queue, we call this **point-to-point (P2P)** messaging. In this mode a message is deleted from a queue as soon as it's successfully received.

If a program publishes a message to be consumed by multiple recipients, that's **publish/subscribe (pub/sub)** mode. A message is published to a particular topic and many subscribers can subscribe to receive it. The topic represents some important news for applications and/or users, for example PriceDropAlert, BreakingNews, et cetera. In pub/sub mode a message is usually deleted from a queue as soon as all subscribers receive it (read about durable subscribers in the section "How to Subscribe for a Topic").

Another good example of a pub/sub application is a chat room. A message published by one person is received by the other people present in the chat room. Developing a chat room with JMS and MOM is a pretty trivial task.

Message delivery can be guaranteed, and MOM, just like the post office, will keep the message in a queue until the receiver gets it. In this mode messages are persistent — the MOM provider stores them in its internal storage, which can be a DBMS or a file system. In a non-guaranteed mode MOM will deliver a message only to the receiving applications that are up-and-running at the moment that the message arrives.

JMS API Overview

JMS is a pretty simple API that includes Java classes that enable you to send and receive messages. Following are the names of the major JMS interfaces and a short description of each. All of them are defined in the package `javax.jms`. All Java EE 6 JMS classes and interfaces are listed at **http://download.oracle.com/javaee/6/api/javax/jms/package-summary.html**, but I'll just give you the summary of the main ones.

- `Queue` is a place to put (or get) your messages. The messages will be retrieved using the first in first out (FIFO) rule. A **message producer** (sender) puts messages in a queue and a **message consumer** (receiver) de-queues them.
- `QueueConnection` is an interface (it extends `Connection`) that represents a particular connection to MOM (and plays a similar role to the JDBC class `Connection`).
- `QueueConnectionFactory` is an object that creates `Connection` objects (just the JDBC class `DataSource` does).
- `QueueSession` is an object that represents a particular session between the client and the MOM server. `QueueConnection` creates a session object.
- `QueueSender` is an object that actually sends messages.
- `QueueReceiver` receives messages.
- `TopicPublisher` publishes messages (it has a similar functionality to `QueueSender`).
- `TopicSubscriber` is an object that receives messages (it has a similar functionality to `QueueReceiver`).
- `Topic` is an object that is used in pub/sub mode to represent some important application event.
- `TopicPublisher` publishes messages to a topic so the `TopicSubscriber`s can subscribe to it.
- `Message` is an object that serves as a wrapper to an application-specific object that can be placed in a JMS queue or published to a topic.

Types of Messages

Every message contains a header and optional body and has facilities for providing additional properties. The header contains the message identification (unique message ID, destination, type, et cetera). The optional properties can be set by a program to tag a message with application-specific data, for example `UrgentOrder`.

The optional body contains a message that has to be delivered. Following are the types of JMS messages that you can place in a mes-

sage body. All these interfaces are inherited from `javax.jms.Message`.

- `TextMessage` is an object that can contain any Java `String`.
- `ObjectMessage` can hold any `Serializable` Java object.
- `BytesMessage` holds an array of bytes.
- `StreamMessage` has a stream of Java primitives.
- `MapMessage` contains any key/value pairs, for example `id=123`.

How to Send a Message

Queues have to be preconfigured in MOM and their names must be known before a program can start sending messages. In the real world a MOM server administrator manages queues and other messaging artifacts. When Java developers get to know the queue parameters, they have a choice of either creating message objects (such as `ConnectionFactory` and `Queue`) programmatically every time they need to send or receive a message, or pre-creating objects bound to some kind of naming service (you'll see examples in Lesson 31).

A program has to perform the following steps to send a message:

1. Create (or get from some naming server) a `ConnectionFactory` object.

2. Create a `Connection` object and call its `start()` method.

3. Create a `Session` object.

4. Create a `Queue` object.

5. Create a `MessageProducer` object.

6. Create one of the `Message` objects (such as `TextMessage`) and put some data in it.

7. Call the `send()` method on the `QueueSender`.

8. Close the `QueueSender`, `Session`, and `Connection` objects to release system resources.

Listing 30-1 shows a code fragment that sends a message to a queue called `TestQueue`. Note that the method `createQueue()` just creates an instance of the `Queue` object locally — the actual destination with the name `TestQueue` has to exist in MOM.



Listing 30-1: Sending a message

```
        Session session=null;
        ConnectionFactory factory;
        QueueConnection connection=null;

        try{
             //The next two lines are specific to Open MQ. Another MOM
             // provider will have their own version of creating connection
             factory = new com.sun.messaging.ConnectionFactory();
             factory.setProperty(ConnectionConfiguration.imqAddressList,
                                 "mq://localhost:7677,mq://localhost:7677");

             connection = factory.createQueueConnection("admin","admin");

             connection.start();

             session = connection.createQueueSession(
                                 false, Session.AUTO_ACKNOWLEDGE);
             Queue ioQueue = session.createQueue("TestQueue");
             MessageProducer queueSender = session.createProducer(ioQueue);

             // Buy 200 shares of IBM at market price
             TextMessage outMsg = session.createTextMessage("IBM 200 Mkt");

             queueSender.send(outMsg);
             queueSender.close();

             System.out.println(
  "Successfully placed an order to purchase 200 shares of IBM");
           }
           catch (JMSException e){
```

```
                System.out.println("Error: " + e.getMessage());
            }
            finally{
               try{
                  session.close();
                  connection.close();
               } catch (Exception e) {
                  System.out.println("Can't close JMS connection/session " +
                                                                  e.getMessage());
              }
            }
```

The code in **Listing 30-1** uses an Open MQ–specific implementation of JMS's `ConnectionFactory`. After you read the section "Administering Objects in Open MQ" later in this lesson, you'll understand why these lines refer to localhost and port 7677.

How to Receive a Message

The program that receives messages is called a **message consumer**. It can be a standalone Java program or a special type of enterprise Java bean called a **message-driven bean (MDB)**. You can receive messages either synchronously, using the `receive()` method, or asynchronously by implementing the `MessageListener` interface and programming a callback `onMessage()`.

Under the hood, the `receive()` method uses a polling mechanism that constantly asks for a message. It blocks program execution until the message is received or the specified time has expired:

```
QueueReceiver queueReceiver= Session.createReceiver(ioQueue);
Message myMessage = queueReceiver.receive();
```

The next line shows how to set a timeout interval of 500 milliseconds:

```
Message myMessage=queueReceiver.receive(500);
```

Using an asynchronous callback `onMessage()` is the best way to receive messages because the message consumer is not sending multiple requests to MOM just to see if the message is in the queue. The callback method `onMessage()` will be called immediately when a message is put in the queue. The listener class must perform the following steps to receive messages:

1. Create (or get from some naming server) the `QueueConnectionFactory` object.

2. Create a `Connection` object and call its `start()` method.

3. Create a `Session` object.

4. Create a `Queue` object.

5. Create a `QueueReceiver` object.

6. If your class implements `MessageListener` (see **Listing 30-2**) write implementation for the callback method `onMessage()`. If you want to get messages synchronously, just call the `QueueReceiver.receive()` method. In this case implementation of the `MessageListener` interface is not needed.

7. Close the `Session` and `Connection` objects to release the system resources.

The sample class `MyReceiver` in **Listing 30-2** shows how to consume messages from the `TestQueue` asynchronously. Its constructor creates JMS objects and registers itself as a message listener. The callback `onMessage()` has code for processing the received messages.



Listing 30-2: Receiving a message

```
package com.practicaljava.lesson30;

import javax.jms.*;
import com.sun.messaging.ConnectionFactory;
import com.sun.messaging.ConnectionConfiguration;

public class MessageReceiver implements MessageListener{

      Session session=null;
       ConnectionFactory factory;
       QueueConnection connection=null;
       MessageConsumer consumer=null;

       MessageReceiver(){
             try{
              factory = new com.sun.messaging.ConnectionFactory();
              factory.setProperty(ConnectionConfiguration.imqAddressList,
                                  "mq://localhost:7677,mq://localhost:7677");
              connection = factory.createQueueConnection("admin","admin");

              connection.start();

              Session session = connection.createQueueSession(
                                  false, Session.AUTO_ACKNOWLEDGE);

             Queue ioQueue = session.createQueue( "TestQueue" );

             consumer = session.createConsumer(ioQueue);
             consumer.setMessageListener(this);

             System.out.println("Listening to the TestQueue...");

            // Don't finish -  wait for messages
            Thread.sleep(100000);

          } catch (InterruptedException e){
            System.out.println("Error: " + e.getMessage());
         }
        catch (JMSException e){
           System.out.println("Error: " + e.getMessage());
```

```
       }
       finally{
          try{
                      connection.close();
          } catch (Exception e) {
               System.out.println("Can't close JMS connection/session "
                                                              + e.getMessage());
          }
      }

 }

  public static void main(String[] args){
         new MessageReceiver();
 }

  public void onMessage(Message msg){
        String msgText;
        try{
            if (msg instanceof TextMessage){
                    msgText = ((TextMessage) msg).getText();
                   System.out.println("Got from the queue: " + msgText);
            }else{
               System.out.println("Got a non-text message");
            }
        }
        catch (JMSException e){
             System.out.println("Error while consuming a message: " +
                                                          e.getMessage());
        }
  }
 }
```

In this code sample the `onMessage()` method checks to ensure that the arrived message is of the `TextMessage` type. All other types of messages can be ignored.

The message acknowledgment mode is defined when the `Session` object is created. The `createSession()` method has two arguments. If the first argument is `true`, the session is transacted, the value of the second argument is irrelevant, and the message can be either committed or rolled back by the consumer.

If the `commit()` method has been called, the message is removed from the queue. The `rollback()` method leaves the message in the queue. If the session is non-transacted, as in our earlier example, the second argument defines the acknowledgement mode.

- `AUTO_ACKNOWLEDGE` mode sends the acknowledgment back as soon as the method `onMessage()` is successfully finished.
- `CLIENT_ACKNOWLEDGE` mode requires explicit acknowledgment, such as `msg.acknowledge()`. (This grants permission to delete the message from the queue.)
- `DUP_OK_ACKNOWLEDGE` mode is used in case the server fails; the same message may be delivered more than once. In some use cases it's acceptable — for example, receiving a price quote twice won't hurt.

If more than one message is processed by the same `Session` object, acknowledgement of one message affects all messages from the same session.

How to Publish a Message

Programs publish messages to topics, which should be created in advance by the MOM system administrator. Multiple subscribers can get messages published to the same topic (this is also known as **one-to-many mode**).

Message publishing is very similar to message sending, but the program should create a `Topic` instead of a `Queue` and a `Publisher` instead of a `Sender`, and the `publish()` method should be called instead of `send()`as in **Listing 30-3**:



Listing 30-3: Publishing a message to a topic

```
...
TopicConnection connection = connectionFactory.createTopicConnection();

TopicSession pubSession = connection.createTopicSession(false,
                                          Session.AUTO_ACKNOWLEDGE);

Topic myTopic = pubSession.createTopic ("Price_Drop_Alerts");

TopicPublisher publisher= pubSession.createPublisher(myTopic);

connection.start();

TextMessage message = pubSession.createTextMessage();
message.setText("Sale in Apple starts  tomorrow");
publisher.publish(message);
```

How to Subscribe for a Topic

Subscribers can be **durable** or **non-durable**. Durable subscribers are guaranteed to receive their messages; they do not have to be active at the time a message arrives. Non-durable subscribers will receive only those messages that come when they are active. This is similar to the way chat rooms operate — you must be online to get messages.

The code snippet in **Listing 30-4** creates a non-durable subscriber. Two modifications have to be made to this code to create a durable subscriber: The client ID has to be assigned to the connection — `connection.setClientID(username)` — and the `createDurableSubscriber(topic)` method should be used instead of `createSubscriber(topic)`.



Listing 30-4: Subscribing to a topic

```
TopicSession subSession =
connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
```

```
subSession.createTopic("Price_Drop_Alerts");

TopicSubscriber subscriber = subSession.createSubscriber(topic);

connection.start();

subscriber.setMessageListener(this);

public void onMessage(Message message) {

String msgText;
try{
    if (msg instanceof TextMessage){
            msgText = ((TextMessage) msg).getText();
           System.out.println("Got " + msgText);
    }else{
       System.out.println("Got a non-text message");
    }
}
catch (JMSException e){
     System.out.println("Error: " + e.getMessage());
}
}
```

Message Selectors

If you have to share a queue with some other applications or developers from your team, use **message selectors** (also known as **filters**) to avoid “stealing” somebody else’s messages. For example:

```
String selector = "StoreName=Apple";
session.createReceiver(queue, selector);
```

In this case the queue listener will de-queue only those messages that have the `String` property `StoreName` with the value `Apple`. Message producers have to set this property:

```
TextMessage outMsg = session.createTextMessage();
outMsg.setText("Super sale starts tomorrow");
outMsg.setStringProperty("StoreName", "Apple");
```

Remember that message selectors slow down the process of retrieval. The messages stay in a queue until the listener with the matching selector picks them up. Selectors really help if your team has a limited number of queues and everyone needs to receive messages without interfering with others. But if someone starts the queue listener without selectors, it just drains the queue.

Administering Objects in Open MQ

To test all code samples from this lesson you need a MOM provider that will transport your messages. I'll be using the open-source MOM provider Open MQ. It's well-known commercial-grade software. As a bonus, it comes with GlassFish. You already have it installed under your `glassfishv3/mq` directory. If you decide to use Open MQ with any other application server you can download it separately from **https://mq.dev.java.net/**.

As I stated earlier in this lesson, I'm not going to use Java EE server — the goal is to test standalone Java clients communicating with Open MQ directly, without the middlemen.

First, open a command (or Terminal) window to the `glassfishv3/mq/bin` directory, and start the Open MQ broker. In Mac OS I enter the following command (in Windows you'll need to run `imqbrokerd.exe`):

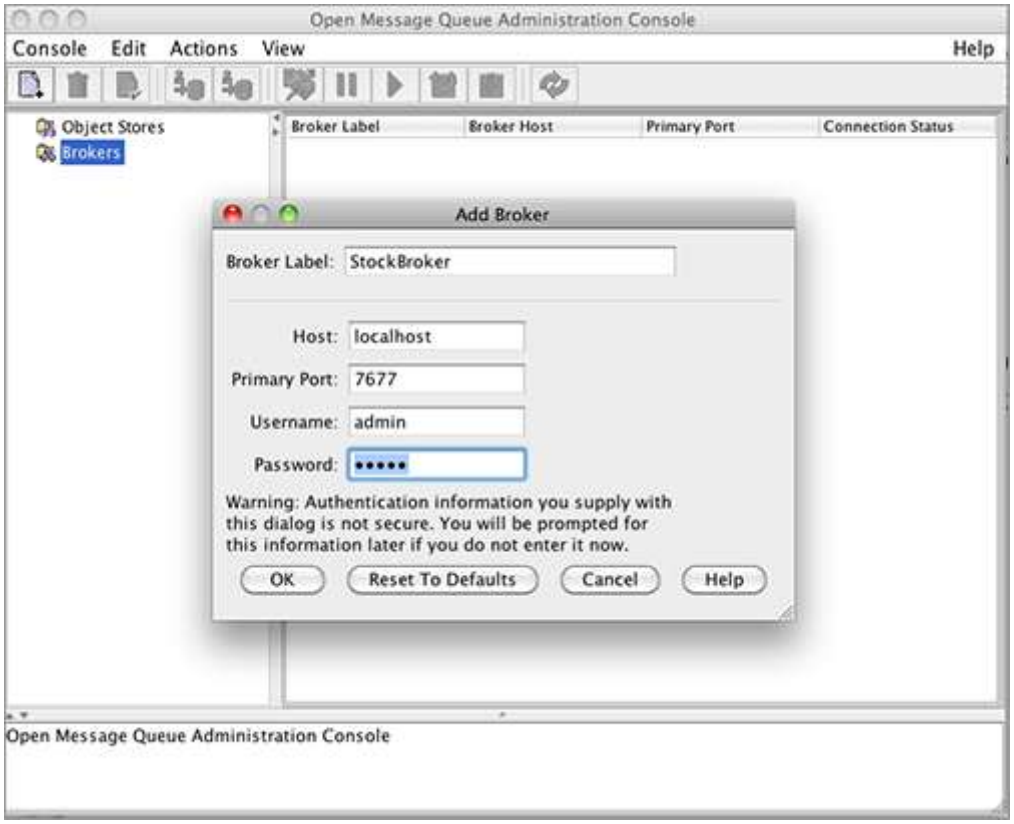
```
./imqbrokerd -port 7677
```

You'll see a prompt informing you that the broker is ready on port 7677. Now open another command window, change to the `glassfishv3/mq/bin` directory again, and start the admin GUI tool imqadmin to create the required messaging destinations:

```
./imqadmin
```

The window of the Open MQ administration console will open. Add a new broker and name it StockBroker, change the port to 7677, enter the password `admin`, and click OK. **Figure 30-2** shows a snapshot of my screen after these steps.

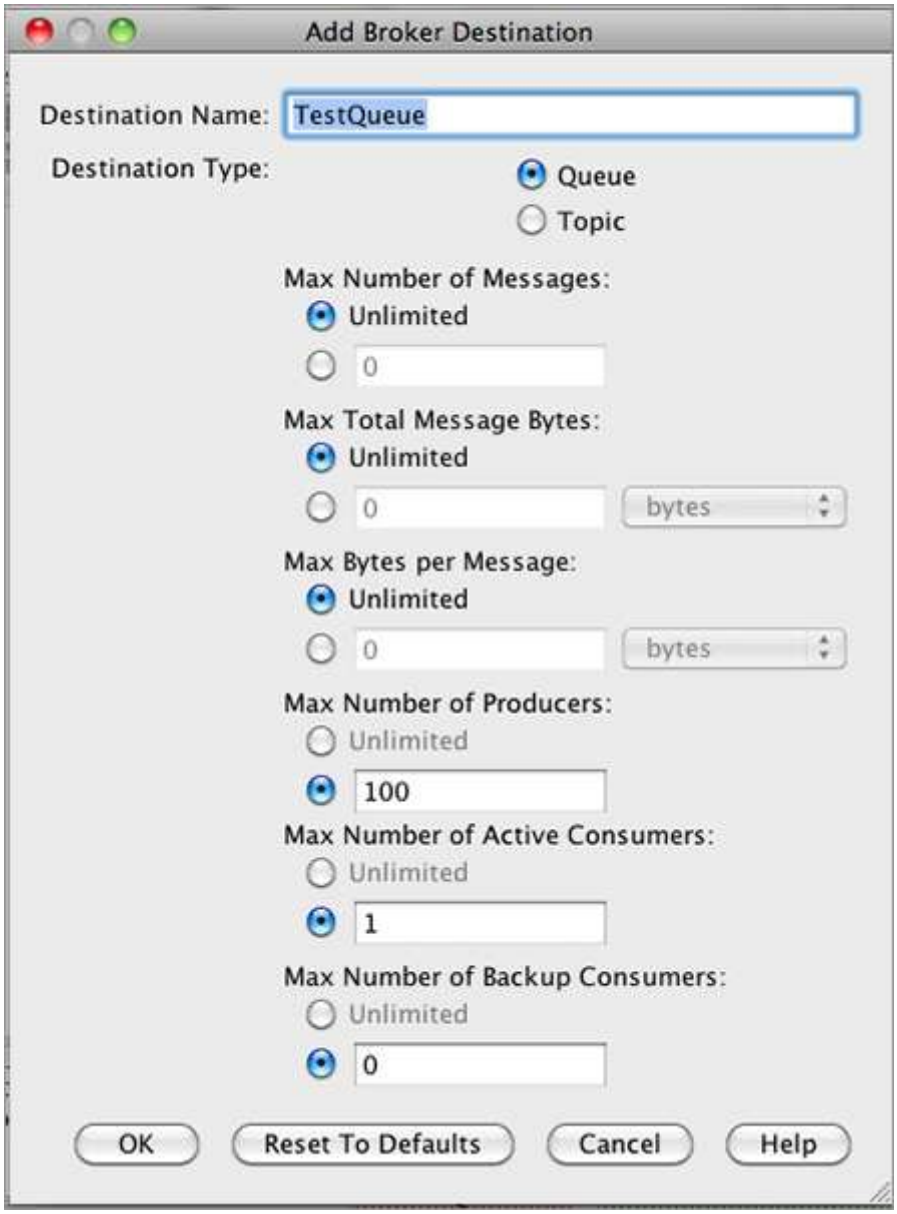
Figure 30-2



Now connect to StockBroker (using the right-click menu) and create a new destination named TestQueue to match the queue name in the

code sample from **Listing 30-1**. I didn't change any settings in the pop-up window shown in **Figure 30-3**. Note the radio button on top that enables you to specify whether you need a queue or a topic.

Figure 30-3



The creation of administrated MOM objects (the TestQueue) is complete; now you're ready to write and test the message sender and receiver.

Try It

The goal is to write two Java programs — one to send messages to the TestQueue and the other to receive them from the TestQueue.

Lesson Requirements

You should have Java and Open MQ installed. You'll also need the JMS classes and their implementation by Open MQ — `jms.jar` and `imq.jar`.



You can download the code and resources for this "Try It" section from the book's web page at **www.wrox.com**. You can find them in the Lesson30 folder in the download.

Hints

Open two console views by clicking the little icon with the plus sign in the lower right of Eclipse to see output of the sender and receiver in different console views. You can also use the pin feature (click the icon with the pin image) to pin each console to a specific application. You can also detach the Eclipse console to display it as a separate window.

Step-by-Step

1. In Eclipse, go to Java perspective and select File ⇒ New ⇒ Create a New Java Project. Name it Lesson30 and click Next. Select the tab Libraries and click Add External JARs. Find `jms.jar` and `imq.jar` in `glassfishv3/mq/lib` and add them to the project. Click Finish. You've created a regular Java SE project with two extra jar files in its CLASSPATH. `jms.jar` contains the standard JMS classes and interfaces, and `imq.jar` has the Open MQ implementation of the JMS API.

2. Create a new class called `MessageSender` with a `main()`method, and add the `import` statements for all JMS classes and the Open MQ implementation of `ConnectionFactory`:

```
import javax.jms.*;
import com.sun.messaging.ConnectionFactory;
import com.sun.messaging.ConnectionConfiguration;
```

3. Enter the code from **Listing 30-1** into the method `main()`.

4. Compile and run the program. You should see the message "Successfully placed an order to purchase 200 shares of IBM." At this point the message is located in the TestQueue configured under the Open MQ server. It'll stay there until some program de-queues it from there.

5. Create another Java class called `MessageReceiver` to look like what is created by the code in **Listing 30-2**.

6. Run `MessageReceiver`. It'll print on the console the message "Listening to the TestQueue" and will be retrieving all the messages from the queue. `Thread.sleep()` will keep this program running.

7. Observe that one console displays the messages of the producer application, and that the consumer properly receives the messages and printing confirmations in another console view.



Please select Lesson 30 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 31

Introducing JNDI

Instead of having distributed Java programs instantiate lots of reusable objects over and over again, it's better if these objects are pre-created and published at a known server, where they can be easily and quickly found. The role of the Java Naming and Directory Interface (JNDI) is to make it easier to find objects in distributed applications. It plays a role similar to that of a company telephone directory assistance service. Various software vendors offer specialized directory assistance software, and JNDI provides a standard API to read from and write to such directories.

In this lesson you'll be introduced to the JNDI concepts and will see how to use JNDI for publishing (and looking up) administered JMS objects.

Java Naming and Directory Interface

A **naming service** enables you to add, change, or delete names of objects that exist in some naming hierarchy so other Java classes can look them up to find their location. One more analogy: In a library, you find the name of the physical location of the book in a directory, and then go to the shelf to pick up the book. A naming service provides a unique name for every entry that is registered (**bound to**) with the service. Every naming service will have one or more **contexts** — think of directories and subdirectories in a file system, where any directory tree with children is a context. The naming tree originates

from a root node, which is also known as an **initial context** (like a root directory on the disk).

A **directory service** enables you to search the naming tree by object attributes rather than object names. One example is that of the domain name system, which is a distributed naming system that takes the domain name of a networked computer or service and returns the IP address and port number of the resource.

To allow clients to do lookups, there has to be a process that initially binds the objects to a naming tree. This can be handled by an independent program that (for example) binds employee names to a directory server of some organization. Java EE servers do a similar job (during start-up) by binding such objects as EJB, Servlets, JMS, and database connection pools to either their internal or external naming servers.

Administering JNDI Objects in GlassFish

Each Java EE application server offers a tool that allows administration of its service modules — we are interested in binding objects to their directory names. When you start GlassFish there is a message on the console: “Waiting for DAS to start…” DAS stands for Domain Administration Server, which authenticates the user with administrative privileges and responds to requests from a graphical web browser-based Admin Console. To use the console enter the following URL in your browser: **http://localhost:4848/**. You’ll be prompted for the user ID (admin) and password, which you selected when you installed GlassFish in Lesson 26.

After a successful log-on you’ll see the console, as in **Figure 31-1**, which enables you to administer various Java objects, but I’ll continue using the example of JMS-related ones. There are two JMS nodes in the directory tree on the left. One is called JMS Resources, and the other Java Message Service. The latter shows the physical queue named

TestQueue that I created using the Open MQ administrative utility in Lesson 30.

The only reason GlassFish automatically knows about the destinations in this MOM is that this application server is integrated with Open MQ. To configure another MOM server you'd need to create a new JMS host by filling out the form shown in **Figure 31-2**.

Figure 31-1

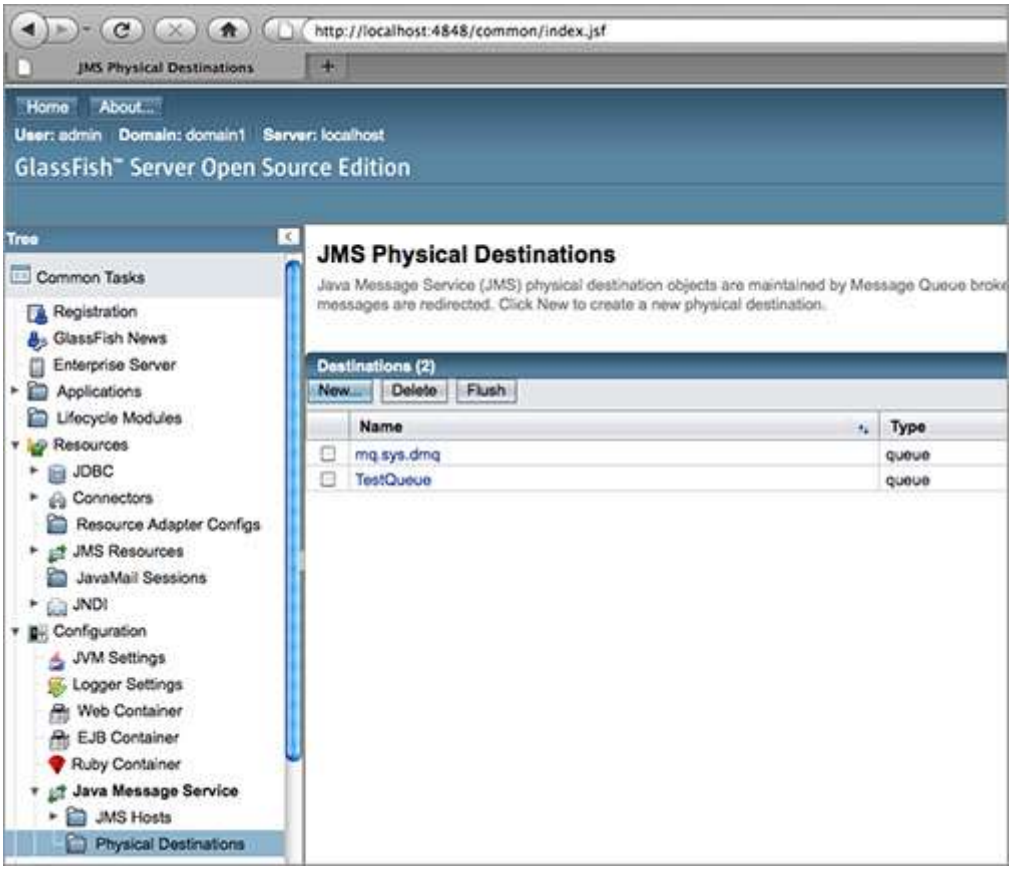
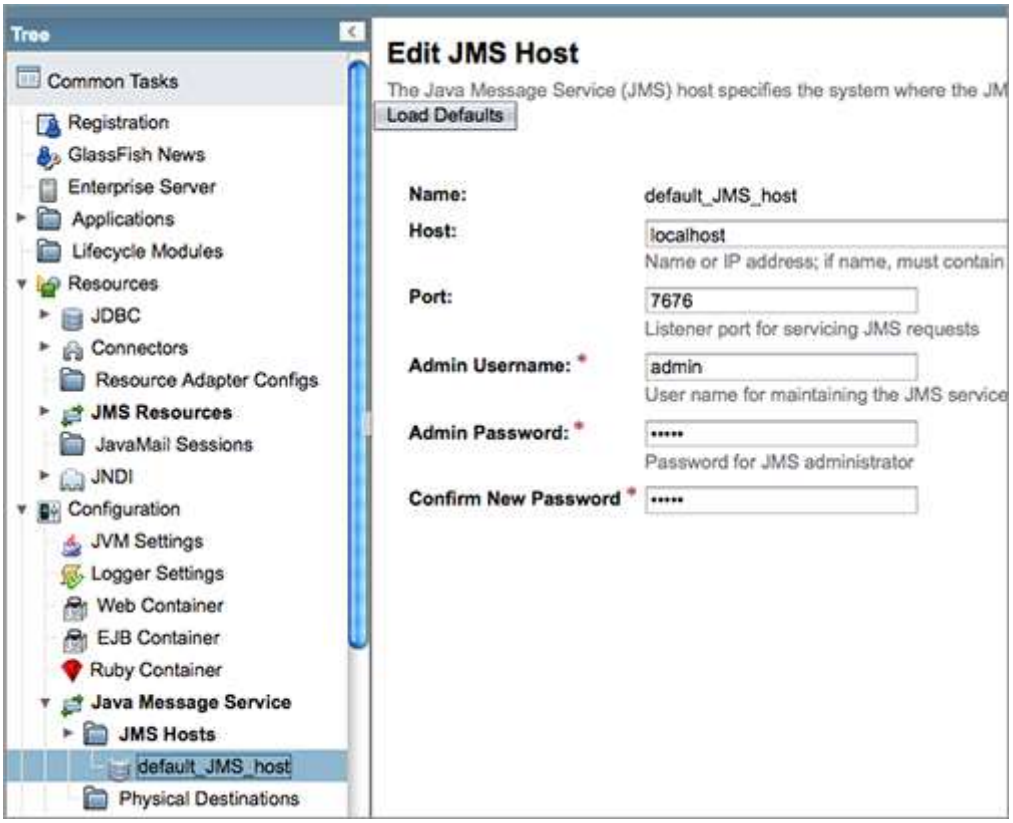


Figure 31-2



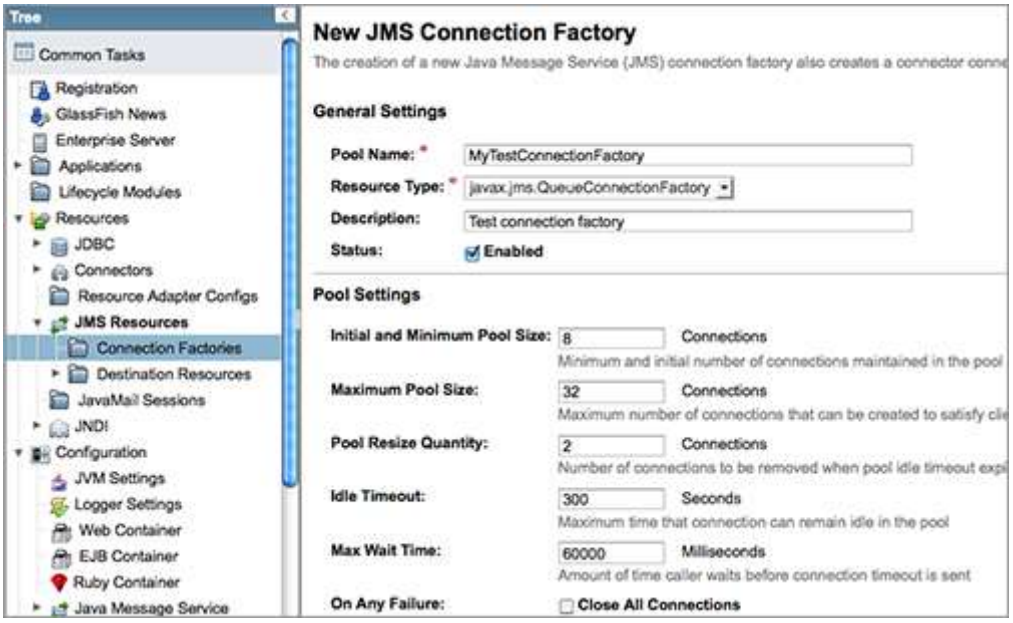
Now you need to create a GlassFish JMS entry to map to the physical MOM queue. Add the new destination resource to JMS Resources (see **Figure 31-3**). I called this JMS resource MyJMSTestQueue and mapped it to TestQueue, created in the previous lesson.

Now create another administered resource, a connection factory. Because the creation and closing of connections and senders are slow operations, you may want to consider writing code for JMS connection pools. Java EE servers enable you to configure such pools. In **Figure 31-4** you can see that the size of the connection pool is configurable. By default, eight JMS connections will be created and, as the number of users increases, up to 32 connections will be created by GlassFish.

Figure 31-3



Figure 31-4



To have JMS implementation as part of a Java EE server you need to download the full profile version (see Lesson 26).

Accessing the GlassFish Naming Service with JNDI

To start searching for an object in the JNDI tree a Java program needs to get a hold of its initial node. There are two use cases to consider here:

- An external Java client needs to find the server that runs the naming service.

- A Java client is running inside the Java EE server that runs the naming service.

In the first case you need to prepare an instance of the Java `Hashtable` or `Properties` object and give it as a parameter to the constructor of the `javax.naming.InitialContext` class. The values to put in such `Properties` objects vary — consult the documentation for your application server. The following code is an example:

```
Properties props = new Properties();

props.setProperty("java.naming.factory.initial",
                   "com.sun.enterprise.naming.SerialInitContextFactory");
props.setProperty("java.naming.factory.url.pkgs","com.sun.enterprise.naming");
props.setProperty("java.naming.factory.state",
                "com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl");
props.setProperty("org.omg.CORBA.ORBInitialHost", "localhost");
props.setProperty("org.omg.CORBA.ORBInitialPort", "8080");

InitialContext ic = new InitialContext(props);
```

In the second case the Java client can just create an instance of `InitialContext` with no argument constructor — the Java EE server knows where to find its naming service:

```
InitialContext ic = new InitialContext();
```

Now you can either perform a lookup starting from the initial context node, or ask the container to inject the reference to the required resource using the syntax of Java annotations.

Revisit the code in Listing 30-1. It creates a `ConnectionFactory` object connecting to a MOM server using classes specific to Open MQ. Now, because TestQueue is known to GlassFish, a Java client wouldn't even need to know what MOM server is in use, but would just look up a standard JMS `ConnectionFactory` in the instance via the JNDI API.

The statement that a Java client is running inside the server may not sound right to you, but any program that's trying to find another object can be considered a client. For example, imagine a web application in which the web browser invokes a Servlet that needs to place an order into a message queue or invoke an EJB. In **Listing 31-1** you can see a Servlet that plays the role of a Java client operating inside a Java EE container.



Listing 31-1: Servlet as Java client

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(description = "The servlet that Sends a message to a queue",
            urlPatterns = { "/quote" }, name="QuoteService")
public class QuoteService extends HttpServlet {

   protected void doGet(HttpServletRequest request,
       HttpServletResponse response) throws ServletException, IOException {

       PrintWriter out = response.getWriter();
       out.println("<html><body bgcolor=yellow>");
      out.println("<h2>Hello from QuoteService</h2>");

       out.println("Sending a message to the TestQueue");
```

```
        MessageSender mySender=new MessageSender();
        mySender.sendMessage("IBM 200 Buy");
    }
}
```

The Servlet receives a request from the client and instantiates the Java class `MessageSender` shown in **Listing 31-2**. Compare the code of this class with the sender from Listing 30-1. Now this code from **Listing 31-1** doesn't contain anything specific to the MOM provider.

If, say, you decide to change the MOM provider to the one that performs better, just re-create the administered JMS objects using the administration console of GlassFish and there is no need to change anything in the code of `MessageSender`. If you decide to switch from GlassFish to another server, just redeploy this Servlet and `MessageSender` in another Java EE application server.



Listing 31-2: Looking up JNDI object and sending message

```
package com.practicaljava.lesson31;

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

public class MessageSender {

void sendMessage(String messageToSend){
 Session session=null;
 ConnectionFactory factory=null;
 Connection connection=null;

 try{
    // Find the JNDI context
    Context jndiContext = new InitialContext();
```

```
    // Look up the factory and the queue
   factory = (ConnectionFactory) jndiContext.lookup("MyTestConnectionFactory");
   Queue ioQueue = (Queue) jndiContext.lookup("MyJMSTestQueue");

   connection = factory.createConnection();

   connection.start();

   session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
   MessageProducer queueSender = session.createProducer(ioQueue);

   // Buy 200 shares of IBM in this example
   TextMessage outMsg = session.createTextMessage(messageToSend);

   queueSender.send(outMsg);
   queueSender.close();

   System.out.println("Successfully placed an order to purchase 200 shares of
     IBM");

  }catch (JMSException e){
                 System.out.println("Error: " + e.getMessage());
  }catch (NamingException e) {

      e.printStackTrace();
  }finally{
    try{
     session.close();
     connection.close();
  } catch (Exception e) {
    System.out.println("Can't close JMS connection/session " + e.getMessage());
  }
  }
 }

 }
```

Run the quote Servlet in Eclipse or direct your web browser to **http://localhost:8080/Lesson31/quote** and the message to purchase 200 shares of IBM will be placed in the physical queue TestQueue in the Open MQ server. If you are not sure if this application is working the way you expect it to, see if there are error messages in the domain's log file in the directory `glassfish/domains/domain1/logs/server.log`. If this program has executed properly, at the end of the log file you'll see the fol-

lowing message: “Successfully placed an order to purchase 200 shares of IBM.”

Injection of JNDI Resources

Even though looking up the JNDI object doesn’t look too difficult, there is even a cleaner and simpler way of providing these resources to your Java EE components: using the annotation `@Resource`. The application server can inject the resource so you don’t need to do a JNDI lookup. For example, injection of `ConnectionFactory` and `Queue` can look like this:

```
import javax.annotation.Resource;
...
@Resource(name="MyTestConnectionFactory")
private ConnectionFactory factory;

@Resource(name="MyJMSTestQueue")
private Queue ioQueue;
```

Resources can be injected into variables, as in the preceding example, or you can put the `@Resource` annotation above the method or a class definition. Depending on its location, the time of injection varies. If you put this annotation at the class level, the resource will be injected during run time when the application looks it up. If you put this annotation above the field or setter method declaration, the resource will be injected in the component when the application is initialized.

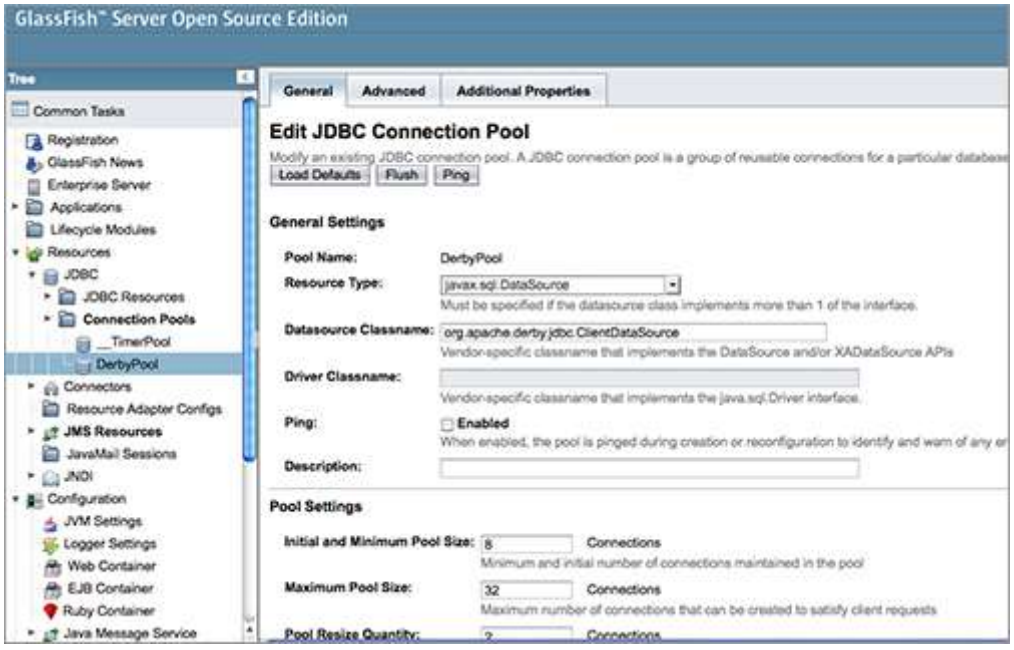
If you need to override resources specified in annotations, you can do it in XML configuration files. You’ll see more examples of using resource injection with EJB in Lesson 32.

DataSource and JNDI

In Lesson 22, while learning JDBC, you were creating database connections before executing any database queries. Imagine that multiple clients send requests to your application server, which has to execute database queries. Creating and closing connections are slow operations, and you want to do them as infrequently as possible.

Typically the administrator of the Java EE server pre-creates the pools of database connections, and configures the minimum and maximum number of connections and some other parameters. **Figure 31-5** shows a snapshot of the GlassFish administration console, where the JNDI entry named DerbyPool represents a pool of JDBC connections. The object `javax.sql.DataSource` is a factory of database connections. The administrator configures this factory as a JNDI resource, specifying what JDBC driver to use, how many connections to create initially, and the maximum number of connections allowed.

Figure 31-5



The user's request will communicate with a Java class deployed on the server, which will find the connection pool by its JNDI name (DerbyPool in this case) and make a `getConnection()` call on this resource. If there are available free connections, the Java class will immediately

get an instance of the `PooledConnection` object (which works the same way as `Connection`). If many users are making concurrent requests, all connections may be taken, and there will be a slight delay until one of the busy connections is returned to the pool. Connections returned to the pool are not destroyed; they are preserved by the application server for future requests.

You can do either a JNDI lookup of the `DataSource` object or inject it to your Java code. If the name of the configured `DataSource` object is DerbyPool, the sample code to obtain a pooled database connection may look as follows:

```
InitialContext ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("DerbyPool");

Connection myConnection = ds.getConnection();
//The rest of the JDBC processing goes here as in Lesson 22

// Closing the connection just returnes it back to the pool
// and makes it available for other Java clients
myConnection.close();
```

Injecting a `DataSource` using the `@Resource` syntax will look as follows:

```
@Resource(name="java:global/DerbyPool")
DataSource ds;
```

Lightweight Directory Access Protocol

LDAP servers are specialized software products that store directory entries in hierarchical trees and are highly optimized for reading. This makes them a good choice for such directory services as employee lists

or phone directories in an organization. Directories are mainly read, not changed, and this is where LDAP servers shine.

From the enterprise Java perspective you should keep LDAP solutions in mind when a really fast lookup of some Java objects is needed, such as with JMS connection factories, queues, or topics. Java developers use the JNDI API to look up and update the entries in LDAP. To use the same analogy as in the previous lesson, JNDI is to LDAP servers as JDBC is to DBMSes.

These are some popular LDAP servers:

- Oracle Directory Server (Oracle)
- Microsoft Active Directory (Microsoft)
- OpenLDAP (open-source, community developed)
- ApacheDS (open-source, Apache)
- OpenDS (open-source, Oracle)

The LDAP directory tree has a root entry, which consists of one or more distinguished names (unique identifiers). Typically the top of the hierarchy is an object with the prefix o for organization. One level below has the prefix ou for organizational unit, cn stands for common name, and so on. Unlike with other naming services, the search string starts from the very lowest hierarchical entry and the root entry has to be specified last. Here's an example of a distinguished name that can be used in a search:

```
cn=jsmith, ou=accounting, o=oracle.com
```

This example corresponds to the following hierarchy in an LDAP tree:

```
o = oracle.com
  ou = accounting
     cn = jsmith
```

The following code snippet prepares the JNDI properties, connects to the LDAP server, and finds the object called `CustomerHome`.

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
                       "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389");
env.put(Context.SECURITY_AUTHENTICATION,"simple");
env.put(Context.SECURITY_PRINCIPAL, "cn=Directory Manager");
env.put(Context.SECURITY_CREDENTIALS,"myPassword");

DirContext ctx = new InitialDirContext(env);

CustomerHome custHome =(CustomerHome) ctx.lookup("cn=CusomerHome,
ou=RealBigProject, o=trump.com");
```

To study distributed applications you can run all the examples from this book (clients and servers) on a single computer, but real-world distributed applications can be constructed in various ways, for example:

- Computer #1 runs the LDAP server.
- Computer #2 runs an application server that has registered (published) some objects with the LDAP server on Computer #1.
- Computer #3 has a client program that finds object references on Computer #1 and invokes their methods on Computer #2.
- Computer #4 has a database management system that is being used by the application server running on Computer #2.

- Computer #5 publishes financial market data, and Computer #2 subscribes to this service.

...and so on, and so on.

In this lesson you've seen how to use JNDI, and some example of why you may want to use it. Comparing the code in Listing 30-1 with the one in **Listing 31-1** is just one example that shows that naming and binding specific physical objects under a generic JNDI name enables you to remove these physical locations from your program code. You'll see more examples of using JNDI in Lesson 32.

Try It

Create a class called `MessageReceiver` to consume messages. It has to perform a JNDI lookup and start listening to TestQueue. When the message arrives `MessageReceiver` should display it. Making a request to the Servlet from **Listing 31-1** has to send a message to the queue, and `MessageReceiver` receives it and prints its content.

Lesson Requirements

You should have Java and GlassFish v3 installed.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find them in the Lesson31 folder in the download.

Step-by-Step

1. Create a new class called `MessageReceiver` that implements `MessageListener` with the code similar to the one shown in Listing 30-2, but this time performing a JNDI lookup of MyTestConnectionFactory and MyJMSTestQueue.

2. Find out how to configure the GlassFish start-up class, and implement it to have `MessageReceiver` instantiated whenever the server starts.

3. Test a send/receive cycle by using the Servlet from **Listing 31-1**.



Please select Lesson 31 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 32

Introduction to Enterprise JavaBeans

This lesson introduces you to one of the Java EE technologies, Enterprise JavaBeans, which can be used for implementing the business tier in a distributed application (refer to Figure 26-1). Lessons 27 and 28 were about various ways of programming the presentation tier on the Web; in Lessons 30 and 31 you learned how to organize communication between the different tiers of the application using messaging. You programmed the application business logic in POJOs.

Now you'll see how to program the business tier in EJBs, which are also POJOs, but managed by an EJB container. With the release of Java EE 6, EJB 3.1 and the Java Persistence API (JPA) 2.0 do offer you a clean and simple way to implement solutions for business logic and data persistence. The features covered in this lesson will work in application servers that support EJB 3.1. Such servers include GlassFish v3, JBoss 6, and WebSphere 8, and more vendors are working on making their application servers Java EE 6–certified. There is also the Apache TomTom bundle, which includes the open-source Apache OpenEJB project; at the time of this writing this bundle partially supports EJB 3.1.

In this lesson I'll introduce you to various types of EJBs, and the next lesson is about persisting data with JPA.

Who Needs EJB Containers?

What's wrong with POJOs? Why not just implement business logic there? You certainly can, but most likely you'd need to spend additional time manually programming a multi-threaded environment for them. The chances are that your application needs transactional support — if the business logic is located in POJO 1 and POJO 2, and the second one fails, you want to roll back whatever has been completed by the first one.

It's great that you know how to program message receivers using JMS, but how can you make this solution scalable? What if you need a couple of dozen message listeners that concurrently de-queue the messages? Don't forget about the tasks of integrating your POJOs with other tiers to perform authentication and JDBC operations.

EJB containers take care of all these infrastructure-related concerns without requiring manual programming on your side. You'll simply configure pools of **message-driven beans** (MDBs) if you need multiple message listeners. You'll turn on transactional support if need be. You don't need to worry about multi-threading problems if your beans operate in an EJB container. The configuration process is as simple as adding Java annotations in the EJB classes. Intercommunication among beans is done via dependency injection or singleton beans.

If you need to scale a distributed application, EJB containers offer clustering and failover support. Security authorization is also the container's responsibility. Without the need to manually code all these infrastructural functionalities, the code of your EJBs becomes really light.

The other interesting service offered by EJB containers is asynchronous method invocation, which enables asynchronous processing without the need for JMS or MOM.

EJB 3.1 supports embeddable containers: You can run EJB applications in a Java SE environment. This is a great idea for tasks such as testing

because there's no need to depend on the readiness of a Java EE server — just test your EJBs locally. Creating an embeddable container is as simple as executing one line of code:

```
EJBContainer myContainer = EJBContainer.createEJBContainer();
```

If the originally selected application server doesn't deliver the performance or reliability you expected, deploy your EJBs in a different Java EE 6–compliant server without changing a line of code.

Types of EJBs

There are two major types of EJB: **session beans** and **message-driven beans**. MDBs specialize in retrieving messages from JMS queues or topics — this is all they can do. But session beans are where your application business logic will reside. There are three types of session beans:

- A **stateless** session bean is one that contains business logic but doesn't support state. In other words, it doesn't "remember" any data specific to the client. If the same client invokes two methods in a row on the stateless bean `FindBooks`, the container may use two separate instances of the `FindBooks` bean, as it doesn't store any intermediate data specific to the client.
- A **stateful** session bean is one that contains business logic and state. The EJB container allocates a specific instance of the session bean to the client and can store results between subsequent method invocations.
- A **singleton** session bean is one guaranteed to be the only instance of its kind in the container. Think of a singleton bean as a global repository, in which one bean can put some data to be used by another bean. Singleton session beans not only provide easy access to

common data but also ensure that there are no race conditions in cases of concurrent access.

Older EJB specifications defined **entity beans** for data persistence. Formally they still exist in the EJB 3.1 specification, but they are being pruned, and most likely will disappear from future versions of Java EE releases.

An EJB container creates and maintains pools of session beans. The instances of the stateless beans are allocated to clients for much less time than stateful ones. Therefore, if you are working on a multiuser application with hundreds or thousands of concurrent requests to the EJB container, stateless beans offer a much more scalable solution, because a smaller pool can serve more users' requests.

But even with stateful session beans the EJB container is playing smart, and those instances sitting in memory without active interaction with the client are being **passivated** — removed from memory and stored on the disk. When the client issues another request to a stateful bean that has been passivated, the container will **activate** it again.

Stateless Session Beans

I'll introduce you to stateless session beans by creating a simple example featuring an EJB that contains the business logic to return the message "Hello World."

The Bean

Having a class that just returns "Hello World" is an obvious case for using a stateless session bean: It'll have only one method and there is no state to remember. **Listing 32-1** shows you how to program such a bean.



Listing 32-1: HelloWorld session bean, take 1

```
@Stateless
public class HelloWorldBean {

    public String sayHello(){
        return "Hello World!";
    }
}
```

Basically, you create a POJO and annotate it with one or more Java annotations. There are no special interfaces to be implemented to turn a POJO into an EJB. Accordingly there are `@Stateful`, `@MessageDriven`, and `@Singleton` annotations to mark other types of session beans. You may use the optional configuration file `ejb-jar.xml` in the application server's directory `META-INF`. This file also enables you to specify metadata for the EJB, and if you need to change the metadata on a production system without recompiling the code, make the changes in `ejb-jar.xml` — they will override the values specified via annotations.

The Client's View

The bean shown in **Listing 32-1** will run on the server, but the client that will invoke the `sayHello()` method can run either in the same JVM (e.g., a servlet or another bean) or in another one (e.g., a standalone Java SE application or Java EE class deployed in another container). Beans may or may not implement interfaces. If `HelloWorldBean` will be used only by clients running in the same JVM, you can mark it with a `@LocalBean` annotation to show that it's a no-interface bean.

If you'd like to expose certain business methods to local clients, you can create an interface and mark it with a `@Local` annotation. Your bean

class has to implement this interface. If you'd like to expose your bean to remote clients, create an interface and mark it as `@Remote`. The bean has to implement this interface.

Local No-Interface Beans

I am planning to use `HelloWorldBean` from the servlet running in the same JVM, so the final version of the code will look like **Listing 32-2**. Any EJB can use regular Java classes to implement business logic: For example, the `sayHello()` method can create instances of other Java classes with the `new` operator and invoke their business methods if need be.



Listing 32-2: HelloWorld session bean, take 2

```
import javax.ejb.LocalBean;
import javax.ejb.Stateless;

@LocalBean
@Stateless
public class HelloWorldBean {

    public String sayHello(){
        return "Hello World!";
    }
}
```

Now let's do it hands-on. Create a new Dynamic Web Project in Eclipse as you did in previous lessons. Name it Lesson32. Then create a new servlet class, `HelloWorldServlet`, in the package `com.practicaljava.lesson32.client` (select the menu File ⇒ New ⇒ Servlet). This servlet will become our client, communicating with the EJB.

Next create a Java class called `HelloWorldBean` in the package `com.practicaljava.lesson32.ejb` by selecting File ⇒ New ⇒ Other ⇒ EJB ⇒ Session Bean (EJB 3.x). Do not select any local or remote business interfaces, and uncheck (when possible) the objects related to EJB 2.x. Eclipse will create a class with a default constructor. Add to this class the method `sayHello()` shown in **Listing 32-2**, and the EJB is finished.

The next step is to inject the `HelloWorldBean` into the servlet code with the `@EJB` annotation:

```
@EJB HelloWorldBean myBean;
```

Eclipse will mark this line with a red error bullet. Right-click it and select Quick Fix to automatically insert two import statements — one for the `@EJB` annotation and the other for `HelloWorldBean`.

Using JNDI remains an alternative to injecting the bean into the client. Java EE 6 supports portable JNDI names that don't depend on the application server's implementation. Instead of the `@EJB` annotation you could use the following code:

```
Context ctx = new InitialContext();
HelloWorldBean myBean = (HelloWorldBean)
                        ctx.lookup("java:global/Lesson32/HelloWorldBean");
```

Now add the following two lines in the `doGet()` method of the servlet to make it invoke the method `sayHello()` on the EJB:

```
PrintWriter out = response.getWriter();
out.println(myBean.sayHello());
```

That's all there is to it. The complete code of the servlet is shown in Listing 32-3.



Listing 32-3: Servlet client for HelloWorldBean

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.practicaljava.lesson32.ejb.HelloWorldBean;

@WebServlet("/HelloWorldServlet")
public class HelloWorldServlet extends HttpServlet {

  @EJB HelloWorldBean myBean;

  protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {

      PrintWriter out = response.getWriter();
      out.println(myBean.sayHello());
  }
}
```

Local Beans

If you'd like to expose the business method to local clients, you can declare the interface marked with the `@Local` annotation and the bean will implement it, as in **Listing 32-4**.



Listing 32-4: Local interface and bean

```
@Local
public interface Greeting {
   public String sayHello();
}

@Stateless
public class HelloWorldBeanL implements Greeting{

    public String sayHello(){
        return "Hello World!";
    }
}
```

Remote Beans

For clients that may access the bean remotely, declare an interface marked with `@Remote` and have your bean class implement it. An EJB can implement both remote and local interfaces, and you can expose different methods for remote and local clients too.



Listing 32-5: The Greeting interface

```
@Remote
```

```
public interface Greeting {   public String sayHello();
}

@Stateless
public class HelloWorldBeanL implements Greeting{

    public String sayHello(){
        return "Hello World!";
    }
}
```

The clients find remote beans by performing JNDI lookups. Because the client and the server may be running in different JVMs, all arguments of the remote methods must be serializable.

The Server

Right-click the GlassFish server in the Servers view and add the project Lesson32 to deploy this Web application. You may also want to turn on automatic redeployments in GlassFish. To do this, double-click the GlassFish server in the Servers view and change the Publishing preferences to Automatically publish when resources change, as in **Figure 32-1**. This will eliminate the need for manual redeployment of the application every time the code changes.

Start GlassFish if it's not running and run `HelloWorldServlet`. You'll see the output produced by the EJB, as in **Figure 32-2**; you can get the same output by entering **http://localhost:8080/Lesson32/HelloWorldServlet** in your web browser. Even though this is a tiny application, nevertheless it demonstrates communication between objects from the presentation and business tiers. Real-world applications have more business logic, but from an architectural perspective they are the same.

Figure 32-1

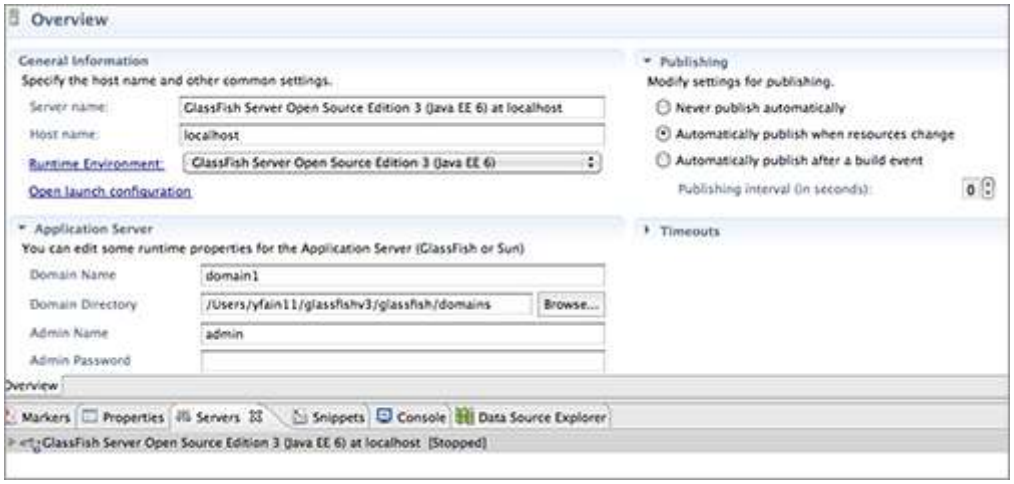
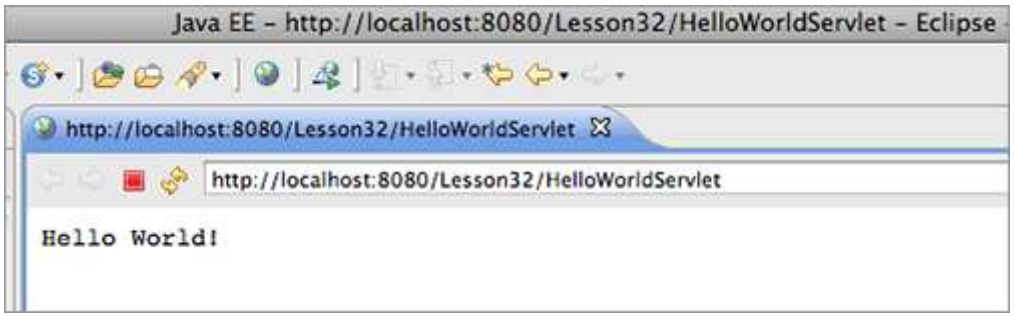


Figure 32-2



Asynchronous Methods

There is one more feature in stateless beans that I'd like you to be aware of: **asynchronous methods**. Imagine that `sayHello()` is a long-running method performing some lengthy calculations and you'd like to call it and continue with other operations without waiting for it to complete. In a Core Java application you would start a thread that would eventually return the `Future` object, as explained in Lesson 21.

In the EJB container, however, you shouldn't create and start threads from the application code, because EJB container takes care of all multi-threading issues for you. The good news is that asynchronous methods offer a solution when you need to introduce parallel processing from EJBs.

Just mark the method with the `@Asynchronous` annotation and have it return an object of type `javax.ejb.AsyncResult`, which is an implementation of

the `Future` interface:

```
@Asynchronous
public <Future>String sayHello(){

        // Some lengthy calculations go here
        //...

        return new AsyncResult<String>("Hello World!");
    }
```

The client would make a call to `sayHello()`, then execute some other code without waiting for `sayHello()` to complete, and at some point would request `Future` by making a blocking call, `get()`, as in the following code snippet:

```
//Asynchronous call
Future<String> myFutureGreeting = myBean.sayHello();
// Some other code that is executed immediately after the above line goes here
//...
}

// Sometime later you can make a blocking call and start
// waiting for the result of sayHello()
String myGreeting = myFutureGreeting.get();
```

You can also use asynchronous methods if a client needs to start more than one method on the EJB to run in parallel threads. The methods don't even need to return values; say one is generating large PDF files and the other prepares shipments based on today's orders. In this case you don't even need to process returned values; just declare the methods as asynchronous and invoke them from the client. They'll start immediately and will run in parallel.

Stateful Session Beans

Although stateless session beans are given to the client just for the time one method execution takes, stateful beans are allocated to the client for longer. They remember the results of the execution of previous method(s). For example, you can use a stateful bean to implement shopping cart functionality, enabling the user to add more than one item to the cart while browsing the company's catalog. When the client ends the session the stateful bean can be allocated to another client.

Suppose you have a stateful EJB called `MyShoppingCart`. The client looks up this bean using JNDI and makes the first call to the method `addItem()`. Then the client continues browsing and adds another item to the shopping cart. Then the client decides to check out and calls the method `placeOrder()`. All these method invocations are done on the same instance of the stateful bean `MyShoppingCart`.

```
MyShoppingCart myCart = (MyShoppingCart)
                        ctx.lookup("java:global/OnlineStore/MyShoppingCart");

// The client is browsing the catalog and finds the first item to buy
...
myCart.addItem(myFirstItem);

// The client continue browsing the catalog and finds the second item to buy
...
myCart.addItem(mySecondItem);

// The client is ready to check out
...

myCart.placeOrder();
```

To complete the shopping process and release the stateful bean for other clients you need to call one of the bean's `MyShoppingCart` methods

marked with the `@Remove` annotation. In the preceding example the method `placeOrder()` should be marked with this annotation. You should also provide another `@Remove` method on the bean to allow the client to cancel the order and release the bean.

There is one more way to release a stateful bean, by using the `@StatefulTimeout` annotation, which enables you to specify how long a bean can stay allocated to the client without any activity. When this time expires the session times out and the bean is released.

Singleton Beans

Pretty often an application needs a place to keep data that is accessible to and shared by all the beans. Another use case for a singleton is to control access to some external resources. For example, if a limited number of connections are available to some external web service, you can create a singleton that will implement throttling for EJBs that need these connections. A singleton bean can be used as global storage (or a cache) for the application — the state of this bean is shared among clients.

Only one singleton bean with any given name can exist per JVM per application. If you need several singletons in an application, give them different names. An EJB container doesn't create pools of singleton beans. To create a singleton EJB just mark it with the `@Singleton` annotation:

```
@Singleton
public class MyGlobalStorage {
...
}
```

When is the singleton bean created? It's up to the EJB container to decide, unless you specifically want to request that this bean be created on application startup. This is called **eager initialization** and there is a special annotation, `@Startup`, for it:

```
@Startup
@Singleton
public class MyGlobalStorage {

   private Map<String, Object> = new HashMap<String, Object>();
...
   addToStorage(String key, Object objToStore){...}

   removeFromStorage(String key){...}
}
```

Let's say that you'd like to create a program that at certain times sends some messages into a queue. Write a singleton bean, request that the EJB container instantiate it on application start-up, and start pushing the messages immediately after the singleton has been constructed. There is another handy annotation, `@PostConstruct`, that will cause the container to invoke a method immediately after the bean's constructor is finished:

```
@Startup
@Singleton
public class MyStockQuoteServer {
...
 @PostConstruct
 void sendPriceQuotes(){
   // The code connecting to the stock prices feed and
   // sending messages to a queue goes here
```

```
 }
}
```

To get access to the business methods of a singleton, the client Java classes will need to call the public static method `getInstance()` on the specific singleton, as shown in the following code snippet. If you'll be implementing the singleton design pattern manually you need to declare a private constructor and a public static `getInstance()` in the class, but otherwise the EJB container will take care of this.

```
MyGlobalStorage.getInstance().addToStorage("emplOfTheMonth", bestEmployee);
```

The EJB container allows concurrent access to singleton beans, and by default it applies the **container-managed concurrency** policy, sparing the developer worry about race conditions and such. You just need to use the `@Lock` annotation, specifying whether you want a resource to be locked during the read or write operation. If you prefer to write thread-synchronization code by yourself, switch to **bean-managed concurrency**. You set the type of concurrency using the `@ConcurrencyManagement` annotation, as shown here:

```
@Singleton
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class MyGlobalStorage {
...
}
```

Deploying EJB

Before deployment to any application server, EJBs are usually packaged into one archive file, which could be a jar, an **Enterprise Archive** (ear), which is a jar with a `.ear` file name extension, or a war. Even such a simple two-class web application as our current example should be compressed and deployed as one file. If your client is a web application it's convenient to keep the EJB inside a war file. Let's create one.

Right-click the Deployment Descriptor section in the Eclipse project Lesson32 and select Export WAR file. In a second you'll get the file `Lesson32.war` that contains both the servlet and the EJB. This file can be deployed in any Java EE 6–compliant application server. This war is really small: under 4 KB! Java EE 6 makes EJB components really lightweight. If you have multiple EJB classes, put them in one or more jars in the `WEB-INF/lib` directory.

If your client is not a small web application, or you want to keep EJBs deployed separately, you can package them inside a separate jar or ear file. The root directory that you'll be using to create ears has to have all compiled Java classes and, if applicable, the optional `ejb-jar.xml` file in the `META-INF` directory. As a matter of fact, you can create an ear file that will contain not only your EJBs, but also the war file.

Message-Driven Beans

MDBs (message-driven beans) perform only one function: They retrieve messages from queues and topics via the JMS API. The clients never need to look them up or invoke their methods. The client just needs to drop a message in a queue or publish it to a topic, and the MDBs that were listening to the messages at these destinations will get invoked automatically.

MDBs must implement the `MessageListener` interface. All configuration parameters can be specified in the parameters of the `@MessageDriven` annotation, as shown in **Listing 32-6**.



Listing 32-6: MDB MyMessageBean

```
@MessageDriven(mappedName="jms/testQueue", activationConfig =  {
        @ActivationConfigProperty(propertyName = "acknowledgeMode",
                                  propertyValue = "Auto-acknowledge"),
        @ActivationConfigProperty(propertyName = "destinationType",
                                  propertyValue = "javax.jms.Queue")
    })
public class MyMessageBean implements MessageListener {

MessageDrivenContext ctx;

  // A no-argument  constructor is required
  public MyListener() {}

  public void onMessage(Message message){
     // The business logic is implemented here.
  }
}
```

When a message appears in the queue named `testQueue` the EJB container picks one of the MDBs from the pool and invokes its method `onMessage()`, passing the message from the queue as an argument. Unlike with the stand-alone message receivers described in Lesson 30, with MDBs the container gives you excellent freebies: distributed transaction processing, automatic pooling, co-location of receivers and other beans, and simple assignment of queues or topics to the receivers in deployment descriptors. In addition, you can easily configure the number of receivers by specifying the pool size in the deployment descriptor.

You can use any client to send a message in a queue. It can be a stand-alone client, as shown in Listing 31-2, a servlet, an EJB, etc.

Timer Service

Many enterprise applications require the use of schedulers to perform certain repetitive tasks at certain times. Cron is a widely used scheduler for Unix-based applications. Windows also has a task scheduler (Control Panel ⇒ Scheduled Tasks). The open-source Quartz Scheduler is also popular among Java developers.

EJB 3.1 includes the `@Schedule` annotation, which takes a calendar-based expression so you can schedule the execution of required functionality in your beans. For example, you can create expressions that will invoke some business method every second, minute, hour, Monday, weekday midnight, etc.

The next code snippet shows how you can create a timer that will invoke the method `getPriceQuotes()` every second during weekdays from 9:30 a.m. to 4 p.m.:

```
@Stateless
public class MyStockQuoteFeed {

   @Schedule(second="*", minute="*", hour="9:30-16:00", dayOfWeek="Mon-Fri")
   public List getPriceQuotes(){
      // The code to connect to price quote feed goes here
      ...
   }
}
```

You can also create timers programmatically using the `TimeService` class and its methods `createTimer()`, `createSingleActionTimer()`, `createIntervalTimer()`, and `createCalendarTimer()`.

In addition to using `@Schedule` and programmatic timers, you can configure timers in the deployment descriptor `ejb-jar.xml`.

Summary

EJB 3.1 is a powerful solution for creating a scalable, easy-to-develop and -deploy, and lightweight solution for enterprise applications. Even a small-scale application can benefit from EJB 3.1. If your application doesn't need all the features mandated by the EJB specification, consider using EJB 3.1 Lite, which is a subset of the full specification.

It's not possible to cover all the features offered by EJB containers in one short lesson. I've introduced you to the main EJB concepts, but if you'd like more in-depth coverage, read Andrew Lee Rubinger and Bill Burke's Enterprise Java Beans 3.1, O'Reilly, 2010, ISBN: 0-596-15802-5.

Try It

The assignment for this lesson is to implement the `StockServer` class as an EJB and to use the timer to automatically generate and print stock price quotes every second. The new quotes should be sent to `testQueue` via the JMS API and consumed by a message-driven bean. Reuse the code of the sample stock server application of the `StockServerImpl` from Listing 25-2 to generate the price quotes.

Lesson Requirements

You should have Java and GlassFish v3 installed.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find them in the Lesson32 folder in the download.

Hint

If you want to push the stock prices to the end users, consider creating a JMS topic (e.g., `PriceQuotes`) to which the method `getQuote()` will publish

the latest prices. The Java client should subscribe to this topic to get the fresh quotes every second.

Step-by-Step

1. In Eclipse project Lesson32 create a new stateless session bean, `StockServerBean`, that includes the method `getQuote()`. The initial version of the bean may look as follows:

```
@Stateless
public class StockServerBean {
  private String price=null;
  private ArrayList<String> nasdaqSymbols = new ArrayList<String>();

  public StockServerBean(){

    // Define some hard-coded NASDAQ symbols
    nasdaqSymbols.add("AAPL");
    nasdaqSymbols.add("MSFT");
    nasdaqSymbols.add("YHOO");
    nasdaqSymbols.add("AMZN");
    nasdaqSymbols.add("MOT");
  }

  public void getQuote(String symbol){

    if(nasdaqSymbols.indexOf(symbol.toUpperCase()) != -1) {

        // Generate a random price for valid symbols
        price = (new Double(Math.random()*100)).toString();
    }
    System.out.println("The price of "+ symbol + " is " + price);
  }
}
```

2. Use the `@Schedule` annotation to have the `getQuote()` method invoked every second.

3. Replace the `println()` statement in the method `getQuote()` with the code sending a text message with the generated price quote to the queue `MyJMSTestQueue` configured in Lesson 31.

4. Create an MDB called `MyPriceConsumer` to retrieve and print messages from the queue `MyJMSTestQueue`.

5. Deploy this application in GlassFish and test it.



Please select Lesson 32 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 33

Introduction to the Java Persistence API

In the previous lesson you learned about various types of Enterprise Java Beans in which you could program the business logic of your application. Now it's time to talk about persisting data. If an online store allows users to place orders with session beans, there should be a mechanism for saving the data too. Typically, the data is persisted in DBMS.

The Java Persistence API (JPA) defines a standard way of mapping the Java classes to their relational database peers. This process is also known as **object-relational mapping (ORM)**. One of the popular third-party ORM frameworks, Hibernate, will be introduced in Lesson 36, but this lesson is a brief introduction to the standard JPA 2.0 implemented by any Java EE 6–compliant server.

The Big Picture

In the past, J2EE specifications recommend using Entity EJB to provide all interactions with databases. Currently, entity beans are undergoing a pruning process, and you should use JPA instead to deal with your application's data querying and persistence. As a matter of fact, JPA 2.0 can be used from Java SE applications too.

JPA enables you to specify and run queries and update data without needing to write SQL statements as you did in Lesson 22 while studying JDBC.

The Java Persistence API enables you to map Java classes to database tables using metadata and perform create, retrieve, update, and delete (CRUD) operations using Java Persistence Query Language (JPQL), the Persistence Criteria API, and native database queries in SQL language. The idea is to create an application-specific domain model as a set of interrelated Java classes and map it to the corresponding data storage (the DBMS).

If a Java class marked with the `@Entity` annotation has no argument constructor you can call it an **entity**. Each entity instance corresponds to a row in a database table, as shown here:

```
@Entity
public class Employee{
 ...
}
```

If you start with an empty database, JPA tools enable you to create database tables based on Java entities. You can also map Java entities to the existing database tables. Just like database tables, Java entities can have one-to-one relationships (such as an *`Employee`* entity with one corresponding *`OfficeAddress`* entity), one-to-many relationships (such as one *`Customer`* with many *`Orders`*), many-to-one relationships (the opposite of one-to-many relationships), and many-to-many relationships (for example, a *`UniversityClass`* has many enrolled *`Students`*, but each *`Student`* can enroll into multiple classes).

Every entity class must define a field containing a unique value, which is the equivalent of a primary key in a database table.

The `EntityManager` class will deal with objects. Before persisting data you can validate the values using the Bean Validation API.

While JPQL provides string-based SQL-like syntax for working with entities, the newer Criteria API enables you to dynamically construct queries from strongly typed objects.

Mapping Objects to Database Tables

You can map Java classes to database tables via annotations, XML configuration files, or both. For example, common fields can be mapped with annotations, and DBMS-specific mapping can be done in XML files. It does not have to be one-to-one mapping; one Java entity can be mapped to a set of columns from more than one database table.

Besides having fields mapped to table columns, Java entities can have embeddable classes, like `Address` in the `Employee` entity.

Typically a database table has a **primary key** — one or more columns that uniquely identify each row. Java entities must have one or more fields making each instance unique. Such an entity ID can be marked with the `@Id` annotation, and you can request your JPA provider to auto-generate the ID by adding the annotation @GeneratedValue. **Listing 33-1** shows an example of an `Employee` entity.



Listing 33-1: Employee entity

```
@Entity
public class Employee{

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
```

```
    @NotNull
    @Size(max=10)
    public String firstName;

    @NotNull
    @Size(min=2, max=20)
    public String lastName;

    @Column(name="boss_name")
    public String managerName;

    @OneToMany (mappedBy = "employee")
    public List<Address> addresses = new ArrayList<Address>();

  }
```

If you don't specify the table name as a parameter of the annotation, JPA assumes that there is a corresponding database table with the same name as the entity class, which in our example is `Employee`. The specified strategy `GenerationType.IDENTITY` means that DBMS has an auto-generated primary key with auto-increment. Many database management systems support either identity columns or sequence objects with similar functionality.

The fields that must have values are marked as `@NotNull`. If an instance of the preceding `Employee` entity won't have values in the `firstName` or `lastName` fields, Bean Validation can catch this and generate an error. The entity fields that don't have to be persisted should be marked with the `@Transient` annotation.

If a table column name is not the same as the name of the entity field, you can specify the column name using `@Column`. According to **Listing 33-1**, the database column name *`boss_name`* corresponds to the field `managerName` of the entity `Employee`.

Not every Java class that corresponds to some data in the database has to be an entity. You can have embeddable classes that define a group of properties that belong to an entity. Let's say a company gives to each

employee a smartphone identified by a phone number and model number. You can create a Java class to represent such a device and mark it with `@Embeddable`:

```
@Embeddable
public class SmartPhone implements Serializable{

   @Size(max=10)
   public String phoneNumber;

   public String model;
}
```

Now the `Employee` entity can embed the property of the `SmartPhone` type along with other fields:

```
@Entity
public class Employee{

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)

  @NotNull
  public String firstName;

 ...
  @Embedded
  public SmartPhone companyPhone;

}
```

The code in **Listing 33-1** illustrates the mapping of one-to-many relations between the entities `Employee` and `Address` (not shown). One employee can have multiple addresses, so `Employee` references a collection of the `Address` entities.

JPQL

JPQL is a SQL-like query language. The main difference is that SQL operates with the DBMS schemas, tables, and stored procedures, but JPQL manipulates with Java objects and their attributes from the domain model. You don't need to know details of the underlying data storage objects to perform JPQL queries.

If you know the queries in advance they can be precompiled; otherwise you can build them dynamically during the run time. Similarly, in JDBC you can use either `PreparedStatement` or `Statement`.

JPQL includes pretty easy-to-remember (and case-insensitive) keywords: `SELECT`, `FROM`, `WHERE`, `ORDER BY`, `GROUP BY`, and so on. Here's how you would write a JPQL query to find all managers who have subordinates with the last name Smith:

```
SELECT e.managerName,
FROM Employee AS e
WHERE e.lastName='Smith'
```

The next query finds all employees who were given iPhones by the firm. Note the dot notation to find the phone model from the embedded class.

```
SELECT e.firstName, e.lastName
FROM Employee AS e
WHERE e.companyPhone.model='iPhone'
```

To select all fields of certain entities (the equivalent of `Select *` in SQL) just specify the alias name of the entity right after the `SELECT` clause:

```
SELECT e FROM Employee AS e
```

The `Employee` and `Address` entities have a one-to-many relationship. If you'd like to find all employees who live in New York, this is the join written in JPQL:

```
SELECT DISTINCT e
FROM Employee AS e JOIN e.addresses as a
WHERE a.cityl='New York'
```

EntityManager

`EntityManager` is the centerpiece of persistence — it will execute all your JPA requests to read from or write into a database. Each instance of `EntityManager` is associated with a set of entities. Such a set is called a **persistence unit**. In Java EE containers the `EntityManager` can be injected as a resource, for example:

```
@PersistenceContext
EntityManager em;
```

In Java SE applications you need to instantiate `EntityManager` programmatically using `EntityManagerFactory`:

```
private EntityManagerFactory factory;
```

```
private static final String PERSISTENCE_CONTEXT_NAME = "employees";
...
factory = Persistence.createEntityManagerFactory(PERSISTENCE_CONTEXT_NAME);
EntityManager em = factory.createEntityManager();
```

The entity manager can create, update, remove, and find entities by IDs or using a query. The code to find an `Employee` entity with the ID `1234` can look like this:

```
Employee employee = em.find(Employee.class, 1234);
```

To create a new row in the `Employee` database table, create an instance of the entity `Employee` and invoke the method `persist()` on the `EntityManager`. To delete a row, call `remove()`. Your application can explicitly begin and commit transactions when the persistence is successfully completed.

```
@PersistenceContext
EntityManagerFactory factory;
EntityManager em;

@Resource
UserTransaction userTransaction;
...
em=factory.createEntityManager();

Employee newEmployee = new Employee();
newEmployee.firstName="Mary";
newEmployee.lastName="Thompson";
...
try{
  userTransaction.begin();
  em.persist(newEmployee);
  em.remove(oldEmployee);

 userTransaction.commit();
```

```
 }
 catch (SystemException e){ //several other exceptions can be thrown here
   e.printStackTrace();
   try{
    userTransaction.rollback();
   } catch(SystemException e1){e1.printStackTrace()}
 }
```

To select the manager name of the employee with the `firstName` Mary and the `lastName` Thompson, ask the `EntityManager` to run the following JPQL query:

```
EntityManager em;
List employees;
...employees = em.createQuery(
"SELECT e.managerName FROM Employee AS e WHERE e.firstName='Mary' "
      + " AND e.lastName='Thompson'").getResultList();
```

This static query works only for employees whose names are Mary Thompson. Note that the method `getResultList()` is invoked on the created query object. If you expect just one entity as a result, call the method `getSingleResult()` instead. To specify the first and last names dynamically, you should use parameters, for example:

```
EntityManager em;
List employees;

String firstName = "Mary";
String lastName = "Thompson";
...employees = em.createQuery(
"SELECT e.managerName FROM Employee AS e WHERE " +
  "e.firstName= :fname AND lastName= :lname")
   .setParameter("lname", lastName)
```

```
    .setParameter("fname", firstName)
    .getResultList();
```

One instance of `EntityManager` manages a **persistence unit** — a set of classes specified in the configuration file *persistence.xml*, which is located in the *META-INF* directory of the deployed EJB jar. If you package the application in the war file, this file has to be located either in the directory *WEB-INF/classes/META-INF* or in a jar under *WEB-INF/lib*.

The file *persistence.xml* specifies the name of the jar file that contains managed persistence classes and their names. It also contains the name of the JDBC data source (not the specific JDBC driver) used for communication with DBMS.

```
<persistence>
    <persistence-unit name="EmployeeManagement">
        <description>This unit manages Acme Employees </description>
        <jta-data-source>jdbc/HRDatabase</jta-data-source>
        <jar-file>MyEmployeeApp.jar</jar-file>
        <class>com.lesson33.Employee</class>
        <class> com.lesson33.Address</class>
    </persistence-unit>
</persistence>
```

A Brief Introduction to the Criteria API

JPQL is a string-based query language, but the Criteria API allows the creation of strongly typed object-based queries. On one hand it's more verbose than JPQL, but on the other there is no need to do data-type conversion when processing query results.

These are some core interfaces in the Criteria API:

- `CriteriaBuilder`: A utility class that can create criteria queries.

- `CriteriaQuery`: This is an object that will contain all parts of the query, such as `SELECT`, `FROM`, and `WHERE`. It's like a memory graph, in which each node represents some clause of the query.
- `Root`: Represents the root of the query.
- `TypedQuery`: A query prepared for execution.
- `Join`: An object that represents a `JOIN` clause.

The next code fragment shows an equivalent of the JPQL query `SELECT e FROM Employee AS e` written using the Criteria API:

```
EntityManager em;
...
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> crQuery = cb.createQuery(Employee.class);
Root<Employee> employee = crQuery.from(Employee.class);
crQuery.select(employee);
TypedQuery<Employee> tQuery= em.createQuery(crQuery);
List<Employee> employeess = tQuery.getResultList( );
```

Start with asking `EntityManager` to create `CriteriaBuilder`, which in turn creates the instance of `CriteriaQuery`. Note that via generics, the `CriteriaQuery` is typed based on the expected results. After that you add instances of required objects (`SELECT`, `FROM`, `WHERE`, and so on) to `CriteriaQuery`.

Finally, the `EntityManager` prepares the executable `TypedQuery` that produces strongly typed results by executing `getResultList()`. If you expect just one record back, use the `getSingleResult()` method. Adding several clauses to the query could look like this:

```
crQuery.select(employee).where(...).orderBy(...);
```

Because we are building the object graph, the order of the query classes in the preceding line is not important. The `Root` object can serve as a starting point for joining entities:

```
Root<Employee> employee = crQuery.from(Employee.class);
Join<Employee, Address> empJoin = employee.join(...);
```

The next example shows how to add the `LIKE` clause to get all employees with the last name Thompson:

```
Root<Employee> employee = crQuery.from(Employee.class);
crQuery.select(employee).where(
           cb.like(employee.<String>)get("lastName"),
           cb.parameter(String.class, "lname"));

TypedQuery<Employee> tQuery= em.createQuery(crQuery);
tQuery.setParameter("lname", "%Thompson%");
List<Employee> employeess = tQuery.getResultList( );
```

Bean Validation

The Bean Validation framework (JSR-303) is a Java API for ensuring that the values in entities are correct. Validation can be performed at different stages of the entity life cycle. Define the constraints on your entities and the validation can be automatically initiated when you are about to create, update, or remove an entity.

Listing 33-1 includes the `@NotNull` and `@Size` constraints defined in the package `javax.validation.constraints`, but you can create your own validation rules as well.

The life-cycle callback methods marked with the annotations `@PrePersist` and `@PreRemove` are invoked on the entity before the `EntityManager` persists or removes this entity. Accordingly, another pair of annotations, `@PostPersist` and `@PostRemove`, are invoked after these operations.

For example, you can put the validation code in the method `transferEmployee()`to ensure that the transfer has been approved by the employee's manager. Throwing an exception invalidates the operation.

```
@PrePersist
public void validateTransfer(){
   if (!transferApproved()){
        throw new TransferException("Manager didn't approve transfer");
  }
}
```

Try It

The GlassFish v3 server includes the binaries of the JPA provider EclipseLink (see **www.eclipse.org/eclipselink**). In this exercise you'll be using EclipseLink to auto-generate Java entities based on the existing database. This will serve as an example of object-relational mapping.

Lesson Requirements

You should have Java, the Eclipse Java EE IDE, and GlassFish v3 installed.



You can download the code and resources for this Try It from the book's web page at www.wrox.com. You can find them in the Lesson33 folder in the download.

Step-by-Step

1. Create a new JPA project in Eclipse by selecting File ⇒ New ⇒ Other ⇒ JPA ⇒ JPA project. Name it Lesson33 and keep the default settings of GlassFish Server Open Source Edition 3 as a target and Minimal JPA 2.0 Configuration as the configuration, as in **Figure 33-1**.

2. A pop-up suggesting the name of the default output folder displays. Don't change anything there; click Next.

3. The JPA Facet window (see **Figure 33-2**) will enable you to select the JPA provider — the libraries that implement JPA specification.

Figure 33-1

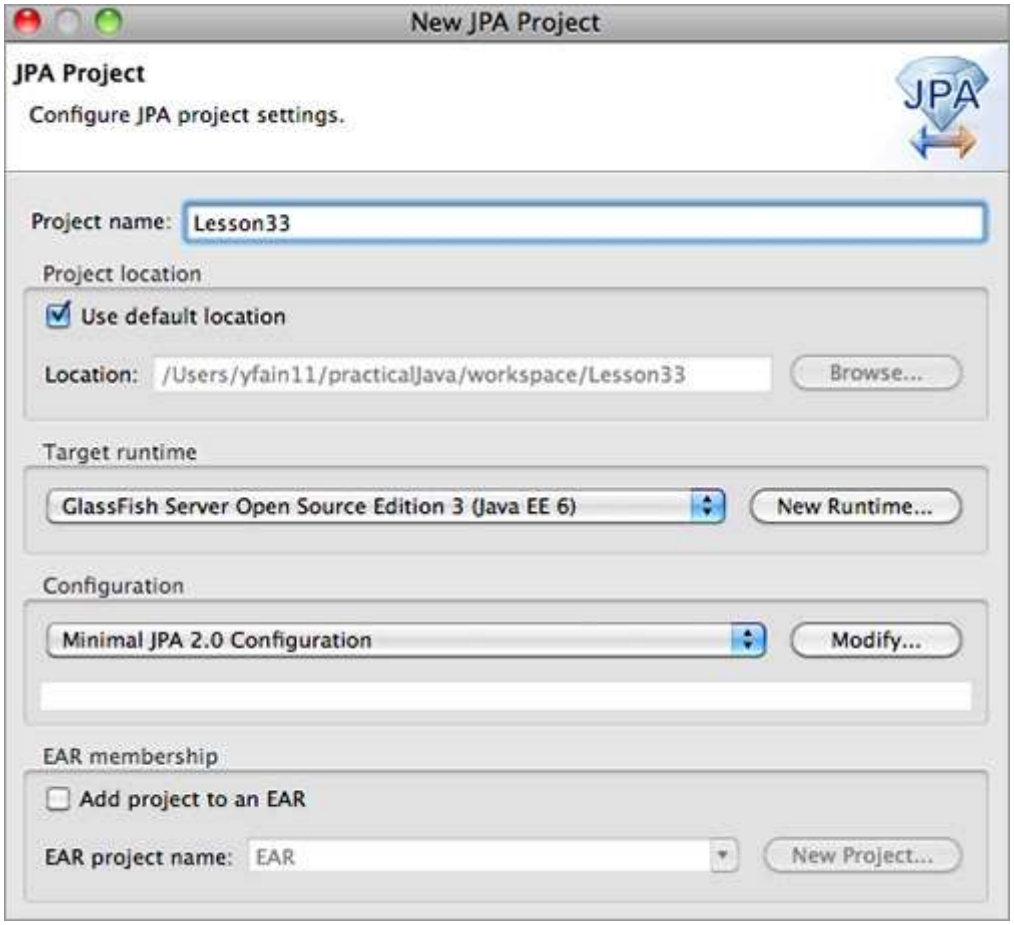
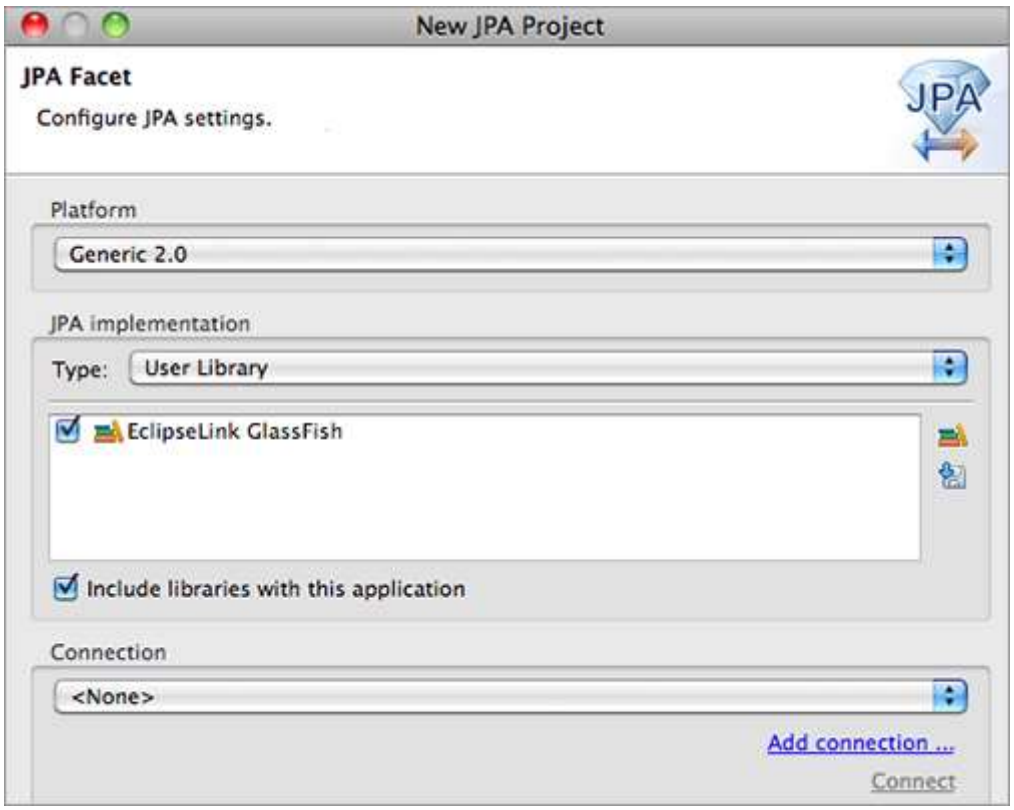

New JPA Project
JPA Project
Configure JPA project settings.
JPA
Project name: Lesson33
Project location
Use default location
Location: /Users/yfain11/practicalJava/workspace/Lesson33
Browse...
Target runtime
GlassFish Server Open Source Edition 3 (Java EE 6)
New Runtime...
Configuration
Minimal JPA 2.0 Configuration
Modify...
EAR membership
Add project to an EAR
EAR project name: EAR
New Project...

Figure 33-2


New JPA Project
JPA Facet
Configure JPA settings.
JPA
Platform
Generic 2.0
JPA implementation
Type: User Library
EclipseLink GlassFish
Include libraries with this application
Connection
<None>
Add connection ...
Connect

The User Library box will initially be empty. Press the little preferences icon next to it and then click New in the Preferences window. You'll need to add the following six jars from the folder `glassfishv3/glassfish/modules` to your new user library:

`org.eclipse.persistence.antlr.jar`

`org.eclipse.persistence.jpa.jar`

`org.eclipse.persistence.asm.jar`

`org.eclipse.persistence.jpa.modelgen.jar`

`org.eclipse.persistence.core.jar`

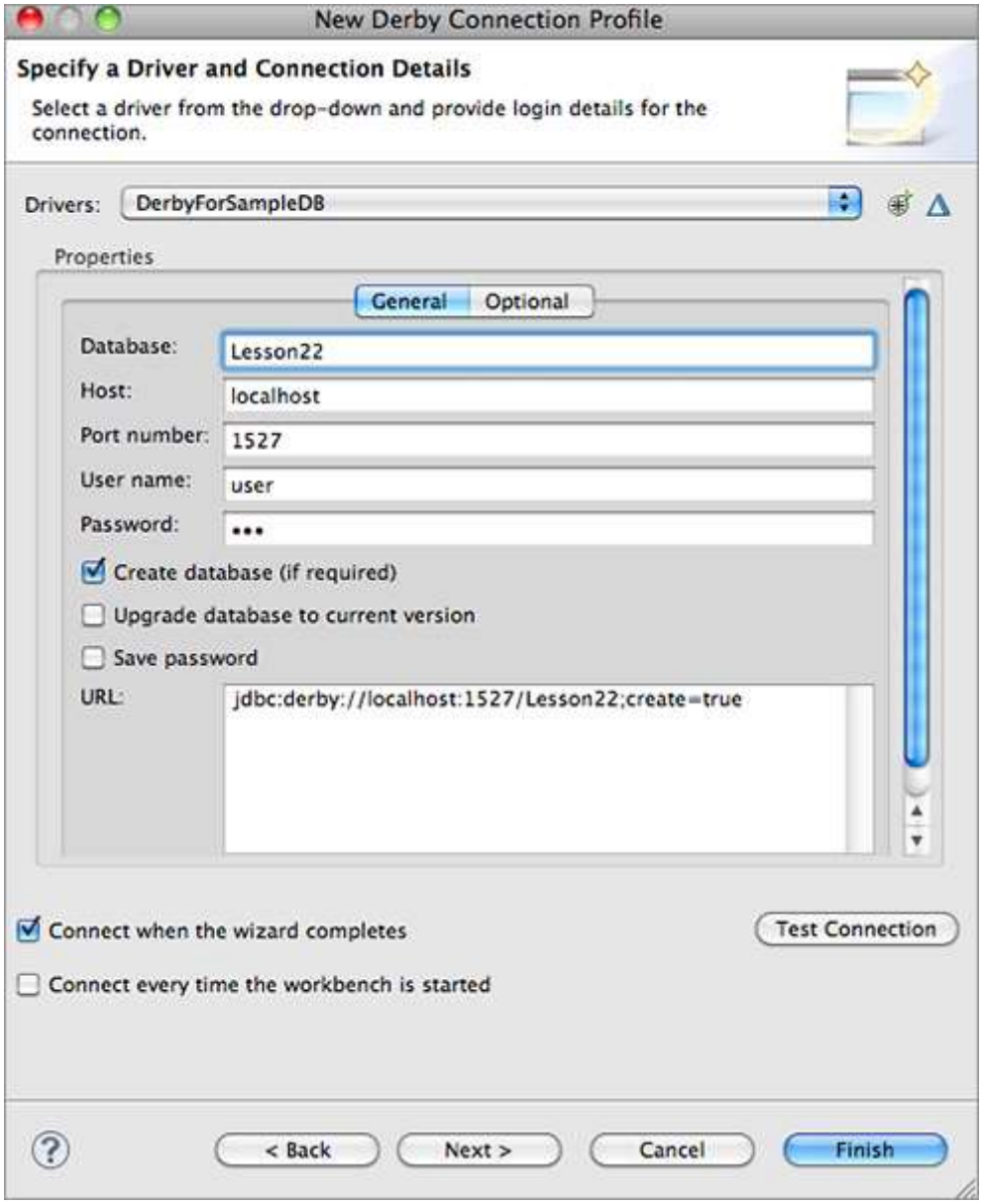
`javax.persistence.jar`

4. Start your Derby database server using the `startNetworkServer` command, as explained in Lesson 22.

5. In the connection drop-down (refer to **Figure 33-2**) select Sample JavaDB Database and click the Connect link. You'll see that Eclipse has opened the Data Source Explorer view showing the sample database that comes with Derby.

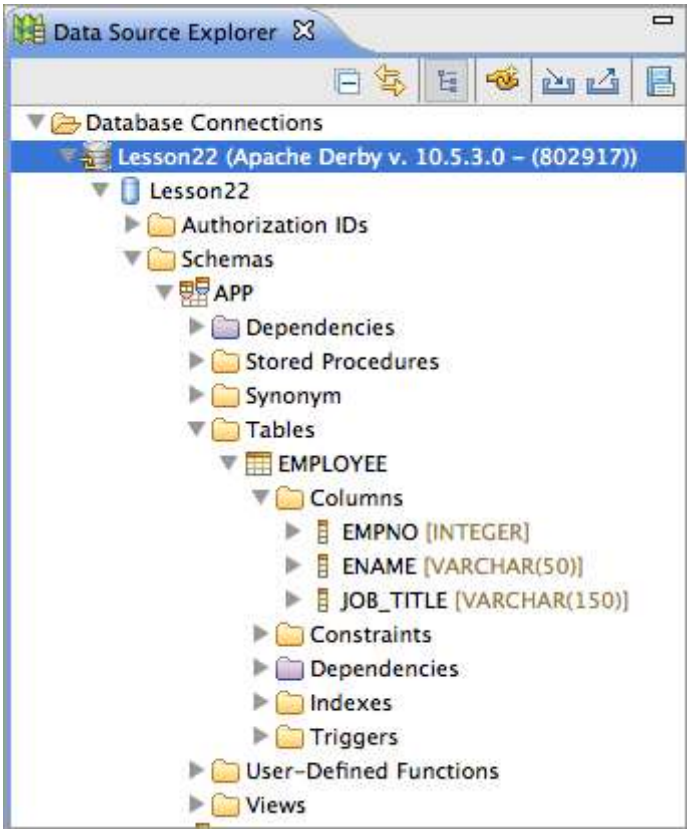
6. Right-click Sample JavaDB Database in the Database Connection view. Configure your new connection in the popup window (see **Figure 33-3**) to the database Lesson22 that you created in Lesson 22. Click Test Connection to ensure you did it right.

Figure 33-3



7. **Figure 33-4** is a snapshot of Eclipse view Data Source Explorer, which shows you the table Employee from Lesson 22. It's located in the APP schema.

Figure 33-4



8. JPA requires database tables to have primary keys, but our table Employee didn't define one. You'll need to fix this by making the column `empno` a primary key. You can do it from the command line via the ij utility (see Lesson 22) by issuing the following command:

```
alter table APP.Employee add primary key (empno);
```

9. Right-click the name of project Lesson33 and select JPA Tools ⇒ Generate Entities from Tables. In the pop-up window select the table Employee and click Finish.

10. Open the folder *src* in your Eclipse project and you'll find there a freshly generated class called `Employee` that will look as follows:

```
import java.io.Serializable;
import javax.persistence.*;
/**
 * The persistent class for the EMPLOYEE database table.
```

```
 *
 */

@Entity
public class Employee implements Serializable {
      private static final long serialVersionUID = 1L;

      @Id
      private int empno;

      private String ename;

 @Column(name="JOB_TITLE")
 private String jobTitle;

 public Employee() {}

 public int getEmpno() {
   return this.empno;
 }

 public void setEmpno(int empno) {
   this.empno = empno;
}

 public String getEname() {
   return this.ename;
 }

 public void setEname(String ename) {
   this.ename = ename;
}

 public String getJobTitle() {
   return this.jobTitle;
}

 public void setJobTitle(String jobTitle) {
   this.jobTitle = jobTitle;
 }
}
```

11. Finally, open the folder *META-INF* and you'll find generated there the file *persistence.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
```