these objects briefly and provides sample commands and code snippets that show how to create and use the objects. The last section briefly describes JMS API exception handling.

Examples that show how to combine all these objects in applications appear in later chapters. For more details, see the JMS API documentation, which you can download from the JMS Web site, `http://java.sun.com/products/jms/`.

3.1 Administered Objects

Two parts of a JMS application—destinations and connection factories—are best maintained administratively rather than programmatically. The technology underlying these objects is likely to be very different from one implementation of the JMS API to another. Therefore, the management of these objects belongs with other administrative tasks that vary from provider to provider.

JMS clients access these objects through interfaces that are portable, so a client application can run with little or no change on more than one implementation of the JMS API. Ordinarily, an administrator configures administered objects in a Java Naming and Directory Interface (JNDI) API namespace, and JMS clients then look them up, using the JNDI API. J2EE applications always use the JNDI API.

With the J2EE Software Development Kit (SDK) version 1.3.1, you use a tool called `j2eeadmin` to perform administrative tasks. For help on the tool, type `j2eeadmin` with no arguments.

3.1.1 Connection Factories

A *connection factory* is the object a client uses to create a connection with a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator. A pair of connection factories come preconfigured with the J2EE SDK and are accessible as soon as you start the service. Each connection factory is an instance of either the `QueueConnectionFactory` or the `TopicConnectionFactory` interface.

With the J2EE SDK, for example, you can use the default connection factory objects, named `QueueConnectionFactory` and `TopicConnectionFactory`, to create connections. You can also create new connection factories by using the following commands:

```
j2eeadmin -addJmsFactory jndi_name queue

j2eeadmin -addJmsFactory jndi_name topic
```

At the beginning of a JMS client program, you usually perform a JNDI API lookup of the connection factory. For example, the following code fragment obtains an `InitialContext` object and uses it to look up the `QueueConnectionFactory` and the `TopicConnectionFactory` by name:

```
Context ctx = new InitialContext();

QueueConnectionFactory queueConnectionFactory =
  (QueueConnectionFactory) ctx.lookup("QueueConnectionFactory");

TopicConnectionFactory topicConnectionFactory =
  (TopicConnectionFactory) ctx.lookup("TopicConnectionFactory");
```

Calling the `InitialContext` method with no parameters results in a search of the current classpath for a vendor-specific file named `jndi.properties`. This file indicates which JNDI API implementation to use and which namespace to use.

3.1.2 Destinations

A *destination* is the object a client uses to specify the target of messages it produces and the source of messages it consumes. In the PTP messaging domain, destinations are called queues, and you use the following J2EE SDK command to create them:

```
j2eeadmin -addJmsDestination queue_name queue
```

In the pub/sub messaging domain, destinations are called topics, and you use the following J2EE SDK command to create them:

```
j2eeadmin -addJmsDestination topic_name topic
```

A JMS application may use multiple queues and/or topics.

In addition to looking up a connection factory, you usually look up a destination. For example, the following line of code performs a JNDI API lookup of the previously created topic `MyTopic` and assigns it to a `Topic` object:

```
Topic myTopic = (Topic) ctx.lookup("MyTopic");
```

The following line of code looks up a queue named `MyQueue` and assigns it to a `Queue` object:

```
Queue myQueue = (Queue) ctx.lookup("MyQueue");
```

3.2 Connections

A *connection* encapsulates a virtual connection with a JMS provider. A connection could represent an open TCP/IP socket between a client and a provider service daemon. You use a connection to create one or more sessions.

Like connection factories, connections come in two forms, implementing either the `QueueConnection` or the `TopicConnection` interface. For example, once you have a `QueueConnectionFactory` or a `TopicConnectionFactory` object, you can use it to create a connection:

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

```
TopicConnection topicConnection =
  topicConnectionFactory.createTopicConnection();
```

When an application completes, you need to close any connections that you have created. Failure to close a connection can cause resources not to be released by the JMS provider. Closing a connection also closes its sessions and their message producers and message consumers.

```
queueConnection.close();

topicConnection.close();
```

Before your application can consume messages, you must call the connection's `start` method; for details, see Section 3.5 on page 27. If you want to stop message delivery temporarily without closing the connection, you call the `stop` method.

3.3 Sessions

A *session* is a single-threaded context for producing and consuming messages. You use sessions to create message producers, message consumers, and messages. Sessions serialize the execution of message listeners; for details, see Section 3.5.1 on page 28.

A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work. For details, see Section 5.2.2 on page 70.

Sessions, like connections, come in two forms, implementing either the `QueueSession` or the `TopicSession` interface. For example, if you created a `TopicConnection` object, you use it to create a `TopicSession`:

```
TopicSession topicSession =
  topicConnection.createTopicSession(false,
  Session.AUTO_ACKNOWLEDGE);
```

The first argument means that the session is not transacted; the second means that the session automatically acknowledges messages when they have been received successfully. (For more information, see Section 5.1.1 on page 62.)

Similarly, you use a `QueueConnection` object to create a `QueueSession`:

```
QueueSession queueSession =
  queueConnection.createQueueSession(true, 0);
```

Here, the first argument means that the session is transacted; the second indicates that message acknowledgment is not specified for transacted sessions.

3.4 Message Producers

A *message producer* is an object created by a session and is used for sending messages to a destination. The PTP form of a message producer implements the `QueueSender` interface. The pub/sub form implements the `TopicPublisher` interface.

For example, you use a `QueueSession` to create a sender for the queue `myQueue`, and you use a `TopicSession` to create a publisher for the topic `myTopic`:

```
QueueSender queueSender = queueSession.createSender(myQueue);

TopicPublisher topicPublisher =
  topicSession.createPublisher(myTopic);
```

You can create an unidentified producer by specifying `null` as the argument to `createSender` or `createPublisher`. With an unidentified producer, you can wait to specify which destination to send the message to until you send or publish a message.

Once you have created a message producer, you can use it to send messages. (You have to create the messages first; see Section 3.6 on page 29.) With a `QueueSender`, you use the `send` method:

```
queueSender.send(message);
```

With a `TopicPublisher`, you use the `publish` method:

```
topicPublisher.publish(message);
```

If you created an unidentified producer, use the overloaded `send` or `publish` method that specifies the destination as the first parameter.

3.5 Message Consumers

A *message consumer* is an object created by a session and is used for receiving messages sent to a destination. A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.

The PTP form of message consumer implements the `QueueReceiver` interface. The pub/sub form implements the `TopicSubscriber` interface.

For example, you use a `QueueSession` to create a receiver for the queue `myQueue`, and you use a `TopicSession` to create a subscriber for the topic `myTopic`:

```
QueueReceiver queueReceiver = queueSession.createReceiver(myQueue);

TopicSubscriber topicSubscriber =
  topicSession.createSubscriber(myTopic);
```

You use the `TopicSession.createDurableSubscriber` method to create a durable topic subscriber. For details, see Section 5.2.1 on page 67.

Once you have created a message consumer, it becomes active, and you can use it to receive messages. You can use the `close` method for a `QueueReceiver` or a `TopicSubscriber` to make the message consumer inactive. Message delivery does not begin until you start the connection you created by calling the `start` method (see Section 3.2 on page 24).

With either a `QueueReceiver` or a `TopicSubscriber`, you use the `receive` method to consume a message synchronously. You can use this method at any time after you call the `start` method:

```
queueConnection.start();
Message m = queueReceiver.receive();

topicConnection.start();
Message m = topicSubscriber.receive(1000); // time out after a second
```

To consume a message asynchronously, you use a message listener, described in Section 3.5.1 on page 28.

3.5.1 Message Listeners

A *message listener* is an object that acts as an asynchronous event handler for messages. This object implements the `MessageListener` interface, which contains one method, `onMessage`. In the `onMessage` method, you define the actions to be taken when a message arrives.

You register the message listener with a specific `QueueReceiver` or `TopicSubscriber` by using the `setMessageListener` method. For example, if you define a class named `TopicListener` that implements the `MessageListener` interface, you can register the message listener as follows:

```
TopicListener topicListener = new TopicListener();
topicSubscriber.setMessageListener(topicListener);
```

After you register the message listener, you call the `start` method on the `QueueConnection` or the `TopicConnection` to begin message delivery. (If you call `start` before you register the message listener, you are likely to 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 takes one argument of type `Message`, which the method can cast to any of the other message types (see Section 3.6.3 on page 31).

A message listener is not specific to a particular destination type. The same listener can obtain messages from either a queue or a topic, depending on whether the listener is set by a `QueueReceiver` or a `TopicSubscriber` object. A message listener does, however, usually expect a specific message type and format. Moreover, if it needs to reply to messages, a message listener must either assume a particular destination type or obtain the destination type of the message and create a producer for that destination type.

Your `onMessage` method should handle all exceptions. It must not throw checked exceptions, and throwing a `RuntimeException`, though possible, is considered a programming error.

The session used to create the message consumer serializes the execution of all message listeners registered with the session. At any time, only one of the session's message listeners is running.

In the J2EE 1.3 platform, a message-driven bean is a special kind of message listener. For details, see Section 6.2 on page 75.

3.5.2 Message Selectors

If your messaging application needs to filter the messages it receives, you can use a JMS API message selector, which allows a message consumer to specify the messages it is interested in. Message selectors assign the work of filtering messages to the JMS provider rather than to the application. For an example of the use of a message selector, see Chapter 8.

A message selector is a `String` that contains an expression. The syntax of the expression is based on a subset of the SQL92 conditional expression syntax. The `createReceiver`, `createSubscriber`, and `createDurableSubscriber` methods each have a form that allows you to specify a message selector as an argument when you create a message consumer.

The message consumer then receives only messages whose headers and properties match the selector. (See Section 3.6.1 on page 29 and Section 3.6.2 on page 30.) A message selector cannot select messages on the basis of the content of the message body.

3.6 Messages

The ultimate purpose of a JMS application is to produce and to consume messages that can then be used by other software applications. JMS messages have a basic format that is simple but highly flexible, allowing you to create messages that match formats used by non-JMS applications on heterogeneous platforms.

A JMS message has three parts:

- A header
- Properties (optional)
- A body (optional)

For complete documentation of message headers, properties, and bodies, see the documentation of the `Message` interface in Chapter 25.

3.6.1 Message Headers

A JMS message header contains a number of predefined fields that contain values that both clients and providers use to identify and to route messages. (Table 3.1 lists the JMS message header fields and indicates how their values are set.) For example, every message has a unique identifier, represented in the header field `JMSMessageID`. The value of another header field, `JMSDestination`, represents the queue or the topic to which the message is sent. Other fields include a timestamp and a priority level.

Each header field has associated setter and getter methods, which are documented in the description of the `Message` interface. Some header fields are intended to be set by a client, but many are set automatically by the `send` or the `publish` method, which overrides any client-set values.

Table 3.1: How JMS Message Header Field Values Are Set

Header Field	**Set By**
`JMSDestination`	`send` or `publish` method
`JMSDeliveryMode`	`send` or `publish` method
`JMSExpiration`	`send` or `publish` method
`JMSPriority`	`send` or `publish` method
`JMSMessageID`	`send` or `publish` method
`JMSTimestamp`	`send` or `publish` method
`JMSCorrelationID`	Client
`JMSReplyTo`	Client
`JMSType`	Client
`JMSRedelivered`	JMS provider

3.6.2 Message Properties

You can create and set properties for messages if you need values in addition to those provided by the header fields. You can use properties to provide compatibility with other messaging systems, or you can use them to create message selectors (see Section 3.5.2 on page 29). For an example of setting a property to be used as a message selector, see Section 8.1.2.3 on page 108.

The JMS API provides some predefined property names that a provider may support. The use of either predefined properties or user-defined properties is optional.

3.6.3 Message Bodies

The JMS API defines five message body formats, also called message types, which allow you to send and to receive data in many different forms and provide compatibility with existing messaging formats. Table 3.2 describes these message types.

Table 3.2: JMS Message Types

Message Type	**Body Contains**
`TextMessage`	A `java.lang.String` object (for example, the contents of an Extensible Markup Language file).
`MapMessage`	A set of name/value pairs, with names as `String` objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
`BytesMessage`	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
`StreamMessage`	A stream of primitive values in the Java programming language, filled and read sequentially.
`ObjectMessage`	A `Serializable` object in the Java programming language.
`Message`	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

The JMS API provides methods for creating messages of each type and for filling in their contents. For example, to create and send a `TextMessage` to a queue, you might use the following statements:

```
TextMessage message = queueSession.createTextMessage();
message.setText(msg_text);     // msg_text is a String
queueSender.send(message);
```

At the consuming end, a message arrives as a generic `Message` object and must be cast to the appropriate message type. You can use one or more getter methods to extract the message contents. The following code fragment uses the `getText` method:

```
Message m = queueReceiver.receive();
if (m instanceof TextMessage) {
    TextMessage message = (TextMessage) m;
    System.out.println("Reading message: " + message.getText());
} else {
    // Handle error
}
```

3.7 Exception Handling

The root class for exceptions thrown by JMS API methods is `JMSException`. Catching `JMSException` provides a generic way of handling all exceptions related to the JMS API. The `JMSException` class includes the following subclasses:

- `IllegalStateException`
- `InvalidClientIDException`
- `InvalidDestinationException`
- `InvalidSelectorException`
- `JMSSecurityException`
- `MessageEOFException`
- `MessageFormatException`
- `MessageNotReadableException`
- `MessageNotWriteableException`
- `ResourceAllocationException`
- `TransactionInProgressException`
- `TransactionRolledBackException`

All the examples in this book catch and handle `JMSException` when it is appropriate to do so.

CHAPTER 4

Writing Simple JMS Client Applications

THIS chapter shows how to create and to run simple JMS client programs. A J2EE application client commonly accesses J2EE components installed in a server based on J2EE technology (“J2EE server”). The clients in this chapter, however, are simple standalone programs that run outside the server as class files. The clients demonstrate the basic tasks that a JMS application must perform:

- Creating a connection and a session
- Creating message producers and consumers
- Sending and receiving messages

In a J2EE application, some of these tasks are performed, in whole or in part, by the EJB container. If you learn about these tasks, you will have a good basis for understanding how a JMS application works on the J2EE platform.

The chapter covers the following topics:

- Setting your environment to run J2EE clients and applications
- A point-to-point example that uses synchronous receives
- A publish/subscribe example that uses a message listener
- Running JMS client programs on multiple systems

Each example consists of two programs: one that sends messages and one that receives them. You can run the programs in two terminal windows.

When you write a JMS application to run in a J2EE component, you use many of the same methods in much the same sequence as you do for a JMS client program. However, there are some significant differences. Chapter 6 describes these differences, and the following chapters provide examples that illustrate them.

4.1 Setting Your Environment for Running Applications

Before you can run the examples, you need to make sure that your environment is set appropriately. Table 4.1 shows how to set the environment variables needed to run J2EE applications on Microsoft Windows and UNIX platforms.

Table 4.1: Environment Settings for Compiling and Running J2EE Applications

Platform	Variable Name	Values
Microsoft Windows	`%JAVA_HOME%`	Directory in which the Java 2 SDK, Standard Edition, version 1.3.1, is installed
	`%J2EE_HOME%`	Directory in which the J2EE SDK 1.3.1 is installed, usually `C:\j2sdkee1.3.1`
	`%CLASSPATH%`	Include the following: `.;%J2EE_HOME%\lib\j2ee.jar;` `%J2EE_HOME%\lib\locale`
	`%PATH%`	Include `%J2EE_HOME%\bin`
UNIX	`$JAVA_HOME`	Directory in which the Java 2 SDK, Standard Edition, version 1.3.1, is installed
	`$J2EE_HOME`	Directory in which the J2EE SDK 1.3.1 is installed, usually `$HOME/j2sdkee1.3.1`
	`$CLASSPATH`	Include the following: `.:$J2EE_HOME/lib/j2ee.jar:` `$J2EE_HOME/lib/locale`
	`$PATH`	Include `$J2EE_HOME/bin`

The appendix provides more examples of client programs that demonstrate additional features of the JMS API. You can download still more examples of JMS client programs from the JMS API Web site, `http://java.sun.com/products/jms/`. If you downloaded the tutorial examples as described in the preface, you will find the examples for this chapter in the directory `jms_tutorial/examples/simple` (on UNIX systems) or `jms_tutorial\examples\simple` (on Microsoft Windows systems).

4.2 A Simple Point-to-Point Example

This section describes the sending and receiving programs in a PTP example that uses the `receive` method to consume messages synchronously. This section then explains how to compile and run the programs, using the J2EE SDK 1.3.1.

4.2.1 Writing the PTP Client Programs

The sending program, `SimpleQueueSender.java`, performs the following steps:

1. Performs a Java Naming and Directory Interface (JNDI) API lookup of the `QueueConnectionFactory` and queue
2. Creates a connection and a session
3. Creates a `QueueSender`
4. Creates a `TextMessage`
5. Sends one or more messages to the queue
6. Sends a control message to indicate the end of the message stream
7. Closes the connection in a `finally` block, automatically closing the session and `QueueSender`

The receiving program, `SimpleQueueReceiver.java`, performs the following steps:

1. Performs a JNDI API lookup of the `QueueConnectionFactory` and queue
2. Creates a connection and a session
3. Creates a `QueueReceiver`

4. Starts the connection, causing message delivery to begin
5. Receives the messages sent to the queue until the end-of-message-stream control message is received
6. Closes the connection in a `finally` block, automatically closing the session and `QueueReceiver`

The `receive` method can be used in several ways to perform a synchronous receive. If you specify no arguments or an argument of `0`, the method blocks indefinitely until a message arrives:

```
Message m = queueReceiver.receive();

Message m = queueReceiver.receive(0);
```

For a simple client program, this may not matter. But if you do not want your program to consume system resources unnecessarily, use a timed synchronous receive. Do one of the following:

- Call the `receive` method with a timeout argument greater than `0`:

  ```
  Message m = queueReceiver.receive(1); // 1 millisecond
  ```

- Call the `receiveNoWait` method, which receives a message only if one is available:

  ```
  Message m = queueReceiver.receiveNoWait();
  ```

The `SimpleQueueReceiver` program uses an indefinite `while` loop to receive messages, calling `receive` with a timeout argument. Calling `receiveNoWait` would have the same effect.

The following subsections show the two programs:

- `SimpleQueueSender.java`
- `SimpleQueueReceiver.java`

4.2.1.1 Sending Messages to a Queue: `SimpleQueueSender.java`

The sending program is `SimpleQueueSender.java`.

```
/**
 * The SimpleQueueSender class consists only of a main method,
 * which sends several messages to a queue.
 *
 * Run this program in conjunction with SimpleQueueReceiver.
 * Specify a queue name on the command line when you run the
 * program.  By default, the program sends one message.  Specify
 * a number after the queue name to send that number of messages.
 */
import javax.jms.*;
import javax.naming.*;

public class SimpleQueueSender {

    /**
     * Main method.
     *
     * @param args     the queue used by the example and,
     *                 optionally, the number of messages to send
     */
    public static void main(String[] args) {
        String                  queueName = null;
        Context                 jndiContext = null;
        QueueConnectionFactory  queueConnectionFactory = null;
        QueueConnection         queueConnection = null;
        QueueSession            queueSession = null;
        Queue                   queue = null;
        QueueSender             queueSender = null;
        TextMessage             message = null;
        final int               NUM_MSGS;
```

```
if ( (args.length < 1) || (args.length > 2) ) {
    System.out.println("Usage: java SimpleQueueSender " +
        "<queue-name> [<number-of-messages>]");
    System.exit(1);
}

queueName = new String(args[0]);
System.out.println("Queue name is " + queueName);
if (args.length == 2){
    NUM_MSGS = (new Integer(args[1])).intValue();
} else {
    NUM_MSGS = 1;
}

/*
 * Create a JNDI API InitialContext object if none exists
 * yet.
 */
try {
    jndiContext = new InitialContext();
} catch (NamingException e) {
    System.out.println("Could not create JNDI API " +
        "context: " + e.toString());
    System.exit(1);
}

/*
 * Look up connection factory and queue.  If either does
 * not exist, exit.
 */
try {
    queueConnectionFactory = (QueueConnectionFactory)
        jndiContext.lookup("QueueConnectionFactory");
    queue = (Queue) jndiContext.lookup(queueName);
} catch (NamingException e) {
    System.out.println("JNDI API lookup failed: " +
        e.toString());
    System.exit(1);
```

```
}

/*
 * Create connection.
 * Create session from connection; false means session is
 * not transacted.
 * Create sender and text message.
 * Send messages, varying text slightly.
 * Send end-of-messages message.
 * Finally, close connection.
 */
try {
    queueConnection =
        queueConnectionFactory.createQueueConnection();
    queueSession =
        queueConnection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);
    queueSender = queueSession.createSender(queue);
    message = queueSession.createTextMessage();
    for (int i = 0; i < NUM_MSGS; i++) {
        message.setText("This is message " + (i + 1));
        System.out.println("Sending message: " +
            message.getText());
        queueSender.send(message);
    }

    /*
     * Send a non-text control message indicating end of
     * messages.
     */
    queueSender.send(queueSession.createMessage());
} catch (JMSException e) {
    System.out.println("Exception occurred: " +
        e.toString());
} finally {
    if (queueConnection != null) {
        try {
            queueConnection.close();
        } catch (JMSException e) {}
```

```
                }
            }
        }
    }
```

Code Example 4.1 `SimpleQueueSender.java`

4.2.1.2 Receiving Messages from a Queue: `SimpleQueueReceiver.java`

The receiving program is `SimpleQueueReceiver.java`.

```
    /**
     * The SimpleQueueReceiver class consists only of a main method,
     * which fetches one or more messages from a queue using
     * synchronous message delivery.  Run this program in conjunction
     * with SimpleQueueSender.  Specify a queue name on the command
     * line when you run the program.
     */
    import javax.jms.*;
    import javax.naming.*;

    public class SimpleQueueReceiver {

        /**
         * Main method.
         *
         * @param args     the queue used by the example
         */
        public static void main(String[] args) {
            String                  queueName = null;
            Context                 jndiContext = null;
            QueueConnectionFactory  queueConnectionFactory = null;
            QueueConnection         queueConnection = null;
            QueueSession            queueSession = null;
            Queue                   queue = null;
```

```
QueueReceiver           queueReceiver = null;
TextMessage             message = null;

/*
 * Read queue name from command line and display it.
 */
if (args.length != 1) {
    System.out.println("Usage: java " +
        "SimpleQueueReceiver <queue-name>");
    System.exit(1);
}
queueName = new String(args[0]);
System.out.println("Queue name is " + queueName);

/*
 * Create a JNDI API InitialContext object if none exists
 * yet.
 */
try {
    jndiContext = new InitialContext();
} catch (NamingException e) {
    System.out.println("Could not create JNDI API " +
        "context: " + e.toString());
    System.exit(1);
}

/*
 * Look up connection factory and queue.  If either does
 * not exist, exit.
 */
try {
    queueConnectionFactory = (QueueConnectionFactory)
        jndiContext.lookup("QueueConnectionFactory");
    queue = (Queue) jndiContext.lookup(queueName);
} catch (NamingException e) {
    System.out.println("JNDI API lookup failed: " +
        e.toString());
    System.exit(1);
}
```

```
/*
 * Create connection.
 * Create session from connection; false means session is
 * not transacted.
 * Create receiver, then start message delivery.
 * Receive all text messages from queue until
 * a non-text message is received indicating end of
 * message stream.
 * Close connection.
 */
try {
    queueConnection =
        queueConnectionFactory.createQueueConnection();
    queueSession =
        queueConnection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);
    queueReceiver = queueSession.createReceiver(queue);
    queueConnection.start();
    while (true) {
        Message m = queueReceiver.receive(1);
        if (m != null) {
            if (m instanceof TextMessage) {
                message = (TextMessage) m;
                System.out.println("Reading message: " +
                    message.getText());
            } else {
                break;
            }
        }
    }
} catch (JMSException e) {
    System.out.println("Exception occurred: " +
        e.toString());
} finally {
    if (queueConnection != null) {
        try {
            queueConnection.close();
        } catch (JMSException e) {}
    }
```

```
            }
        }
    }
```

Code Example 4.2 `SimpleQueueReceiver.java`

4.2.2 Compiling the PTP Clients

To compile the PTP example, do the following.

1. Make sure that you have set the environment variables shown in Table 4.1 on page 34.
2. At a command line prompt, compile the two source files:

```
javac SimpleQueueSender.java
javac SimpleQueueReceiver.java
```

4.2.3 Starting the JMS Provider

When you use the J2EE SDK 1.3.1, your JMS provider is the SDK. At another command line prompt, start the J2EE server as follows:

```
j2ee -verbose
```

Wait until the server displays the message “J2EE server startup complete.”

4.2.4 Creating the JMS Administered Objects

In the window in which you compiled the clients, use the `j2eeadmin` command to create a queue named `MyQueue`. The last argument tells the command what kind of destination to create.

```
j2eeadmin -addJmsDestination MyQueue queue
```

To make sure that the queue has been created, use the following command:

```
j2eeadmin -listJmsDestination
```

This example uses the default `QueueConnectionFactory` object supplied with the J2EE SDK 1.3.1. With a different J2EE product, you might need to create a connection factory yourself.

4.2.5 Running the PTP Clients

Run the clients as follows.

1. Run the `SimpleQueueSender` program, sending three messages. You need to define a value for `jms.properties`.

   - On a Microsoft Windows system, type the following command on a single line:

   ```
   java -Djms.properties=%J2EE_HOME%\config\jms_client.properties
   SimpleQueueSender MyQueue 3
   ```

   - On a UNIX system, type the following command on a single line:

   ```
   java -Djms.properties=$J2EE_HOME/config/jms_client.properties
   SimpleQueueSender MyQueue 3
   ```

   The output of the program looks like this:

   ```
   Queue name is MyQueue
   Sending message: This is message 1
   Sending message: This is message 2
   Sending message: This is message 3
   ```

2. In the same window, run the `SimpleQueueReceiver` program, specifying the queue name. The `java` commands look like this:

   - Microsoft Windows systems:

   ```
   java -Djms.properties=%J2EE_HOME%\config\jms_client.properties
   SimpleQueueReceiver MyQueue
   ```

   - UNIX systems:

   ```
   java -Djms.properties=$J2EE_HOME/config/jms_client.properties
   SimpleQueueReceiver MyQueue
   ```

The output of the program looks like this:

```
Queue name is MyQueue
Reading message: This is message 1
Reading message: This is message 2
Reading message: This is message 3
```

3. Now try running the programs in the opposite order. Start the `SimpleQueue-Receiver` program. It displays the queue name and then appears to hang, waiting for messages.
4. In a different terminal window, run the `SimpleQueueSender` program. When the messages have been sent, the `SimpleQueueReceiver` program receives them and exits.

4.2.6 Deleting the Queue

You can delete the queue you created as follows:

```
j2eeadmin -removeJmsDestination MyQueue
```

You will use it again in Section 4.4.1 on page 57, however.

4.3 A Simple Publish/Subscribe Example

This section describes the publishing and subscribing programs in a pub/sub example that uses a message listener to consume messages asynchronously. This section then explains how to compile and run the programs, using the J2EE SDK 1.3.1.

4.3.1 Writing the Pub/Sub Client Programs

The publishing program, `SimpleTopicPublisher.java`, performs the following steps:

1. Performs a JNDI API lookup of the `TopicConnectionFactory` and topic
2. Creates a connection and a session
3. Creates a `TopicPublisher`

4. Creates a `TextMessage`
5. Publishes one or more messages to the topic
6. Closes the connection, which automatically closes the session and `TopicPublisher`

The receiving program, `SimpleTopicSubscriber.java`, performs the following steps:

1. Performs a JNDI API lookup of the `TopicConnectionFactory` and topic
2. Creates a connection and a session
3. Creates a `TopicSubscriber`
4. Creates an instance of the `TextListener` class and registers it as the message listener for the `TopicSubscriber`
5. Starts the connection, causing message delivery to begin
6. Listens for the messages published to the topic, stopping when the user enters the character `q` or `Q`
7. Closes the connection, which automatically closes the session and `TopicSubscriber`

The message listener, `TextListener.java`, follows these steps:

1. When a message arrives, the `onMessage` method is called automatically.
2. The `onMessage` method converts the incoming message to a `TextMessage` and displays its content.

The following subsections show the three source files:

- `SimpleTopicPublisher.java`
- `SimpleTopicSubscriber.java`
- `TextListener.java`

4.3.1.1 Publishing Messages to a Topic: `SimpleTopicPublisher.java`

The publisher program is `SimpleTopicPublisher.java`.

```
/**
 * The SimpleTopicPublisher class consists only of a main method,
 * which publishes several messages to a topic.
 *
 * Run this program in conjunction with SimpleTopicSubscriber.
 * Specify a topic name on the command line when you run the
 * program.  By default, the program sends one message.
 * Specify a number after the topic name to send that number
 * of messages.
 */
import javax.jms.*;
import javax.naming.*;

public class SimpleTopicPublisher {

    /**
     * Main method.
     *
     * @param args     the topic used by the example and,
     *                 optionally, the number of messages to send
     */
    public static void main(String[] args) {
        String                  topicName = null;
        Context                 jndiContext = null;
        TopicConnectionFactory  topicConnectionFactory = null;
        TopicConnection         topicConnection = null;
        TopicSession            topicSession = null;
        Topic                   topic = null;
        TopicPublisher          topicPublisher = null;
        TextMessage             message = null;
        final int               NUM_MSGS;

        if ( (args.length < 1) || (args.length > 2) ) {
            System.out.println("Usage: java " +
            "SimpleTopicPublisher <topic-name> " +
            "[<number-of-messages>]");
        System.exit(1);
    }
    topicName = new String(args[0]);
    System.out.println("Topic name is " + topicName);
    if (args.length == 2){
        NUM_MSGS = (new Integer(args[1])).intValue();
    } else {
        NUM_MSGS = 1;
    }

    /*
     * Create a JNDI API InitialContext object if none exists
     * yet.
     */
    try {
        jndiContext = new InitialContext();
    } catch (NamingException e) {
        System.out.println("Could not create JNDI API " +
            "context: " + e.toString());
        e.printStackTrace();
        System.exit(1);
    }

    /*
     * Look up connection factory and topic.  If either does
     * not exist, exit.
     */
    try {
        topicConnectionFactory = (TopicConnectionFactory)
            jndiContext.lookup("TopicConnectionFactory");
        topic = (Topic) jndiContext.lookup(topicName);
    } catch (NamingException e) {
        System.out.println("JNDI API lookup failed: " +
            e.toString());
        e.printStackTrace();
        System.exit(1);
    }
```

```
            /*
             * Create connection.
             * Create session from connection; false means session is
             * not transacted.
             * Create publisher and text message.
             * Send messages, varying text slightly.
             * Finally, close connection.
             */
            try {
                topicConnection =
                    topicConnectionFactory.createTopicConnection();
                topicSession =
                    topicConnection.createTopicSession(false,
                        Session.AUTO_ACKNOWLEDGE);
                topicPublisher = topicSession.createPublisher(topic);
                message = topicSession.createTextMessage();
                for (int i = 0; i < NUM_MSGS; i++) {
                    message.setText("This is message " + (i + 1));
                    System.out.println("Publishing message: " +
                        message.getText());
                    topicPublisher.publish(message);
                }
            } catch (JMSException e) {
                System.out.println("Exception occurred: " +
                    e.toString());
            } finally {
                if (topicConnection != null) {
                    try {
                        topicConnection.close();
                    } catch (JMSException e) {}
                }
            }
        }
    }
```

Code Example 4.3 `SimpleTopicPublisher.java`

4.3.1.2 Receiving Messages Asynchronously: `SimpleTopicSubscriber.java`

The subscriber program is `SimpleTopicSubscriber.java`.

```
/**
 * The SimpleTopicSubscriber class consists only of a main
 * method, which receives one or more messages from a topic using
 * asynchronous message delivery.  It uses the message listener
 * TextListener.  Run this program in conjunction with
 * SimpleTopicPublisher.
 *
 * Specify a topic name on the command line when you run the
 * program. To end the program, enter Q or q on the command line.
 */
import javax.jms.*;
import javax.naming.*;
import java.io.*;

public class SimpleTopicSubscriber {

    /**
     * Main method.
     *
     * @param args     the topic used by the example
     */
    public static void main(String[] args) {
        String                   topicName = null;
        Context                  jndiContext = null;
        TopicConnectionFactory   topicConnectionFactory = null;
        TopicConnection          topicConnection = null;
        TopicSession             topicSession = null;
        Topic                    topic = null;
        TopicSubscriber          topicSubscriber = null;
        TextListener             topicListener = null;
        TextMessage              message = null;
        InputStreamReader        inputStreamReader = null;
        char                     answer = '\0';
```

```
/*
 * Read topic name from command line and display it.
 */
if (args.length != 1) {
    System.out.println("Usage: java " +
        "SimpleTopicSubscriber <topic-name>");
    System.exit(1);
}
topicName = new String(args[0]);
System.out.println("Topic name is " + topicName);

/*
 * Create a JNDI API InitialContext object if none exists
 * yet.
 */
try {
    jndiContext = new InitialContext();
} catch (NamingException e) {
    System.out.println("Could not create JNDI API " +
        "context: " + e.toString());
    e.printStackTrace();
    System.exit(1);
}

/*
 * Look up connection factory and topic.  If either does
 * not exist, exit.
 */
try {
    topicConnectionFactory = (TopicConnectionFactory)
        jndiContext.lookup("TopicConnectionFactory");
    topic = (Topic) jndiContext.lookup(topicName);
} catch (NamingException e) {
    System.out.println("JNDI API lookup failed: " +
        e.toString());
    e.printStackTrace();
    System.exit(1);
}
```

```
/*
 * Create connection.
 * Create session from connection; false means session is
 * not transacted.
 * Create subscriber.
 * Register message listener (TextListener).
 * Receive text messages from topic.
 * When all messages have been received, enter Q to quit.
 * Close connection.
 */
try {
    topicConnection =
        topicConnectionFactory.createTopicConnection();
    topicSession =
        topicConnection.createTopicSession(false,
            Session.AUTO_ACKNOWLEDGE);
    topicSubscriber =
        topicSession.createSubscriber(topic);
    topicListener = new TextListener();
    topicSubscriber.setMessageListener(topicListener);
    topicConnection.start();
    System.out.println("To end program, enter Q or q, " +
        "then <return>");
    inputStreamReader = new InputStreamReader(System.in);
    while (!((answer == 'q') || (answer == 'Q'))) {
        try {
            answer = (char) inputStreamReader.read();
        } catch (IOException e) {
            System.out.println("I/O exception: "
                + e.toString());
        }
    }
} catch (JMSException e) {
    System.out.println("Exception occurred: " +
        e.toString());
} finally {
    if (topicConnection != null) {
        try {
            topicConnection.close();
```

```
                    } catch (JMSException e) {}
                }
            }
        }
    }
```

Code Example 4.4 `SimpleTopicSubscriber.java`

4.3.1.3 The Message Listener: `TextListener.java`

The message listener is `TextListener.java`.

```
/**
 * The TextListener class implements the MessageListener
 * interface by defining an onMessage method that displays
 * the contents of a TextMessage.
 *
 * This class acts as the listener for the SimpleTopicSubscriber
 * class.
 */
import javax.jms.*;

public class TextListener implements MessageListener {

    /**
     * Casts the message to a TextMessage and displays its text.
     *
     * @param message     the incoming message
     */

    public void onMessage(Message message) {
        TextMessage msg = null;

        try {
            if (message instanceof TextMessage) {
                msg = (TextMessage) message;
```

```
                    System.out.println("Reading message: " +
                        msg.getText());
                } else {
                    System.out.println("Message of wrong type: " +
                        message.getClass().getName());
                }
            } catch (JMSException e) {
                System.out.println("JMSException in onMessage(): " +
                    e.toString());
            } catch (Throwable t) {
                System.out.println("Exception in onMessage():" +
                    t.getMessage());
            }
        }
    }
```

Code Example 4.5 `TextListener.java`

4.3.2 Compiling the Pub/Sub Clients

To compile the pub/sub example, do the following.

1. Make sure that you have set the environment variables shown in Table 4.1 on page 34.
2. Compile the programs and the message listener class:

```
javac SimpleTopicPublisher.java
javac SimpleTopicSubscriber.java
javac TextListener.java
```

4.3.3 Starting the JMS Provider

If you did not do so before, start the J2EE server in another terminal window:

```
j2ee -verbose
```

Wait until the server displays the message “J2EE server startup complete.”

4.3.4 Creating the JMS Administered Objects

In the window in which you compiled the clients, use the `j2eeadmin` command to create a topic named `MyTopic`. The last argument tells the command what kind of destination to create.

```
j2eeadmin -addJmsDestination MyTopic topic
```

To verify that the queue has been created, use the following command:

```
j2eeadmin -listJmsDestination
```

This example uses the default `TopicConnectionFactory` object supplied with the J2EE SDK 1.3.1. With a different J2EE product, you might need to create a connection factory.

4.3.5 Running the Pub/Sub Clients

Run the clients as follows.

1. Run the `SimpleTopicSubscriber` program, specifying the topic name. You need to define a value for `jms.properties`.

   - On a Microsoft Windows system, type the following command on a single line:

```
java -Djms.properties=%J2EE_HOME%\config\jms_client.properties
SimpleTopicSubscriber MyTopic
```

   - On a UNIX system, type the following command on a single line:

```
java -Djms.properties=$J2EE_HOME/config/jms_client.properties
SimpleTopicSubscriber MyTopic
```

   The program displays the following lines and appears to hang:

```
Topic name is MyTopic
To end program, enter Q or q, then <return>
```

2. In another terminal window, run the `SimpleTopicPublisher` program, publishing three messages. The `java` commands look like this:
   - Microsoft Windows systems:

   ```
   java -Djms.properties=%J2EE_HOME%\config\jms_client.properties
   SimpleTopicPublisher MyTopic 3
   ```

   - UNIX systems:

   ```
   java -Djms.properties=$J2EE_HOME/config/jms_client.properties
   SimpleTopicPublisher MyTopic 3
   ```

   The output of the program looks like this:

   ```
   Topic name is MyTopic
   Publishing message: This is message 1
   Publishing message: This is message 2
   Publishing message: This is message 3
   ```

   In the other window, the program displays the following:

   ```
   Reading message: This is message 1
   Reading message: This is message 2
   Reading message: This is message 3
   ```

   Enter `Q` or `q` to stop the program.

4.3.6 Deleting the Topic and Stopping the Server

1. You can delete the topic you created as follows:

   ```
   j2eeadmin -removeJmsDestination MyTopic
   ```

   You will use it again in Section 4.4.2 on page 58, however.
2. If you wish, you can stop the J2EE server as well:

   ```
   j2ee -stop
   ```

4.4 Running JMS Client Programs on Multiple Systems

JMS client programs can communicate with each other when they are running on different systems in a network. The systems must be visible to each other by name—the UNIX host name or the Microsoft Windows computer name—and must both be running the J2EE server.

This section explains how to produce and to consume messages in two different situations:

- When a J2EE server is running on both systems
- When a J2EE server is running on only one system

4.4.1 Communicating Between Two J2EE Servers

Suppose that you want to run the `SimpleQueueSender` program on one system, `mars`, and the `SimpleQueueReceiver` program on another system, `earth`. To do so, follow these steps.

1. Start the J2EE server on both systems. Enter the following command in a terminal window on each system:

   ```
   j2ee -verbose
   ```

2. On `earth`, create a `QueueConnectionFactory` object, using a command like the following:

   ```
   j2eeadmin -addJmsFactory jms/EarthQCF queue
   ```

3. On `mars`, create a connection factory with the same name that points to the server on `earth`. Enter, on one line, a command like the following:

   ```
   j2eeadmin -addJmsFactory jms/EarthQCF queue -props
   url=corbaname:iiop:earth:1050#earth
   ```

4. In each source program, change the line that looks up the connection factory so that it refers to the new connection factory:

   ```
   queueConnectionFactory =
       (QueueConnectionFactory) jndiContext.lookup("jms/EarthQCF");
   ```

5. Recompile the programs; then run them by using the instructions in Section 4.2.5 on page 44. Because both connection factories have the same name, you can run either the sender or the receiver on either system. (Note: A bug in the JMS provider in the J2EE SDK may cause a runtime failure to create a connection to systems that use the Dynamic Host Configuration Protocol [DHCP] to obtain an IP address.)

You can run the pub/sub example in the same way by creating a `TopicConnectionFactory` object on both systems. For an example showing how to deploy J2EE applications on two different systems, see Chapter 10.

4.4.2 Communicating Between a J2EE Server and a System Not Running a J2EE Server

In order for two standalone client programs to communicate, both must have the J2EE SDK installed locally. However, the J2EE server does not have to be running on both systems. Suppose that you want to run the `SimpleTopicPublisher` and the `SimpleTopicSubscriber` programs on two systems called `earth` and `mars`, as in Section 4.4.1, but that the J2EE server will be running only on `earth`. To specify the system running the server, you can either

- Use the command line, which allows you to access different applications on different servers for maximum flexibility
- Set a configurable property, which allows applications to run only on the system specified in the property

When the server is running only on the remote system, you do *not* have to create a connection factory on the local system that refers to the remote system.

The procedure for using the command line is as follows:

1. Start the J2EE server on `earth`:

   ```
   j2ee -verbose
   ```

2. Set the `J2EE_HOME` environment variable and the classpath on `mars` so that they point to the J2EE SDK 1.3.1 installation on `mars` (see Table 4.1 on page 34).

3. To access a client program on a system running the server from a client program on a system not running the server, use the following option, where `*hostname*` is the name of the system running the J2EE server:

   ```
   -Dorg.omg.CORBA.ORBInitialHost=hostname
   ```

   This option allows you to access the naming service on the remote system. For example, if the server is running on `earth`, use a command like the following to run the `SimpleTopicSubscriber` program on `mars`. Make sure that the destination you specify exists on the server running on `earth`.

   - On a Microsoft Windows system, type the following command on a single line:

   ```
   java -Djms.properties=%J2EE_HOME%\config\jms_client.properties
   -Dorg.omg.CORBA.ORBInitialHost=earth SimpleTopicSubscriber MyTopic
   ```

   - On a UNIX system, type the following command on a single line:

   ```
   java -Djms.properties=$J2EE_HOME/config/jms_client.properties
   -Dorg.omg.CORBA.ORBInitialHost=earth SimpleTopicSubscriber MyTopic
   ```

If all the remote programs you need to access are on the same system, you can edit the file `%J2EE_HOME%\config\orb.properties` (on Microsoft Windows systems) or `$J2EE_HOME/config/orb.properties` (on UNIX systems) on the local system. The second line of this file looks like this:

```
host=localhost
```

Change `localhost` to the name of the system on which the J2EE server is running—for example, `earth`:

```
host=earth
```

You can now run the client program as before, but you do not need to specify the option `-Dorg.omg.CORBA.ORBInitialHost`.

CHAPTER 5

Creating Robust JMS Applications

THIS chapter explains how to use features of the JMS API to achieve the level of reliability and performance your application requires. Many JMS applications cannot tolerate dropped or duplicate messages and require that every message be received once and only once.

The most reliable way to produce a message is to send a `PERSISTENT` message within a transaction. JMS messages are `PERSISTENT` by default. A *transaction* is a unit of work into which you can group a series of operations, such as message sends and receives, so that the operations either all succeed or all fail. For details, see Section 5.1.2 on page 64 and Section 5.2.2 on page 70.

The most reliable way to consume a message is to do so within a transaction, either from a nontemporary queue—in the PTP messaging domain—or from a durable subscription—in the pub/sub messaging domain. For details, see Section 5.1.5 on page 66, Section 5.2.1 on page 67, and Section 5.2.2 on page 70.

For other applications, a lower level of reliability can reduce overhead and improve performance. You can send messages with varying priority levels—see Section 5.1.3 on page 65—and you can set them to expire after a certain length of time (see Section 5.1.4 on page 65).

The JMS API provides several ways to achieve various kinds and degrees of reliability. This chapter divides them into two categories:

- Basic reliability mechanisms
- Advanced reliability mechanisms

The following sections describe these features as they apply to JMS clients. Some of the features work differently in J2EE applications; in these cases, the differences are noted here and are explained in detail in Chapter 6.

5.1 Using Basic Reliability Mechanisms

The basic mechanisms for achieving or affecting reliable message delivery are as follows:

- **Controlling message acknowledgment.** You can specify various levels of control over message acknowledgment.
- **Specifying message persistence.** You can specify that messages are persistent, meaning that they must not be lost in the event of a provider failure.
- **Setting message priority levels.** You can set various priority levels for messages, which can affect the order in which the messages are delivered.
- **Allowing messages to expire.** You can specify an expiration time for messages, so that they will not be delivered if they are obsolete.
- **Creating temporary destinations.** You can create temporary destinations that last only for the duration of the connection in which they are created.

5.1.1 Controlling Message Acknowledgment

Until a JMS message has been acknowledged, it is not considered to be successfully consumed. The successful consumption of a message ordinarily takes place in three stages.

1. The client receives the message.
2. The client processes the message.
3. The message is acknowledged. Acknowledgment is initiated either by the JMS provider or by the client, depending on the session acknowledgment mode.

In transacted sessions (see Section 5.2.2 on page 70), acknowledgment happens automatically when a transaction is committed. If a transaction is rolled back, all consumed messages are redelivered.

In nontransacted sessions, when and how a message is acknowledged depends on the value specified as the second argument of the `createQueueSession` or `createTopicSession` method. The three possible argument values are:

- `Session.AUTO_ACKNOWLEDGE`. The session automatically acknowledges a client's receipt of a message either when the client has successfully returned from a call to `receive` or when the `MessageListener` it has called to process the message returns successfully. A synchronous receive in an `AUTO_ACKNOWLEDGE` session is the one exception to the rule that message consumption is a three-stage process. In this case, the receipt and acknowledgment take place in one step, followed by the processing of the message.
- `Session.CLIENT_ACKNOWLEDGE`. A client acknowledges a message by calling the message's `acknowledge` method. In this mode, acknowledgment takes place on the session level: Acknowledging a consumed message automatically acknowledges the receipt of all messages that have been consumed by its session. For example, if a message consumer consumes ten messages and then acknowledges the fifth message delivered, all ten messages are acknowledged.
- `Session.DUPS_OK_ACKNOWLEDGE`. This option instructs the session to lazily acknowledge the delivery of messages. This is likely to result in the delivery of some duplicate messages if the JMS provider fails, so it should be used only by consumers that can tolerate duplicate messages. (If it redelivers a message, the JMS provider must set the value of the `JMSRedelivered` message header to `true`.) This option can reduce session overhead by minimizing the work the session does to prevent duplicates.

If messages have been received but not acknowledged when a `QueueSession` terminates, the JMS provider retains them and redelivers them when a consumer next accesses the queue. The provider also retains unacknowledged messages for a terminated `TopicSession` with a durable `TopicSubscriber`. (See Section 5.2.1 on page 67.) Unacknowledged messages for a nondurable `TopicSubscriber` are dropped when the session is closed.

If you use a queue or a durable subscription, you can use the `Session.recover` method to stop a nontransacted session and restart it with its first unacknowledged message. In effect, the session's series of delivered messages is reset to the point after its last acknowledged message. The messages it now delivers may be different from those that were originally delivered, if messages have expired or higher-priority messages have arrived. For a nondurable `TopicSubscriber`, the provider may drop unacknowledged messages when its session is recovered.

The sample program in Section A.3 on page 250 demonstrates two ways to ensure that a message will not be acknowledged until processing of the message is complete.

5.1.2 Specifying Message Persistence

The JMS API supports two delivery modes for messages to specify whether messages are lost if the JMS provider fails. These delivery modes are fields of the `DeliveryMode` interface.

- The `PERSISTENT` delivery mode, which is the default, instructs the JMS provider to take extra care to ensure that a message is not lost in transit in case of a JMS provider failure. A message sent with this delivery mode is logged to stable storage when it is sent.
- The `NON_PERSISTENT` delivery mode does not require the JMS provider to store the message or otherwise guarantee that it is not lost if the provider fails.

You can specify the delivery mode in either of two ways.

- You can use the `setDeliveryMode` method of the `MessageProducer` interface—the parent of the `QueueSender` and the `TopicPublisher` interfaces—to set the delivery mode for all messages sent by that producer.
- You can use the long form of the `send` or the `publish` method to set the delivery mode for a specific message. The second argument sets the delivery mode. For example, the following `publish` call sets the delivery mode for `message` to `NON_PERSISTENT`:

  ```
  topicPublisher.publish(message, DeliveryMode.NON_PERSISTENT, 3,
      10000);
  ```

  The third and fourth arguments set the priority level and expiration time, which are described in the next two subsections.

If you do not specify a delivery mode, the default is `PERSISTENT`. Using the `NON_PERSISTENT` delivery mode may improve performance and reduce storage overhead, but you should use it only if your application can afford to miss messages.

5.1.3 Setting Message Priority Levels

You can use message priority levels to instruct the JMS provider to deliver urgent messages first. You can set the priority level in either of two ways.

- You can use the `setPriority` method of the `MessageProducer` interface to set the priority level for all messages sent by that producer.
- You can use the long form of the `send` or the `publish` method to set the priority level for a specific message. The third argument sets the priority level. For example, the following `publish` call sets the priority level for `message` to 3:

```
topicPublisher.publish(message, DeliveryMode.NON_PERSISTENT, 3,
    10000);
```

The ten levels of priority range from 0 (lowest) to 9 (highest). If you do not specify a priority level, the default level is 4. A JMS provider tries to deliver higher-priority messages before lower-priority ones but does not have to deliver messages in exact order of priority.

5.1.4 Allowing Messages to Expire

By default, a message never expires. If a message will become obsolete after a certain period, however, you may want to set an expiration time. You can do this in either of two ways.

- You can use the `setTimeToLive` method of the `MessageProducer` interface to set a default expiration time for all messages sent by that producer.
- You can use the long form of the `send` or the `publish` method to set an expiration time for a specific message. The fourth argument sets the expiration time in milliseconds. For example, the following `publish` call sets a time to live of 10 seconds:

```
topicPublisher.publish(message, DeliveryMode.NON_PERSISTENT, 3,
    10000);
```

If the specified *`timeToLive`* value is `0`, the message never expires.

When the message is published, the specified *`timeToLive`* is added to the current time to give the expiration time. Any message not delivered before the specified expiration time is destroyed. The destruction of obsolete messages conserves storage and computing resources.

5.1.5 Creating Temporary Destinations

Normally, you create JMS destinations—queues and topics—administratively rather than programmatically. Your JMS or J2EE provider includes a tool that you use to create and to remove destinations, and it is common for destinations to be long lasting.

The JMS API also enables you to create destinations—`TemporaryQueue` and `TemporaryTopic` objects—that last only for the duration of the connection in which they are created. You create these destinations dynamically, using the `QueueSession.createTemporaryQueue` and the `TopicSession.createTemporary-Topic` methods.

The only message consumers that can consume from a temporary destination are those created by the same connection that created the destination. Any message producer can send to the temporary destination. If you close the connection that a temporary destination belongs to, the destination is closed and its contents lost.

You can use temporary destinations to implement a simple request/reply mechanism. If you create a temporary destination and specify it as the value of the `JMSReplyTo` message header field when you send a message, the consumer of the message can use the value of the `JMSReplyTo` field as the destination to which it sends a reply and can also reference the original request by setting the `JMSCorrelationID` header field of the reply message to the value of the `JMSMessageID` header field of the request. For examples, see Chapters 9 and 10.

5.2 Using Advanced Reliability Mechanisms

The more advanced mechanisms for achieving reliable message delivery are the following:

- **Creating durable subscriptions.** You can create durable topic subscriptions, which receive messages published while the subscriber is not active. Durable subscriptions offer the reliability of queues to the publish/subscribe message domain.

- **Using local transactions.** You can use local transactions, which allow you to group a series of sends and receives into an atomic unit of work. Transactions are rolled back if they fail at any time.

5.2.1 Creating Durable Subscriptions

To make sure that a pub/sub application receives all published messages, use `PERSISTENT` delivery mode for the publishers. In addition, use durable subscriptions for the subscribers.

The `TopicSession.createSubscriber` method creates a nondurable subscriber. A nondurable subscriber can receive only messages that are published while it is active.

At the cost of higher overhead, you can use the `TopicSession.createDurableSubscriber` method to create a durable subscriber. A durable subscription can have only one active subscriber at a time.

A durable subscriber registers a durable subscription with a unique identity that is retained by the JMS provider. Subsequent subscriber objects with the same identity resume the subscription in the state in which it was left by the previous subscriber. If a durable subscription has no active subscriber, the JMS provider retains the subscription's messages until they are received by the subscription or until they expire.

You establish the unique identity of a durable subscriber by setting the following:

- A client ID for the connection
- A topic and a subscription name for the subscriber

You set the client ID administratively for a client-specific connection factory using the `j2eeadmin` command. For example:

```
j2eeadmin -addJmsFactory MY_CON_FAC topic -props clientID=MyID
```

After using this connection factory to create the connection and the session, you call the `createDurableSubscriber` method with two arguments—the topic and a string that specifies the name of the subscription:

```
String subName = "MySub";
TopicSubscriber topicSubscriber =
  topicSession.createDurableSubscriber(myTopic, subName);
```

The subscriber becomes active after you start the `TopicConnection`. Later on, you might close the `TopicSubscriber`:

```
topicSubscriber.close();
```

The JMS provider stores the messages published to the topic, as it would store messages sent to a queue. If the program or another application calls `create-DurableSubscriber` with the same connection factory and its client ID, the same topic, and the same subscription name, the subscription is reactivated, and the JMS provider delivers the messages that were published while the subscriber was inactive.

To delete a durable subscription, first close the subscriber, and then use the `unsubscribe` method, with the subscription name as the argument:

```
topicSubscriber.close();
topicSession.unsubscribe("MySub");
```

The `unsubscribe` method deletes the state that the provider maintains for the subscriber.

Figures 5.1 and 5.2 show the difference between a nondurable and a durable subscriber. With an ordinary, nondurable, subscriber, the subscriber and the subscription are coterminous and, in effect, identical. When a subscriber is closed, the subscription ends as well. Here, `create` stands for a call to `TopicSession.create-Subscriber`, and `close` stands for a call to `TopicSubscriber.close`. Any messages published to the topic between the time of the first `close` and the time of the second `create` are not consumed by the subscriber. In Figure 5.1, the subscriber consumes messages M1, M2, M5, and M6, but messages M3 and M4 are lost.

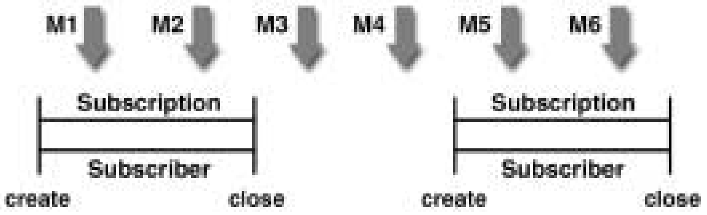

Figure 5.1 Nondurable Subscribers and Subscriptions

With a durable subscriber, the subscriber can be closed and recreated, but the subscription continues to exist and to hold messages until the application calls the `unsubscribe` method. In Figure 5.2, `create` stands for a call to `TopicSession.createDurableSubscriber`, `close` stands for a call to `TopicSubscriber.close`, and `unsubscribe` stands for a call to `TopicSession.unsubscribe`. Messages published while the subscriber is closed are received when the subscriber is created again. So even though messages M2, M4, and M5 arrive while the subscriber is closed, they are not lost.

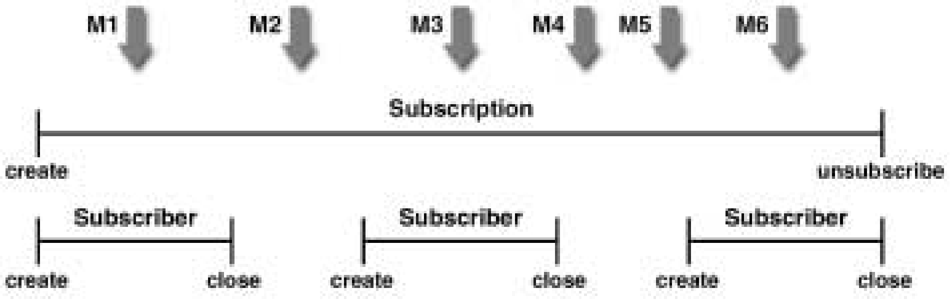

Figure 5.2 A Durable Subscriber and Subscription

See Chapter 8 for an example of a J2EE application that uses durable subscriptions. See Section A.1 on page 215 for an example of a client application that uses durable subscriptions.

5.2.2 Using JMS API Local Transactions

You can group a series of operations together into an atomic unit of work called a transaction. If any one of the operations fails, the transaction can be rolled back, and the operations can be attempted again from the beginning. If all the operations succeed, the transaction can be committed.

In a JMS client, you can use local transactions to group message sends and receives. The JMS API `Session` interface provides `commit` and `rollback` methods that you can use in a JMS client. A transaction commit means that all produced messages are sent and all consumed messages are acknowledged. A transaction rollback means that all produced messages are destroyed and all consumed messages are recovered and redelivered unless they have expired (see Section 5.1.4 on page 65).

A transacted session is always involved in a transaction. As soon as the `commit` or the `rollback` method is called, one transaction ends and another transaction begins. Closing a transacted session rolls back its transaction in progress, including any pending sends and receives.

In an Enterprise JavaBeans component, you cannot use the `Session.commit` and `Session.rollback` methods. Instead, you use distributed transactions, which are described in Chapter 6.

You can combine several sends and receives in a single JMS API local transaction. If you do so, you need to be careful about the order of the operations. You will have no problems if the transaction consists of all sends or all receives or if the receives come before the sends. But if you try to use a request-reply mechanism, whereby you send a message and then try to receive a reply to the sent message in the same transaction, the program will hang, because the send cannot take place until the transaction is committed. Because a message sent during a transaction is not actually sent until the transaction is committed, the transaction cannot contain any receives that depend on that message's having been sent.

It is also important to note that the production and the consumption of a message cannot both be part of the same transaction. The reason is that the transactions take place between the clients and the JMS provider, which intervenes between the production and the consumption of the message. Figure 5.3 illustrates this interaction.

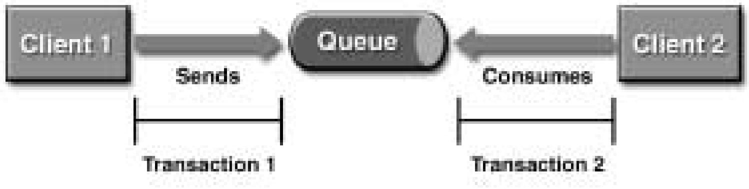

Figure 5.3 Using JMS API Local Transactions

The sending of one or more messages to a queue by Client 1 can form a single transaction, because it forms a single set of interactions with the JMS provider. Similarly, the receiving of one or more messages from the queue by Client 2 also forms a single transaction. But because the two clients have no direct interaction, no transactions take place between them. Another way of putting this is that the act of producing and/or consuming messages in a session can be transactional, but the act of producing and consuming a specific message across different sessions cannot be transactional.

This is the fundamental difference between messaging and synchronized processing. Instead of tightly coupling the sending and receiving of data, message producers and consumers use an alternative approach to reliability, one that is built on a JMS provider's ability to supply a once-and-only-once message delivery guarantee.

When you create a session, you specify whether it is transacted. The first argument to the `createQueueSession` and the `createTopicSession` methods is a `boolean` value. A value of `true` means that the session is transacted; a value of `false` means that it is not transacted. The second argument to these methods is the acknowledgment mode, which is relevant only to nontransacted sessions (see Section 5.1.1 on page 62). If the session is transacted, the second argument is ignored, so it is a good idea to specify `0` to make the meaning of your code clear. For example:

```
topicSession = topicConnection.createTopicSession(true, 0);
```

Because the `commit` and the `rollback` methods for local transactions are associated with the session, you cannot combine queue and topic operations in a single transaction. For example, you cannot receive a message from a queue and then publish a related message to a topic in the same transaction, because the `QueueReceiver` and the `TopicPublisher` are associated with a `QueueSession` and a

`TopicSession`, respectively. You can, however, receive from one queue and send to another queue in the same transaction, assuming that you use the same `QueueSession` to create the `QueueReceiver` and the `QueueSender`. You can pass a client program's session to a message listener's constructor function and use it to create a message producer, so that you can use the same session for receives and sends in asynchronous message consumers. For an example of the use of JMS API local transactions, see Section A.2 on page 225.

CHAPTER 6

Using the JMS API in a J2EE Application

THIS chapter describes the ways in which using the JMS API in a J2EE application differs from using it in a standalone client application:

- Using enterprise beans to produce and to synchronously receive messages
- Using message-driven beans to receive messages asynchronously
- Managing distributed transactions
- Using application clients and Web components

This chapter assumes that you have some knowledge of the J2EE platform and J2EE components. If you have not already done so, you may wish to read the J2EE Tutorial (`http://java.sun.com/j2ee/tutorial/`) or the Java 2 Platform, Enterprise Edition Specification, v1.3 (available from `http://java.sun.com/j2ee/download.html`).

6.1 Using Enterprise Beans to Produce and to Synchronously Receive Messages

A J2EE application that produces messages or synchronously receives them may use any kind of enterprise bean to perform these operations. The example in Chapter 8 uses a stateless session bean to publish messages to a topic.

Because a blocking synchronous receive ties up server resources, it is not a good programming practice to use such a `receive` call in an enterprise bean.

Instead, use a timed synchronous receive, or use a message-driven bean to receive messages asynchronously. For details about blocking and timed synchronous receives, see Section 4.2.1 on page 35.

Using the JMS API in a J2EE application is in many ways similar to using it in a standalone client. The main differences are in administered objects, resource management, and transactions.

6.1.1 Administered Objects

The Platform Specification recommends that you use `java:comp/env/jms` as the environment subcontext for Java Naming and Directory Interface (JNDI) API lookups of connection factories and destinations. With the J2EE SDK 1.3.1, you use the Application Deployment Tool, commonly known as the deploytool, to specify JNDI API names that correspond to those in your source code.

Instead of looking up a JMS API connection factory or destination each time it is used in a method, it is recommended that you look up these instances once in the enterprise bean's `ejbCreate` method and cache them for the lifetime of the enterprise bean.

6.1.2 Resource Management

A JMS API resource is a JMS API connection or a JMS API session. In general, it is important to release JMS resources when they are no longer being used. Here are some useful practices to follow.

- If you wish to maintain a JMS API resource only for the life span of a business method, it is a good idea to close the resource in a `finally` block within the method.
- If you would like to maintain a JMS API resource for the life span of an enterprise bean instance, it is a good idea to use the component's `ejbCreate` method to create the resource and to use the component's `ejbRemove` method to close the resource. If you use a stateful session bean or an entity bean and you wish to maintain the JMS API resource in a cached state, you must close the resource in the `ejbPassivate` method and set its value to `null`, and you must create it again in the `ejbActivate` method.

6.1.3 Transactions

Instead of using local transactions, you use the deploytool to specify container-managed transactions for bean methods that perform sends and receives, allowing the EJB container to handle transaction demarcation. (You can use bean-managed transactions and the `javax.transaction.UserTransaction` interface's transaction demarcation methods, but you should do so only if your application has special requirements and you are an expert in using transactions. Usually, container-managed transactions produce the most efficient and correct behavior.)

6.2 Using Message-Driven Beans

As we noted in Section 1.4 on page 12, the J2EE platform supports a new kind of enterprise bean, the message-driven bean, which allows J2EE applications to process JMS messages asynchronously. Session beans and entity beans allow you to send messages and to receive them synchronously but not asynchronously.

A message-driven bean is a message listener that can reliably consume messages from a queue or a durable subscription. The messages may be sent by any J2EE component—from an application client, another enterprise bean, or a Web component—or from an application or a system that does not use J2EE technology.

Like a message listener in a standalone JMS client, a message-driven bean contains an `onMessage` method that is called automatically when a message arrives. Like a message listener, a message-driven bean class may implement helper methods invoked by the `onMessage` method to aid in message processing.

A message-driven bean differs from a standalone client's message listener in five ways, however.

- The EJB container automatically performs several setup tasks that a standalone client has to do:
  - Creating a message consumer (a `QueueReceiver` or a `TopicSubscriber`) to receive the messages. You associate the message-driven bean with a destination and a connection factory at deployment time. If you want to specify a durable subscription or use a message selector, you do this at deployment time also.
  - Registering the message listener. (You must not call `setMessageListener`.)

    - Specifying a message acknowledgment mode. (For details, see Section 6.3 on page 77.)

- Your bean class must implement the `javax.ejb.MessageDrivenBean` and the `javax.jms.MessageListener` interfaces.
- Your bean class must implement the `ejbCreate` method in addition to the `onMessage` method. The method has the following signature:

  ```
  public void ejbCreate() {}
  ```

  If your message-driven bean produces messages or does synchronous receives from another destination, you use its `ejbCreate` method to look up JMS API connection factories and destinations and to create the JMS API connection.
- Your bean class must implement an `ejbRemove` method. The method has the following signature:

  ```
  public void ejbRemove() {}
  ```

  If you used the message-driven bean's `ejbCreate` method to create the JMS API connection, you ordinarily use the `ejbRemove` method to close the connection.
- Your bean class must implement the `setMessageDrivenContext` method. A `MessageDrivenContext` object provides some additional methods that you can use for transaction management. The method has the following signature:

  ```
  public void setMessageDrivenContext(MessageDrivenContext mdc) {}
  ```

See Section 7.1.2 on page 85 for a simple example of a message-driven bean.

The main difference between a message-driven bean and other enterprise beans is that a message-driven bean has no home or remote interface. Rather, it has only a bean class.

A message-driven bean is similar in some ways to a stateless session bean: its instances are relatively short-lived and retain no state for a specific client. The instance variables of the message-driven bean instance can contain some state across the handling of client messages: for example, a JMS API connection, an open database connection, or an object reference to an enterprise bean object.

Like a stateless session bean, a message-driven bean can have many interchangeable instances running at the same time. The container can pool these instances to allow streams of messages to be processed concurrently. Concurrency can affect the order in which messages are delivered, so you should write your application to handle messages that arrive out of sequence.

To create a new instance of a message-driven bean, the container instantiates the bean and then

1. Calls the `setMessageDrivenContext` method to pass the context object to the instance
2. Calls the instance's `ejbCreate` method

Figure 6.1 shows the life cycle of a message-driven bean.

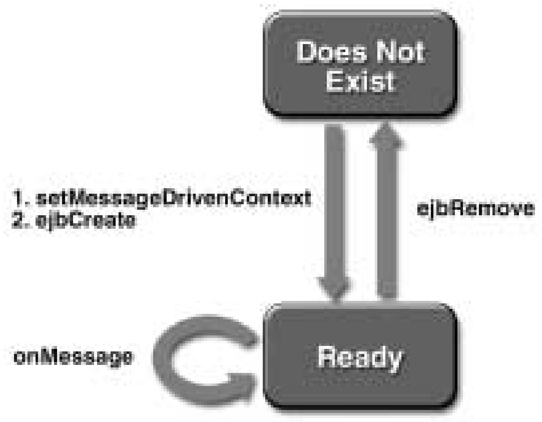

Figure 6.1 Life Cycle of a Message-Driven Bean

6.3 Managing Distributed Transactions

JMS client applications use JMS API local transactions, described in Section 5.2.2 on page 70, which allow the grouping of sends and receives within a specific JMS session. J2EE applications commonly use distributed transactions in order to ensure the integrity of accesses to external resources. For example, distributed transactions allow multiple applications to perform atomic updates on the same database, and they allow a single application to perform atomic updates on multiple databases.

In a J2EE application that uses the JMS API, you can use transactions to combine message sends or receives with database updates and other resource manager operations. You can access resources from multiple application components within a single transaction. For example, a servlet may start a transaction, access multiple databases, invoke an enterprise bean that sends a JMS message, invoke another enterprise bean that modifies an EIS system using the Connector architecture, and finally commit the transaction. Your application cannot, however, both send a JMS message and receive a reply to it within the same transaction; the restriction described in Section 5.2.2 on page 70 still applies.

Distributed transactions can be either of two kinds:

- **Container-managed transactions.** The EJB container controls the integrity of your transactions without your having to call `commit` or `rollback`. Container-managed transactions are recommended for J2EE applications that use the JMS API. You can specify appropriate transaction attributes for your enterprise bean methods.

  Use the `Required` transaction attribute to ensure that a method is always part of a transaction. If a transaction is in progress when the method is called, the method will be part of that transaction; if not, a new transaction will be started before the method is called and will be committed when the method returns.

- **Bean-managed transactions.** You can use these in conjunction with the `javax.transaction.UserTransaction` interface, which provides its own `commit` and `rollback` methods that you can use to delimit transaction boundaries.

You can use either container-managed transactions or bean-managed transactions with message-driven beans.

To ensure that all messages are received and handled within the context of a transaction, use container-managed transactions and specify the `Required` transaction attribute for the `onMessage` method. This means that a new transaction will be started before the method is called and will be committed when the method returns. An `onMessage` call is always a separate transaction, because there is never a transaction in progress when the method is called.

When you use container-managed transactions, you can call the following `MessageDrivenContext` methods:

- `setRollbackOnly`. Use this method for error handling. If an exception occurs, `setRollbackOnly` marks the current transaction so that the only possible outcome of the transaction is a rollback.
- `getRollbackOnly`. Use this method to test whether the current transaction has been marked for rollback.

If you use bean-managed transactions, the delivery of a message to the `onMessage` method takes place outside of the distributed transaction context. The transaction begins when you call the `UserTransaction.begin` method within the `onMessage` method and ends when you call `UserTransaction.commit`. If you call `UserTransaction.rollback`, the message is not redelivered, whereas calling `setRollbackOnly` for container-managed transactions does cause a message to be redelivered.

Neither the JMS API Specification nor the Enterprise JavaBeans Specification (available from `http://java.sun.com/products/ejb/`) specifies how to handle calls to JMS API methods outside transaction boundaries. The Enterprise JavaBeans Specification does state that the EJB container is responsible for acknowledging a message that is successfully processed by the `onMessage` method of a message-driven bean that uses bean-managed transactions. Using bean-managed transactions allows you to process the message by using more than one transaction or to have some parts of the message processing take place outside a transaction context. In most cases, however, container-managed transactions provide greater reliability and are therefore preferable.

When you create a session in an enterprise bean, the container ignores the arguments you specify, because it manages all transactional properties for enterprise beans. It is still a good idea to specify arguments of `true` and `0` to the `createQueueSession` or the `createTopicSession` method to make this situation clear:

```
queueSession = queueConnection.createQueueSession(true, 0);
```

When you use container-managed transactions, you usually specify the `Required` transaction attribute for your enterprise bean's business methods.

You do not specify a message acknowledgment mode when you create a message-driven bean that uses container-managed transactions. The container acknowledges the message automatically when it commits the transaction.

If a message-driven bean uses bean-managed transactions, the message receipt cannot be part of the bean-managed transaction, so the container acknowledges the message outside of the transaction. When you package a message-driven bean using the deploytool, the Message-Driven Bean Settings dialog box allows you to specify the acknowledgment mode, which can be either `AUTO_ACKNOWLEDGE` (the default) or `DUPS_OK_ACKNOWLEDGE`.

If the `onMessage` method throws a `RuntimeException`, the container does not acknowledge processing the message. In that case, the JMS provider will redeliver the unacknowledged message in the future.

6.4 Using the JMS API with Application Clients and Web Components

An application client can use the JMS API in much the same way a standalone client program does. It can produce messages, and it can consume messages by using either synchronous receives or message listeners. See Chapter 7 for an example of an application client that produces messages; see Chapters 9 and 10 for examples of using application clients to produce and to consume messages.

The J2EE Platform Specification does not define how Web components implement a JMS provider. In the J2EE SDK 1.3.1, a Web component—one that uses either the Java Servlet API or JavaServerPages™ (JSP™) technology—may send messages and consume them synchronously but may not consume them asynchronously.

Because a blocking synchronous receive ties up server resources, it is not a good programming practice to use such a `receive` call in a Web component. Instead, use a timed synchronous receive. For details about blocking and timed synchronous receives, see Section 4.2.1 on page 35.

CHAPTER 7

A Simple J2EE Application that Uses the JMS API

THIS chapter explains how to write, compile, package, deploy, and run a simple J2EE application that uses the JMS API. The application in this chapter uses the following components:

- An application client that sends several messages to a queue
- A message-driven bean that asynchronously receives and processes the messages

The chapter covers the following topics:

- Writing and compiling the application components
- Creating and packaging the application
- Deploying and running the application

If you downloaded the tutorial examples as described in the preface, you will find the source code files for this chapter in `jms_tutorial/examples/client_mdb` (on UNIX systems) or `jms_tutorial\examples\client_mdb` (on Microsoft Windows systems). The directory `ear_files` in the `examples` directory contains a built application called `SampleMDBApp.ear`. If you run into difficulty at any time, you can open this file in the deploytool and compare that file to your own version.

7.1 Writing and Compiling the Application Components

The first and simplest application contains the following components:

- An application client that sends several messages to a queue
- A message-driven bean that asynchronously receives and processes the messages

Figure 7.1 illustrates the structure of this application.

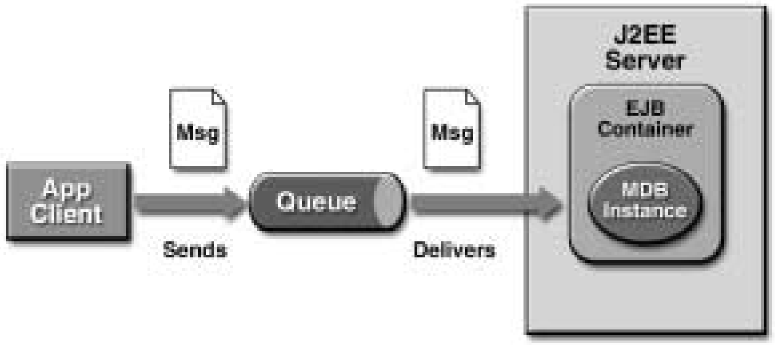

Figure 7.1 A Simple J2EE Application: Client to Message-Driven Bean

The application client sends messages to the queue, which is created administratively, using the `j2eeadmin` command. The JMS provider—in this case, the J2EE server—delivers the messages to the message-driven bean instances, which then process the messages.

Writing and compiling the components of this application involve

- Coding the application client
- Coding the message-driven bean
- Compiling the source files

7.1.1 Coding the Application Client: `SimpleClient.java`

The application client class, `SimpleClient.java`, is almost identical to `Simple-QueueSender.java` in Section 4.2.1.1 on page 37. The only significant differences are as follows.

- You do not specify the queue name on the command line. Instead, the client obtains the queue name through a Java Naming and Directory Interface (JNDI) API lookup.
- You do not specify the number of messages on the command line; the number of messages is set at 3 for simplicity, and no end-of-messages message is sent.
- The JNDI API lookup uses the `java:/comp/env/jms` naming context.

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

/**
 * The SimpleClient class sends several messages to a queue.
 */
public class SimpleClient {

    /**
     * Main method.
     */
    public static void main(String[] args) {
        Context                 jndiContext = null;
        QueueConnectionFactory  queueConnectionFactory = null;
        QueueConnection         queueConnection = null;
        QueueSession            queueSession = null;
        Queue                   queue = null;
        QueueSender             queueSender = null;
        TextMessage             message = null;
        final int               NUM_MSGS = 3;

        /*
         * Create a JNDI API InitialContext object.
         */
        try {
            jndiContext = new InitialContext();
        } catch (NamingException e) {
            System.out.println("Could not create JNDI API " +
                "context: " + e.toString());
            System.exit(1);
        }

        /*
         * Look up connection factory and queue.  If either does
         * not exist, exit.
         */
        try {
            queueConnectionFactory = (QueueConnectionFactory)
jndiContext.lookup("java:comp/env/jms/MyQueueConnectionFactory");
            queue = (Queue)
               jndiContext.lookup("java:comp/env/jms/QueueName");
        } catch (NamingException e) {
            System.out.println("JNDI API lookup failed: " +
                e.toString());
            System.exit(1);
        }

        /*
         * Create connection.
         * Create session from connection; false means session is
         * not transacted.
         * Create sender and text message.
         * Send messages, varying text slightly.
         * Finally, close connection.
         */
        try {
            queueConnection =
                queueConnectionFactory.createQueueConnection();
            queueSession =
                queueConnection.createQueueSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            queueSender = queueSession.createSender(queue);
            message = queueSession.createTextMessage();
```

```
                for (int i = 0; i < NUM_MSGS; i++) {
                    message.setText("This is message " + (i + 1));
                    System.out.println("Sending message: " +
                        message.getText());
                    queueSender.send(message);
                }
            } catch (JMSException e) {
                System.out.println("Exception occurred: " +
                    e.toString());
            } finally {
                if (queueConnection != null) {
                    try {
                        queueConnection.close();
                    } catch (JMSException e) {}
                }
                System.exit(0);
            }
        }
    }
```

Code Example 7.1 `SimpleClient.java`

7.1.2 Coding the Message-Driven Bean: `MessageBean.java`

The message-driven bean class, `MessageBean.java`, implements the methods `setMessageDrivenContext`, `ejbCreate`, `onMessage`, and `ejbRemove`. The `onMessage` method, almost identical to that of `TextListener.java` in Section 4.3.1.3 on page 53, casts the incoming message to a `TextMessage` and displays the text. The only significant difference is that it calls the `MessageDrivenContext.setRollbackOnly` method in case of an exception. This method rolls back the transaction so that the message will be redelivered.

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

```
/**
 * The MessageBean class is a message-driven bean.  It implements
 * the javax.ejb.MessageDrivenBean and javax.jms.MessageListener
 * interfaces. It is defined as public (but not final or
 * abstract).  It defines a constructor and the methods
 * setMessageDrivenContext, ejbCreate, onMessage, and
 * ejbRemove.
 */
public class MessageBean implements MessageDrivenBean,
        MessageListener {

    private transient MessageDrivenContext mdc = null;
    private Context context;

    /**
     * Constructor, which is public and takes no arguments.
     */
    public MessageBean() {
        System.out.println("In MessageBean.MessageBean()");
    }

    /**
     * setMessageDrivenContext method, declared as public (but
     * not final or static), with a return type of void, and
     * with one argument of type javax.ejb.MessageDrivenContext.
     *
     * @param mdc    the context to set
     */
    public void setMessageDrivenContext(MessageDrivenContext mdc)
    {
        System.out.println("In " +
            "MessageBean.setMessageDrivenContext()");
       this.mdc = mdc;
    }

    /**
     * ejbCreate method, declared as public (but not final or
     * static), with a return type of void, and with no
     * arguments.
     */
    public void ejbCreate() {
       System.out.println("In MessageBean.ejbCreate()");
    }

    /**
     * onMessage method, declared as public (but not final or
     * static), with a return type of void, and with one argument
     * of type javax.jms.Message.
     *
     * Casts the incoming Message to a TextMessage and displays
     * the text.
     *
     * @param inMessage    the incoming message
     */
    public void onMessage(Message inMessage) {
        TextMessage msg = null;

        try {
            if (inMessage instanceof TextMessage) {
                msg = (TextMessage) inMessage;
                System.out.println("MESSAGE BEAN: Message " +
                    "received: " + msg.getText());
            } else {
                System.out.println("Message of wrong type: " +
                    inMessage.getClass().getName());
            }
        } catch (JMSException e) {
            System.err.println("MessageBean.onMessage: " +
                "JMSException: " + e.toString());
            mdc.setRollbackOnly();
        } catch (Throwable te) {
            System.err.println("MessageBean.onMessage: " +
                "Exception: " + te.toString());
        }
    }
```

```
        /**
         * ejbRemove method, declared as public (but not final or
         * static), with a return type of void, and with no
         * arguments.
         */
        public void ejbRemove() {
            System.out.println("In MessageBean.remove()");
        }
    }
```

Code Example 7.2 `MessageBean.java`

7.1.3 Compiling the Source Files

To compile the files in the application, go to the `client_mdb` directory and do the following.

1. Make sure that you have set the environment variables shown in Table 4.1 on page 34: `JAVA_HOME`, `J2EE_HOME`, `CLASSPATH`, and `PATH`.
2. At a command line prompt, compile the source files:

```
javac *.java
```

7.2 Creating and Packaging the Application

Creating and packaging this application involve several steps:

1. Starting the J2EE server and the Application Deployment Tool
2. Creating a queue
3. Creating the J2EE application
4. Packaging the application client
5. Packaging the message-driven bean
6. Checking the JNDI API names (“JNDI names”)

7.2.1 Starting the J2EE Server and the Deploytool

Before you can create and package the application, you must start the J2EE server and the deploytool. Follow these steps.

1. At the command line prompt, start the J2EE server:

   ```
   j2ee -verbose
   ```

   Wait until the server displays the message “J2EE server startup complete.”

   (To stop the server, type `j2ee -stop`.)

2. At another command line prompt, start the deploytool:

   ```
   deploytool
   ```

   (To access the tool’s context-sensitive help, press F1.)

7.2.2 Creating a Queue

In Section 4.2.4 on page 43, you used the `j2eeadmin` command to create a queue. This time, you will create it by using the deploytool, as follows.

1. In the deploytool, select the Tools menu.
2. From the Tools menu, choose Server Configuration.
3. Under the JMS folder, select Destinations.
4. In the JMS Queue Destinations area, click Add.
5. In the text field, enter `jms/MyQueue`. (We will observe the J2EE convention of placing the queue in the `jms` namespace.)
6. Click OK.
7. Verify that the queue was created:

   ```
   j2eeadmin -listJmsDestination
   ```

7.2.3 Creating the J2EE Application

Create a new J2EE application called `MDBApp` and store it in the file named `MDBApp.ear`. Follow these steps.

1. In the deploytool, select the File menu.
2. From the File menu, choose New → Application.
3. Click Browse next to the Application File Name field and use the file chooser to locate the directory `client_mdb`.
4. In the File Name field, enter `MDBApp`.
5. Click New Application.
6. Click OK.

A diamond icon labeled `MDBApp` appears in the tree view on the left side of the deploytool window. The full path name of `MDBApp.ear` appears in the General tabbed pane on the right side.

7.2.4 Packaging the Application Client

In this section, you will run the New Application Client Wizard of the deploytool to package the application client. The New Application Client Wizard does the following:

- Identifies the application toward which the client is targeted
- Identifies the main class for the application client
- Identifies the queue and the connection factory referenced by the application client

To start the New Application Client Wizard, follow these steps.

1. In the tree view, select `MDBApp`.
2. From the File menu, choose New → Application Client. The wizard displays a series of dialog boxes.

7.2.4.1 Introduction Dialog Box

1. Read this explanatory text for an overview of the wizard's features.
2. Click Next.

7.2.4.2 JAR File Contents Dialog Box

1. In the combo box labeled Create Archive Within Application, select `MDBApp`.
2. Click the Edit button next to the Contents text area.
3. In the dialog box Edit Contents of <Application Client>, choose the `client_mdb` directory. If the directory is not already in the Starting Directory field, type it in the field, or locate it by browsing through the Available Files tree.
4. Select `SimpleClient.class` from the Available Files tree area and click Add.
5. Click OK.
6. Click Next.

7.2.4.3 General Dialog Box

1. Verify that the Main Class and the Display Name are both `SimpleClient`.
2. In the Callback Handler Class combo box, verify that container-managed authentication is selected.
3. Click Next.

7.2.4.4 Environment Entries Dialog Box

Click Next.

7.2.4.5 Enterprise Bean References Dialog Box

Click Next.

7.2.4.6 Resource References Dialog Box

In this dialog box, you associate the JNDI API context name for the connection factory in the `SimpleClient.java` source file with the name of the `QueueConnectionFactory`. You also specify container authentication for the connection factory resource, defining the user name and the password that the user must enter in order to be able to create a connection. Follow these steps.

1. Click Add.
2. In the Coded Name field, enter `jms/MyQueueConnectionFactory`—the logical name referenced by `SimpleClient`.
3. In the Type field, select `javax.jms.QueueConnectionFactory`.
4. In the Authentication field, select Container.
5. In the Sharable field, make sure that the checkbox is checked. This allows the container to optimize connections.
6. In the JNDI Name field, enter `jms/QueueConnectionFactory`.
7. In the User Name field, enter `j2ee`.
8. In the Password field, enter `j2ee`.
9. Click Next.

7.2.4.7 JMS Destination References Dialog Box

In this dialog box, you associate the JNDI API context name for the queue in the `SimpleClient.java` source file with the name of the queue you created using `j2eeadmin`. Follow these steps.

1. Click Add.
2. In the Coded Name field, enter `jms/QueueName`—the logical name referenced by `SimpleClient`.
3. In the Type field, select `javax.jms.Queue`.
4. In the JNDI Name field, enter `jms/MyQueue`.
5. Click Next.

7.2.4.8 Review Settings Dialog Box

1. Check the settings for the deployment descriptor.
2. Click Finish.

7.2.5 Packaging the Message-Driven Bean

In this section, you will run the New Enterprise Bean Wizard of the deploytool to perform these tasks:

- Create the bean's deployment descriptor
- Package the deployment descriptor and the bean class in an enterprise bean JAR file
- Insert the enterprise bean JAR file into the application's `MDBApp.ear` file

  To start the New Enterprise Bean Wizard, follow these steps.

1. In the tree view, select `MDBApp`.
2. From the File menu, choose New → Enterprise Bean. The wizard displays a series of dialog boxes.

7.2.5.1 Introduction Dialog Box

Click Next.

7.2.5.2 EJB JAR Dialog Box

1. In the combo box labeled JAR File Location, verify that Create New JAR File in Application and `MDBApp` are selected.
2. In the JAR Display Name field, verify that the name is `Ejb1`, the default display name. Representing the enterprise bean JAR file that contains the bean, this name will be displayed in the tree view.
3. Click the Edit button next to the Contents text area.
4. In the dialog box Edit Contents of Ejb1, choose the `client_mdb` directory. If the directory is not already in the Starting Directory field, type it in the field, or locate it by browsing through the Available Files tree.

5. Select the `MessageBean.class` file from the Available Files tree area and click Add.
6. Click OK.
7. Click Next.

7.2.5.3 General Dialog Box

1. In the Bean Type combo box, select the Message-Driven radio button.
2. Under Enterprise Bean Class, select `MessageBean`. The combo boxes for the local and remote interfaces are grayed out.
3. In the Enterprise Bean Name field, enter `MessageBean`. This name will represent the message-driven bean in the tree view. The display name does not have to be different from the bean class name.
4. Click Next.

7.2.5.4 Transaction Management Dialog Box

In this dialog box, you specify how transactions for the `onMessage` method should be handled. Although an ordinary enterprise bean has six possible transaction attributes, a message-driven bean has only two. (The others are meaningful only if there might be a preexisting transaction context, which doesn't exist for a message-driven bean.) Follow these steps.

1. Select the Container-Managed radio button.
2. In the Transaction Attribute field opposite the `onMessage` method, verify that Required is selected.
3. Click Next.

7.2.5.5 Message-Driven Bean Settings Dialog Box

In this dialog box, you specify the deployment properties for the bean. Because you are using container-managed transactions, the Acknowledgment field is grayed out. Follow these steps.

1. In the Destination Type combo box, select Queue.
2. In the Destination field, select `jms/MyQueue`.

3. In the Connection Factory field, select `jms/QueueConnectionFactory`.
4. Click Next.

7.2.5.6 Environment Entries Dialog Box

Click Next.

7.2.5.7 Enterprise Bean References Dialog Box

Click Next.

7.2.5.8 Resource References Dialog Box

Click Next. (You do not need to specify the connection factory for the message-driven bean, because it is not referred to in the message-driven bean code. You specified the connection factory in the Message-Driven Bean Settings dialog box.)

7.2.5.9 Resource Environment References Dialog Box

Click Next. (You do not need to specify the queue name here, because it is not referred to in the message-driven bean code. You specified it in the Message-Driven Bean Settings dialog box.)

7.2.5.10 Security Dialog Box

Use the default Security Identity setting for a message-driven bean, Run As Specified Role. Click Next.

7.2.5.11 Review Settings Dialog Box

1. Check the settings for the deployment descriptor.
2. Click Finish.

7.2.6 Checking the JNDI Names

Verify that the JNDI names for the application components are correct.

- You give the JNDI name of the destination—in this case, the queue—to the message-driven bean component.
- You check to make sure that the context names for the connection factory and the destination are correctly matched to their JNDI names.

1. In the tree view, select the `MDBApp` application.
2. Select the JNDI Names tabbed pane.
3. Verify that the JNDI names appear as shown in Tables 7.1 and 7.2.

Table 7.1: Application Pane

Component Type	Component	JNDI Name
EJB	`MessageBean`	`jms/MyQueue`

Table 7.2: References Pane

Ref. Type	Referenced By	Reference Name	JNDI Name
Resource	`SimpleClient`	`jms/MyQueue-ConnectionFactory`	`jms/Queue-ConnectionFactory`
Env Resource	`SimpleClient`	`jms/QueueName`	`jms/MyQueue`

7.3 Deploying and Running the Application

Deploying and running this application involve several steps:

1. Looking at the deployment descriptor
2. Adding the server, if necessary
3. Deploying the application
4. Running the client

5. Undeploying the application
6. Removing the application and stopping the server

7.3.1 Looking at the Deployment Descriptor

As you package an application, the deploytool creates a deployment descriptor in accordance with the packaging choices you make. To see some of the JMS API-related elements of the enterprise bean deployment descriptor, follow these steps.

1. Select `Ejb1` in the tree view.
2. Choose Descriptor Viewer from the Tools menu.
3. Select `SimpleClient` in the tree view and repeat step 2.

In the Deployment Descriptor Viewer window, click Save As if you want to save the contents as an XML file for future reference. Table 7.3 describes the elements that are related to the JMS API.

Table 7.3: JMS API-Related Deployment Descriptor Elements

Element Name	**Description**
`<message-driven>`	Declares a message-driven bean.
`<message-selector>`	Specifies the JMS API message selector to be used in determining which messages a message-driven bean is to receive.
`<message-driven-destination>`	Tells the Deployer whether a message-driven bean is intended for a queue or a topic, and if it is intended for a topic, whether the subscription is durable. Contains the element `<destination-type>` and optionally, for topics, `<subscription-durability>`.
`<destination-type>`	Specifies the type of the JMS API destination (either `javax.jms.Queue` or `javax.jms.Topic`). In this case, the value is `javax.jms.Queue`.

(continued)

Table 7.3: JMS API-Related Deployment Descriptor Elements (Cont.)

Element Name	Description
`<subscription-durability>`	Optionally specifies whether a topic subscription is intended to be durable or nondurable.
`<resource-env-ref>`	Declares an enterprise bean's reference to an administered object associated with a resource in the enterprise bean's environment—in this case, a JMS API destination. Contains the elements `<resource-env-ref-name>` and `<resource-env-ref-type>`.
`<resource-env-ref-name>`	Specifies the name of a resource environment reference; its value is the environment entry name used in the enterprise bean code—in this case, `jms/QueueName`.
`<resource-env-ref-type>`	Specifies the type of a resource environment reference—in this case, `javax.jms.Queue`.
`<resource-ref>`	Contains a declaration of the enterprise bean's reference to an external resource—in this case, a JMS API connection factory. Contains the elements `<res-ref-name>`, `<res-type>`, and `<res-auth>`.
`<res-ref-name>`	Specifies the name of a resource manager connection factory reference—in this case, `jms/MyQueueConnectionFactory`.
`<res-type>`	Specifies the type of the data source—in this case, `javax.jms.QueueConnectionFactory`.
`<res-auth>`	Specifies whether the enterprise bean code signs on programmatically to the resource manager (Application) or whether the Container will sign on to the resource manager on behalf of the bean. In the latter case, the Container uses information that is supplied by the Deployer.
`<res-sharing-scope>`	Specifies whether connections obtained through the given resource manager connection factory reference can be shared. In this case, the value is `Shareable`.

7.3.2 Adding the Server

Before you can deploy the application, you must make available to the deploytool the J2EE server you started in Section 7.2.1 on page 89. Because you started the J2EE server before you started the deploytool, the server, named `localhost`, probably appears in the tree under `Servers`. If it does not, do the following.

1. From the File menu, choose Add Server.
2. In the Add Server dialog box, enter `localhost` in the Server Name field.
3. Click OK. A `localhost` node appears under `Servers` in the tree view.

7.3.3 Deploying the Application

You have now created an application that consists of an application client and a message-driven bean. To deploy the application, perform the following steps.

1. From the File menu, choose Save to save the application.
2. From the Tools menu, choose Deploy.
3. In the Introduction dialog box, verify that the Object to Deploy selection is `MDBApp` and that the Target Server selection is `localhost`.
4. Click Next.
5. In the JNDI Names dialog box, verify that the JNDI names are correct.
6. Click Next.
7. Click Finish.
8. In the Deployment Progress dialog box, click OK when the “Deployment of MDBApp is complete” message appears.
9. In the tree view, expand `Servers` and select `localhost`. Verify that `MDBApp` is deployed.

7.3.4 Running the Client

To run the client, you use the `MDBApp.ear` file that you created in Section 7.2.3 on page 90. Make sure that you are in the directory `client_mdb`. Then perform the following steps.

1. At the command line prompt, enter the following:

   ```
   runclient -client MDBApp.ear -name SimpleClient
   ```

2. When the Login for user: dialog box appears, enter `j2ee` for the user name and `j2ee` for the password.
3. Click OK.

The client program runs in the command window, generating output that looks like this:

```
Binding name:'java:comp/env/jms/QueueName'
Binding name:'java:comp/env/jms/MyQueueConnectionFactory'
Java(TM) Message Service 1.0.2 Reference Implementation (build b14)
Sending message: This is message 1
Sending message: This is message 2
Sending message: This is message 3
Unbinding name:'java:comp/env/jms/QueueName'
Unbinding name:'java:comp/env/jms/MyQueueConnectionFactory'
```

Output from the application appears in the window in which you started the J2EE server. By default, the server creates three instances of the `MessageBean` to receive messages.

```
In MessageBean.MessageBean()
In MessageBean.setMessageDrivenContext()
In MessageBean.ejbCreate()
MESSAGE BEAN: Message received: This is message 1
In MessageBean.MessageBean()
In MessageBean.setMessageDrivenContext()
In MessageBean.ejbCreate()
In MessageBean.MessageBean()
In MessageBean.setMessageDrivenContext()
```

```
In MessageBean.ejbCreate()
MESSAGE BEAN: Message received: This is message 2
MESSAGE BEAN: Message received: This is message 3
```

7.3.5 Undeploying the Application

To undeploy the J2EE application, follow these steps.

1. In the tree view, select `localhost`.
2. Select `MDBApp` in the Deployed Objects area.
3. Click Undeploy.
4. Answer Yes in the confirmation dialog.

7.3.6 Removing the Application and Stopping the Server

To remove the application from the deploytool, follow these steps.

1. Select `MDBApp` in the tree view.
2. Select Close from the File menu.

To delete the queue you created, enter the following at the command line prompt:

```
j2eeadmin -removeJmsDestination jms/MyQueue
```

To stop the J2EE server, use the following command:

```
j2ee -stop
```

To exit the deploytool, choose Exit from the File menu.

CHAPTER 8

A J2EE Application that Uses the JMS API with a Session Bean

THIS chapter explains how to write, compile, package, deploy, and run a J2EE application that uses the JMS API in conjunction with a session bean. The application contains the following components:

- An application client that calls an enterprise bean
- A session bean that publishes several messages to a topic
- A message-driven bean that receives and processes the messages, using a durable topic subscriber and a message selector

The chapter covers the following topics:

- Writing and compiling the application components
- Creating and packaging the application
- Deploying and running the application

If you downloaded the tutorial examples as described in the preface, you will find the source code files for this chapter in `jms_tutorial/examples/client_ses_mdb` (on UNIX systems) or `jms_tutorial\examples\client_ses_mdb` (on Microsoft Windows systems). The directory `ear_files` in the `examples` directory contains a built application called `SamplePubSubApp.ear`. If you run into difficulty at any time, you can open this file in the deploytool and compare that file to your own version.

8.1 Writing and Compiling the Application Components

This application demonstrates how to send messages from an enterprise bean—in this case, a session bean—rather than from an application client, as in the example in Chapter 7. Figure 8.1 illustrates the structure of this application.

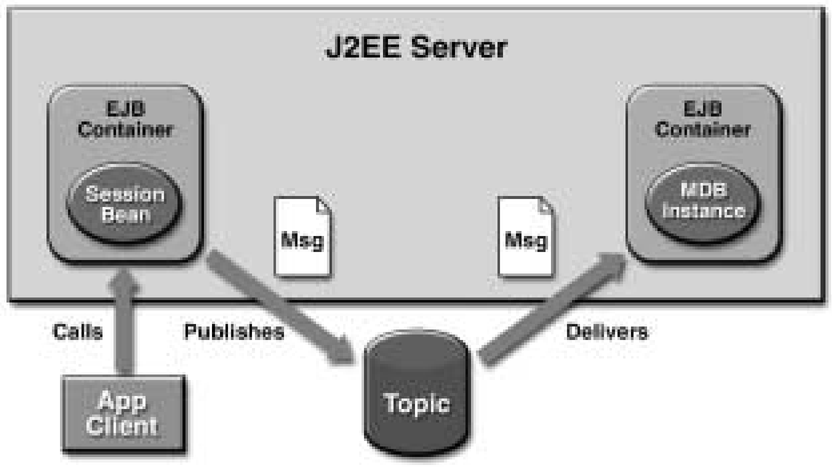

Figure 8.1 A J2EE Application: Client to Session Bean to Message-Driven Bean

The Publisher enterprise bean in this example is the enterprise-application equivalent of a wire-service news feed that categorizes news events into six news categories. The message-driven bean could represent a newsroom, where the Sports desk, for example, would set up a subscription for all news events pertaining to sports news.

The application client in the example obtains a handle to the Publisher enterprise bean's home interface and calls the enterprise bean's business method. The enterprise bean creates 18 text messages. For each message, it sets a `String` property randomly to one of six values representing the news categories and then publishes the message to a topic. The message-driven bean uses a message selector for the property to limit which of the published messages it receives.

Writing and compiling the components of the application involve

- Coding the application client
- Coding the Publisher session bean
- Coding the message-driven bean
- Compiling the source files

8.1.1 Coding the Application Client: `MyAppClient.java`

The application client program, `MyAppClient.java`, performs no JMS API operations and so is simpler than the client program in Chapter 7. The program obtains a handle to the Publisher enterprise bean's home interface, using the Java Naming and Directory Interface (JNDI) API naming context `java:comp/env`. The program then creates an instance of the bean and calls the bean's business method twice.

```
import javax.ejb.EJBHome;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import javax.jms.*;

/**
 * The MyAppClient class is the client program for this J2EE
 * application.  It obtains a reference to the home interface
 * of the Publisher enterprise bean and creates an instance of
 * the bean.  After calling the publisher's publishNews method
 * twice, it removes the bean.
 */
public class MyAppClient {

    public static void main (String[] args) {
        MyAppClient client = new MyAppClient();
        client.doTest();
        System.exit(0);
    }

    public void doTest() {
            try {

                Context ic = new InitialContext();

                System.out.println("Looking up EJB reference");
                java.lang.Object objref =
                    ic.lookup("java:comp/env/ejb/MyEjbReference");
                System.err.println("Looked up home");

                PublisherHome pubHome = (PublisherHome)
                    PortableRemoteObject.narrow(objref,
                        PublisherHome.class);
                System.err.println("Narrowed home");

                /*
                 * Create bean instance, invoke business method
                 * twice, and remove bean instance.
                 */
                Publisher phr = pubHome.create();
                System.err.println("Got the EJB");
                phr.publishNews();
                phr.publishNews();
                phr.remove();
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
```

Code Example 8.1 `MyAppClient.java`

8.1.2 Coding the Publisher Session Bean

The Publisher bean is a stateless session bean with one `create` method and one business method. The Publisher bean uses remote interfaces rather than local interfaces because it is accessed from outside the EJB container.

8.1.2.1 The Remote Home Interface: `PublisherHome.java`

The remote home interface source file is `PublisherHome.java`.

```
import java.rmi.RemoteException;
import javax.ejb.EJBHome;
import javax.ejb.CreateException;

/**
 * Home interface for Publisher enterprise bean.
 */
public interface PublisherHome extends EJBHome {
    Publisher create() throws RemoteException, CreateException;
}
```

Code Example 8.2 `PublisherHome.java`

8.1.2.2 The Remote Interface: `Publisher.java`

The remote interface, `Publisher.java`, declares a single business method, `publishNews`.

```
import javax.ejb.*;
import java.rmi.RemoteException;

/**
 * Remote interface for Publisher enterprise bean. Declares one
 * business method.
 */
public interface Publisher extends EJBObject {
    void publishNews() throws RemoteException;
}
```

Code Example 8.3 `Publisher.java`

8.1.2.3 The Bean Class: `PublisherBean.java`

The bean class, `PublisherBean.java`, implements the `publishNews` method and its helper method `chooseType`. The bean class also implements the required methods `ejbCreate`, `setSessionContext`, `ejbRemove`, `ejbActivate`, and `ejbPassivate`.

The `ejbCreate` method of the bean class allocates resources—in this case, by looking up the `TopicConnectionFactory` and the topic and creating the `TopicConnection`. The business method `publishNews` creates a `TopicSession` and a `TopicPublisher` and publishes the messages.

The `ejbRemove` method must deallocate the resources that were allocated by the `ejbCreate` method. In this case, the `ejbRemove` method closes the `TopicConnection`.

```
import java.rmi.RemoteException;
import java.util.*;
import javax.ejb.*;
import javax.naming.*;
import javax.jms.*;

/**
 * Bean class for Publisher enterprise bean. Defines publishNews
 * business method as well as required methods for a stateless
 * session bean.
 */
public class PublisherBean implements SessionBean {
    SessionContext      sc = null;
    TopicConnection     topicConnection = null;
    Topic               topic = null;
    final static String messageTypes[] = {"Nation/World",
        "Metro/Region", "Business", "Sports", "Living/Arts",
        "Opinion"};
```

```
public PublisherBean() {
    System.out.println("In PublisherBean() (constructor)");
}

/**
 * Sets the associated session context. The container calls
 * this method after the instance creation.
 */
public void setSessionContext(SessionContext sc) {
    this.sc = sc;
}

/**
 * Instantiates the enterprise bean.  Creates the
 * TopicConnection and looks up the topic.
 */
public void ejbCreate() {
    Context context = null;
    TopicConnectionFactory topicConnectionFactory = null;

    System.out.println("In PublisherBean.ejbCreate()");
    try {
        context = new InitialContext();
        topic = (Topic)
            context.lookup("java:comp/env/jms/TopicName");

        // Create a TopicConnection
        topicConnectionFactory = (TopicConnectionFactory)
context.lookup("java:comp/env/jms/MyTopicConnectionFactory");
        topicConnection =
            topicConnectionFactory.createTopicConnection();
    } catch (Throwable t) {
        // JMSException or NamingException could be thrown
        System.err.println("PublisherBean.ejbCreate:" +
            "Exception: " + t.toString());
    }
}

/**
```

```
     * Chooses a message type by using the random number
     * generator found in java.util.  Called by publishNews().
     *
     * @return   the String representing the message type
     */
    private String chooseType() {
        int    whichMsg;
        Random rgen = new Random();

        whichMsg = rgen.nextInt(messageTypes.length);
        return messageTypes[whichMsg];
    }

    /**
     * Creates TopicSession, publisher, and message.  Publishes
     * messages after setting their NewsType property and using
     * the property value as the message text. Messages are
     * received by MessageBean, a message-driven bean that uses a
     * message selector to retrieve messages whose NewsType
     * property has certain values.
     */
    public void publishNews() throws EJBException {
        TopicSession   topicSession = null;
        TopicPublisher topicPublisher = null;
        TextMessage    message = null;
        int            numMsgs = messageTypes.length * 3;
        String         messageType = null;

        try {
            topicSession =
                topicConnection.createTopicSession(true, 0);
            topicPublisher = topicSession.createPublisher(topic);
            message = topicSession.createTextMessage();
            for (int i = 0; i < numMsgs; i++) {
                messageType = chooseType();
                message.setStringProperty("NewsType",
                    messageType);
                message.setText("Item " + i + ": " +
                    messageType);
                System.out.println("PUBLISHER: Setting " +
                    "message text to: " + message.getText());
                topicPublisher.publish(message);
            }
        } catch (Throwable t) {
            // JMSException could be thrown
            System.err.println("PublisherBean.publishNews: " +
                 "Exception: " + t.toString());
            sc.setRollbackOnly();
        } finally {
            if (topicSession != null) {
                try {
                    topicSession.close();
                } catch (JMSException e) {}
            }
        }
    }

    /**
     * Closes the TopicConnection.
     */
    public void ejbRemove() throws RemoteException {
        System.out.println("In PublisherBean.ejbRemove()");
        if (topicConnection != null) {
            try {
                topicConnection.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    public void ejbActivate() {}
    public void ejbPassivate() {}
}
```

Code Example 8.4 `PublisherBean.java`

8.1.3 Coding the Message-Driven Bean: `MessageBean.java`

The message-driven bean class, `MessageBean.java`, is identical to the one in Section 7.1.2 on page 85.

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

/**
 * The MessageBean class is a message-driven bean.  It implements
 * the javax.ejb.MessageDrivenBean and javax.jms.MessageListener
 * interfaces. It is defined as public (but not final or
 * abstract).  It defines a constructor and the methods
 * setMessageDrivenContext, ejbCreate, onMessage, and
 * ejbRemove.
*/
public class MessageBean implements MessageDrivenBean,
    MessageListener {

    private transient MessageDrivenContext mdc = null;
    private Context context;

    /**
     * Constructor, which is public and takes no arguments.
     */
    public MessageBean() {
        System.out.println("In MessageBean.MessageBean()");
    }

    /**
     * setMessageDrivenContext method, declared as public (but
     * not final or static), with a return type of void, and
     * with one argument of type javax.ejb.MessageDrivenContext.
     *
     * @param mdc    the context to set
     */
```

```
public void setMessageDrivenContext(MessageDrivenContext mdc)
{
    System.out.println("In " +
        "MessageBean.setMessageDrivenContext()");
    this.mdc = mdc;
}

/**
 * ejbCreate method, declared as public (but not final or
 * static), with a return type of void, and with no
 * arguments.
 */
public void ejbCreate() {
    System.out.println("In MessageBean.ejbCreate()");
}



/**
 * onMessage method, declared as public (but not final or
 * static), with a return type of void, and with one argument
 * of type javax.jms.Message.
 *
 * Casts the incoming Message to a TextMessage and displays
 * the text.
 *
 * @param inMessage    the incoming message
 */
public void onMessage(Message inMessage) {
    TextMessage msg = null;

    try {
        if (inMessage instanceof TextMessage) {
            msg = (TextMessage) inMessage;
            System.out.println("MESSAGE BEAN: Message " +
                "received: " + msg.getText());
        } else {
            System.out.println("Message of wrong type: " +
                inMessage.getClass().getName());
                }
            } catch (JMSException e) {
                System.err.println("MessageBean.onMessage: " +
                    "JMSException: " + e.toString());
                mdc.setRollbackOnly();
            } catch (Throwable te) {
                System.err.println("MessageBean.onMessage: " +
                    "Exception: " + te.toString());
            }
        }

        /**
         * ejbRemove method, declared as public (but not final or
         * static), with a return type of void, and with no
         * arguments.
         */
        public void ejbRemove() {
            System.out.println("In MessageBean.remove()");
        }
    }
```

Code Example 8.5 `MessageBean.java`

8.1.4 Compiling the Source Files

To compile all the files in the application, go to the directory `client_ses_mdb` and do the following.

1. Make sure that you have set the environment variables shown in Table 4.1 on page 34: `JAVA_HOME`, `J2EE_HOME`, `CLASSPATH`, and `PATH`.
2. At a command line prompt, compile the source files:

```
javac *.java
```

8.2 Creating and Packaging the Application

Creating and packaging this application involve several steps:

1. Starting the J2EE server and the deploytool*
2. Creating a topic
3. Creating a connection factory
4. Creating the J2EE application
5. Packaging the application client
6. Packaging the session bean
7. Packaging the message-driven bean
8. Specifying the JNDI names

Step 1, marked with an asterisk (*), is not needed if the server and deploytool are still running.

8.2.1 Starting the J2EE Server and the Deploytool

Before you can create and package the application, you must start the J2EE server and the deploytool. Follow these steps.

1. At the command line prompt, start the J2EE server:

   ```
   j2ee -verbose
   ```

   Wait until the server displays the message “J2EE server startup complete.” (To stop the server, type `j2ee -stop`.)

2. At another command line prompt, start the deploytool:

   ```
   deploytool
   ```

   (To access the tool’s context-sensitive help, press F1.)

8.2.2 Creating a Topic

In Section 4.3.4 on page 55, you used the `j2eeadmin` command to create a topic. This time, you will create the topic by using the deploytool. Follow these steps.

1. In the deploytool, select the Tools menu.
2. From the Tools menu, choose Server Configuration.
3. Under the JMS folder, select Destinations.
4. In the JMS Topic Destinations area, click Add.
5. In the text field, enter `jms/MyTopic`. (We will observe the J2EE convention of placing the topic in the `jms` namespace.)
6. Click OK.
7. If you wish, you can verify that the topic was created:

   ```
   j2eeadmin -listJmsDestination
   ```

8.2.3 Creating a Connection Factory

For this application, you create a new connection factory. This application will use a durable subscriber, so you need a connection factory that has a client ID. (For more information, see Section 5.2.1 on page 67.) Follow these steps.

1. At the command line prompt, enter the following command (all on one line):

   ```
   j2eeadmin -addJmsFactory jms/DurableTopicCF topic -props
   clientID=MyID
   ```

2. Verify that the connection factory was created:

   ```
   j2eeadmin -listJmsFactory
   ```

You can also create connection factories by using the deploytool's Server Configuration dialog.

8.2.4 Creating the J2EE Application

Create a new J2EE application called `PubSubApp` and store it in the file named `PubSubApp.ear`. Follow these steps.

1. In the deploytool, select the File menu.
2. From the File menu, choose New → Application.
3. Click Browse next to the Application File Name field and use the file chooser to locate the directory `client_ses_mdb`.
4. In the File Name field, enter `PubSubApp`.
5. Click New Application.
6. Click OK.

A diamond icon labeled `PubSubApp` appears in the tree view on the left side of the deploytool window. The full path name of `PubSubApp.ear` appears in the General tabbed pane on the right side.

8.2.5 Packaging the Application Client

In this section, you will run the New Application Client Wizard of the deploytool to package the application client. To start the New Application Client Wizard, follow these steps.

1. In the tree view, select `PubSubApp`.
2. From the File menu, choose New → Application Client. The wizard displays a series of dialog boxes.

8.2.5.1 Introduction Dialog Box

Click Next.

8.2.5.2 JAR File Contents Dialog Box

1. In the combo box labeled Create Archive Within Application, select `PubSubApp`.
2. Click the Edit button next to the Contents text area.

3. In the dialog box Edit Contents of <Application Client>, choose the `client_ses_mdb` directory. If the directory is not already in the Starting Directory field, type it in the field, or locate it by browsing through the Available Files tree.
4. Select the `MyAppClient.class` file from the Available Files tree area and click Add.
5. Click OK.
6. Click Next.

8.2.5.3 General Dialog Box

1. In the Application Client combo box, select `MyAppClient` in the Main Class field, and enter `MyAppClient` in the Display Name field.
2. In the Callback Handler Class combo box, verify that container-managed authentication is selected.
3. Click Next.

8.2.5.4 Environment Entries Dialog Box

Click Next.

8.2.5.5 Enterprise Bean References Dialog Box

In this dialog box, you associate the JNDI API context name for the EJB reference in the `MyAppClient.java` source file with the home and remote interfaces of the Publisher enterprise bean. Follow these steps.

1. Click Add.
2. In the Coded Name column, enter `ejb/MyEjbReference`.
3. In the Type column, select Session.
4. In the Interfaces column, select Remote.
5. In the Home Interface column, enter `PublisherHome`.
6. In the Local/Remote Interface column, enter `Publisher`.

7. In the Deployment Settings combo box, select JNDI Name. In the JNDI Name field, enter `MyPublisher`.
8. Click Finish. You do not need to enter anything in the other dialog boxes.

8.2.6 Packaging the Session Bean

In this section, you will run the New Enterprise Bean Wizard of the deploytool to package the session bean. Follow these steps.

1. In the tree view, select `PubSubApp`.
2. From the File menu, choose New → Enterprise Bean. The wizard displays a series of dialog boxes.

8.2.6.1 Introduction Dialog Box

Click Next.

8.2.6.2 EJB JAR Dialog Box

1. In the combo box labeled JAR File Location, verify that Create New JAR File in Application and `PubSubApp` are selected.
2. In the JAR Display Name field, verify that the name is `Ejb1`, the default display name. Representing the enterprise bean JAR file that contains the bean, this name will be displayed in the tree view.
3. Click the Edit button next to the Contents text area.
4. In the dialog box Edit Contents of Ejb1, choose the `client_ses_mdb` directory. If the directory is not already in the Starting Directory field, type it in the field, or locate it by browsing through the Available Files tree.
5. Select the files `Publisher.class`, `PublisherBean.class`, and `PublisherHome.class` from the Available Files tree area and click Add.
6. Click OK.
7. Click Next.

8.2.6.3 General Dialog Box

1. In the Bean Type combo box, select the Session radio button.
2. Select the Stateless radio button.
3. In the Enterprise Bean Class combo box, select `PublisherBean`.
4. In the Enterprise Bean Name field, enter `PublisherEJB`.
5. In the Remote Interfaces combo box, select `PublisherHome` for Remote Home Interface and `Publisher` for Remote Interface. Ignore the Local Interfaces combo box.
6. Click Next.

8.2.6.4 Transaction Management Dialog Box

1. Select the Container-Managed radio button.
2. In the Transaction Attribute field opposite the `publishNews` method, verify that Required is selected.
3. Click Next.

8.2.6.5 Environment Entries Dialog Box

Click Next.

8.2.6.6 Enterprise Bean References Dialog Box

Click Next.

8.2.6.7 Resource References Dialog Box

1. Click Add.
2. In the Coded Name field, enter `jms/MyTopicConnectionFactory`.
3. In the Type field, select `javax.jms.TopicConnectionFactory`.
4. In the Authentication field, select Container.
5. In the JNDI Name field, enter `jms/DurableTopicCF`.
6. In the User Name field, enter `j2ee`.

7. In the Password field, enter `j2ee`.
8. Click Next.

8.2.6.8 Resource Environment References Dialog Box

1. Click Add.
2. In the Coded Name field, enter `jms/TopicName`—the logical name referenced by the `PublisherBean`.
3. In the Type field, select `javax.jms.Topic`.
4. In the JNDI Name field, enter `jms/MyTopic`.
5. Click Next.

8.2.6.9 Security Dialog Box

Use the default Security Identity setting for a session or an entity bean, Use Caller ID. Click Next.

8.2.6.10 Review Settings Dialog Box

1. Check the settings for the deployment descriptor.
2. Click Finish.

8.2.7 Packaging the Message-Driven Bean

In this section, you will run the New Enterprise Bean Wizard of the deploytool to package the message-driven bean. To start the New Enterprise Bean Wizard, follow these steps.

1. In the tree view, select `PubSubApp`.
2. From the File menu, choose New → Enterprise Bean. The wizard displays a series of dialog boxes.

8.2.7.1 Introduction Dialog Box

Click Next.

8.2.7.2 EJB JAR Dialog Box

1. In the combo box labeled JAR File Location, verify that Create New JAR File in Application and `PubSubApp` are selected.
2. In the JAR Display Name field, verify that the name is `Ejb2`, the default display name.
3. Click the Edit button next to the Contents text area.
4. In the dialog box Edit Contents of Ejb2, choose the `client_ses_mdb` directory. If the directory is not already in the Starting Directory field, type it in the field, or locate it by browsing through the Available Files tree.
5. Select the `MessageBean.class` file from the Available Files tree area and click Add.
6. Click OK.
7. Click Next.

8.2.7.3 General Dialog Box

1. In the Bean Type combo box, select the Message-Driven radio button.
2. In the Enterprise Bean Class combo box, select `MessageBean`.
3. In the Enterprise Bean Name field, enter `MessageBean`.
4. Click Next.

8.2.7.4 Transaction Management Dialog Box

1. Select the Container-Managed radio button.
2. In the Transaction Type field opposite the `onMessage` method, verify that Required is selected.
3. Click Next.

8.2.7.5 Message-Driven Bean Settings Dialog Box

1. In the Destination Type combo box, select Topic.
2. Check the Durable Subscriber checkbox.

3. In the Subscription Name field, enter `MySub`.
4. In the Destination field, select `jms/MyTopic`.
5. In the Connection Factory field, select `jms/DurableTopicCF`.
6. In the JMS Message Selector field, enter the following exactly as shown:

   ```
   NewsType = 'Opinion' OR NewsType = 'Sports'
   ```

   This will cause the message-driven bean to receive only messages whose `NewsType` property is set to one of these values.
7. Click Finish. You do not need to enter anything in the other dialog boxes.

8.2.8 Specifying the JNDI Names

Verify that the JNDI names are correct, and add one for the `PublisherEJB` component. Follow these steps.

1. In the tree view, select the `PubSubApp` application.
2. Select the JNDI Names tabbed pane.
3. Make sure that the JNDI names appear as shown in Tables 8.1 and 8.2. You will need to enter `MyPublisher` as the JNDI name for the `PublisherEJB` component.

Table 8.1: Application Pane

Component Type	**Component**	**JNDI Name**
EJB	`MessageBean`	`jms/MyTopic`
EJB	`PublisherEJB`	`MyPublisher`

Table 8.2: References Pane

Ref. Type	Referenced By	Reference Name	JNDI Name
Resource	`PublisherEJB`	`jms/MyTopic-ConnectionFactory`	`jms/DurableTopicCF`
Env Resource	`PublisherEJB`	`jms/TopicName`	`jms/MyTopic`
EJB Ref	`MyAppClient`	`ejb/MyEjbReference`	`MyPublisher`

8.3 Deploying and Running the Application

Deploying and running this application involve several steps:

1. Adding the server, if necessary
2. Deploying the application
3. Running the client
4. Undeploying the application
5. Removing the application and stopping the server

8.3.1 Adding the Server

Before you can deploy the application, you must make available to the deploytool the J2EE server you started in Section 8.2.1 on page 115. Because you started the J2EE server before you started the deploytool, the server, named `localhost`, probably appears in the tree under `Servers`. If it does not, do the following.

1. From the File menu, choose Add Server.
2. In the Add Server dialog box, enter `localhost` in the Server Name field.
3. Click OK. A `localhost` node appears under `Servers` in the tree view.

8.3.2 Deploying the Application

To deploy the application, perform the following steps.

1. From the File menu, choose Save to save the application.
2. From the Tools menu, choose Deploy.
3. In the Introduction dialog box, verify that the Object to Deploy selection is `PubSubApp` and that the Target Server selection is `localhost`.
4. Click Next.
5. In the JNDI Names dialog box, verify that the JNDI names are correct.
6. Click Next.
7. Click Finish.
8. In the Deployment Progress dialog box, click OK when the “Deployment of PubSubApp is complete” message appears.
9. In the tree view, expand `Servers` and select `localhost`. Verify that `PubSubApp` is deployed.

8.3.3 Running the Client

To run the client, perform the following steps.

1. At the command line prompt, enter the following:

   ```
   runclient -client PubSubApp.ear -name MyAppClient -textauth
   ```

2. At the login prompts, enter `j2ee` as the user name and `j2ee` as the password.
3. Click OK.

The client program runs in the command window and has output that looks like this:

```
Binding name:‘java:comp/env/ejb/MyEjbReference‘
Looking up EJB reference
Looked up home
Narrowed home
```

```
Got the EJB
Unbinding name:‘java:comp/env/ejb/MyEjbReference‘
```

Output from the application appears in the window in which you started the J2EE server. Suppose that the last few messages from the Publisher session bean look like this:

```
PUBLISHER: Setting message text to: Item 13: Opinion
PUBLISHER: Setting message text to: Item 14: Sports
PUBLISHER: Setting message text to: Item 15: Nation/World
PUBLISHER: Setting message text to: Item 16: Living/Arts
PUBLISHER: Setting message text to: Item 17: Opinion
```

Because of the message selector, the last few messages received by the message-driven bean will look like this:

```
MESSAGE BEAN: Message received: Item 13: Opinion
MESSAGE BEAN: Message received: Item 14: Sports
MESSAGE BEAN: Message received: Item 17: Opinion
```

8.3.4 Undeploying the Application

To undeploy the J2EE application, follow these steps.

1. In the tree view, select `localhost`.
2. Select `PubSubApp` in the Deployed Objects area.
3. Click Undeploy.
4. Answer Yes in the confirmation dialog.

8.3.5 Removing the Application and Stopping the Server

To remove the application from the deploytool, follow these steps.

1. Select `PubSubApp` in the tree view.
2. Select Close from the File menu.

To delete the topic you created, enter the following at the command line prompt:

```
j2eeadmin -removeJmsDestination jms/MyTopic
```

To delete the connection factory you created, enter the following:

```
j2eeadmin -removeJmsFactory jms/DurableTopicCF
```

To stop the J2EE server, use the following command:

```
j2ee -stop
```

CHAPTER 9

A J2EE Application that Uses the JMS API with an Entity Bean

THIS chapter explains how to write, compile, package, deploy, and run a J2EE application that uses the JMS API with an entity bean. The application uses the following components:

- An application client that both sends and receives messages
- Three message-driven beans
- An entity bean that uses container-managed persistence

The chapter covers the following topics:

- An overview of the application
- Writing and compiling the application components
- Creating and packaging the application
- Deploying and running the application

If you downloaded the tutorial examples as described in the preface, you will find the source code files for this chapter in `jms_tutorial/examples/client_mdb_ent` (on UNIX systems) or `jms_tutorial\examples\client_mdb_ent` (on Microsoft Windows systems). The directory `ear_files` in the `examples` directory contains a built application called `SampleNewHireApp.ear`. If you run into difficulty at any time, you can open this file in the deploytool and compare that file to your own version.

9.1 Overview of the Human Resources Application

This application simulates, in a simplified way, the work flow of a company's human resources (HR) department when it processes a new hire. This application also demonstrates how to use the J2EE platform to accomplish a task that many JMS client applications perform.

A JMS client must often wait for several messages from various sources. It then uses the information in all these messages to assemble a message that it then sends to another destination. (The common term for this process is *joining messages*.) Such a task must be transactional, with all the receives and the send as a single transaction. If all the messages are not received successfully, the transaction can be rolled back. For a client example that illustrates this task, see Section A.2 on page 225.

A message-driven bean can process only one message at a time in a transaction. To provide the ability to join messages, a J2EE application can have the message-driven bean store the interim information in an entity bean. The entity bean can then determine whether all the information has been received; when it has, the entity bean can create and send the message to the other destination. Once it has completed its task, the entity bean can be removed.

The basic steps of the application are as follows.

1. The HR department's application client generates an employee ID for each new hire and then publishes a message containing the new hire's name and employee ID. The client then creates a temporary queue with a message listener that waits for a reply to the message.
2. Two message-driven beans process each message: One bean assigns the new hire's office number, and one bean assigns the new hire's equipment. The first bean to process the message creates an entity bean to store the information it has generated. The second bean locates the existing entity bean and adds its information.
3. When both the office and the equipment have been assigned, the entity bean sends to the reply queue a message describing the assignments. The application client's message listener retrieves the information. The entity bean also

sends to a Schedule queue a message that contains a reference to the entity bean.

4. The Schedule message-driven bean receives the message from the entity bean. This message serves as a notification that the entity bean has finished joining all messages. The message contains the primary key to look up the entity bean instance that aggregates the data of the joined messages. The message-driven bean accesses information from the entity bean to complete its task and then removes the entity bean instance.

Figure 9.1 illustrates the structure of this application. An actual HR application would have more components, of course; other beans could set up payroll and benefits records, schedule orientation, and so on.

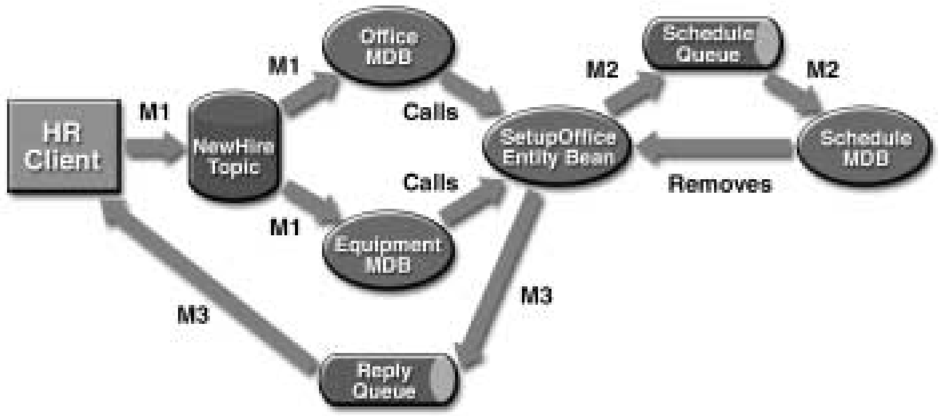

Figure 9.1 A J2EE Application: Client to Message-Driven Beans to Entity Beans

9.2 Writing and Compiling the Application Components

Writing and compiling the components of the application involve

- Coding the application client
- Coding the message-driven beans
- Coding the entity bean
- Compiling the source files

9.2.1 Coding the Application Client: `HumanResourceClient.java`

The application client program, `HumanResourceClient.java`, performs the following steps:

1. Uses the Java Naming and Directory Interface (JNDI) API naming context `java:comp/env` to look up a `TopicConnectionFactory`, a `QueueConnectionFactory`, and a topic
2. Creates a `TemporaryQueue` to receive notification of processing that occurs, based on new-hire events it has published
3. Creates a `QueueReceiver` for the `TemporaryQueue`, sets the `QueueReceiver`'s message listener, and starts the connection
4. Creates a `TopicPublisher` and a `MapMessage`
5. Creates five new employees with randomly generated names, positions, and ID numbers (in sequence) and publishes five messages containing this information

The message listener, `HRListener`, waits for messages that contain the assigned office and equipment for each employee. When a message arrives, the message listener displays the information received and checks to see whether all five messages have arrived yet. When they have, the message listener notifies the main program, which then exits.

```
import javax.jms.*;
import javax.naming.*;
import java.util.*;

/**
 * The HumanResourceClient class is the client program for this
 * J2EE application. It publishes a message describing a new
 * hire business event that other departments can act upon. It
 * also listens for a message reporting the completion of the
 * other departments' actions and displays the results.
*/
public class HumanResourceClient {
    static Object     waitUntilDone = new Object();
static SortedSet  outstandingRequests =
    Collections.synchronizedSortedSet(new TreeSet());

public static void main (String[] args) {
    InitialContext          ic = null;
    TopicConnectionFactory  topicConnectionFactory = null;
    TopicConnection         tConnection = null;
    TopicSession            tSession = null;
    Topic                   pubTopic = null;
    TopicPublisher          tPublisher = null;
    MapMessage              message = null;
    QueueConnectionFactory  queueConnectionFactory = null;
    QueueConnection         qConnection = null;
    QueueSession            qSession = null;
    Queue                   replyQueue = null;
    QueueReceiver           qReceiver = null;

    /*
     * Create a JNDI API InitialContext object.
     */
    try {
        ic = new InitialContext();
    } catch (NamingException e) {
        System.err.println("HumanResourceClient: " +
            "Could not create JNDI API context: " +
            e.toString());
        System.exit(1);
    }

    /*
     * Look up connection factories and topic.  If any do not
     * exist, exit.
     */
    try {
        topicConnectionFactory = (TopicConnectionFactory)
       ic.lookup("java:comp/env/jms/TopicConnectionFactory");
        pubTopic =
         (Topic) ic.lookup("java:comp/env/jms/NewHireTopic");
        queueConnectionFactory = (QueueConnectionFactory)
       ic.lookup("java:comp/env/jms/QueueConnectionFactory");
    } catch (NamingException e) {
          System.err.println("HumanResourceClient: " +
              "JNDI API lookup failed: " + e.toString());
          System.exit(1);
    }

    /*
     * Create topic and queue connections.
     * Create sessions from connections for the publisher
     *   and receiver; false means session is not
     *   transacted.
     * Create temporary queue and receiver, set message
     *   listener, and start connection.
     * Create publisher and MapMessage.
     * Publish new hire business events.
     * Wait for all messages to be processed.
     * Finally, close connection.
     */
    try {
        Random rand = new Random();
        int nextHireID = rand.nextInt(100);

        String[] positions = { "Programmer",
            "Senior Programmer", "Manager", "Director" };
        String[] firstNames = { "Fred", "Robert", "Tom",
            "Steve", "Alfred", "Joe", "Jack", "Harry",
            "Bill", "Gertrude", "Jenny", "Polly", "Ethel",
            "Mary", "Betsy", "Carol", "Edna", "Gwen" };
        String[] lastNames = { "Astaire", "Preston", "Tudor",
            "Stuart", "Drake", "Jones", "Windsor",
            "Hapsburg", "Robinson", "Lawrence", "Wren",
            "Parrott", "Waters", "Martin", "Blair",
            "Bourbon", "Merman", "Verdon" };

        tConnection =
            topicConnectionFactory.createTopicConnection();
```

```
tSession = tConnection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);

qConnection =
    queueConnectionFactory.createQueueConnection();
qSession = qConnection.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);
replyQueue = qSession.createTemporaryQueue();
qReceiver = qSession.createReceiver(replyQueue);
qReceiver.setMessageListener(new HRListener());
qConnection.start();

tPublisher = tSession.createPublisher(pubTopic);

message = tSession.createMapMessage();
message.setJMSReplyTo(replyQueue);
for (int i = 0; i < 5; i++) {
    int currentHireID = nextHireID++;
        String.valueOf(currentHireID));
    message.setString("Name",
        firstNames[rand.nextInt(firstNames.length)]
        + " " +
        lastNames[rand.nextInt(lastNames.length)]);
    message.setString("Position",
        positions[rand.nextInt(positions.length)]);
    System.out.println("PUBLISHER: Setting hire " +
        "ID to " + message.getString("HireID") +
        ", name " + message.getString("Name") +
        ", position " +
        message.getString("Position"));
    tPublisher.publish(message);
 outstandingRequests.add(new Integer(currentHireID));
}

System.out.println("Waiting for " +
    outstandingRequests.size() + " message(s)");
synchronized (waitUntilDone) {
    waitUntilDone.wait();
}
```

```
        } catch (Exception e) {
            System.err.println("HumanResourceClient: " +
                "Exception: " + e.toString());
        } finally {
            if (tConnection != null) {
                try {
                    tConnection.close();
                } catch (Exception e) {
                    System.err.println("HumanResourceClient: " +
                        "Close exception: " + e.toString());
                }
            }
            if (qConnection != null) {
                try {
                    qConnection.close();
                } catch (Exception e) {
                    System.out.println("HumanResourceClient: " +
                                       "Close exception: " +
                                       e.toString());
                }
            }
            System.exit(0);
        }
    }

    /**
     * The HRListener class implements the MessageListener
     * interface by defining an onMessage method.
     */
    static class HRListener implements MessageListener {

        /**
         * onMessage method, which displays the contents of a
         * MapMessage describing the results of processing the
         * new employee, then removes the employee ID from the
         * list of outstanding requests.
         *
         * @param message    the incoming message
         */
```

```
            public void onMessage(Message message) {
                MapMessage msg = (MapMessage) message;
                try {
                    System.out.println("New hire event processed:");
                    Integer id =
                        Integer.valueOf(msg.getString("employeeId"));
                    System.out.println("  Name: " +
                        msg.getString("employeeName"));
                    System.out.println("  Equipment: " +
                        msg.getString("equipmentList"));
                    System.out.println("  Office number: " +
                        msg.getString("officeNumber"));
                    outstandingRequests.remove(id);
                } catch (JMSException je) {
                    System.out.println("HRListener.onMessage(): " +
                        "Exception: " + je.toString());
                }

                if (outstandingRequests.size() == 0) {
                    synchronized(waitUntilDone) {
                        waitUntilDone.notify();
                    }
                } else {
                    System.out.println("Waiting for " +
                        outstandingRequests.size() + " message(s)");
                }
            }
        }
    }
```

Code Example 9.1 `HumanResourceClient.java`

9.2.2 Coding the Message-Driven Beans

This example uses three message-driven beans. Two of them, `ReserveEquipmentMsgBean.java` and `ReserveOfficeMsgBean.java`, take the following steps.

1. The `ejbCreate` method gets a handle to the home interface of the entity bean.
2. The `onMessage` method retrieves the information in the message. The `ReserveEquipmentMsgBean`'s `onMessage` method chooses equipment, based on the new hire's position; the `ReserveOfficeMsgBean`'s `onMessage` method randomly generates an office number.
3. After a slight delay to simulate real-world processing hitches, the `onMessage` method calls a helper method, `compose`.
4. The `compose` method either creates or finds, by primary key, the `SetupOffice` entity bean and uses it to store the equipment or the office information in the database.

```
import java.io.Serializable;
import java.rmi.RemoteException;
import javax.rmi.PortableRemoteObject;
import javax.ejb.*;
import javax.naming.*;
import javax.jms.*;
import java.util.Random;
/**
 * The ReserveEquipmentMsgBean class is a message-driven bean.
 * It implements the javax.ejb.MessageDrivenBean and
 * javax.jms.MessageListener interfaces. It is defined as public
 * (but not final or abstract).  It defines a constructor and the
 * methods ejbCreate, onMessage, setMessageDrivenContext, and
 * ejbRemove.
 */
public class ReserveEquipmentMsgBean implements
        MessageDrivenBean, MessageListener {

    private transient MessageDrivenContext mdc = null;
    private SetupOfficeLocalHome soLocalHome = null;
    private Random processingTime = new Random();
```

```
/**
 * Constructor, which is public and takes no arguments.
 */
public ReserveEquipmentMsgBean() {
    System.out.println("In " +
        "ReserveEquipmentMsgBean.ReserveEquipmentMsgBean()");
}

/**
 * setMessageDrivenContext method, declared as public (but
 * not final or static), with a return type of void, and with
 * one argument of type javax.ejb.MessageDrivenContext.
 *
 * @param mdc    the context to set
 */
public void setMessageDrivenContext(MessageDrivenContext mdc)
{
    System.out.println("In " +
        "ReserveEquipmentMsgBean.setMessageDrivenContext()");
    this.mdc = mdc;
}

/**
 * ejbCreate method, declared as public (but not final or
 * static), with a return type of void, and with no
 * arguments. It looks up the entity bean and gets a handle
 * to its home interface.
 */
public void ejbCreate() {
    System.out.println("In " +
        "ReserveEquipmentMsgBean.ejbCreate()");
    try {
        Context initial = new InitialContext();
        Object objref =
          initial.lookup("java:comp/env/ejb/MyEjbReference");
        soLocalHome = (SetupOfficeLocalHome)
            PortableRemoteObject.narrow(objref,
                SetupOfficeLocalHome.class);
    } catch (Exception ex) {
            System.err.println("ReserveEquipmentMsgBean." +
                "ejbCreate: Exception: " + ex.toString());
        }
    }

    /**
     * onMessage method, declared as public (but not final or
     * static), with a return type of void, and with one argument
     * of type javax.jms.Message.
     *
     * Casts the incoming Message to a MapMessage, retrieves its
     * contents, and assigns equipment appropriate to the new
     * hire's position.  Calls the compose method to store the
     * information in the entity bean.
     *
     * @param inMessage    the incoming message
     */
    public void onMessage(Message inMessage) {
        MapMessage msg = null;
        String key = null;
        String name = null;
        String position = null;
        String equipmentList = null;

        try {
            if (inMessage instanceof MapMessage) {
                msg = (MapMessage) inMessage;
                System.out.println("  ReserveEquipmentMsgBean:" +
                    " Message received.");
                key = msg.getString("HireID");
                name = msg.getString("Name");
                position = msg.getString("Position");

                if (position.equals("Programmer")) {
                    equipmentList = "Desktop System";
                } else if (position.equals("Senior Programmer")){
                    equipmentList = "Laptop";
                } else if (position.equals("Manager")) {
                    equipmentList = "Pager";
                } else if (position.equals("Director")) {
                    equipmentList = "Java Phone";
                } else {
                    equipmentList = "Baton";
                }

                // Simulate processing time taking 1 to 10 seconds.
                Thread.sleep( processingTime.nextInt(10) * 1000);
                compose(key, name, equipmentList, msg);
            } else {
                System.err.println("Message of wrong type: " +
                    inMessage.getClass().getName());
            }
        } catch (JMSException e) {
            System.err.println("ReserveEquipmentMsgBean." +
                "onMessage: JMSException: " + e.toString());
            mdc.setRollbackOnly();
        } catch (Throwable te) {
            System.err.println("ReserveEquipmentMsgBean." +
                "onMessage: Exception: " + te.toString());
        }
    }

    /**
     * compose method, helper to onMessage method.
     *
     * Locates the row of the database represented by the primary
     * key and adds the equipment allocated for the new hire.
     *
     * @param key           employee ID, primary key
     * @param name          employee name
     * @param equipmentList equipment allocated based on position
     * @param msg           the message received
     */
    void compose (String key, String name, String equipmentList,
            Message msg) {
        int num = 0;
        SetupOffice so = null;
```

```
            try {
                try {
                    so = soLocalHome.findByPrimaryKey(key);
                    System.out.println("  ReserveEquipmentMsgBean:" +
                        " Found join entity bean for employeeId " +
                        key);
                } catch (ObjectNotFoundException onfe) {
                    System.err.println("  ReserveEquipmentMsgBean:" +
                        " Creating join entity bean for " +
                        " employeeId " + key);
                    so = soLocalHome.createLocal(key, name);
                }
                so.doEquipmentList(equipmentList, msg);
                System.out.println("  ReserveEquipmentMsgBean: " +
                    "employeeId " + key + " (" +
                    so.getEmployeeName() + ") has the following " +
                    "equipment: " + so.getEquipmentList());
            } catch (Exception ex) {
                System.err.println(" ReserveEquipmentMsgBean." +
                    "compose: Exception: " + ex.toString());
                mdc.setRollbackOnly();
            }
        }

        /**
         * ejbRemove method, declared as public (but not final or
         * static), with a return type of void, and with no
         * arguments.
         */
        public void ejbRemove() {
            System.out.println("In " +
                "ReserveEquipmentMsgBean.ejbRemove()");
        }
    }
```

Code Example 9.2 `ReserveEquipmentMsgBean.java`

```
import java.io.Serializable;
import java.rmi.RemoteException;
import javax.rmi.PortableRemoteObject;
import javax.ejb.*;
import javax.naming.*;
import javax.jms.*;
import java.util.Random;

/**
 * The ReserveOfficeMsgBean class is a message-driven bean. It
 * implements the javax.ejb.MessageDrivenBean and
 * javax.jms.MessageListener interfaces. It is defined as public
 * (but not final or abstract).  It defines a constructor and the
 * methods ejbCreate, onMessage, setMessageDrivenContext, and
 * ejbRemove.
 */
public class ReserveOfficeMsgBean implements MessageDrivenBean,
    MessageListener {

    private transient MessageDrivenContext mdc = null;
    private SetupOfficeLocalHome soLocalHome = null;
    private Random processingTime = new Random();

    /**
     * Constructor, which is public and takes no arguments.
     */
    public ReserveOfficeMsgBean() {
        System.out.println("In " +
            "ReserveOfficeMsgBean.ReserveOfficeMsgBean()");
    }

    /**
     * setMessageDrivenContext method, declared as public (but
     * not final or static), with a return type of void, and with
     * one argument of type javax.ejb.MessageDrivenContext.
     *
     * @param mdc    the context to set
     */
    public void setMessageDrivenContext(MessageDrivenContext mdc)
    {
        System.out.println("In " +
            "ReserveOfficeMsgBean.setMessageDrivenContext()");
        this.mdc = mdc;
    }

    /**
     * ejbCreate method, declared as public (but not final or
     * static), with a return type of void, and with no
     * arguments. It looks up the entity bean and gets a handle
     * to its home interface.
     */
    public void ejbCreate() {
        System.out.println("In " +
            "ReserveOfficeMsgBean.ejbCreate()");
        try {
            Context initial = new InitialContext();
            Object objref =
              initial.lookup("java:comp/env/ejb/MyEjbReference");
            soLocalHome = (SetupOfficeLocalHome)
                PortableRemoteObject.narrow(objref,
                    SetupOfficeLocalHome.class);
        } catch (Exception ex) {
            System.err.println("ReserveOfficeMsgBean." +
                "ejbCreate: Exception: " + ex.toString());
        }
    }

    /**
     * onMessage method, declared as public (but not final or
     * static), with a return type of void, and with one argument
     * of type javax.jms.Message.
     *
     * Casts the incoming Message to a MapMessage, retrieves its
     * contents, and assigns the new hire to an office. Calls the
     * compose method to store the information in the entity
     * bean.
     *
     * @param inMessage    the incoming message
     */
    public void onMessage(Message inMessage) {
        MapMessage msg = null;
        String key = null;
        String name = null;
        String position = null;
        int officeNumber = 0;

        try {
            if (inMessage instanceof MapMessage) {
                msg = (MapMessage) inMessage;
                System.out.println("  >>> ReserveOfficeMsgBean:" +
                    " Message received.");
                key = msg.getString("HireID");
                name = msg.getString("Name");
                position = msg.getString("Position");

                officeNumber = new Random().nextInt(300) + 1;

                // Simulate processing time taking 1 to 10 seconds.
                Thread.sleep( processingTime.nextInt(10) * 1000);
                compose(key, name, officeNumber, msg);
            } else {
                System.err.println("Message of wrong type: " +
                    inMessage.getClass().getName());
            }
        } catch (JMSException e) {
            System.err.println("ReserveOfficeMsgBean." +
                "onMessage: JMSException: " + e.toString());
            mdc.setRollbackOnly();
        } catch (Throwable te) {
            System.err.println("ReserveOfficeMsgBean." +
                "onMessage: Exception: " + te.toString());
        }
    }
```

```
/**
 * compose method, helper to onMessage method.
 *
 * Locates the row of the database represented by the primary
 * key and adds the office number allocated for the new hire.
 *
 * @param key           employee ID, primary key
 * @param name          employee name
 * @param officeNumber  office number
 * @param msg           the message received
 */
void compose (String key, String name, int officeNumber,
        Message msg) {
    int num = 0;
    SetupOffice so = null;

    try {
        try {
            so = soLocalHome.findByPrimaryKey(key);
            System.out.println("  ReserveOfficeMsgBean: " +
                "Found join entity bean for employeeId " +
                key);
        } catch (ObjectNotFoundException onfe) {
            System.out.println("  ReserveOfficeMsgBean: " +
                "Creating join entity bean for " +
                "employeeId " + key);
            so = soLocalHome.createLocal(key, name);
        }
        so.doOfficeNumber(officeNumber, msg);
        System.out.println("  ReserveOfficeMsgBean: " +
            "employeeId " + key + " (" +
            so.getEmployeeName() + ") has the following " +
            "office: " + so.getOfficeNumber());
    } catch (Exception ex) {
        System.err.println(" ReserveOfficeMsgBean." +
            "compose: Exception: " + ex.toString());
        mdc.setRollbackOnly();
    }
}
```

```
    /**
     * ejbRemove method, declared as public (but not final or
     * static), with a return type of void, and with no
     * arguments.
     */
    public void ejbRemove() {
        System.out.println("In " +
            "ReserveOfficeMsgBean.ejbRemove()");
    }
}
```

Code Example 9.3 `ReserveOfficeMsgBean.java`

The third message-driven bean, `ScheduleMsgBean.java`, is notified when the `SetupOfficeBean` entity bean instance has aggregated data from all messages needed to set up an office. The message contains the primary key to look up the correct composite entity bean instance. The `ScheduleMsgBean`'s `onMessage` method then schedules the office setup, based on the information aggregated in the entity bean instance. Finally, the `ScheduleMsgBean`'s `onMessage` method removes the entity bean instance.

```
import java.rmi.RemoteException;
import javax.rmi.PortableRemoteObject;
import javax.ejb.*;
import javax.naming.*;
import javax.jms.*;
import java.util.Random;

/**
 * The ScheduleMsgBean class is a message-driven bean.
 * It implements the javax.ejb.MessageDrivenBean and
 * javax.jms.MessageListener interfaces. It is defined as public
 * (but not final or abstract).  It defines a constructor and the
 * methods ejbCreate, onMessage, setMessageDrivenContext, and
 * ejbRemove.
 */
public class ScheduleMsgBean implements MessageDrivenBean,
    MessageListener {

    private transient MessageDrivenContext mdc = null;

    private SetupOfficeLocalHome soLocalHome = null;

    /**
     * Constructor, which is public and takes no arguments.
     */
    public ScheduleMsgBean() {
        System.out.println("In " +
            "ScheduleMsgBean.ScheduleMsgBean()");
    }

    /**
     * setMessageDrivenContext method, declared as public (but
     * not final or static), with a return type of void, and with
     * one argument of type javax.ejb.MessageDrivenContext.
     *
     * @param mdc    the context to set
     */
    public void setMessageDrivenContext(MessageDrivenContext mdc)
    {
        System.out.println("In " +
            "ScheduleMsgBean.setMessageDrivenContext()");
        this.mdc = mdc;
    }

    /**
     * ejbCreate method, declared as public (but not final or
     * static), with a return type of void, and with no arguments.
     * It looks up the entity bean and gets a handle to its home
     * interface.
     */

    public void ejbCreate() {
        System.out.println("In ScheduleMsgBean.ejbCreate()");
    try {
        Context initial = new InitialContext();
        Object objref =
   initial.lookup("java:comp/env/ejb/CompositeEjbReference");
        soLocalHome = (SetupOfficeLocalHome)
            PortableRemoteObject.narrow(objref,
                SetupOfficeLocalHome.class);
    } catch (Exception ex) {
        System.err.println("ScheduleMsgBean.ejbCreate: " +
             "Exception: " + ex.toString());
    }
}

/**
 * onMessage method, declared as public (but not final or
 * static), with a return type of void, and with one argument
 * of type javax.jms.Message.
 *
 * Casts the incoming Message to a TextMessage, retrieves its
 * handle to the SetupOffice entity bean, and schedules
 * office setup based on information joined in the entity
 * bean. When finished with data, deletes the entity bean.
 *
 * @param inMessage    the incoming message
 */
public void onMessage(Message inMessage) {
    String key = null;
    SetupOffice setupOffice = null;

    try {
        if (inMessage instanceof TextMessage) {
            System.out.println("  ScheduleMsgBean:" +
                " Message received.");
            key = ((TextMessage)inMessage).getText();
            System.out.println("  ScheduleMsgBean: " +
                "Looking up SetupOffice bean by primary " +
                "key=" + key);
            setupOffice = soLocalHome.findByPrimaryKey(key);
```

```
                /*
                 * Schedule office setup using contents of
                 * SetupOffice entity bean.
                 */
                System.out.println("  ScheduleMsgBean: " +
                    "SCHEDULE employeeId=" +
                    setupOffice.getEmployeeId() + ", Name=" +
                    setupOffice.getEmployeeName() +
                    " to be set up in office #" +
                    setupOffice.getOfficeNumber() + " with " +
                    setupOffice.getEquipmentList());

                // All done. Remove SetupOffice entity bean.
                setupOffice.remove();
            } else {
                System.err.println("Message of wrong type: " +
                    inMessage.getClass().getName());
            }
        } catch (JMSException e) {
            System.err.println("ScheduleMsgBean.onMessage: " +
                "JMSException: " + e.toString());
            mdc.setRollbackOnly();

        } catch (Throwable te) {
            System.err.println("ScheduleMsgBean.onMessage: " +
                "Exception: " + te.toString());
        }
    }

    /**
     * ejbRemove method, declared as public (but not final or
     * static), with a return type of void, and with no
     * arguments.
     */
    public void ejbRemove() {
        System.out.println("In ScheduleMsgBean.ejbRemove()");
    }
```

```
}
```

Code Example 9.4 `ScheduleMsgBean.java`

9.2.3 Coding the Entity Bean

The `SetupOffice` bean is an entity bean that uses a local interface. The local interface allows the entity bean and the message-driven beans to be packaged in the same EJB JAR file for maximum efficiency. The entity bean has these components:

- The local home interface, `SetupOfficeLocalHome.java`
- The local interface, `SetupOffice.java`
- The bean class, `SetupOfficeBean.java`

9.2.3.1 The Local Home Interface: `SetupOfficeLocalHome.java`

The local home interface source file is `SetupOfficeLocalHome.java`. It declares the create method, called `createLocal` for a bean that uses a local interface, and one finder method, `findByPrimaryKey`.

```
import java.rmi.RemoteException;
import java.util.Collection;
import javax.ejb.*;

public interface SetupOfficeLocalHome extends EJBLocalHome {

    public SetupOffice createLocal(String hireID, String name)
        throws CreateException;
    public SetupOffice findByPrimaryKey(String hireID)
        throws FinderException;
}
```

Code Example 9.5 `SetupOfficeLocalHome.java`

9.2.3.2 The Local Interface: `SetupOffice.java`

The local interface, `SetupOffice.java`, declares several business methods that get and manipulate new-hire data.

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

public interface SetupOffice extends EJBLocalObject {

    public String getEmployeeId();
    public String getEmployeeName();
    public String getEquipmentList();
    public int    getOfficeNumber();

    public void   doEquipmentList(String list, Message msg)
        throws JMSException;
    public void   doOfficeNumber(int number, Message msg)
        throws JMSException;
}
```

Code Example 9.6 `SetupOffice.java`

9.2.3.3 The Bean Class: `SetupOfficeBean.java`

The bean class, `SetupOfficeBean.java`, implements the business methods and their helper method, `checkIfSetupComplete`. The bean class also implements the required methods `ejbCreateLocal`, `ejbPostCreateLocal`, `setEntityContext`, `unsetEntityContext`, `ejbRemove`, `ejbActivate`, `ejbPassivate`, `ejbLoad`, and `ejb-Store`. The `ejbFindByPrimaryKey` method is generated automatically.

The only methods called by the message-driven beans are the business methods declared in the local interface, the `findByPrimaryKey` method, and the `createLocal` method. The entity bean uses container-managed persistence, so all database calls are generated automatically.

```
import java.io.*;
import java.util.*;
import javax.ejb.*;
import javax.naming.*;
import javax.jms.*;

/**
 * The SetupOfficeBean class implements the business methods of
 * the entity bean.  Because the bean uses version 2.0 of
 * container-managed persistence, the bean class and the
 * accessor methods for fields to be persisted are all declared
 * abstract.
 */
public abstract class SetupOfficeBean implements EntityBean {

    abstract public String getEmployeeId();
    abstract public void setEmployeeId(String id);

    abstract public String getEmployeeName();
    abstract public void setEmployeeName(String name);

    abstract public int getOfficeNumber();
    abstract public void setOfficeNumber(int officeNum);

    abstract public String getEquipmentList();
    abstract public void setEquipmentList(String equip);

    abstract public byte[] getSerializedReplyDestination();
    abstract public void setSerializedReplyDestination(byte[]
        byteArray);

    abstract public String getReplyCorrelationMsgId();
    abstract public void setReplyCorrelationMsgId(String msgId);
    /*
     * There should be a list of replies for each message being
     * joined.  This example is joining the work of separate
     * departments on the same original request, so it is all
     * right to have only one reply destination.  In theory, this
     * should be a set of destinations, with one reply for each
     * unique destination.
     *
     * Because a Destination is not a data type that can be
     * persisted, the persisted field is a byte array,
     * serializedReplyDestination, that is created and accessed
     * with the setReplyDestination and getReplyDestination
     * methods.
     */

    transient private Destination      replyDestination;
    transient private Queue            scheduleQueue;
    transient private QueueConnection  queueConnection;
    private EntityContext context;

    /**
     * The getReplyDestination method extracts the
     * replyDestination from the serialized version that is
     * persisted, using a ByteArrayInputStream and
     * ObjectInputStream to read the object and casting it to a
     * Destination object.
     *
     * @return    the reply destination
     */
    private Destination getReplyDestination() {
        ByteArrayInputStream bais = null;
        ObjectInputStream ois = null;
        byte[] srd = null;

        srd = getSerializedReplyDestination();
        if (replyDestination == null && srd != null) {
            try {
                bais = new ByteArrayInputStream(srd);
                ois = new ObjectInputStream(bais);
                replyDestination = (Destination)ois.readObject();
                ois.close();
            } catch (IOException io) {
            } catch (ClassNotFoundException cnfe) {}
    }
    return replyDestination;
}

/**
 * The setReplyDestination method serializes the reply
 * destination so that it can be persisted.  It uses a
 * ByteArrayOutputStream and an ObjectOutputStream.
 *
 * @param replyDestination    the reply destination
 */
private void setReplyDestination(Destination
        replyDestination) {
    ByteArrayOutputStream baos = null;
    ObjectOutputStream oos = null;
    this.replyDestination = replyDestination;
    try {
        baos = new ByteArrayOutputStream();
        oos = new ObjectOutputStream(baos);
        oos.writeObject(replyDestination);
        oos.close();
        setSerializedReplyDestination(baos.toByteArray());
    } catch (IOException io) {
    }
}

/**
 * The doEquipmentList method stores the assigned equipment
 * in the database and retrieves the reply destination, then
 * determines if setup is complete.
 *
 * @param list      assigned equipment
 * @param msg       the message received
 */
public void doEquipmentList(String list, Message msg)
        throws JMSException {
    setEquipmentList(list);
    setReplyDestination(msg.getJMSReplyTo());
    setReplyCorrelationMsgId(msg.getJMSMessageID());
```

```
        System.out.println("  SetupOfficeBean." +
            "doEquipmentList: equipment is " +
            getEquipmentList() + " (office number " +
            getOfficeNumber() + ")");
        checkIfSetupComplete();
    }

    /**
     * The doOfficeNumber method stores the assigned office
     * number in the database and retrieves the reply
     * destination, then determines if setup is complete.
     *
     * @param officeNum assigned office
     * @param msg       the message received
     */
    public void doOfficeNumber(int officeNum, Message msg)
            throws JMSException {
        setOfficeNumber(officeNum);
        setReplyDestination(msg.getJMSReplyTo());
        setReplyCorrelationMsgId(msg.getJMSMessageID());
        System.out.println("  SetupOfficeBean." +
            "doOfficeNumber: office number is " +
            getOfficeNumber() + " (equipment " +
            getEquipmentList() + ")");
        checkIfSetupComplete();
    }

    /**
     * The checkIfSetupComplete method determines whether
     * both the office and the equipment have been assigned.  If
     * so, it sends messages to the schedule queue and the reply
     * queue with the information about the assignments.
     */

    private void checkIfSetupComplete() {
        QueueConnection qCon = null;
        QueueSession    qSession = null;
        QueueSender     qSender = null;
        TextMessage     schedMsg = null;
```

```
MapMessage      replyMsg = null;

if (getEquipmentList() != null &&
    getOfficeNumber() != -1) {
    System.out.println("  SetupOfficeBean." +
        "checkIfSetupComplete: SCHEDULE" +
        " employeeId=" + getEmployeeId() + ", Name=" +
        getEmployeeName() + " to be set up in office #" +
        getOfficeNumber() + " with " +
        getEquipmentList());

    try {
        qCon = getQueueConnection();
    } catch (Exception ex) {
        throw new EJBException("Unable to connect to " +
            "JMS provider: " + ex.toString());
    }

    try {
        /*
         * Compose and send message to schedule office
         * setup queue.
         */
        qSession = qCon.createQueueSession(true, 0);
        qSender = qSession.createSender(null);
        schedMsg =
            qSession.createTextMessage(getEmployeeId());
        qSender.send(scheduleQueue, schedMsg);

        /*
         * Send reply to messages aggregated by this
         * composite entity bean.
         */
        replyMsg = qSession.createMapMessage();
        replyMsg.setString("employeeId",
            getEmployeeId());
        replyMsg.setString("employeeName",
            getEmployeeName());
            replyMsg.setString("equipmentList",
                getEquipmentList());
            replyMsg.setInt("officeNumber",
                getOfficeNumber());
    replyMsg.setJMSCorrelationID(getReplyCorrelationMsgId());
            qSender.send((Queue)getReplyDestination(),
                replyMsg);
        } catch (JMSException je) {
            System.err.println("SetupOfficeBean." +
                "checkIfSetupComplete: " + "JMSException: " +
                je.toString());
        }
    }
}

/**
 * ejbCreateLocal method, declared as public (but not final
 * or static).  Stores the available information about the
 * new hire in the database.
 *
 * @param newhireID   ID assigned to the new hire
 * @param name        name of the new hire
 *
 * @return            null (required for CMP 2.0)
 */
public String ejbCreateLocal(String newhireID, String name)
    throws CreateException {

    setEmployeeId(newhireID);
    setEmployeeName(name);
    setEquipmentList(null);
    setOfficeNumber(-1);

    this.queueConnection = null;
    return null;
}

public void ejbRemove() {
    closeQueueConnection();
```

```
        System.out.println("  SetupOfficeBean.ejbRemove: " +
            "REMOVING SetupOffice bean employeeId=" +
            getEmployeeId() + ", Name=" + getEmployeeName());
    }

    public void setEntityContext(EntityContext context) {
        this.context = context;
    }

    public void unsetEntityContext() {
        this.context = null;
    }

    public void ejbActivate() {
        setEmployeeId((String) context.getPrimaryKey());
    }

    public void ejbPassivate() {
        setEmployeeId(null);
        closeQueueConnection();
    }

    public void ejbLoad() {}

    public void ejbStore() {}

   public void ejbPostCreateLocal(String newhireID, String name) {}

    /**
     * The getQueueConnection method, called by the
     * checkIfSetupComplete method, looks up the schedule queue
     * and connection factory and creates a QueueConnection.
     *
     * @return   a QueueConnection object
     */
    private QueueConnection getQueueConnection()
            throws NamingException, JMSException {
```

```
        if (queueConnection == null) {
            InitialContext ic = new InitialContext();

            QueueConnectionFactory queueConnectionFactory =
                (QueueConnectionFactory)
           ic.lookup("java:comp/env/jms/QueueConnectionFactory");
            scheduleQueue =
            (Queue) ic.lookup("java:comp/env/jms/ScheduleQueue");
            queueConnection =
                queueConnectionFactory.createQueueConnection();
        }
        return queueConnection;
    }

    /**
     * The closeQueueConnection method, called by the ejbRemove
     * and ejbPassivate methods, closes the QueueConnection that
     * was created by the getQueueConnection method.
     */
    private void closeQueueConnection() {
        if (queueConnection != null) {
            try {
                queueConnection.close();
            } catch (JMSException je) {
                System.err.println("SetupOfficeBean." +
                    "closeQueueConnection: JMSException: " +
                    je.toString());
            }
            queueConnection = null;
        }
    }
}
```

Code Example 9.7 `SetupOfficeBean.java`

9.2.4 Compiling the Source Files

To compile all the files in the application, go to the directory `client_mdb_ent` and do the following.

1. Make sure that you have set the environment variables shown in Table 4.1 on page 34: `JAVA_HOME`, `J2EE_HOME`, `CLASSPATH`, and `PATH`.
2. At a command line prompt, compile the source files:

   ```
   javac *.java
   ```

9.3 Creating and Packaging the Application

Creating and packaging this application involve several steps:

1. Starting the J2EE server and the deploytool*
2. Creating a queue
3. Starting the Cloudscape database server
4. Creating the J2EE application
5. Packaging the application client
6. Packaging the Equipment message-driven bean
7. Packaging the Office message-driven bean
8. Packaging the Schedule message-driven bean
9. Packaging the entity bean
10. Specifying the entity bean deployment settings
11. Specifying the JNDI names

Step 1, marked with an asterisk (*), is not needed if the server and the deploytool are running.

9.3.1 Starting the J2EE Server and the Deploytool

Before you can create and package the application, you must start the J2EE server and the deploytool. Follow these steps.

1. At the command line prompt, start the J2EE server:

   ```
   j2ee -verbose
   ```

   Wait until the server displays the message “J2EE server startup complete.”

   (To stop the server, type `j2ee -stop`.)

2. At another command line prompt, start the deploytool:

   ```
   deploytool
   ```

   (To access the tool’s context-sensitive help, press F1.)

9.3.2 Creating a Queue

For this application, you publish messages by using one of the topics that the J2EE server creates automatically. You create a queue to process the notification that the composite entity bean has aggregated the group of related messages that it was joining. Follow these steps.

1. In the deploytool, select the Tools menu.
2. From the Tools menu, choose Server Configuration.
3. Under the JMS folder, select Destinations.
4. In the JMS Queue Destinations area, click Add.
5. In the text field, enter `jms/ScheduleQueue`.
6. Click OK.
7. If you wish, you can verify that the queue was created:

   ```
   j2eeadmin -listJmsDestination
   ```

9.3.3 Starting the Cloudscape Database Server

The Cloudscape software is included with the J2EE SDK download bundle. You may also run this example with databases provided by other vendors.

From the command line prompt, run the Cloudscape database server:

```
cloudscape -start
```

9.3.4 Creating the J2EE Application

Create a new J2EE application, called `NewHireApp`, and store it in the file named `NewHireApp.ear`. Follow these steps.

1. In the deploytool, select the File menu.
2. From the File menu, choose New → Application.
3. Click Browse next to the Application File Name field, and use the file chooser to locate the directory `client_mdb_ent`.
4. In the File Name field, enter `NewHireApp`.
5. Click New Application.
6. Click OK.

A diamond icon labeled `NewHireApp` appears in the tree view on the left side of the deploytool window. The full path name of `NewHireApp.ear` appears in the General tabbed pane on the right side.

9.3.5 Packaging the Application Client

In this section, you will run the New Application Client Wizard of the deploytool to package the application client. To start the New Application Client Wizard, follow these steps.

1. In the tree view, select `NewHireApp`.
2. From the File menu, choose New → Application Client. The wizard displays a series of dialog boxes.

9.3.5.1 Introduction Dialog Box

Click Next.

9.3.5.2 JAR File Contents Dialog Box

1. In the combo box labeled Create Archive Within Application, select `NewHireApp`.
2. Click the Edit button next to the Contents text area.
3. In the dialog box Edit Contents of <Application Client>, choose the `client_mdb_ent` directory. If the directory is not already in the Starting Directory field, type it in the field, or locate it by browsing through the Available Files tree.
4. Select `HumanResourceClient.class` and `HumanResourceClient$HRListener.class` from the Available Files tree area and click Add.
5. Click OK.
6. Click Next.

9.3.5.3 General Dialog Box

1. In the Application Client combo box, select `HumanResourceClient` in the Main Class field, and enter `HumanResourceClient` in the Display Name field.
2. In the Callback Handler Class combo box, verify that container-managed authentication is selected.
3. Click Next.

9.3.5.4 Environment Entries Dialog Box

Click Next.

9.3.5.5 Enterprise Bean References Dialog Box

Click Next.

9.3.5.6 Resource References Dialog Box

In this dialog box, you associate the JNDI API context names for the connection factories in the `HumanResourceClient.java` source file with the names of the `TopicConnectionFactory` and the `QueueConnectionFactory`. You also specify container authentication for the connection factory resources, defining the user name and the password that the user must enter in order to be able to create a connection. Follow these steps.

1. Click Add.
2. In the Coded Name field, enter `jms/TopicConnectionFactory`—the logical name referenced by `HumanResourceClient`.
3. In the Type field, select `javax.jms.TopicConnectionFactory`.
4. In the Authentication field, select Container.
5. In the Sharable field, make sure that the checkbox is selected. This allows the container to optimize connections.
6. In the JNDI Name field, enter `jms/TopicConnectionFactory`.
7. In the User Name field, enter `j2ee`.
8. In the Password field, enter `j2ee`.
9. Click Add.
10. In the Coded Name field, enter `jms/QueueConnectionFactory`—the logical name referenced by `HumanResourceClient`.
11. In the Type field, select `javax.jms.QueueConnectionFactory`.
12. In the Authentication field, select Container.
13. In the Sharable field, make sure that the checkbox is selected.
14. In the JNDI Name field, enter `jms/QueueConnectionFactory`.
15. In the User Name field, enter `j2ee`. (If the user name and the password appear to be filled in already, make sure that you follow the instructions at the end of Section 9.3.5.8 after you exit the Wizard.)
16. In the Password field, enter `j2ee`.
17. Click Next.

9.3.5.7 JMS Destination References Dialog Box

In this dialog box, you associate the JNDI API context name for the topic in the `HumanResourceClient.java` source file with the name of the default topic. You do not specify the queue, because it is a temporary queue created programmatically rather than administratively and does not have to be specified in the deployment descriptor. Follow these steps.

1. Click Add.
2. In the Coded Name field, enter `jms/NewHireTopic`—the logical name for the publisher topic referenced by `HumanResourceClient`.
3. In the Type field, select `javax.jms.Topic`.
4. In the JNDI Name field, enter `jms/Topic` (the default topic).
5. Click Next.

9.3.5.8 Review Settings Dialog Box

1. Check the settings for the deployment descriptor.
2. Click Finish.

   After you exit the Wizard, do the following.

1. Select the `HumanResourceClient` node in the tree.
2. Select the Resource Refs tabbed pane.
3. Select the second entry in the table, `jms/QueueConnectionFactory`.
4. See whether the User Name and Password fields are filled in. If they are blank, enter `j2ee` in each field.
5. Choose Save from the File menu to save the application.

9.3.6 Packaging the Equipment Message-Driven Bean

In this section, you will run the New Enterprise Bean Wizard of the deploytool to package the first message-driven bean. To start the New Enterprise Bean Wizard, follow these steps.

1. In the tree view, select `NewHireApp`.
2. From the File menu, choose New ➞ Enterprise Bean. The wizard displays a series of dialog boxes.

9.3.6.1 Introduction Dialog Box

Click Next.

9.3.6.2 EJB JAR Dialog Box

1. In the combo box labeled JAR File Location, verify that Create New JAR File in Application and `NewHireApp` are selected.
2. In the JAR Display Name field, verify that the name is `Ejb1`, the default display name. Representing the enterprise bean JAR file that contains the bean, this name will be displayed in the tree view.
3. Click the Edit button next to the Contents text area.
4. In the dialog box Edit Contents of Ejb1, choose the `client_mdb_ent` directory. If the directory is not already in the Starting Directory field, type it in the field, or locate it by browsing through the Available Files tree.
5. Select the `ReserveEquipmentMsgBean.class` file from the Available Files tree area and click Add.
6. Click OK.
7. Click Next.

9.3.6.3 General Dialog Box

1. In the Bean Type combo box, select the Message-Driven radio button.
2. Under Enterprise Bean Class, select `ReserveEquipmentMsgBean`.
3. In the Enterprise Bean Name field, enter `EquipmentMDB`.
4. Click Next.

9.3.6.4 Transaction Management Dialog Box

1. Select the Container-Managed radio button.
2. In the Transaction Attribute field opposite the `onMessage` method, verify that Required is selected.
3. Click Next.

9.3.6.5 Message-Driven Bean Settings Dialog Box

1. In the Destination Type combo box, select Topic.
2. In the Destination field, select `jms/Topic`.
3. In the Connection Factory field, select `jms/TopicConnectionFactory`.
4. Click Next.

9.3.6.6 Environment Entries Dialog Box

Click Next.

9.3.6.7 Enterprise Bean References Dialog Box

1. Click Add.
2. In the Coded Name column, enter `ejb/MyEjbReference`.
3. In the Type column, select Entity.
4. In the Interfaces column, select Local.
5. In the Home Interface column, enter `SetupOfficeLocalHome`.
6. In the Local/Remote Interface column, enter `SetupOffice`.
7. In the Deployment Settings combo box, select Enterprise Bean Name. In the Enterprise Bean Name field, enter `SetupOfficeEJB`.
8. Click Finish. You do not need to enter anything in the other dialog boxes.

9.3.7 Packaging the Office Message-Driven Bean

In this section, you will run the New Enterprise Bean Wizard of the deploytool to package the second message-driven bean. To start the New Enterprise Bean Wizard, follow these steps.

1. In the tree view, select `NewHireApp`.
2. From the File menu, choose New → Enterprise Bean.

9.3.7.1 Introduction Dialog Box

Click Next.

9.3.7.2 EJB JAR Dialog Box

1. In the combo box labeled JAR File Location, select Add to Existing JAR File and select Ejb1 (NewHireApp).
2. Click the Edit button next to the Contents text area.
3. In the dialog box Edit Contents of Ejb1, choose the directory `client_mdb_ent`. If the directory is not already in the Starting Directory field, type it in the field, or locate it by browsing through the Available Files tree.
4. Select the `ReserveOfficeMsgBean.class` file from the Available Files tree area and click Add.
5. Click OK.
6. Click Next.

9.3.7.3 General Dialog Box

1. In the Bean Type combo box, select the Message-Driven radio button.
2. Under Enterprise Bean Class, select `ReserveOfficeMsgBean`. The combo boxes for the local and remote interfaces are grayed out.
3. In the Enterprise Bean Name field, enter `OfficeMDB`. This name will represent the message-driven bean in the tree view.
4. Click Next.

9.3.7.4 Transaction Management Dialog Box

1. Select the Container-Managed radio button.
2. In the Transaction Attribute field opposite the `onMessage` method, verify that Required is selected.
3. Click Next.

9.3.7.5 Message-Driven Bean Settings Dialog Box

1. In the Destination Type combo box, select Topic.
2. In the Destination field, select `jms/Topic`.
3. In the Connection Factory field, select `jms/TopicConnectionFactory`.
4. Click Next.

9.3.7.6 Environment Entries Dialog Box

Click Next.

9.3.7.7 Enterprise Bean References Dialog Box

1. Click Add.
2. In the Coded Name column, enter `ejb/MyEjbReference`.
3. In the Type column, select Entity.
4. In the Interfaces column, select Local.
5. In the Home Interface column, enter `SetupOfficeLocalHome`.
6. In the Local/Remote Interface column, enter `SetupOffice`.
7. In the Deployment Settings combo box, select Enterprise Bean Name. In the Enterprise Bean Name field, enter `SetupOfficeEJB`.
8. Click Finish. You do not need to enter anything in the other dialog boxes.

9.3.8 Packaging the Schedule Message-Driven Bean

In this section, you will run the New Enterprise Bean Wizard of the deploytool to package the third message-driven bean. To start the New Enterprise Bean Wizard, follow these steps.

1. In the tree view, select `NewHireApp`.
2. From the File menu, choose New → Enterprise Bean.

9.3.8.1 Introduction Dialog Box

Click Next.

9.3.8.2 EJB JAR Dialog Box

1. In the combo box labeled JAR File Location, select Add to Existing JAR File and select Ejb1 (NewHireApp).
2. Click the Edit button next to the Contents text area.
3. In the dialog box Edit Contents of Ejb1, choose the directory `client_mdb_ent`. If the directory is not already in the Starting Directory field, type it in the field, or locate it by browsing through the Available Files tree.
4. Select the `ScheduleMsgBean.class` file from the Available Files tree area and click Add.
5. Click OK.
6. Click Next.

9.3.8.3 General Dialog Box

1. In the Bean Type combo box, select the Message-Driven radio button.
2. Under Enterprise Bean Class, select `ScheduleMsgBean`. The combo boxes for the local and remote interfaces are grayed out.
3. In the Enterprise Bean Name field, enter `ScheduleMDB`. This name will represent the message-driven bean in the tree view.
4. Click Next.

9.3.8.4 Transaction Management Dialog Box

1. Select the Container-Managed radio button.
2. In the Transaction Attribute field opposite the `onMessage` method, verify that Required is selected.
3. Click Next.

9.3.8.5 Message-Driven Bean Settings Dialog Box

1. In the Destination Type combo box, select Queue.
2. In the Destination field, select `jms/ScheduleQueue`.
3. In the Connection Factory field, select `jms/QueueConnectionFactory`.
4. Click Next.

9.3.8.6 Environment Entries Dialog Box

Click Next.

9.3.8.7 Enterprise Bean References Dialog Box

1. Click Add.
2. In the Coded Name column, enter `ejb/CompositeEjbReference`.
3. In the Type column, select Entity.
4. In the Interfaces column, select Local.
5. In the Home Interface column, enter `SetupOfficeLocalHome`.
6. In the Local/Remote Interface column, enter `SetupOffice`.
7. In the Deployment Settings combo box, select Enterprise Bean Name. In the Enterprise Bean Name field, enter `SetupOfficeEJB`.
8. Click Finish. You do not need to enter anything in the other dialog boxes.

9.3.9 Packaging the Entity Bean

In this section, you will run the New Enterprise Bean Wizard of the deploytool to package the entity bean. To start the New Enterprise Bean Wizard, follow these steps.

1. In the tree view, select `NewHireApp`.
2. From the File menu, choose New → Enterprise Bean.

9.3.9.1 Introduction Dialog Box

Click Next.

9.3.9.2 EJB JAR Dialog Box

1. In the combo box labeled JAR File Location, select Add to Existing JAR File and select Ejb1 (NewHireApp).
2. Click the Edit button next to the Contents text area.
3. In the dialog box Edit Contents of Ejb1, choose the directory `client_mdb_ent`. If the directory is not already in the Starting Directory field, type it in the field, or locate it by browsing through the Available Files tree.
4. Select the following files from the Available Files tree area and click Add: `SetupOfficeLocalHome.class`, `SetupOffice.class`, and `SetupOffice-Bean.class`.
5. Click OK.
6. Click Next.

9.3.9.3 General Dialog Box

1. In the Bean Type combo box, select the Entity radio button.
2. In the Enterprise Bean Class combo box, select `SetupOfficeBean`.
3. In the Enterprise Bean Name field, enter `SetupOfficeEJB`.
4. In the Local Interfaces combo box, select `SetupOfficeLocalHome` for Local Home Interface and `SetupOffice` for Local Interface.
5. Click Next.

9.3.9.4 Entity Settings Dialog Box

1. Select the radio button labeled Container managed persistence (2.0).
2. Select the checkboxes next to all six fields in the Fields To Be Persisted area: `employeeId`, `employeeName`, `equipmentList`, `officeNumber`, `serialized-ReplyDestination`, and `replyCorrelationMsgId`.
3. In the Abstract Schema Name field, enter `SetupOfficeSchema`.
4. In the Primary Key Class field, enter `java.lang.String`.
5. In the Primary Key Field Name field, select `employeeId`.
6. Click Next.

9.3.9.5 Transaction Management Dialog Box

1. Select the Container-Managed radio button.
2. For all methods, verify that Required is set in the Transaction Attribute column opposite the Local and Local Home radio buttons.
3. Click Next.

9.3.9.6 Environment Entries Dialog Box

Click Next.

9.3.9.7 Enterprise Bean References Dialog Box

Click Next.

9.3.9.8 Resource References Dialog Box

In this dialog box, you specify the connection factory for the Schedule queue and for the reply. Follow these steps.

1. Click Add.
2. In the Coded Name field, enter `jms/QueueConnectionFactory`.
3. In the Type field, select `javax.jms.QueueConnectionFactory`.
4. In the Authentication field, select Container.

5. In the Sharable field, make sure that the checkbox is selected.
6. In the JNDI Name field, enter `jms/QueueConnectionFactory`.
7. In the User Name field, enter `j2ee`.
8. In the Password field, enter `j2ee`.
9. Click Next.

9.3.9.9 Resource Environment References Dialog Box

1. Click Add.
2. In the Coded Name field, enter `jms/ScheduleQueue`—the logical name referenced by `SetupOfficeBean`.
3. In the Type field, select `javax.jms.Queue`.
4. In the JNDI Name field, enter `jms/ScheduleQueue`.
5. Click Next.

9.3.9.10 Security Dialog Box

Use the default Security Identity setting for a session or entity bean, Use Caller ID. Click Next.

9.3.9.11 Review Settings Dialog Box

1. Check the settings for the deployment descriptor.
2. Click Finish.

9.3.10 Specifying the Entity Bean Deployment Settings

Generate the SQL for the entity bean and create the table. Follow these steps.

1. In the tree view, select the `SetupOfficeEJB` entity bean.
2. Select the Entity tabbed pane.
3. Click Deployment Settings.

4. In the Deployment Settings dialog box, perform these steps.
   a. In the Database Table combo box, select the two checkboxes labeled Create table on deploy and Delete table on undeploy.
   b. Click Database Settings.
   c. In the Deployment Settings dialog box that appears, enter `jdbc/Cloudscape` in the Database JNDI Name field. Do not enter a user name or a password.
   d. Click OK.
   e. Click Generate Default SQL.
   f. When the SQL Generation Complete dialog appears, click OK.
   g. Click OK in the dialog box.
5. Choose Save from the File menu to save the application.

9.3.11 Specifying the JNDI Names

Verify that the JNDI names are correct, and add one for the `SetupOfficeEJB` component. Follow these steps.

1. In the tree view, select the `NewHireApp` application.
2. Select the JNDI Names tabbed pane.
3. Make sure that the JNDI names appear as shown in Tables 9.1 and 9.2. You will need to enter `SetupOfficeEJB` as the JNDI name for the `SetupOfficeEJB` component.

Table 9.1: Application Pane

Component Type	**Component**	**JNDI Name**
EJB	`SetupOfficeEJB`	`SetupOfficeEJB`
EJB	`EquipmentMDB`	`jms/Topic`
EJB	`OfficeMDB`	`jms/Topic`
EJB	`ScheduleMDB`	`jms/ScheduleQueue`

Table 9.2: References Pane

Ref. Type	Referenced By	Reference Name	JNDI Name
Resource	`SetupOfficeEJB`	`jms/Queue-ConnectionFactory`	`jms/Queue-ConnectionFactory`
Env Resource	`SetupOfficeEJB`	`jms/ScheduleQueue`	`jms/ScheduleQueue`
Resource	`HumanResource-Client`	`jms/Topic-ConnectionFactory`	`jms/Topic-ConnectionFactory`
Resource	`HumanResource-Client`	`jms/Queue-ConnectionFactory`	`jms/Queue-ConnectionFactory`
Env Resource	`HumanResource-Client`	`jms/NewHireTopic`	`jms/Topic`
Resource	`EJB1[CMP]`		`jdbc/Cloudscape`

9.4 Deploying and Running the Application

Deploying and running the application involve several steps:

1. Adding the server, if necessary
2. Deploying the application
3. Running the client
4. Undeploying the application
5. Removing the application and stopping the server

9.4.1 Adding the Server

Before you can deploy the application, you must make available to the deploytool the J2EE server you started in Section 9.3.1 on page 162. Because you started the J2EE server before you started the deploytool, the server, named `localhost`, probably appears in the tree under `Servers`. If it does not, do the following.

1. From the File menu, choose Add Server.
2. In the Add Server dialog box, enter `localhost` in the Server Name field.
3. Click OK. A `localhost` node appears under `Servers` in the tree view.

9.4.2 Deploying the Application

To deploy the application, perform the following steps.

1. From the File menu, choose Save to save the application.
2. From the Tools menu, choose Deploy.
3. In the Introduction dialog box, verify that the Object to Deploy selection is `NewHireApp` and that the Target Server selection is `localhost`.
4. Click Next.
5. In the JNDI Names dialog box, verify that the JNDI names are correct.
6. Click Next.
7. Click Finish.
8. In the Deployment Progress dialog box, click OK when the “Deployment of NewHireApp is complete” message appears.
9. In the tree view, expand `Servers` and select `localhost`. Verify that `NewHireApp` is deployed.

9.4.3 Running the Client

To run the client, perform the following steps.

1. At the command line prompt, enter the following:

   ```
   runclient -client NewHireApp.ear -name HumanResourceClient -textauth
   ```

2. At the login prompts, enter `j2ee` as the user name and `j2ee` as the password.
3. Click OK.

The client program runs in the command window, and output from the application appears in the window in which you started the J2EE server.

9.4.4 Undeploying the Application

To undeploy the J2EE application, follow these steps.

1. In the tree view, select `localhost`.
2. Select `NewHireApp` in the Deployed Objects area.
3. Click Undeploy.
4. Answer Yes in the confirmation dialog.

9.4.5 Removing the Application and Stopping the Server

To remove the application from the deploytool, follow these steps.

1. Select `NewHireApp` in the tree view.
2. Select Close from the File menu.

To delete the queue you created, enter the following at the command line prompt:

```
j2eeadmin -removeJmsDestination jms/ScheduleQueue
```

To stop the J2EE server, use the following command:

```
j2ee -stop
```

To stop the Cloudscape database server, use the following command:

```
cloudscape -stop
```

To exit the deploytool, choose Exit from the File menu.

CHAPTER 10

An Application Example that Uses Two J2EE Servers

THIS chapter explains how to write, compile, package, deploy, and run a pair of J2EE applications that use the JMS API and run on two J2EE servers. A common practice is to deploy different components of an enterprise application on different systems within a company, and this example illustrates on a small scale how to do this for an application that uses the JMS API.

The applications use the following components:

- An application client that uses two connection factories—one ordinary one and one that is configured to communicate with the remote server—to create two publishers and two subscribers and to publish and to consume messages
- A message-driven bean that is deployed twice—once on the local server and once on the remote one—to process the messages and to send replies

In this chapter, the term *local server* means the server on which the application client is deployed. The term *remote server* means the server on which only the message-driven bean is deployed.

Another possible situation is that an application deployed on a J2EE server must be accessed from another system on which no J2EE server is running. The last section of this chapter discusses how to handle this situation.

The chapter covers the following topics:

- An overview of the applications
- Writing and compiling the application components

- Creating and packaging the applications
- Deploying and running the applications
- Accessing a J2EE application from a remote system that is not running a J2EE server

If you downloaded the tutorial examples as described in the preface, you will find the source code files for this chapter in `jms_tutorial/examples/multi_server` (on UNIX systems) or `jms_tutorial\examples\multi_server` (on Microsoft Windows systems). The directory `ear_files` in the `examples` directory contains two built applications, called `SampleMultiApp.ear` and `SampleReplyBeanApp.ear`. If you run into difficulty at any time, you can open one of these files in the deploytool and compare that file to your own version.

10.1 Overview of the Applications

This pair of applications is somewhat similar to the application in Chapter 7 in that the only components are a client and a message-driven bean. However, the applications here use these components in more complex ways. One application consists of the application client. The other application contains only the message-driven bean and is deployed twice, once on each server.

The basic steps of the applications are as follows.

1. The administrator starts two J2EE servers.
2. On the local server, the administrator creates a connection factory to communicate with the remote server.
3. The application client uses two connection factories—a preconfigured one and the one just created—to create two connections, sessions, publishers, and subscribers. Each publisher publishes five messages.
4. The local and the remote message-driven beans each receive five messages and send replies.
5. The client's message listener consumes the replies.

Figure 10.1 illustrates the structure of this application.

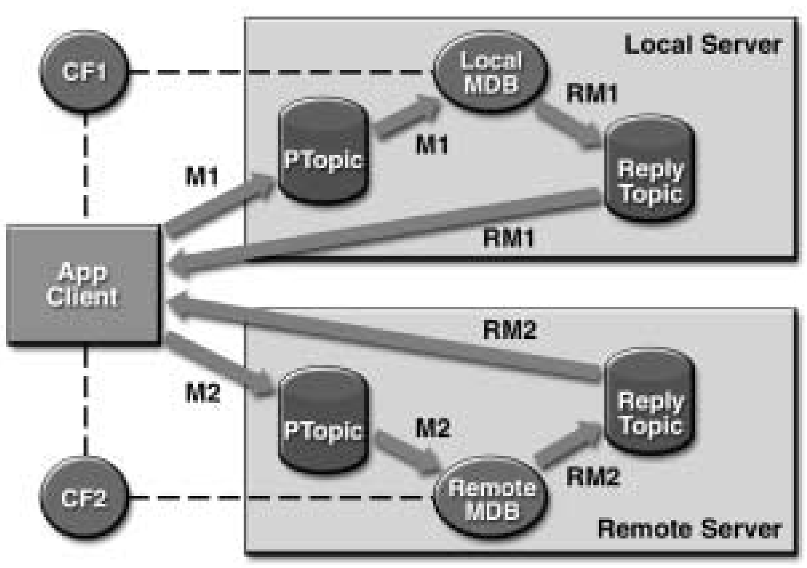

Figure 10.1 A J2EE Two-Server Application

10.2 Writing and Compiling the Application Components

Writing and compiling the components of the applications involve

- Coding the application client
- Coding the message-driven bean
- Compiling the source files

10.2.1 Coding the Application Client: `MultiAppServerRequester.java`

The application client class, `MultiAppServerRequester.java`, does the following.

1. It uses the Java Naming and Directory Interface (JNDI) API naming context `java:comp/env` to look up two connection factories and a topic.
2. For each connection factory, it creates a connection, a publisher session, a publisher, a subscriber session, a subscriber, and a temporary topic for replies.
3. Each subscriber sets its message listener, `ReplyListener`, and starts the connection.

4. Each publisher publishes five messages and creates a list of the messages the listener should expect.
5. When each reply arrives, the message listener displays its contents and removes it from the list of expected messages.
6. When all the messages have arrived, the client exits.

```
import javax.jms.*;
import javax.naming.*;
import java.util.*;

/**
 * The MultiAppServerRequester class is the client program for
 * this J2EE application.  It publishes a message to two
 * different JMS providers and waits for a reply.
 */
public class MultiAppServerRequester {
    static Object     waitUntilDone = new Object();
    static SortedSet  outstandingRequests1 =
        Collections.synchronizedSortedSet(new TreeSet());
    static SortedSet  outstandingRequests2 =
        Collections.synchronizedSortedSet(new TreeSet());

    public static void main (String[] args) {
        InitialContext          ic = null;
        TopicConnectionFactory  tcf1 = null;  // App Server 1
        TopicConnectionFactory  tcf2 = null;  // App Server 2
        TopicConnection         tc1 = null;
        TopicConnection         tc2 = null;
        TopicSession            pubSession1 = null;
        TopicSession            pubSession2 = null;
        TopicPublisher          topicPublisher1 = null;
        TopicPublisher          topicPublisher2 = null;
        Topic                   pTopic = null;
        TemporaryTopic          replyTopic1 = null;
        TemporaryTopic          replyTopic2 = null;
        TopicSession            subSession1 = null;
        TopicSession            subSession2 = null;
```

```
TopicSubscriber        topicSubscriber1 = null;
TopicSubscriber        topicSubscriber2 = null;
TextMessage            message = null;

/*
 * Create a JNDI API InitialContext object.
 */
try {
    ic = new InitialContext();
} catch (NamingException e) {
    System.err.println("Could not create JNDI API " +
        "context: " + e.toString());
    e.printStackTrace();
    System.exit(1);
}

/*
 * Look up connection factories and topic.  If any do not
 * exist, exit.
 */
try {
    tcf1 = (TopicConnectionFactory)
  ic.lookup("java:comp/env/jms/TopicConnectionFactory1");
    tcf2 = (TopicConnectionFactory)
  ic.lookup("java:comp/env/jms/TopicConnectionFactory2");
  pTopic = (Topic) ic.lookup("java:comp/env/jms/PTopic");
} catch (NamingException e) {
    System.err.println("JNDI API lookup failed: " +
        e.toString());
    e.printStackTrace();
    System.exit(1);
}

try {
    // Create two TopicConnections.
    tc1 = tcf1.createTopicConnection();
    tc2 = tcf2.createTopicConnection();
```

```
// Create TopicSessions for publishers.
pubSession1 =
    tc1.createTopicSession(false,
        Session.AUTO_ACKNOWLEDGE);
pubSession2 =
    tc2.createTopicSession(false,
        Session.AUTO_ACKNOWLEDGE);

// Create temporary topics for replies.
replyTopic1 = pubSession1.createTemporaryTopic();
replyTopic2 = pubSession2.createTemporaryTopic();

// Create TopicSessions for subscribers.
subSession1 =
    tc1.createTopicSession(false,
        Session.AUTO_ACKNOWLEDGE);
subSession2 =
    tc2.createTopicSession(false,
        Session.AUTO_ACKNOWLEDGE);

/*
 * Create subscribers, set message listeners, and
 * start connections.
 */
topicSubscriber1 =
    subSession1.createSubscriber(replyTopic1);
topicSubscriber2 =
    subSession2.createSubscriber(replyTopic2);
topicSubscriber1.setMessageListener(new
    ReplyListener(outstandingRequests1));
topicSubscriber2.setMessageListener(new
    ReplyListener(outstandingRequests2));
tc1.start();
tc2.start();

// Create publishers.
topicPublisher1 =
    pubSession1.createPublisher(pTopic);
```

```
topicPublisher2 =
    pubSession2.createPublisher(pTopic);

/*
 * Create and send two sets of messages, one set to
 * each app server, at 1.5-second intervals.  For
 * each message, set the JMSReplyTo message header to
 * a reply topic, and set an id property.  Add the
 * message ID to the list of outstanding requests for
 * the message listener.
 */
message = pubSession1.createTextMessage();
int id = 1;
for (int i = 0; i < 5; i++) {
    message.setJMSReplyTo(replyTopic1);
    message.setIntProperty("id", id);
    message.setText("text: id=" + id +
        " to local app server");
    topicPublisher1.publish(message);
    System.out.println("Published message: " +
        message.getText());
 outstandingRequests1.add(message.getJMSMessageID());
    id++;
    Thread.sleep(1500);
    message.setJMSReplyTo(replyTopic2);
    message.setIntProperty("id", id);
    message.setText("text: id=" + id +
        " to remote app server");
    try {
        topicPublisher2.publish(message);
    System.out.println("Published message: " +
        message.getText());
 outstandingRequests2.add(message.getJMSMessageID());
    } catch (Exception e) {
        System.err.println("Exception: Caught " +
            "failed publish to " +
            "topicConnectionFactory2");
        e.printStackTrace();
    }
```

```
                id++;
                Thread.sleep(1500);
            }

            /*
             * Wait for replies.
             */
            System.out.println("Waiting for " +
                outstandingRequests1.size() + " message(s) " +
                "from local app server");
            System.out.println("Waiting for " +
                outstandingRequests2.size() + " message(s) " +
                "from remote app server");
            while (outstandingRequests1.size() > 0 ||
                   outstandingRequests2.size() > 0 ) {
                synchronized (waitUntilDone) {
                    waitUntilDone.wait();
                }
            }
            System.out.println("Finished");

        } catch (Exception e) {
            System.err.println("Exception occurred: " +
                e.toString());
            e.printStackTrace();
        } finally {
            System.out.println("Closing connection 1");
            if (tc1 != null) {
                try {
                    tc1.close();
                } catch (Exception e) {
                    System.err.println("Error closing " +
                        "connection 1: " + e.toString());
                }
            }
            System.out.println("Closing connection 2");
            if (tc2 != null) {
                try {
                    tc2.close();
            } catch (Exception e) {
                System.err.println("Error closing " +
                    "connection 2: " + e.toString());
            }
        }
        System.exit(0);
    }
}

/**
 * The ReplyListener class is instantiated with a set of
 * outstanding requests.
 */
static class ReplyListener implements MessageListener {
    SortedSet outstandingRequests = null;

    /**
     * Constructor for ReplyListener class.
     *
     * @param outstandingRequests   set of outstanding
     *                              requests
     */
    ReplyListener(SortedSet outstandingRequests) {
        this.outstandingRequests = outstandingRequests;
    }

    /**
     * onMessage method, which displays the contents of the
     * id property and text and uses the JMSCorrelationID to
     * remove from the list of outstanding requests the
     * message to which this message is a reply.  If this is
     * the last message, it notifies the client.
     *
     * @param message     the incoming message
     */
    public void onMessage(Message message) {
        TextMessage  tmsg = (TextMessage) message;
        String       txt = null;
```

```
                int          id = 0;
                String       correlationID = null;

                try {
                    id = tmsg.getIntProperty("id");
                    txt = tmsg.getText();
                    correlationID = tmsg.getJMSCorrelationID();
                } catch (JMSException e) {
                    System.err.println("ReplyListener.onMessage: " +
                        "JMSException: " + e.toString());
                }
                System.out.println("ReplyListener: Received " +
                    "message: id=" + id + ", text=" + txt);
                outstandingRequests.remove(correlationID);

                if (outstandingRequests.size() == 0) {
                    synchronized(waitUntilDone) {
                        waitUntilDone.notify();
                    }
                } else {
                    System.out.println("ReplyListener: Waiting " +
                        "for " + outstandingRequests.size() +
                        " message(s)");
                }
            }
        }
    }
```

Code Example 10.1 `MultiAppServerRequester.java`

10.2.2 Coding the Message-Driven Bean: `ReplyMsgBean.java`

The `onMessage` method of the message-driven bean class, `ReplyMsgBean.java`, does the following:

1. Casts the incoming message to a `TextMessage` and displays the text
2. Creates a connection, session, and publisher for the reply message

3. Publishes the message to the reply topic
4. Closes the connection

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

/**
 * The ReplyMsgBean class is a message-driven bean. It
 * implements the javax.ejb.MessageDrivenBean and
 * javax.jms.MessageListener interfaces. It is defined as public
 * (but not final or abstract).  It defines a constructor and the
 * methods ejbCreate, onMessage, setMessageDrivenContext, and
 * ejbRemove.
 */
public class ReplyMsgBean implements MessageDrivenBean,
        MessageListener {

    private transient MessageDrivenContext mdc = null;
    private transient TopicConnectionFactory tcf = null;

    /**
     * Constructor, which is public and takes no arguments.
     */
    public ReplyMsgBean() {
        System.out.println("In " +
            "ReplyMsgBean.ReplyMsgBean()");
    }

    /**
     * setMessageDrivenContext method, declared as public (but
     * not final or static), with a return type of void, and
     * with one argument of type javax.ejb.MessageDrivenContext.
     *
     * @param mdc    the context to set
     */
```

```
public void setMessageDrivenContext(MessageDrivenContext mdc)
{
    System.out.println("In " +
        "ReplyMsgBean.setMessageDrivenContext()");
    this.mdc = mdc;
}

/**
 * ejbCreate method, declared as public (but not final or
 * static), with a return type of void, and with no
 * arguments. It looks up the topic connection factory.
 */
public void ejbCreate() {
    System.out.println("In ReplyMsgBean.ejbCreate()");
    try {
        Context initial = new InitialContext();
        tcf = (TopicConnectionFactory)
  initial.lookup("java:comp/env/jms/TopicConnectionFactory");
    } catch (Exception ex) {
        System.err.println("ReplyMsgBean.ejbCreate: " +
            "Exception: " + ex.toString());
    }
}

/**
 * onMessage method, declared as public (but not final or
 * static), with a return type of void, and with one argument
 * of type javax.jms.Message.
 *
 * It displays the contents of the message and creates a
 * connection, session, and publisher for the reply, using
 * the JMSReplyTo field of the incoming message as the
 * destination.  It creates and publishes a reply message,
 * setting the JMSCorrelationID header field to the message
 * ID of the incoming message, and the id property to that of
 * the incoming message.  It then closes the topic
 * connection.
 *
 * @param inMessagethe incoming message
 */
public void onMessage(Message inMessage) {
    TextMessage msg = null;
    TopicConnection tc = null;
    TopicSession ts = null;
    TopicPublisher tp = null;
    TextMessage replyMsg = null;

    try {
        if (inMessage instanceof TextMessage) {
            msg = (TextMessage) inMessage;
            System.out.println("  ReplyMsgBean: " +
                "Received message: " + msg.getText());
            tc = tcf.createTopicConnection();
            ts = tc.createTopicSession(true, 0);

            tp =
              ts.createPublisher((Topic)msg.getJMSReplyTo());
            replyMsg =
                ts.createTextMessage("ReplyMsgBean " +
                    "processed message: " + msg.getText());
         replyMsg.setJMSCorrelationID(msg.getJMSMessageID());
            replyMsg.setIntProperty("id",
                msg.getIntProperty("id"));
            tp.publish(replyMsg);
            tc.close();
        } else {
            System.err.println("Message of wrong type: " +
                inMessage.getClass().getName());
        }
    } catch (JMSException e) {
        System.err.println("ReplyMsgBean.onMessage: " +
            "JMSException: " + e.toString());
    } catch (Throwable te) {
        System.err.println("ReplyMsgBean.onMessage: " +
            "Exception: " + te.toString());
    }
}
```

```
        /**
         * ejbRemove method, declared as public (but not final or
         * static), with a return type of void, and with no
         * arguments.
         */
        public void ejbRemove() {
            System.out.println("In ReplyMsgBean.ejbRemove()");
        }
    }
```

Code Example 10.2 `ReplyMsgBean.java`

10.2.3 Compiling the Source Files

To compile the files in the application, go to the directory `multi_server` and do the following.

1. Make sure that you have set the environment variables shown in Table 4.1 on page 34: `JAVA_HOME`, `J2EE_HOME`, `CLASSPATH`, and `PATH`.
2. At a command line prompt, compile the source files:

```
javac MultiAppServerRequester.java
javac ReplyMsgBean.java
```

10.3 Creating and Packaging the Application

Creating and packaging this application involve several steps:

1. Starting the J2EE servers and the deploytool
2. Creating a connection factory
3. Creating the first J2EE application
4. Packaging the application client
5. Creating the second J2EE application

6. Packaging the message-driven bean
7. Checking the JNDI names

10.3.1 Starting the J2EE Servers and the Deploytool

Before you can create and package the application, you must start the local and remote J2EE servers and the deploytool. Follow these steps.

1. At a command line prompt on the local system, start the J2EE server:

   ```
   j2ee -verbose
   ```

   Wait until the server displays the message “J2EE server startup complete.”

   (To stop the server, type `j2ee -stop`.)

2. At another command line prompt on the local system, start the deploytool:

   ```
   deploytool
   ```

   (To access the tool’s context-sensitive help, press F1.)

3. At a command line prompt on the remote system, start the J2EE server:

   ```
   j2ee -verbose
   ```

10.3.2 Creating a Connection Factory

For this example, you create on the local system a connection factory that allows the client to communicate with the remote server. If you downloaded the tutorial examples as described in the preface, you will find in the `multi_server` directory a Microsoft Windows script called `setup.bat` and a UNIX script called `setup.sh`. You can use one of these scripts to create the connection factory on the local system. The command in `setup.bat` looks like this:

```
call j2eeadmin -addJmsFactory jms/RemoteTCF topic -props
url=corbaname:iiop:%1:1050#%1
```

The UNIX command in `setup.sh` looks like this:

```
#!/bin/sh -x
j2eeadmin -addJmsFactory jms/RemoteTCF topic -props
url=corbaname:iiop:$1:1050#$1
```

1. To run the script, specify the name of the remote server as an argument. Use the host name that is visible to you on your network; do not use an IP address. For example, if the remote system is named `mars`, enter the following:

   ```
   setup.bat mars
   ```

   or

   ```
   setup.sh mars
   ```

2. Verify that the connection factory was created:

   ```
   j2eeadmin -listJmsFactory
   ```

   One line of the output looks like this (it appears on one line):

   ```
   < JMS Cnx Factory : jms/RemoteTCF , Topic ,
   [ url=corbaname:iiop:mars:1050#mars ] >
   ```

10.3.3 Creating the First J2EE Application

Create a new J2EE application called `MultiApp` and store it in the file named `MultiApp.ear`. Follow these steps.

1. In the deploytool, select the File menu.
2. From the File menu, choose New → Application.
3. Click Browse next to the Application File Name field, and use the file chooser to locate the directory `multi_server`.
4. In the File Name field, enter `MultiApp`.
5. Click New Application.
6. Click OK.

A diamond icon labeled `MultiApp` appears in the tree view on the left side of the deploytool window. The full path name of `MultiApp.ear` appears in the General tabbed pane on the right side.

10.3.4 Packaging the Application Client

In this section, you will run the New Application Client Wizard of the deploytool to package the application client. To start the New Application Client Wizard, follow these steps.

1. In the tree view, select `MultiApp`.
2. From the File menu, choose New → Application Client. The wizard displays a series of dialog boxes.

10.3.4.1 Introduction Dialog Box

Click Next.

10.3.4.2 JAR File Contents Dialog Box

1. In the combo box labeled Create Archive Within Application, select `MultiApp`.
2. Click the Edit button next to the Contents text area.
3. In the dialog box Edit Contents of <Application Client>, choose the `multi_server` directory. If the directory is not already in the Starting Directory field, type it in the field, or locate it by browsing through the Available Files tree.
4. Select `MultiAppServerRequester.class` and `MultiAppServerRequester$ReplyListener.class` from the Available Files tree area and click Add.
5. Click OK.
6. Click Next.

10.3.4.3 General Dialog Box

1. In the Application Client combo box, select `MultiAppServerRequester` in the Main Class field, and enter `MultiAppServerRequester` in the Display Name field.
2. In the Callback Handler Class combo box, verify that container-managed authentication is selected.
3. Click Next.

10.3.4.4 Environment Entries Dialog Box

Click Next.

10.3.4.5 Enterprise Bean References Dialog Box

Click Next.

10.3.4.6 Resource References Dialog Box

In this dialog box, you associate the JNDI API context names for the connection factories in the `MultiAppServerRequester.java` source file with the names of the local and remote connection factories. You also specify container authentication for the connection factory resources, defining the user name and the password that the user must enter in order to be able to create a connection. Follow these steps.

1. Click Add.
2. In the Coded Name field, enter `jms/TopicConnectionFactory1`—the first logical name referenced by `MultiAppServerRequester`.
3. In the Type field, select `javax.jms.TopicConnectionFactory`.
4. In the Authentication field, select Container.
5. In the Sharable field, make sure that the checkbox is selected. This allows the container to optimize connections.
6. In the JNDI Name field, enter `jms/TopicConnectionFactory`.
7. In the User Name field, enter `j2ee`.
8. In the Password field, enter `j2ee`.
9. Click Add.

10. In the Coded Name field, enter `jms/TopicConnectionFactory2`—the other logical name referenced by `MultiAppServerRequester`.
11. In the Type field, select `javax.jms.TopicConnectionFactory`.
12. In the Authentication field, select Container.
13. In the Sharable field, make sure that the checkbox is selected.
14. In the JNDI Name field, enter `jms/RemoteTCF`.
15. In the User Name field, enter `j2ee`. (If the user name and the password appear to be filled in already, make sure that you follow the instructions at the end of Section 10.3.4.8 after you exit the Wizard.)
16. In the Password field, enter `j2ee`.
17. Click Next.

10.3.4.7 JMS Destination References Dialog Box

In this dialog box, you associate the JNDI API context name for the topic in the `MultiAppServerRequester.java` source file with the name of the default topic. The client code also uses a reply topic, but it is a temporary topic created programmatically rather than administratively and does not have to be specified in the deployment descriptor. Follow these steps.

1. Click Add.
2. In the Coded Name field, enter `jms/PTopic`—the logical name for the publisher topic referenced by `MultiAppServerRequester`.
3. In the Type field, select `javax.jms.Topic`.
4. In the JNDI Name field, enter `jms/Topic`—the preconfigured topic.
5. Click Next.

10.3.4.8 Review Settings Dialog Box

1. Check the settings for the deployment descriptor.
2. Click Finish.

After you exit the Wizard, do the following.

1. Select the `MultiAppServerRequester` node in the tree.
2. Select the Resource Refs tabbed pane.
3. Select the second entry in the table, `jms/TopicConnectionFactory2`.
4. If the User Name and the Password fields are blank, enter `j2ee` in each field.
5. Choose Save from the File menu to save the application.

10.3.5 Creating the Second J2EE Application

Create a new J2EE application, called `ReplyBeanApp`, and store it in the file named `ReplyBeanApp.ear`. Follow these steps.

1. In the deploytool, select the File menu.
2. From the File menu, choose New → Application.
3. Click Browse next to the Application File Name field, and use the file chooser to locate the directory `multi_server`.
4. In the File Name field, enter `ReplyBeanApp`.
5. Click New Application.
6. Click OK.

A diamond icon labeled `ReplyBeanApp` appears in the tree view on the left side of the deploytool window. The full path name of `ReplyBeanApp.ear` appears in the General tabbed pane on the right side.

10.3.6 Packaging the Message-Driven Bean

In this section, you will run the New Enterprise Bean Wizard of the deploytool to package the message-driven bean. To start the New Enterprise Bean Wizard, follow these steps.

1. In the tree view, select `ReplyBeanApp`.
2. From the File menu, choose New → Enterprise Bean.

10.3.6.1 Introduction Dialog Box

Click Next.

10.3.6.2 EJB JAR Dialog Box

1. In the combo box labeled JAR File Location, verify that Create New JAR File in Application and `ReplyBeanApp` are selected.
2. In the JAR Display Name field, verify that the name is `Ejb1`, the default display name.
3. Click the Edit button next to the Contents text area.
4. In the dialog box Edit Contents of Ejb1, choose the `multi_server` directory. If the directory is not already in the Starting Directory field, type it in the field, or locate it by browsing through the Available Files tree.
5. Select the `ReplyMsgBean.class` file from the Available Files tree area and click Add.
6. Click OK.
7. Click Next.

10.3.6.3 General Dialog Box

1. In the Bean Type combo box, select the Message-Driven radio button.
2. Under Enterprise Bean Class, select `ReplyMsgBean`. The combo boxes for the local and remote interfaces are grayed out.
3. In the Enterprise Bean Name field, enter `ReplyMDB`. This name will represent the message-driven bean in the tree view.
4. Click Next.

10.3.6.4 Transaction Management Dialog Box

1. Select the Container-Managed radio button.
2. In the Transaction Attribute field opposite the `onMessage` method, verify that Required is selected.
3. Click Next.

10.3.6.5 Message-Driven Bean Settings Dialog Box

1. In the Destination Type combo box, select Topic.
2. In the Destination field, select `jms/Topic`.
3. In the Connection Factory field, select `jms/TopicConnectionFactory`.
4. Click Next.

10.3.6.6 Environment Entries Dialog Box

Click Next.

10.3.6.7 Enterprise Bean References Dialog Box

Click Next.

10.3.6.8 Resource References Dialog Box

1. Click Add.
2. In the Coded Name field, enter `jms/TopicConnectionFactory`.
3. In the Type field, select `javax.jms.TopicConnectionFactory`.
4. In the Authentication field, select Container.
5. In the JNDI Name field, enter `jms/TopicConnectionFactory`.
6. In the User Name field, enter `j2ee`.
7. In the Password field, enter `j2ee`.
8. Click Finish. You do not need to enter anything in the other dialog boxes.

10.3.7 Checking the JNDI Names

Verify that the JNDI names for the application components are correct. To do so, do the following.

1. In the tree view, select the `MultiApp` application.
2. Select the JNDI Names tabbed pane.
3. Verify that the JNDI names appear as shown in Table 10.1.

Table 10.1: References Pane

Ref. Type	**Referenced By**	**Reference Name**	**JNDI Name**
Resource	`MultiAppServer-Requester`	`jms/Topic-ConnectionFactory1`	`jms/Topic-ConnectionFactory`
Resource	`MultiAppServer-Requester`	`jms/Topic-ConnectionFactory2`	`jms/RemoteTCF`
Env Resource	`MultiAppServer-Requester`	`jms/PTopic`	`jms/Topic`

4. In the tree view, select the `ReplyBeanApp` application.
5. Select the JNDI Names tabbed pane.
6. Verify that the JNDI names appear as shown in Tables 10.2 and 10.3.

Table 10.2: Application Pane

Component Type	**Component**	**JNDI Name**
EJB	`ReplyMDB`	`jms/Topic`

Table 10.3: References Pane

Ref. Type	**Referenced By**	**Reference Name**	**JNDI Name**
Resource	`ReplyMDB`	`jms/Topic-ConnectionFactory`	`jms/Topic-ConnectionFactory`

10.4 Deploying and Running the Applications

Deploying and running this application involve several steps:

1. Adding the server
2. Deploying the applications
3. Running the client
4. Undeploying the applications
5. Removing the applications and stopping the server

10.4.1 Adding the Server

Before you can deploy the application, you must make available to the deploytool both the J2EE servers you started in Section 10.3.1 on page 195. To add the remote server, follow these steps.

1. From the File menu, choose Add Server.
2. In the Add Server dialog box, enter the name of the remote system in the Server Name field. Use the same name you specified when you ran the `setup` script in Section 10.3.2 on page 195.
3. Click OK.

A node with the name of the remote system appears under `Servers` in the tree view.

Because you started the local J2EE server before you started the deploytool, the server, named `localhost`, probably appears in the tree under `Servers`. If it does not, do the following.

1. From the File menu, choose Add Server.
2. In the Add Server dialog box, enter `localhost` in the Server Name field.
3. Click OK.

The `localhost` node appears under `Servers` in the tree view.

10.4.2 Deploying the Applications

To deploy the `MultiApp` application, perform the following steps.

1. In the tree view, select the `MultiApp` application.
2. From the Tools menu, choose Deploy.
3. In the Introduction dialog box, verify that the Object to Deploy selection is `MultiApp`, and select `localhost` as the Target Server.
4. Click Next.
5. In the JNDI Names dialog box, verify that the JNDI names are correct.
6. Click Next.
7. Click Finish.
8. In the Deployment Progress dialog box, click OK when the “Deployment of MultiApp is complete” message appears.
9. In the tree view, expand `Servers` and select the host name. Verify that `MultiApp` is deployed.

To deploy the `ReplyBeanApp` application on the local server, perform the following steps.

1. In the tree view, select the `ReplyBeanApp` application.
2. From the Tools menu, choose Deploy.
3. In the Introduction dialog box, verify that the Object to Deploy selection is `ReplyBeanApp`, and select the local server as the Target Server.
4. Click Next.
5. In the JNDI Names dialog box, verify that the JNDI names are correct.
6. Click Next.
7. Click Finish.
8. In the Deployment Progress dialog box, click OK when the “Deployment of ReplyBeanApp is complete” message appears.
9. In the tree view, expand `Servers` and select the host name. Verify that `ReplyBeanApp` is deployed.

10. Repeat steps 1–9 for the remote server, selecting the remote server as the Target Server in step 3.

10.4.3 Running the Client

To run the client, perform the following steps.

1. At a command line prompt on the local system, enter the following on one line:

```
runclient -client MultiApp.ear -name MultiAppServerRequester
-textauth
```

2. At the login prompts, enter `j2ee` as the user name and `j2ee` as the password.
3. Click OK.

The client program runs in the command window. Output from the message-driven beans appears on both the local and the remote systems, in the windows in which you started each J2EE server.

10.4.4 Undeploying the Applications

To undeploy the J2EE applications, follow these steps.

1. In the tree view, select `localhost` under Servers.
2. Select `MultiApp` in the Deployed Objects area.
3. Click Undeploy.
4. Answer Yes in the confirmation dialog.
5. Repeat steps 1–4 for `ReplyBeanApp` on both the local and the remote servers.

10.4.5 Removing the Applications and Stopping the Servers

To remove the applications from the deploytool, follow these steps.

1. Select `MultiApp` in the tree view.
2. Select Close from the File menu.
3. Repeat these steps for `ReplyBeanApp`.

To delete the connection factory you created, enter the following at a command line prompt on the local system:

```
j2eeadmin -removeJmsFactory jms/RemoteTCF
```

To stop the J2EE servers, use the following command on each system:

```
j2ee -stop
```

To exit the deploytool, choose Exit from the File menu.

10.5 Accessing a J2EE Application from a Remote System that Is Not Running a J2EE Server

To run an application installed on a J2EE server from a system that is not running a J2EE server, you perform tasks similar to those described in Section 4.4.2 on page 58. Again, the J2EE SDK must be installed on both systems. You may also want to use the `runclient` command to run an application client installed on a remote system.

This section describes both of these situations:

- Accessing a J2EE application from a standalone client
- Using `runclient` to access a remote application client

10.5.1 Accessing a J2EE Application from a Standalone Client

You can run a standalone client that uses messages to communicate with a J2EE application. For example, you can use the deploytool to deploy the `ReplyBeanApp` application on a system running the J2EE server, then use a standalone client to publish messages to the topic that the `ReplyMsgBean` is listening on and receive replies on a temporary topic.

For example, suppose that the `ReplyBeanApp` application is deployed on the server running on the system `earth`, and suppose that the standalone client is named `PubSub` and will run on the system `mars`. Section 10.5.1.1 shows the client program.

To specify the remote system on the command line, you use a command line just like the one in Section 4.4.2 (you do so after setting your environment variables as shown in Table 4.1 on page 34).

- On a Microsoft Windows system, type the following command on a single line:

```
java -Djms.properties=%J2EE_HOME%\config\jms_client.properties
-Dorg.omg.CORBA.ORBInitialHost=earth PubSub jms/Topic
```

- On a UNIX system, type the following command on a single line:

```
java -Djms.properties=$J2EE_HOME/config/jms_client.properties
-Dorg.omg.CORBA.ORBInitialHost=earth PubSub jms/Topic
```

If all the remote applications you need to access are deployed on the same server, you can edit the file `%J2EE_HOME%\config\orb.properties` (on Microsoft Windows systems) or `$J2EE_HOME/config/orb.properties` (on UNIX systems) on the local system. The second line of this file looks like this:

```
host=localhost
```

Change `localhost` to the name of the system on which the remote applications are deployed (for example, `earth`):

```
host=earth
```

You can now run the client program as before, but you do not need to specify the option `-Dorg.omg.CORBA.ORBInitialHost`.

10.5.1.1 The Sample Client Program: `PubSub.java`

The sample client program `PubSub.java` can publish messages to a topic that the `ReplyMsgBean` is listening on and receive the message bean's replies.

```
/**
 * The PubSub class consists of
 *
 *  - A main method, which publishes several messages to a topic
 *    and creates a subscriber and a temporary topic on which
 *    to receive replies
 *  - A TextListener class that receives the replies
 *
 * Run this program in conjunction with ReplyBeanApp.
 * Specify a topic name on the command line when you run the
 * program.  By default, the program sends one message.
 * Specify a number after the topic name to send that number
 * of messages.
 *
 * To end the program, enter Q or q on the command line.
 */
import javax.jms.*;
import javax.naming.*;
import java.io.*;

public class PubSub {

    /**
     * Main method.
     *
     * @param args     the topic used by the example and,
     *                 optionally, the number of messages to send
     */
    public static void main(String[] args) {
        String                  topicName = null;
        Context                 jndiContext = null;
        TopicConnectionFactory  topicConnectionFactory = null;
        TopicConnection         topicConnection = null;
```

```
TopicSession            topicSession = null;
Topic                   topic = null;
Topic                   replyTopic = null;
TopicPublisher          topicPublisher = null;
TopicSubscriber         topicSubscriber = null;
TextMessage             message = null;
InputStreamReader       inputStreamReader = null;
char                    answer = '\0';
final int               NUM_MSGS;

if ( (args.length < 1) || (args.length > 2) ) {
    System.out.println("Usage: java " +
        "PubSub <topic-name> " +
        "[<number-of-messages>]");
    System.exit(1);
}
topicName = new String(args[0]);
System.out.println("Topic name is " + topicName);
if (args.length == 2){
    NUM_MSGS = (new Integer(args[1])).intValue();
} else {
    NUM_MSGS = 1;
}

/*
 * Create a JNDI API InitialContext object if none exists
 * yet.
 */
try {
    jndiContext = new InitialContext();
} catch (NamingException e) {
    System.out.println("Could not create JNDI API " +
        "context: " + e.toString());
    e.printStackTrace();
    System.exit(1);
}

/*
 * Look up connection factory and topic.  If either does
 * not exist, exit.
 */
try {
    topicConnectionFactory = (TopicConnectionFactory)
        jndiContext.lookup("jms/TopicConnectionFactory");
    topic = (Topic) jndiContext.lookup(topicName);
} catch (NamingException e) {
    System.out.println("JNDI API lookup failed: " +
        e.toString());
    e.printStackTrace();
    System.exit(1);
}

/*
 * Create connection.
 * Create session from connection; false means session is
 * not transacted.
 * Create publisher, temporary topic, and text message,
 *  setting JMSReplyTo field to temporary topic and
 *  setting an id property.
 * Send messages, varying text slightly.
 * Create subscriber and set message listener to receive
 *  replies.
 * When all messages have been received, enter Q to quit.
 * Finally, close connection.
 */
try {
    topicConnection =
        topicConnectionFactory.createTopicConnection();
    topicSession =
        topicConnection.createTopicSession(false,
            Session.AUTO_ACKNOWLEDGE);
    topicPublisher = topicSession.createPublisher(topic);
    replyTopic = topicSession.createTemporaryTopic();
    message = topicSession.createTextMessage();
    message.setJMSReplyTo(replyTopic);
    int id = 1;
    for (int i = 0; i < NUM_MSGS; i++) {
        message.setText("This is message " + id);
```

```
            message.setIntProperty("id", id);
            System.out.println("Publishing message: " +
                message.getText());
            topicPublisher.publish(message);
            id++;
        }

        topicSubscriber =
            topicSession.createSubscriber(replyTopic);
      topicSubscriber.setMessageListener(new TextListener());
        topicConnection.start();
        System.out.println("To end program, enter Q or q, " +
            "then <return>");
        inputStreamReader = new InputStreamReader(System.in);
        while (!((answer == 'q') || (answer == 'Q'))) {
            try {
                answer = (char) inputStreamReader.read();
            } catch (IOException e) {
                System.out.println("I/O exception: "
                    + e.toString());
            }
        }
    } catch (JMSException e) {
        System.out.println("Exception occurred: " +
            e.toString());
    } finally {
        if (topicConnection != null) {
            try {
                topicConnection.close();
            } catch (JMSException e) {}
        }
    }
}

/**
 * The TextListener class implements the MessageListener
 * interface by defining an onMessage method that displays
 * the contents and id property of a TextMessage.
 *
```

```
     * This class acts as the listener for the PubSub
     * class.
     */
    static class TextListener implements MessageListener {

        /**
         * Casts the message to a TextMessage and displays its
         * text.
         *
         * @param message     the incoming message
         */
        public void onMessage(Message message) {
            TextMessage  msg = null;
            String       txt = null;
            int          id = 0;

            try {
                if (message instanceof TextMessage) {
                    msg = (TextMessage) message;
                    id = msg.getIntProperty("id");
                    txt = msg.getText();
                    System.out.println("Reading message: id=" +
                        id + ", text=" + txt);
                } else {
                    System.out.println("Message of wrong type: "
                        + message.getClass().getName());
                }
            } catch (JMSException e) {
                System.out.println("JMSException in onMessage():"
                    + e.toString());
            } catch (Throwable t) {
                System.out.println("Exception in onMessage():" +
                    t.getMessage());
            }
        }
    }
```

```
}
```

Code Example 10.3 `PubSub.java`

10.5.2 Using `runclient` to Access a Remote Application Client

If you need to run a J2EE application that contains an application client and that is deployed on a remote system, you can use the `runclient` command to do so. For example, if you deploy both `ReplyBeanApp` and `MultiApp` on the server running on `earth`, the steps are as follows.

1. Make sure that the `multi_server` directory on `earth` is accessible to you via the file system so that the `runclient` command can find it.
2. Follow the instructions in Section 10.3.2 on page 195 and create on `mars` a connection factory that will refer to the corresponding connection factory on `earth`.
3. Set the `host` property in the `orb.properties` file in the J2EE SDK on `mars`, as described in Section 10.5.1, because the `runclient` command does not allow you to specify the `ORBInitialHost` value:

   ```
   host=earth
   ```

4. Go to the `multi_server` directory on `earth`—or specify the complete path to the `MultiApp.ear` file—and issue the `runclient` command (on one line):

   ```
   runclient -client MultiApp.ear -name MultiAppServerRequester
   -textauth
   ```

APPENDIX A

JMS Client Examples

THIS appendix contains a number of sample programs that illustrate JMS API concepts and features. The samples are as follows:

- `DurableSubscriberExample.java`, a program that illustrates the use of durable subscriptions
- `TransactedExample.java`, a program that shows how to use transactions in standalone applications
- `AckEquivExample.java`, a program that illustrates acknowledgment modes
- `SampleUtilities.java`, a utility class containing methods called by the other sample programs

The programs are all self-contained threaded applications. The programs include producer and consumer classes that send and receive messages. If you downloaded the tutorial examples as described in the preface, you will find the examples for this chapter in the directory `jms_tutorial/examples/appendix` (on UNIX systems) or `jms_tutorial\examples\appendix` (on Microsoft Windows systems). You can compile and run the examples using the instructions in Chapter 4.

A.1 Durable Subscriptions

The `DurableSubscriberExample.java` program shows how durable subscriptions work. It demonstrates that a durable subscription is active even when the subscriber is not active. The program contains a `DurableSubscriber` class, a

`MultiplePublisher` class, a main method, and a method that instantiates the classes and calls their methods in sequence.

The program begins like any publish/subscribe program: The subscriber starts, the publisher publishes some messages, and the subscriber receives them. At this point, the subscriber closes itself. The publisher then publishes some messages while the subscriber is not active. The subscriber then restarts and receives the messages.

Before you run this program, create a connection factory with a client ID. You can use a command similar to the one shown in Section 8.2.3 on page 116. Then specify the connection factory name and the topic name on the command line when you run the program, as in the following sample command, which should all be on one line, for a Microsoft Windows system:

```
java -Djms.properties=%J2EE_HOME%\config\jms_client.properties
DurableSubscriberExample DurableTopicCF jms/Topic
```

The output looks something like this:

```
Connection factory name is DurableTopicCF
Topic name is jms/Topic
Java(TM) Message Service 1.0.2 Reference Implementation (build b14)
Starting subscriber
PUBLISHER: Publishing message: Here is a message 1
PUBLISHER: Publishing message: Here is a message 2
PUBLISHER: Publishing message: Here is a message 3
SUBSCRIBER: Reading message: Here is a message 1
SUBSCRIBER: Reading message: Here is a message 2
SUBSCRIBER: Reading message: Here is a message 3
Closing subscriber
PUBLISHER: Publishing message: Here is a message 4
PUBLISHER: Publishing message: Here is a message 5
PUBLISHER: Publishing message: Here is a message 6
Starting subscriber
SUBSCRIBER: Reading message: Here is a message 4
SUBSCRIBER: Reading message: Here is a message 5
SUBSCRIBER: Reading message: Here is a message 6
Closing subscriber
Unsubscribing from durable subscription
```

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

public class DurableSubscriberExample {
    String      conFacName = null;
    String      topicName = null;
    static int  startindex = 0;

    /**
     * The DurableSubscriber class contains a constructor, a
     * startSubscriber method, a closeSubscriber method, and a
     * finish method.
     *
     * The class fetches messages asynchronously, using a message
     * listener, TextListener.
     */
    public class DurableSubscriber {
        Context                    jndiContext = null;
        TopicConnectionFactory     topicConnectionFactory = null;
        TopicConnection            topicConnection = null;
        TopicSession               topicSession = null;
        Topic                      topic = null;
        TopicSubscriber            topicSubscriber = null;
        TextListener               topicListener = null;

        /**
         * The TextListener class implements the MessageListener
         * interface by defining an onMessage method for the
         * DurableSubscriber class.
         */
        private class TextListener implements MessageListener {
            final SampleUtilities.DoneLatch  monitor =
                new SampleUtilities.DoneLatch();

            /**
             * Casts the message to a TextMessage and displays
             * its text. A non-text message is interpreted as the
         * end of the message stream, and the message
         * listener sets its monitor state to all done
         * processing messages.
         *
         * @param message    the incoming message
         */
        public void onMessage(Message message) {
            if (message instanceof TextMessage) {
                TextMessage  msg = (TextMessage) message;

                try {
                    System.out.println("SUBSCRIBER: " +
                        "Reading message: " + msg.getText());
                } catch (JMSException e) {
                    System.err.println("Exception in " +
                        "onMessage(): " + e.toString());
                }
            } else {
                monitor.allDone();
            }
        }
    }

    /**
     * Constructor: looks up a connection factory and topic
     * and creates a connection and session.
     */
    public DurableSubscriber() {

        /*
         * Create a JNDI API InitialContext object if none
         * exists yet.
         */
        try {
            jndiContext = new InitialContext();

        } catch (NamingException e) {
            System.err.println("Could not create JNDI API " +
                "context: " + e.toString());
            System.exit(1);
        }

        /*
         * Look up connection factory and topic.  If either
         * does not exist, exit.
         */
        try {
            topicConnectionFactory = (TopicConnectionFactory)
                jndiContext.lookup(conFacName);
        } catch (NamingException e) {
            System.err.println("JNDI API lookup failed: " +
                e.toString());
            System.exit(1);
        }

        try {
            topicConnection =
              topicConnectionFactory.createTopicConnection();
            topicSession =
                topicConnection.createTopicSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            topic = SampleUtilities.getTopic(topicName,
                topicSession);
        } catch (Exception e) {
            System.err.println("Connection problem: " +
                e.toString());
            if (topicConnection != null) {
                try {
                    topicConnection.close();
                } catch (JMSException ee) {}
            }
        System.exit(1);
        }
    }

    /**
     * Stops connection, then creates durable subscriber,
     * registers message listener (TextListener), and starts
```

```
 * message delivery; listener displays the messages
 * obtained.
 */
public void startSubscriber() {
    try {
        System.out.println("Starting subscriber");
        topicConnection.stop();
        topicSubscriber =
            topicSession.createDurableSubscriber(topic,
                "MakeItLast");
        topicListener = new TextListener();
       topicSubscriber.setMessageListener(topicListener);
        topicConnection.start();
    } catch (JMSException e) {
        System.err.println("Exception occurred: " +
            e.toString());
    }
}

/**
 * Blocks until publisher issues a control message
 * indicating end of publish stream, then closes
 * subscriber.
 */
public void closeSubscriber() {
    try {
        topicListener.monitor.waitTillDone();
        System.out.println("Closing subscriber");
        topicSubscriber.close();
    } catch (JMSException e) {
        System.err.println("Exception occurred: " +
            e.toString());
    }
}

/**
 * Closes the connection.
 */
```

```
    public void finish() {
        if (topicConnection != null) {
            try {
                System.out.println("Unsubscribing from " +
                    "durable subscription");
                topicSession.unsubscribe("MakeItLast");
                topicConnection.close();
            } catch (JMSException e) {}
        }
    }
}

/**
 * The MultiplePublisher class publishes several messages to
 * a topic. It contains a constructor, a publishMessages
 * method, and a finish method.
 */
public class MultiplePublisher {
    TopicConnection  topicConnection = null;
    TopicSession     topicSession = null;
    Topic            topic = null;
    TopicPublisher   topicPublisher = null;

    /**
     * Constructor: looks up a connection factory and topic
     * and creates a connection, session, and publisher.
     */
    public MultiplePublisher() {
        TopicConnectionFactory topicConnectionFactory = null;

        try {
            topicConnectionFactory =
                SampleUtilities.getTopicConnectionFactory();
            topicConnection =
              topicConnectionFactory.createTopicConnection();

            topicSession =
                topicConnection.createTopicSession(false,
                    Session.AUTO_ACKNOWLEDGE);
        topic =
            SampleUtilities.getTopic(topicName,
                topicSession);
        topicPublisher =
            topicSession.createPublisher(topic);
    } catch (Exception e) {
        System.err.println("Connection problem: " +
            e.toString());
        if (topicConnection != null) {
            try {
                topicConnection.close();
            } catch (JMSException ee) {}
        }
    System.exit(1);
    }
}

/**
 * Creates text message.
 * Sends some messages, varying text slightly.
 * Messages must be persistent.
 */
public void publishMessages() {
    TextMessage  message = null;
    int          i;
    final int    NUMMSGS = 3;
    final String MSG_TEXT =
                     new String("Here is a message");

    try {
        message = topicSession.createTextMessage();
        for (i = startindex;
                 i < startindex + NUMMSGS; i++) {
            message.setText(MSG_TEXT + " " + (i + 1));
            System.out.println("PUBLISHER: Publishing " +
                "message: " + message.getText());
            topicPublisher.publish(message);
        }
```

```
                /*
                 * Send a non-text control message indicating end
                 * of messages.
                 */
            topicPublisher.publish(topicSession.createMessage());
                startindex = i;
            } catch (JMSException e) {
                System.err.println("Exception occurred: " +
                    e.toString());
            }
        }

        /**
         * Closes the connection.
         */
        public void finish() {
            if (topicConnection != null) {
                try {
                    topicConnection.close();
                } catch (JMSException e) {}
            }
        }
    }

    /**
     * Instantiates the subscriber and publisher classes.
     * Starts the subscriber; the publisher publishes some
     *   messages.
     * Closes the subscriber; while it is closed, the publisher
     *   publishes some more messages.
     * Restarts the subscriber and fetches the messages.
     * Finally, closes the connections.
     */
    public void run_program() {
        DurableSubscriber  durableSubscriber =
                                    new DurableSubscriber();
        MultiplePublisher  multiplePublisher =
                                    new MultiplePublisher();
            durableSubscriber.startSubscriber();
            multiplePublisher.publishMessages();
            durableSubscriber.closeSubscriber();
            multiplePublisher.publishMessages();
            durableSubscriber.startSubscriber();
            durableSubscriber.closeSubscriber();
            multiplePublisher.finish();
            durableSubscriber.finish();
        }

        /**
         * Reads the topic name from the command line, then calls the
         * run_program method.
         *
         * @param args    the topic used by the example
         */
        public static void main(String[] args) {
            DurableSubscriberExample  dse =
                                      new DurableSubscriberExample();

            if (args.length != 2) {
            System.out.println("Usage: java " +
                "DurableSubscriberExample " +
                "<connection_factory_name> <topic_name>");
            System.exit(1);
        }
        dse.conFacName = new String(args[0]);
            System.out.println("Connection factory name is " +
                dse.conFacName);
            dse.topicName = new String(args[1]);
            System.out.println("Topic name is " + dse.topicName);

        dse.run_program();
        }
    }
```

Code Example A.1 `DurableSubscriberExample.java`

A.2 Transactions

The `TransactedExample.java` program demonstrates the use of transactions in a JMS client application. The program represents a highly simplified e-Commerce application, in which the following things happen.

1. A retailer sends a message to the vendor order queue, ordering a quantity of computers, and waits for the vendor's reply.
2. The vendor receives the retailer's order message and places an order message into each of its suppliers' order queues, all in one transaction. This JMS transaction combines one synchronous receive with multiple sends.
3. One supplier receives the order from its order queue, checks its inventory, and sends the items ordered to the destination named in the order message's `JMSReplyTo` field. If it does not have enough in stock, the supplier sends what it has. The synchronous receive and the send take place in one JMS transaction.
4. The other supplier receives the order from its order queue, checks its inventory, and sends the items ordered to the destination named in the order message's `JMSReplyTo` field. If it does not have enough in stock, the supplier sends what it has. The synchronous receive and the send take place in one JMS transaction.
5. The vendor receives the replies from the suppliers from its confirmation queue and updates the state of the order. Messages are processed by an asynchronous message listener; this step illustrates using JMS transactions with a message listener.
6. When all outstanding replies are processed for a given order, the vendor sends a message notifying the retailer whether it can fulfill the order.
7. The retailer receives the message from the vendor.

Figure A.1 illustrates these steps.

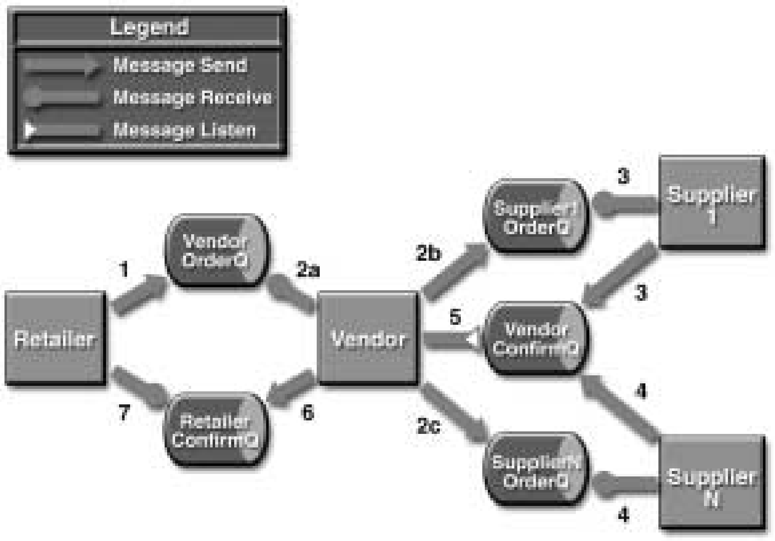

Figure A.1 Transactions: JMS Client Example

The program contains five classes: `Retailer`, `Vendor`, `GenericSupplier`, `VendorMessageListener`, and `Order`. The program also contains a main method and a method that runs the threads of the `Retail`, `Vendor`, and two supplier classes.

All the messages use the `MapMessage` message type. Synchronous receives are used for all message reception except for the case of the vendor processing the replies of the suppliers. These replies are processed asynchronously and demonstrate how to use transactions within a message listener.

At random intervals, the `Vendor` class throws an exception to simulate a database problem and cause a rollback.

All classes except `Retailer` use transacted sessions.

The program uses five queues. Before you run the program, create the queues and name them `A`, `B`, `C`, `D` and `E`.

When you run the program, specify on the command line the number of computers to be ordered. For example, on a Microsoft Windows system:

```
java -Djms.properties=%J2EE_HOME%\config\jms_client.properties
TransactedExample 3
```

The output looks something like this:

```
Quantity to be ordered is 3
Java(TM) Message Service 1.0.2 Reference Implementation (build b14)
Java(TM) Message Service 1.0.2 Reference Implementation (build b14)
Java(TM) Message Service 1.0.2 Reference Implementation (build b14)
Java(TM) Message Service 1.0.2 Reference Implementation (build b14)
Retailer: ordered 3 computer(s)
Vendor: JMSException occurred: javax.jms.JMSException: Simulated
database concurrent access exception
javax.jms.JMSException: Simulated database concurrent access excep-
tion at TransactedExample$Vendor.run(TransactedExample.java:300)
  Vendor: rolled back transaction 1
Vendor: Retailer ordered 3 Computer(s)
Vendor: ordered 3 Monitor(s)
Vendor: ordered 3 Hard Drive(s)
Hard Drive Supplier: Vendor ordered 3 Hard Drive(s)
Hard Drive Supplier: sent 3 Hard Drive(s)
  Vendor: committed transaction 1
Monitor Supplier: Vendor ordered 3 Monitor(s)
Monitor Supplier: sent 3 Monitor(s)
  Hard Drive Supplier: committed transaction
  Monitor Supplier: committed transaction
Vendor: Completed processing for order 1
Vendor: sent 3 computer(s)
  Vendor: committed transaction 2
Retailer: Order filled
Retailer: placing another order
Retailer: ordered 6 computer(s)
Vendor: Retailer ordered 6 Computer(s)
Vendor: ordered 6 Monitor(s)
Vendor: ordered 6 Hard Drive(s)
  Vendor: committed transaction 1
Monitor Supplier: Vendor ordered 6 Monitor(s)
Hard Drive Supplier: Vendor ordered 6 Hard Drive(s)
Hard Drive Supplier: sent 6 Hard Drive(s)
Monitor Supplier: sent 0 Monitor(s)
  Monitor Supplier: committed transaction
  Hard Drive Supplier: committed transaction
```

```
Vendor: Completed processing for order 2
Vendor: unable to send 6 computer(s)
  Vendor: committed transaction 2
Retailer: Order not filled
```

```
import java.util.*;
import javax.jms.*;

public class TransactedExample {
    public static String  vendorOrderQueueName = null;
    public static String  retailerConfirmQueueName = null;
    public static String  monitorOrderQueueName = null;
    public static String  storageOrderQueueName = null;
    public static String  vendorConfirmQueueName = null;

    /**
     * The Retailer class orders a number of computers by sending
     * a message to a vendor.  It then waits for the order to be
     * confirmed.
     *
     * In this example, the Retailer places two orders, one for
     * the quantity specified on the command line and one for
     * twice that number.
     *
     * This class does not use transactions.
     */
    public static class Retailer extends Thread {
        int  quantity = 0;

        /**
         * Constructor.  Instantiates the retailer with the
         * quantity of computers being ordered.
         *
         * @param q    the quantity specified in the program
         *             arguments
         */
```

```
public Retailer(int q) {
    quantity = q;
}

/**
 * Runs the thread.
 */
public void run() {
    QueueConnectionFactory queueConnectionFactory = null;
    QueueConnection        queueConnection = null;
    QueueSession           queueSession = null;
    Queue                  vendorOrderQueue = null;
    Queue                  retailerConfirmQueue = null;
    QueueSender            queueSender = null;
    MapMessage             outMessage = null;
    QueueReceiver          orderConfirmReceiver = null;
    MapMessage             inMessage = null;

    try {
        queueConnectionFactory =
            SampleUtilities.getQueueConnectionFactory();
        queueConnection =
          queueConnectionFactory.createQueueConnection();
        queueSession =
            queueConnection.createQueueSession(false,
                Session.AUTO_ACKNOWLEDGE);
        vendorOrderQueue =
           SampleUtilities.getQueue(vendorOrderQueueName,
                queueSession);
        retailerConfirmQueue =
       SampleUtilities.getQueue(retailerConfirmQueueName,
            queueSession);
    } catch (Exception e) {
        System.err.println("Connection problem: " +
            e.toString());
        System.err.println("Program assumes five " +
            "queues named A B C D E");
    if (queueConnection != null) {
        try {
            queueConnection.close();
        } catch (JMSException ee) {}
    }
    System.exit(1);
}

/*
 * Create non-transacted session and sender for
 * vendor order queue.
 * Create message to vendor, setting item and
 * quantity values.
 * Send message.
 * Create receiver for retailer confirmation queue.
 * Get message and report result.
 * Send an end-of-message-stream message so vendor
 * will stop processing orders.
 */
try {
    queueSender =
        queueSession.createSender(vendorOrderQueue);
    outMessage = queueSession.createMapMessage();
    outMessage.setString("Item", "Computer(s)");
    outMessage.setInt("Quantity", quantity);
    outMessage.setJMSReplyTo(retailerConfirmQueue);
    queueSender.send(outMessage);
    System.out.println("Retailer: ordered " +
        quantity + " computer(s)");

    orderConfirmReceiver =
   queueSession.createReceiver(retailerConfirmQueue);
    queueConnection.start();
    inMessage =
        (MapMessage) orderConfirmReceiver.receive();
    if (inMessage.getBoolean("OrderAccepted")
            == true) {
        System.out.println("Retailer: Order filled");
    } else {
```

```
                System.out.println("Retailer: Order not " +
                    "filled");
            }

            System.out.println("Retailer: placing another " +
                "order");
            outMessage.setInt("Quantity", quantity * 2);
            queueSender.send(outMessage);
            System.out.println("Retailer: ordered " +
                outMessage.getInt("Quantity") +
                " computer(s)");
            inMessage =
                (MapMessage) orderConfirmReceiver.receive();
            if (inMessage.getBoolean("OrderAccepted")
                    == true) {
                System.out.println("Retailer: Order filled");
            } else {
                System.out.println("Retailer: Order not " +
                    "filled");
            }

            /*
             * Send a non-text control message indicating end
             * of messages.
             */
            queueSender.send(queueSession.createMessage());
        } catch (Exception e) {
            System.err.println("Retailer: Exception " +
                "occurred: " + e.toString());
            e.printStackTrace();
        } finally {
            if (queueConnection != null) {
                try {
                    queueConnection.close();
                } catch (JMSException e) {}
            }
        }
    }
}
```

```
/**
 * The Vendor class uses one transaction to receive the
 * computer order from the retailer and order the needed
 * number of monitors and disk drives from its suppliers.
 * At random intervals, it throws an exception to simulate a
 * database problem and cause a rollback.
 *
 * The class uses an asynchronous message listener to process
 * replies from suppliers. When all outstanding supplier
 * inquiries complete, it sends a message to the Retailer
 * accepting or refusing the order.
 */
public static class Vendor extends Thread {
    Random  rgen = new Random();
    int     throwException = 1;

    /**
     * Runs the thread.
     */
    public void run() {
        QueueConnectionFactory queueConnectionFactory = null;
        QueueConnection        queueConnection = null;
        QueueSession           queueSession = null;
        QueueSession           asyncQueueSession = null;
        Queue                  vendorOrderQueue = null;
        Queue                  monitorOrderQueue = null;
        Queue                  storageOrderQueue = null;
        Queue                  vendorConfirmQueue = null;
        QueueReceiver        vendorOrderQueueReceiver = null;
        QueueSender          monitorOrderQueueSender = null;
        QueueSender          storageOrderQueueSender = null;
        MapMessage           orderMessage = null;
        QueueReceiver      vendorConfirmQueueReceiver = null;
        VendorMessageListener   listener = null;
        Message                 inMessage = null;
        MapMessage              vendorOrderMessage = null;
        Message                 endOfMessageStream = null;
        Order                   order = null;
        int                     quantity = 0;
```

```
try {
    queueConnectionFactory =
        SampleUtilities.getQueueConnectionFactory();
    queueConnection =
      queueConnectionFactory.createQueueConnection();
    queueSession =
        queueConnection.createQueueSession(true, 0);
    asyncQueueSession =
        queueConnection.createQueueSession(true, 0);
    vendorOrderQueue =
       SampleUtilities.getQueue(vendorOrderQueueName,
            queueSession);
    monitorOrderQueue =
      SampleUtilities.getQueue(monitorOrderQueueName,
            queueSession);
    storageOrderQueue =
      SampleUtilities.getQueue(storageOrderQueueName,
            queueSession);
    vendorConfirmQueue =
     SampleUtilities.getQueue(vendorConfirmQueueName,
            queueSession);
} catch (Exception e) {
    System.err.println("Connection problem: " +
        e.toString());
    System.err.println("Program assumes five " +
        "queues named A B C D E");
    if (queueConnection != null) {
        try {
            queueConnection.close();
        } catch (JMSException ee) {}
    }
    System.exit(1);
}

try {
    /*
     * Create receiver for vendor order queue, sender
     * for supplier order queues, and message to send
     * to suppliers.
```

```
             */
            vendorOrderQueueReceiver =
               queueSession.createReceiver(vendorOrderQueue);
            monitorOrderQueueSender =
                queueSession.createSender(monitorOrderQueue);
            storageOrderQueueSender =
                queueSession.createSender(storageOrderQueue);
            orderMessage = queueSession.createMapMessage();

            /*
             * Configure an asynchronous message listener to
             * process supplier replies to inquiries for
             * parts to fill order.  Start delivery.
             */
            vendorConfirmQueueReceiver =
        asyncQueueSession.createReceiver(vendorConfirmQueue);
            listener =
                new VendorMessageListener(asyncQueueSession,
                    2);
     vendorConfirmQueueReceiver.setMessageListener(listener);
            queueConnection.start();

            /*
             * Process orders in vendor order queue.
             * Use one transaction to receive order from
             * order queue and send messages to suppliers'
             * order queues to order components to fulfill
             * the order placed with the vendor.
             */
            while (true) {
                try {

                    // Receive an order from a retailer.
                    inMessage =
                        vendorOrderQueueReceiver.receive();
                    if (inMessage instanceof MapMessage) {
                        vendorOrderMessage =
                            (MapMessage) inMessage;
                    } else {
                        /*
                         * Message is an end-of-message-
                         * stream message from retailer.
                         * Send similar messages to
                         * suppliers, then break out of
                         * processing loop.
                         */
                        endOfMessageStream =
                            queueSession.createMessage();
endOfMessageStream.setJMSReplyTo(vendorConfirmQueue);
    monitorOrderQueueSender.send(endOfMessageStream);
    storageOrderQueueSender.send(endOfMessageStream);
                        queueSession.commit();
                        break;
                    }

                    /*
                     * A real application would check an
                     * inventory database and order only the
                     * quantities needed.  Throw an exception
                     * every few times to simulate a database
                     * concurrent-access exception and cause
                     * a rollback.
                     */
                    if (rgen.nextInt(3) == throwException) {
                        throw new JMSException("Simulated " +
                        "database concurrent access " +
                        "exception");
                    }

                    /*
                     * Record retailer order as a pending
                     * order.
                     */
                    order = new Order(vendorOrderMessage);

                    /*
                     * Set order number and reply queue for
                     * outgoing message.
```

```
         */
        orderMessage.setInt("VendorOrderNumber",
            order.orderNumber);
  orderMessage.setJMSReplyTo(vendorConfirmQueue);
quantity = vendorOrderMessage.getInt("Quantity");
        System.out.println("Vendor: Retailer " +
            "ordered " + quantity + " " +
           vendorOrderMessage.getString("Item"));

        // Send message to monitor supplier.
        orderMessage.setString("Item",
            "Monitor");
        orderMessage.setInt("Quantity",
            quantity);
      monitorOrderQueueSender.send(orderMessage);
        System.out.println("Vendor: ordered " +
            quantity + " " +
            orderMessage.getString("Item") +
            "(s)");

        /*
         * Reuse message to send to storage
         * supplier, changing only item name.
         */
        orderMessage.setString("Item",
            "Hard Drive");
      storageOrderQueueSender.send(orderMessage);
        System.out.println("Vendor: ordered " +
            quantity + " " +
            orderMessage.getString("Item") +
            "(s)");

        // Commit session.
        queueSession.commit();
        System.out.println("  Vendor: " +
            "committed transaction 1");
    } catch(JMSException e) {
        System.err.println("Vendor: " +
            "JMSException occurred: " +
                        e.toString());
                    e.printStackTrace();
                    queueSession.rollback();
                    System.err.println("  Vendor: rolled " +
                        "back transaction 1");
                }
            }

            // Wait till suppliers get back with answers.
            listener.monitor.waitTillDone();
        } catch (JMSException e) {
            System.err.println("Vendor: Exception " +
                "occurred: " + e.toString());
            e.printStackTrace();
        } finally {
            if (queueConnection != null) {
                try {
                    queueConnection.close();
                } catch (JMSException e) {}
            }
        }
    }
}

/**
 * The Order class represents a Retailer order placed with a
 * Vendor. It maintains a table of pending orders.
 */
public static class Order {
    private static Hashtable pendingOrders = new Hashtable();
    private static int       nextOrderNumber = 1;
    private static final int  PENDING_STATUS   = 1;
    private static final int  CANCELLED_STATUS = 2;
    private static final int  FULFILLED_STATUS = 3;
    int                       status;
    public final int          orderNumber;
    public int                quantity;

    // Original order from retailer
```

```
public final MapMessage   order;
// Reply from supplier
public MapMessage         monitor = null;
// Reply from supplier
public MapMessage         storage = null;

/**
 * Returns the next order number and increments the
 * static variable that holds this value.
 *
 * @return    the next order number
 */
private static int getNextOrderNumber() {
    int  result = nextOrderNumber;
    nextOrderNumber++;
    return result;
}

/**
 * Constructor.  Sets order number; sets order and
 * quantity from incoming message. Sets status to
 * pending, and adds order to hash table of pending
 * orders.
 *
 * @param order    the message containing the order
 */
public Order(MapMessage order) {
    this.orderNumber = getNextOrderNumber();
    this.order = order;
    try {
        this.quantity = order.getInt("Quantity");
    } catch (JMSException je) {
        System.err.println("Unexpected error. Message " +
            "missing Quantity");
        this.quantity = 0;
    }
    status = PENDING_STATUS;
    pendingOrders.put(new Integer(orderNumber), this);
}
```

```
/**
 * Returns the number of orders in the hash table.
 *
 * @return    the number of pending orders
 */
public static int outstandingOrders() {
    return pendingOrders.size();
}

/**
 * Returns the order corresponding to a given order
 * number.
 *
 * @param orderNumber   the number of the requested order
 * @return              the requested order
 */
public static Order getOrder(int orderNumber) {
    return (Order)
        pendingOrders.get(new Integer(orderNumber));
}

/**
 * Called by the onMessage method of the
 * VendorMessageListener class to process a reply from
 * a supplier to the Vendor.
 *
 * @param component    the message from the supplier
 * @return             the order with updated status
 *                     information
 */
public Order processSubOrder(MapMessage component) {
    String  itemName = null;

    // Determine which subcomponent this is.
    try {
        itemName = component.getString("Item");
    } catch (JMSException je) {
        System.err.println("Unexpected exception. " +
            "Message missing Item");
        }
        if (itemName.compareTo("Monitor") == 0) {
            monitor = component;
        } else if (itemName.compareTo("Hard Drive") == 0 ) {
            storage = component;
        }

        /*
         * If notification for all subcomponents has been
         * received, verify the quantities to compute if able
         * to fulfill order.
         */
        if ( (monitor != null) && (storage != null) ) {
            try {
                if (quantity > monitor.getInt("Quantity")) {
                    status = CANCELLED_STATUS;
             } else if (quantity >
                            storage.getInt("Quantity")) {
                    status = CANCELLED_STATUS;
                } else {
                    status = FULFILLED_STATUS;
                }
            } catch (JMSException je) {
                System.err.println("Unexpected exception: " +
                    je.toString());
                status = CANCELLED_STATUS;
            }

            /*
             * Processing of order is complete, so remove it
             * from pending-order list.
             */
            pendingOrders.remove(new Integer(orderNumber));
        }
        return this;
    }

    /**
     * Determines if order status is pending.
```

```
         *
         * @return    true if order is pending, false if not
         */
        public boolean isPending() {
            return status == PENDING_STATUS;
        }

        /**
         * Determines if order status is cancelled.
         *
         * @return    true if order is cancelled, false if not
         */
        public boolean isCancelled() {
            return status == CANCELLED_STATUS;
        }

        /**
         * Determines if order status is fulfilled.
         *
         * @return    true if order is fulfilled, false if not
         */
        public boolean isFulfilled() {
            return status == FULFILLED_STATUS;
        }
    }

    /**
     * The VendorMessageListener class processes an order
     * confirmation message from a supplier to the vendor.
     *
     * It demonstrates the use of transactions within message
     * listeners.
     */
    public static class VendorMessageListener
            implements MessageListener {
        final SampleUtilities.DoneLatch  monitor =
                                 new SampleUtilities.DoneLatch();
        private final                    QueueSession session;
        int                              numSuppliers;
```

```
/**
 * Constructor.  Instantiates the message listener with
 * the session of the consuming class (the vendor).
 *
 * @param qs               the session of the consumer
 * @param numSuppliers    the number of suppliers
 */
public VendorMessageListener(QueueSession qs,
                             int numSuppliers) {
    this.session = qs;
    this.numSuppliers = numSuppliers;
}

/**
 * Casts the message to a MapMessage and processes the
 * order. A message that is not a MapMessage is
 * interpreted as the end of the message stream, and the
 * message listener sets its monitor state to all done
 * processing messages.
 *
 * Each message received represents a fulfillment message
 * from a supplier.
 *
 * @param message    the incoming message
 */
public void onMessage(Message message) {

    /*
     * If message is an end-of-message-stream message and
     * this is the last such message, set monitor status
     * to all done processing messages and commit
     * transaction.
     */
    if (! (message instanceof MapMessage)) {
        if (Order.outstandingOrders() == 0) {
            numSuppliers--;
            if (numSuppliers == 0) {
       monitor.allDone();
   }
```

```
      }
      try {
          session.commit();
      } catch (JMSException je) {}
      return;
  }

  /*
   * Message is an order confirmation message from a
   * supplier.
   */
  int orderNumber = -1;
  try {
      MapMessage component = (MapMessage) message;

      /*
       * Process the order confirmation message and
       * commit the transaction.
       */
      orderNumber =
          component.getInt("VendorOrderNumber");
      Order order =
Order.getOrder(orderNumber).processSubOrder(component);
      session.commit();

      /*
       * If this message is the last supplier message,
       * send message to Retailer and commit
       * transaction.
       */
      if (! order.isPending()) {
          System.out.println("Vendor: Completed " +
              "processing for order " +
              order.orderNumber);
          Queue replyQueue =
              (Queue) order.order.getJMSReplyTo();
          QueueSender qs =
              session.createSender(replyQueue);
```

```
                MapMessage retailerConfirmMessage =
                    session.createMapMessage();
                if (order.isFulfilled()) {
           retailerConfirmMessage.setBoolean("OrderAccepted",
                        true);
                    System.out.println("Vendor: sent " +
                        order.quantity + " computer(s)");
                } else if (order.isCancelled()) {
           retailerConfirmMessage.setBoolean("OrderAccepted",
                        false);
                    System.out.println("Vendor: unable to " +
                        "send " + order.quantity +
                        " computer(s)");
                }
                qs.send(retailerConfirmMessage);
                session.commit();
                System.out.println("  Vendor: committed " +
                    "transaction 2");
            }
        } catch (JMSException je) {
            je.printStackTrace();
            try {
                session.rollback();
            } catch (JMSException je2) {}
        } catch (Exception e) {
            e.printStackTrace();
            try {
                session.rollback();
            } catch (JMSException je2) {}
        }
    }
}

/**
 * The GenericSupplier class receives an item order from the
 * vendor and sends a message accepting or refusing it.
 */
public static class GenericSupplier extends Thread {
    final String  PRODUCT_NAME;
final String  IN_ORDER_QUEUE;
int           quantity = 0;

/**
 * Constructor.  Instantiates the supplier as the
 * supplier for the kind of item being ordered.
 *
 * @param itemName    the name of the item being ordered
 * @param inQueue     the queue from which the order is
 *                    obtained
 */
public GenericSupplier(String itemName, String inQueue) {
    PRODUCT_NAME = itemName;
    IN_ORDER_QUEUE = inQueue;
}

/**
 * Checks to see if there are enough items in inventory.
 * Rather than go to a database, it generates a random
 * number related to the order quantity, so that some of
 * the time there won't be enough in stock.
 *
 * @return    the number of items in inventory
 */
public int checkInventory() {
    Random  rgen = new Random();

    return (rgen.nextInt(quantity * 5));
}

/**
 * Runs the thread.
 */
public void run() {
    QueueConnectionFactory queueConnectionFactory = null;
    QueueConnection        queueConnection = null;
    QueueSession           queueSession = null;
    Queue                  orderQueue = null;
    QueueReceiver          queueReceiver = null;
```

```
Message                 inMessage = null;
MapMessage              orderMessage = null;
MapMessage              outMessage = null;

try {
    queueConnectionFactory =
        SampleUtilities.getQueueConnectionFactory();
    queueConnection =
      queueConnectionFactory.createQueueConnection();
    queueSession =
        queueConnection.createQueueSession(true, 0);
    orderQueue =
        SampleUtilities.getQueue(IN_ORDER_QUEUE,
            queueSession);
} catch (Exception e) {
    System.err.println("Connection problem: " +
        e.toString());
    System.err.println("Program assumes five " +
        "queues named A B C D E");
    if (queueConnection != null) {
        try {
            queueConnection.close();
        } catch (JMSException ee) {}
    }
    System.exit(1);
}

/*
 * Create receiver for order queue and start message
 * delivery.
 */
try {
    queueReceiver =
        queueSession.createReceiver(orderQueue);
    queueConnection.start();
} catch (JMSException je) {}

/*
 * Keep checking supplier order queue for order
```

```
         * request until end-of-message-stream message is
         * received. Receive order and send an order
         * confirmation as one transaction.
         */
        while (true) {
            try {
                inMessage = queueReceiver.receive();
                if (inMessage instanceof MapMessage) {
                    orderMessage = (MapMessage) inMessage;
                } else {
                    /*
                     * Message is an end-of-message-stream
                     * message. Send a similar message to
                     * reply queue, commit transaction, then
                     * stop processing orders by breaking out
                     * of loop.
                     */
                    QueueSender queueSender =
                        queueSession.createSender((Queue)
                            inMessage.getJMSReplyTo());
              queueSender.send(queueSession.createMessage());
                    queueSession.commit();
                    break;
                }

                /*
                 * Extract quantity ordered from order
                 * message.
                 */
                quantity = orderMessage.getInt("Quantity");
                System.out.println(PRODUCT_NAME +
                    " Supplier: Vendor ordered " + quantity +
                    " " + orderMessage.getString("Item") +
                    "(s)");

                /*
                 * Create sender and message for reply queue.
                 * Set order number and item; check inventory
                 * and set quantity available.
```

```
                     * Send message to vendor and commit
                     * transaction.
                     */
                    QueueSender queueSender =
                        queueSession.createSender((Queue)
                            orderMessage.getJMSReplyTo());
                    outMessage = queueSession.createMapMessage();
                    outMessage.setInt("VendorOrderNumber",
                       orderMessage.getInt("VendorOrderNumber"));
                    outMessage.setString("Item", PRODUCT_NAME);
                    int numAvailable = checkInventory();
                    if (numAvailable >= quantity) {
                        outMessage.setInt("Quantity", quantity);
                    } else {
                        outMessage.setInt("Quantity",
                            numAvailable);
                    }
                    queueSender.send(outMessage);
                    System.out.println(PRODUCT_NAME +
                        " Supplier: sent " +
                        outMessage.getInt("Quantity") + " " +
                        outMessage.getString("Item") + "(s)");
                    queueSession.commit();
                    System.out.println("  " + PRODUCT_NAME +
                        " Supplier: committed transaction");
                } catch (Exception e) {
                    System.err.println(PRODUCT_NAME +
                        " Supplier: Exception occurred: " +
                        e.toString());
                    e.printStackTrace();
                }
            }
            if (queueConnection != null) {
                try {
                    queueConnection.close();
                } catch (JMSException e) {}
            }
        }
    }
```

```
/**
 * Creates the Retailer and Vendor classes and the two
 * supplier classes, then starts the threads.
 *
 * @param quantity    the quantity specified on the command
 *                    line
 */
public static void run_threads(int quantity) {
    Retailer         r = new Retailer(quantity);
    Vendor           v = new Vendor();
    GenericSupplier  ms = new GenericSupplier("Monitor",
                             monitorOrderQueueName);
    GenericSupplier  ss = new GenericSupplier("Hard Drive",
                             storageOrderQueueName);

    r.start();
    v.start();
    ms.start();
    ss.start();
    try {
        r.join();
        v.join();
        ms.join();
        ss.join();
    } catch (InterruptedException e) {}
}

/**
 * Reads the order quantity from the command line, then
 * calls the run_threads method to execute the program
 * threads.
 *
 * @param args    the quantity of computers being ordered
 */
public static void main(String[] args) {
    TransactedExample  te = new TransactedExample();
    int                quantity = 0;

    if (args.length != 1) {
```

```
                System.out.println("Usage: java TransactedExample " +
                    "<integer>");
                System.out.println("Program assumes five queues " +
                    "named A B C D E");
                System.exit(1);
            }
            te.vendorOrderQueueName = new String("A");
            te.retailerConfirmQueueName = new String("B");
            te.monitorOrderQueueName = new String("C");
            te.storageOrderQueueName = new String("D");
            te.vendorConfirmQueueName = new String("E");
            quantity = (new Integer(args[0])).intValue();
            System.out.println("Quantity to be ordered is " +
                quantity);
            if (quantity > 0) {
                te.run_threads(quantity);
            } else {
                System.out.println("Quantity must be positive and " +
                    "nonzero");
            }
        }
    }
```

Code Example A.2 `TransactedExample.java`

A.3 Acknowledgment Modes

The `AckEquivExample.java` program shows how the following two scenarios both ensure that a message will not be acknowledged until processing of it is complete:

- Using an asynchronous receiver—a message listener—in an `AUTO_ACKNOWLEDGE` session
- Using a synchronous receiver in a `CLIENT_ACKNOWLEDGE` session

With a message listener, the automatic acknowledgment happens when the `onMessage` method returns—that is, after message processing has finished. With a synchronous receiver, the client acknowledges the message after processing is complete. (If you use `AUTO_ACKNOWLEDGE` with a synchronous receive, the acknowledgment happens immediately after the `receive` call; if any subsequent processing steps fail, the message cannot be redelivered.)

The program contains a `SynchSender` class, a `SynchReceiver` class, an `AsynchSubscriber` class with a `TextListener` class, a `MultiplePublisher` class, a `main` method, and a method that runs the other classes' threads.

The program needs two queues, a topic, and a connection factory with a client ID, similar to the one in the example in Section A.1 on page 215. You can use existing administered objects or create new ones. Edit the names at the beginning of the source file before compiling if you do not use the objects already specified. You can run the program with a command on one line similar to the following example for UNIX systems:

```
java -Djms.properties=$J2EE_HOME/config/jms_client.properties
AckEquivExample
```

The output looks like this:

```
java -Djms.properties=$J2EE_HOME/config/jms_client.properties
AckEquivExample
Queue name is controlQueue
Queue name is jms/Queue
Topic name is jms/Topic
Connection factory name is DurableTopicCF
Java(TM) Message Service 1.0.2 Reference Implementation (build b14)
Java(TM) Message Service 1.0.2 Reference Implementation (build b14)
  SENDER: Created client-acknowledge session
  RECEIVER: Created client-acknowledge session
  SENDER: Sending message: Here is a client-acknowledge message
  RECEIVER: Processing message: Here is a client-acknowledge message
  RECEIVER: Now I'll acknowledge the message
SUBSCRIBER: Created auto-acknowledge session
PUBLISHER: Created auto-acknowledge session
PUBLISHER: Receiving synchronize messages from controlQueue; count
= 1
SUBSCRIBER: Sending synchronize message to controlQueue
PUBLISHER: Received synchronize message;  expect 0 more
PUBLISHER: Publishing message: Here is an auto-acknowledge message 1
```

```
PUBLISHER: Publishing message: Here is an auto-acknowledge message 2
PUBLISHER: Publishing message: Here is an auto-acknowledge message 3
SUBSCRIBER: Processing message: Here is an auto-acknowledge message
1
SUBSCRIBER: Processing message: Here is an auto-acknowledge message
2
SUBSCRIBER: Processing message: Here is an auto-acknowledge message
3
```

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

public class AckEquivExample {
    final String  CONTROL_QUEUE = "controlQueue";
    final String  queueName = "jms/Queue";
    final String  topicName = "jms/Topic";
    final String  conFacName = "DurableTopicCF";

    /**
     * The SynchSender class creates a session in
     * CLIENT_ACKNOWLEDGE mode and sends a message.
     */
    public class SynchSender extends Thread {

        /**
         * Runs the thread.
         */
        public void run() {
            QueueConnectionFactory queueConnectionFactory = null;
            QueueConnection        queueConnection = null;
            QueueSession           queueSession = null;
            Queue                  queue = null;
            QueueSender            queueSender = null;
            final String           MSG_TEXT =
              new String("Here is a client-acknowledge message");
            TextMessage             message = null;
```

```
try {
    queueConnectionFactory =
        SampleUtilities.getQueueConnectionFactory();
    queueConnection =
      queueConnectionFactory.createQueueConnection();
    queueSession =
        queueConnection.createQueueSession(false,
            Session.CLIENT_ACKNOWLEDGE);
    queue = SampleUtilities.getQueue(queueName,
        queueSession);
} catch (Exception e) {
    System.err.println("Connection problem: " +
        e.toString());
    if (queueConnection != null) {
        try {
            queueConnection.close();
        } catch (JMSException ee) {}
    }
    System.exit(1);
}

/*
 * Create client-acknowledge sender.
 * Create and send message.
 */

try {
    System.out.println("  SENDER: Created " +
        "client-acknowledge session");
    queueSender = queueSession.createSender(queue);
    message = queueSession.createTextMessage();
    message.setText(MSG_TEXT);
    System.out.println("  SENDER: Sending " +
        "message: " + message.getText());
    queueSender.send(message);
} catch (JMSException e) {
    System.err.println("Exception occurred: " +
        e.toString());
} finally {
```

```
                if (queueConnection != null) {
                    try {
                        queueConnection.close();
                    } catch (JMSException e) {}
                }
            }
        }
    }

    /**
     * The SynchReceiver class creates a session in
     * CLIENT_ACKNOWLEDGE mode and receives the message sent by
     * the SynchSender class.
     */
    public class SynchReceiver extends Thread {

        /**
         * Runs the thread.
         */
        public void run() {
            QueueConnectionFactory queueConnectionFactory = null;
            QueueConnection        queueConnection = null;
            QueueSession           queueSession = null;
            Queue                  queue = null;
            QueueReceiver          queueReceiver = null;
            TextMessage            message = null;
            try {
                queueConnectionFactory =
                    SampleUtilities.getQueueConnectionFactory();
                queueConnection =
                  queueConnectionFactory.createQueueConnection();
                queueSession =
                    queueConnection.createQueueSession(false,
                        Session.CLIENT_ACKNOWLEDGE);
                queue =
                    SampleUtilities.getQueue(queueName,
                        queueSession);
            } catch (Exception e) {
                System.err.println("Connection problem: " +
                    e.toString());
                if (queueConnection != null) {
                    try {
                        queueConnection.close();
                    } catch (JMSException ee) {}
                }
                System.exit(1);
            }

            /*
             * Create client-acknowledge receiver.
             * Receive message and process it.
             * Acknowledge message.
             */
             try {
                System.out.println("  RECEIVER: Created " +
                    "client-acknowledge session");
                queueReceiver =
                    queueSession.createReceiver(queue);
                queueConnection.start();
                message = (TextMessage) queueReceiver.receive();
                System.out.println("  RECEIVER: Processing " +
                    "message: " + message.getText());
                System.out.println("  RECEIVER: Now I'll " +
                    "acknowledge the message");
                message.acknowledge();
            } catch (JMSException e) {
                System.err.println("Exception occurred: " +
                    e.toString());
            } finally {
                if (queueConnection != null) {
                    try {
                        queueConnection.close();
                    } catch (JMSException e) {}
                }
            }
        }
    }
```

```
/**
 * The AsynchSubscriber class creates a session in
 * AUTO_ACKNOWLEDGE mode and fetches several messages from a
 * topic asynchronously, using a message listener,
 * TextListener.
 *
 * Each message is acknowledged after the onMessage method
 * completes.
 */
public class AsynchSubscriber extends Thread {

    /**
     * The TextListener class implements the MessageListener
     * interface by defining an onMessage method for the
     * AsynchSubscriber class.
     */
    private class TextListener implements MessageListener {
        final SampleUtilities.DoneLatch  monitor =
            new SampleUtilities.DoneLatch();

        /**
         * Casts the message to a TextMessage and displays
         * its text. A non-text message is interpreted as the
         * end of the message stream, and the message
         * listener sets its monitor state to all done
         * processing messages.
         *
         * @param message    the incoming message
         */
        public void onMessage(Message message) {
            if (message instanceof TextMessage) {
                TextMessage  msg = (TextMessage) message;

                try {
                    System.out.println("SUBSCRIBER: " +
                        "Processing message: " +
                        msg.getText());
                } catch (JMSException e) {
                    System.err.println("Exception in " +
                        "onMessage(): " + e.toString());
                }
            } else {
                monitor.allDone();
            }
        }
    }

    /**
     * Runs the thread.
     */
    public void run() {
        Context                jndiContext = null;
        TopicConnectionFactory topicConnectionFactory = null;
        TopicConnection        topicConnection = null;
        TopicSession           topicSession = null;
        Topic                  topic = null;
        TopicSubscriber        topicSubscriber = null;
        TextListener           topicListener = null;

        /*
         * Create a JNDI API InitialContext object if none
         * exists yet.
         */
        try {
            jndiContext = new InitialContext();
        } catch (NamingException e) {
            System.err.println("Could not create JNDI API " +
                "context: " + e.toString());
            System.exit(1);
        }

        /*
         * Look up connection factory and topic.  If either
         * does not exist, exit.
         */
```

```
try {
    topicConnectionFactory = (TopicConnectionFactory)
        jndiContext.lookup(conFacName);
    topicConnection =
      topicConnectionFactory.createTopicConnection();
    topicSession =
        topicConnection.createTopicSession(false,
            Session.AUTO_ACKNOWLEDGE);
    System.out.println("SUBSCRIBER: Created " +
        "auto-acknowledge session");
    topic = SampleUtilities.getTopic(topicName,
        topicSession);
} catch (Exception e) {
    System.err.println("Connection problem: " +
        e.toString());
    if (topicConnection != null) {
        try {
            topicConnection.close();
        } catch (JMSException ee) {}
    }
    System.exit(1);
}

/*
 * Create auto-acknowledge subscriber.
 * Register message listener (TextListener).
 * Start message delivery.
 * Send synchronize message to publisher, then wait
 *   till all messages have arrived.
 * Listener displays the messages obtained.
 */
try {
    topicSubscriber =
        topicSession.createDurableSubscriber(topic,
            "AckSub");
    topicListener = new TextListener();
   topicSubscriber.setMessageListener(topicListener);
    topicConnection.start();
```

```
            // Let publisher know that subscriber is ready.
            try {
       SampleUtilities.sendSynchronizeMessage("SUBSCRIBER: ",
                    CONTROL_QUEUE);
            } catch (Exception e) {
                System.err.println("Queue probably " +
                    "missing: " + e.toString());
                if (topicConnection != null) {
                    try {
                        topicConnection.close();
                    } catch (JMSException ee) {}
                }
                System.exit(1);
            }

            /*
             * Asynchronously process messages.
             * Block until publisher issues a control message
             * indicating end of publish stream.
             */
            topicListener.monitor.waitTillDone();
            topicSubscriber.close();
            topicSession.unsubscribe("AckSub");
        } catch (JMSException e) {
            System.err.println("Exception occurred: " +
                e.toString());
        } finally {
            if (topicConnection != null) {
                try {
                    topicConnection.close();
                } catch (JMSException e) {}
            }
        }
    }
}

/**
 * The MultiplePublisher class creates a session in
 * AUTO_ACKNOWLEDGE mode and publishes three messages
 * to a topic.
 */
public class MultiplePublisher extends Thread {

    /**
     * Runs the thread.
     */
    public void run() {
        TopicConnectionFactory topicConnectionFactory = null;
        TopicConnection        topicConnection = null;
        TopicSession           topicSession = null;
        Topic                  topic = null;
        TopicPublisher         topicPublisher = null;
        TextMessage            message = null;
        final int              NUMMSGS = 3;
        final String           MSG_TEXT =
           new String("Here is an auto-acknowledge message");

        try {
            topicConnectionFactory =
                SampleUtilities.getTopicConnectionFactory();
            topicConnection =
              topicConnectionFactory.createTopicConnection();
            topicSession =
                topicConnection.createTopicSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            System.out.println("PUBLISHER: Created " +
                "auto-acknowledge session");
            topic =
                SampleUtilities.getTopic(topicName,
                    topicSession);
        } catch (Exception e) {
            System.err.println("Connection problem: " +
                e.toString());
            if (topicConnection != null) {
                try {
                    topicConnection.close();
                } catch (JMSException ee) {}
            }
            System.exit(1);
        }

        /*
         * After synchronizing with subscriber, create
         *   publisher.
         * Send 3 messages, varying text slightly.
         * Send end-of-messages message.
         */
        try {
            /*
             * Synchronize with subscriber.  Wait for message
             * indicating that subscriber is ready to receive
             * messages.
             */
            try {
    SampleUtilities.receiveSynchronizeMessages("PUBLISHER: ",
                    CONTROL_QUEUE, 1);
            } catch (Exception e) {
                System.err.println("Queue probably " +
                    "missing: " + e.toString());
                if (topicConnection != null) {
                    try {
                        topicConnection.close();
                    } catch (JMSException ee) {}
                }
                System.exit(1);
            }

            topicPublisher =
                topicSession.createPublisher(topic);
            message = topicSession.createTextMessage();
            for (int i = 0; i < NUMMSGS; i++) {
                message.setText(MSG_TEXT + " " + (i + 1));
                System.out.println("PUBLISHER: Publishing " +
                    "message: " + message.getText());
                topicPublisher.publish(message);
            }
```

```
                /*
                 * Send a non-text control message indicating
                 * end of messages.
                 */
            topicPublisher.publish(topicSession.createMessage());
            } catch (JMSException e) {
                System.err.println("Exception occurred: " +
                    e.toString());
            } finally {
                if (topicConnection != null) {
                    try {
                        topicConnection.close();
                    } catch (JMSException e) {}
                }
            }
        }
    }

    /**
     * Instantiates the sender, receiver, subscriber, and
     * publisher classes and starts their threads.
     * Calls the join method to wait for the threads to die.
     */
    public void run_threads() {
        SynchSender        synchSender = new SynchSender();
        SynchReceiver      synchReceiver = new SynchReceiver();
        AsynchSubscriber   asynchSubscriber =
                                 new AsynchSubscriber();
        MultiplePublisher  multiplePublisher =
                                new MultiplePublisher();

        synchSender.start();
        synchReceiver.start();
        try {
            synchSender.join();
            synchReceiver.join();
        } catch (InterruptedException e) {}

        asynchSubscriber.start();
        multiplePublisher.start();
        try {
            asynchSubscriber.join();
            multiplePublisher.join();
        } catch (InterruptedException e) {}
    }

    /**
     * Reads the queue and topic names from the command line,
     * then calls the run_threads method to execute the program
     * threads.
     *
     * @param args    the topic used by the example
     */
    public static void main(String[] args) {
        AckEquivExample  aee = new AckEquivExample();

        if (args.length != 0) {
            System.out.println("Usage: java AckEquivExample");
            System.exit(1);
        }
        System.out.println("Queue name is " + aee.queueName);
        System.out.println("Topic name is " + aee.topicName);
        System.out.println("Connection factory name is " +
            aee.conFacName);

        aee.run_threads();
    }
}
```

Code Example A.3 `AckEquivExample.java`

A.4 Utility Class

The `SampleUtilities` class, in `SampleUtilities.java`, is a utility class for the other sample programs. It contains the following methods:

- `getQueueConnectionFactory`
- `getTopicConnectionFactory`
- `getQueue`
- `getTopic`
- `jndiLookup`
- `receiveSynchronizeMessages`
- `sendSynchronizeMessages`

It also contains the class `DoneLatch`, which has the following methods:

- `waitTillDone`
- `allDone`

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

public class SampleUtilities {
    public static final String  QUEUECONFAC =
                                        "QueueConnectionFactory";
    public static final String  TOPICCONFAC =
                                        "TopicConnectionFactory";
    private static Context      jndiContext = null;

    /**
     * Returns a QueueConnectionFactory object.
     *
     * @return    a QueueConnectionFactory object
```

```
 * @throws    javax.naming.NamingException (or other
 *            exception) if name cannot be found
 */
public static QueueConnectionFactory
        getQueueConnectionFactory() throws Exception {
    return (QueueConnectionFactory) jndiLookup(QUEUECONFAC);
}

/**
 * Returns a TopicConnectionFactory object.
 *
 * @return    a TopicConnectionFactory object
 * @throws    javax.naming.NamingException (or other
 *            exception) if name cannot be found
 */
public static TopicConnectionFactory
        getTopicConnectionFactory() throws Exception {
    return (TopicConnectionFactory) jndiLookup(TOPICCONFAC);
}

/**
 * Returns a Queue object.
 *
 * @param name       String specifying queue name
 * @param session    a QueueSession object
 *
 * @return           a Queue object
 * @throws           javax.naming.NamingException (or other
 *                   exception) if name cannot be found
 */
public static Queue getQueue(String name,
        QueueSession session) throws Exception {
    return (Queue) jndiLookup(name);
}

/**
 * Returns a Topic object.
 *
 * @param name       String specifying topic name
```

```
     * @param session    a TopicSession object
     *
     * @return           a Topic object
     * @throws           javax.naming.NamingException (or other
     *                   exception) if name cannot be found
     */
    public static Topic getTopic(String name,
            TopicSession session) throws Exception {
        return (Topic) jndiLookup(name);
    }

    /**
     * Creates a JNDI API InitialContext object if none exists
     * yet. Then looks up the string argument and returns the
     * associated object.
     *
     * @param name    the name of the object to be looked up
     *
     * @return        the object bound to name
     * @throws        javax.naming.NamingException (or other
     *                exception) if name cannot be found
     */
    public static Object jndiLookup(String name)
        throws NamingException {
        Object    obj = null;

        if (jndiContext == null) {
            try {
                jndiContext = new InitialContext();
            } catch (NamingException e) {
                System.err.println("Could not create JNDI API " +
                    "context: " + e.toString());
                throw e;
            }
        }
        try {
           obj = jndiContext.lookup(name);
        } catch (NamingException e) {
            System.err.println("JNDI API lookup failed: " +
                e.toString());
            throw e;
        }
        return obj;
    }

    /**
     * Waits for 'count' messages on controlQueue before
     * continuing.  Called by a publisher to make sure that
     * subscribers have started before it begins publishing
     * messages.
     *
     * If controlQueue does not exist, the method throws an
     * exception.
     *
     * @param prefix    prefix (publisher or subscriber) to be
     *                  displayed
     * @param controlQueueName  name of control queue
     * @param count     number of messages to receive
     */
    public static void receiveSynchronizeMessages(String prefix,
            String controlQueueName, int count)
            throws Exception {
        QueueConnectionFactory  queueConnectionFactory = null;
        QueueConnection         queueConnection = null;
        QueueSession            queueSession = null;
        Queue                   controlQueue = null;
        QueueReceiver           queueReceiver = null;

        try {
            queueConnectionFactory =
                SampleUtilities.getQueueConnectionFactory();
            queueConnection =
                queueConnectionFactory.createQueueConnection();
            queueSession =
                queueConnection.createQueueSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            controlQueue = getQueue(controlQueueName,
                queueSession);
                queueConnection.start();
            } catch (Exception e) {
                System.err.println("Connection problem: " +
                    e.toString());
                if (queueConnection != null) {
                    try {
                        queueConnection.close();
                    } catch (JMSException ee) {}
                }
                throw e;
            }

            try {
                System.out.println(prefix +
                    "Receiving synchronize messages from " +
                    controlQueueName + "; count = " + count);
                queueReceiver =
                    queueSession.createReceiver(controlQueue);
                while (count > 0) {
                    queueReceiver.receive();
                    count--;
                    System.out.println(prefix +
                        "Received synchronize message; " +
                        " expect " + count + " more");
                }
            } catch (JMSException e) {
                System.err.println("Exception occurred: " +
                    e.toString());
                throw e;
            } finally {
                if (queueConnection != null) {
                    try {
                        queueConnection.close();
                    } catch (JMSException e) {}
                }
            }
        }

        /**
```

```
     * Sends a message to controlQueue.  Called by a subscriber
     * to notify a publisher that it is ready to receive
     * messages.
     * <p>
     * If controlQueue doesn't exist, the method throws an
     * exception.
     *
     * @param prefix    prefix (publisher or subscriber) to be
     *                  displayed
     * @param controlQueueName  name of control queue
     */
    public static void sendSynchronizeMessage(String prefix,
            String controlQueueName)
            throws Exception {
        QueueConnectionFactory  queueConnectionFactory = null;
        QueueConnection         queueConnection = null;
        QueueSession            queueSession = null;
        Queue                   controlQueue = null;
        QueueSender             queueSender = null;
        TextMessage             message = null;

        try {
            queueConnectionFactory =
                SampleUtilities.getQueueConnectionFactory();
            queueConnection =
                queueConnectionFactory.createQueueConnection();
            queueSession =
                queueConnection.createQueueSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            controlQueue = getQueue(controlQueueName,
                queueSession);
        } catch (Exception e) {
            System.err.println("Connection problem: " +
                e.toString());
            if (queueConnection != null) {
                try {
                    queueConnection.close();
                } catch (JMSException ee) {}
            }
        throw e;
    }
    try {
        queueSender =
            queueSession.createSender(controlQueue);
        message = queueSession.createTextMessage();
        message.setText("synchronize");
        System.out.println(prefix +
            "Sending synchronize message to " +
            controlQueueName);
        queueSender.send(message);
    } catch (JMSException e) {
        System.err.println("Exception occurred: " +
            e.toString());
        throw e;
    } finally {
        if (queueConnection != null) {
            try {
                queueConnection.close();
            } catch (JMSException e) {}
        }
    }
}

/**
 * Monitor class for asynchronous examples.  Producer signals
 * end of message stream; listener calls allDone() to notify
 * consumer that the signal has arrived, while consumer calls
 * waitTillDone() to wait for this notification.
 */
static public class DoneLatch {
    boolean  done = false;
    /**
     * Waits until done is set to true.
     */
    public void waitTillDone() {
        synchronized (this) {
            while (! done) {
                try {
                    this.wait();
                } catch (InterruptedException ie) {}
            }
        }
    }

    /**
     * Sets done to true.
     */
    public void allDone() {
        synchronized (this) {
            done = true;
            this.notify();
        }
    }
}
}
```

Code Example A.4 `SampleUtilities.java`

Index