If you properly deploy this servlet on `MyBooks.com`, the HTML client from Listing 27-1 will get as a response a web page containing the text "Hello from FindBooks." Note that the import statements include classes from the `javax.servlet` package, which is not included with Java SE — you need Java EE 6 SDK installed. GlassFish Server comes with Java EE 6, but if you are using a different container you can get it from `www.oracle.com/technetwork/java/javaee/downloads/index.html`.

HOW TO DEPLOY A SERVLET

The annotation `@WebServlet` is where you specify servlet deployment parameters. Prior to Java EE 6 you needed to specify deployment parameters in an XML file, but now that's optional. Our `FindBooks` servlet uses the deployment parameters `urlPatterns`, and `name`. The former is used to match one of the servlets deployed on the server to the URL. The value *`/books`* means that whenever the client sends a request containing the pattern `books` in its URL, the request has to be redirected to the `FindBooks` servlet. For example, the servlet container will send the request `http://localhost:8080/books` to the `FindBooks` servlet.

Each application server and servlet container has a directory known as a *document root*. It is used not only for servlet-based websites, but also for the deployment of static HTML files. For example, if you put the HTML file *`TermsAndConditions.html`* in the subfolder *`legal`* of the document root of the server `MyBooks.com`, users would need to direct their web browsers to `www.mybooks.com/legal/TermsAndConditions.html`.

In the GlassFish application server the default document root is the directory *`/glassfish/domains/domain1/docroot`*. In Apache Tomcat it's the directory *`webapps`*. If you are planning to create a servlet, its deployment directory will also be located in the document root, but it will contain the subdirectories *`WEB-INF`* and *`META-INF`*.

`WEB-INF` will have the subdirectories *`classes`* and *`lib`* and the optional file *`web.xml`*. The latter is optional if you can specify all deployment parameters in annotations. But if you have both annotations and `web.xml`, the values in this file will override the corresponding values in the annotations. This allows changing deployment parameters without requiring recompilation of the servlets. The *`WEB-INF`* directory also may have some container-specific files. For example, GlassFish has the file `sun-web.xml`, in which it stores the *context root*, which is the root directory of your Web application as well as the part of the URL that's used to access your application in the Web browser (`/Lesson27` in our case).

The directory *`META-INF`* may have files containing metadata about this web application, like *`manifest.mf`* or other data specific to a third-party framework or application server content. For example, Apache Tomcat has a file called *`context.xml`* where you may find information about the JDBC driver. This directory may also contain the file *`web-fragments.xml`*, which is a module of deployment mainly used by third-party frameworks to store their specific deployment parameters.

This is what the directory structure of the application deployed in the document root directory can look like:

```
document root dir
     WEB-INF
          Classes
             Com
```

```
                Practicaljava
                  lesson27
                     FindBooks.class
          Lib
    META-INF
        manifest.mf
```

The class `com.practicaljava.lesson27.FindBooks` was compiled into the directory *`classes`*. If you had some third-party jar files you could add them to the *`lib`* directory.

When your web application is complete, most likely it'll consist of multiple files, and typically the entire directory structure is deployed as one compressed file with the extension *`.war`*, which stands for web archive. Such files can be created manually, by your IDE plug-ins, or by one of the build tools like Ant or Maven. Later in this lesson I'll show you how to create a war file with Eclipse for Java EE Developers.

INSTALLING THE GLASSFISH PLUG-IN FOR ECLIPSE

It's a lot more convenient to develop and deploy Web applications when you don't need to leave the IDE. Eclipse for Java EE is a good fit for the development of web projects. NetBeans IDE and IntelliJIDEA are also good tools for Java EE developers. Because I'll be using GlassFish Server, let's install the GlassFish plug-in that's available for Eclipse IDE Helios or later.

From Eclipse's Help menu select Eclipse Marketplace. A new window will pop up: Enter `GlassFish` in the Find field. This search will return several GlassFish-related plug-ins; pick the Tools Bundle for Eclipse. This is a bundle that comes as two plug-ins. Press Install and follow the instructions. At the end of the installation, Eclipse will restart showing a GlassFish welcome window; close it and switch to the Java EE perspective.

Now configure GlassFish Server. The Eclipse view will allow you to start and stop the server and deploy applications without needing to leave Eclipse. Select File ➪ New ➪ Other and pick Server in the opened window. Select the option GlassFish Open Source Edition 3 (Java EE 6).

The next pop-up will ask you about a default JRE (it must be Java 1.6 or above) and where GlassFish Server should be located. According to my installation (see Lesson 26) I specified */Users/yfain11/glassfishv3/glassfish* and pressed Next. Accept default parameters for *`domain1`* in the next window and press Finish. Eclipse will start GlassFish Server, asking you for the administrator's ID and password that you specified when installing GlassFish in the previous lesson.

If you did everything right your Eclipse Servers view should look like Figure 27-2, and the Console view should show output similar to the following:

```
Waiting for DAS to start ..............
Started domain: domain1
Domain location: /Users/yfain11/glassfishv3/glassfish/domains/domain1
Log file: /Users/yfain11/glassfishv3/glassfish/domains/domain1/logs/server.log
Admin port for the domain: 4848
Command start-domain executed successfully.
```

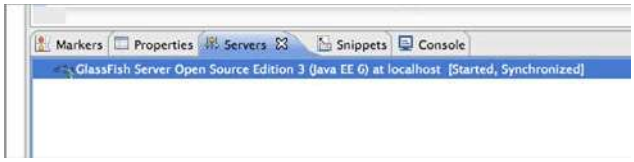


FIGURE 27-2

HOW TO CREATE A SERVLET WITH ECLIPSE

In the File ➪ New menu of Eclipse, select the option Create Dynamic Web Project. In the pop-up window name it **Lesson27**. Note that by default the GlassFish target run time has been selected (see Figure 27-3). If you have another servlet container configured in your Eclipse installation, you can deploy it under a different run time (such as Tomcat).

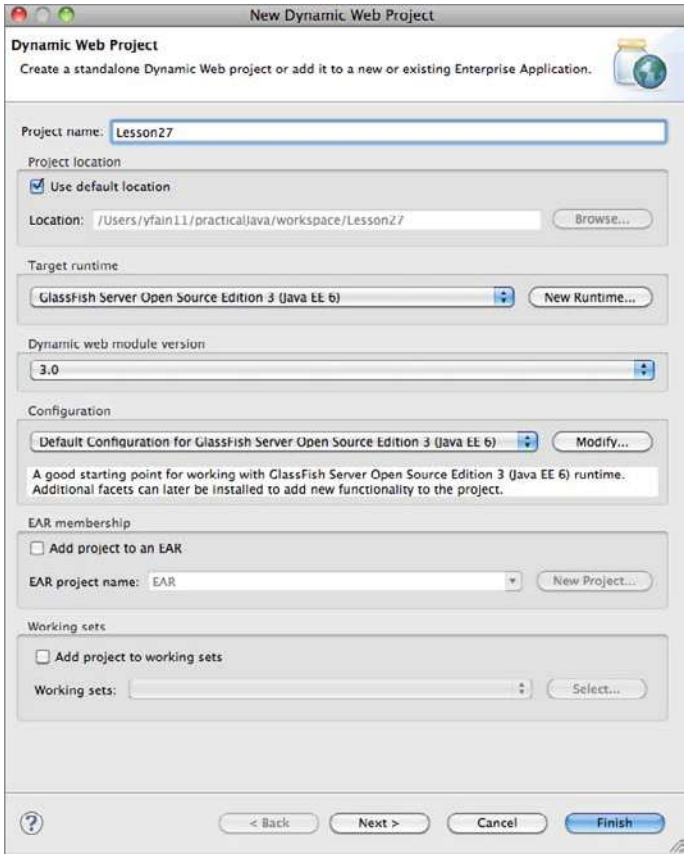


FIGURE 27-3

Click Finish, and it'll create a new Eclipse project, which is not the same as the projects you've been creating so far. It has a subfolder called *WebContent* that contains the *WEB-INF* and *META-INF* directories that will be used for deployment in the servlet container of your choice.

Right-click the project name and select New ➪ Servlet, then specify `com.practicaljava.lesson27` as the name of the package, and the class name as `FindBooks`. Click Next and edit the URL mapping field to be `/book`. Click Next again. The next window will ask you which method stubs you'd like to have auto-generated; keep the default `doGet()` and `doPost()`. Finally, click Finish, and the code in Listing 27-3 will be generated (I removed the comments).



LISTING 27-3: Generated FindBooks servlet

```
package com.practicaljava.lesson27;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(description = "The servlet that searches for books",
            urlPatterns = { "/books" })
public class FindBooks extends HttpServlet {
      private static final long serialVersionUID = 1L;

    public FindBooks() {
        super();
        // TODO Auto-generated constructor stub
    }

      protected void doGet(HttpServletRequest request,
         HttpServletResponse response) throws ServletException, IOException{
            // TODO Auto-generated method stub
      }

      protected void doPost(HttpServletRequest request,
         HttpServletResponse response) throws ServletException, IOException{
            // TODO Auto-generated method stub
      }
}
```

Add the following two lines in the `doGet()` method to get access to the output stream and send the message "Hello from FindBooks":

```
PrintWriter out = response.getWriter();
out.println("Hello from FindBooks");
```

Don't forget to add the import statement for `PrintWriter`. The final step is to deploy the servlet under GlassFish Server. Open the Servers view, right-click the server, and select Add and Remove

from the menu. Select the Lesson27 project in the left panel and add it to the right one (see Figure 27-4). Click Finish.

The servlet deployment is finished. Now right-click `FindBooks` in the Eclipse project, select Run on Server ➪ GlassFish and click Finish. Eclipse will run its internal web browser and will display the message shown in Figure 27-5. Please note that the URL ends with `/books` according to the specified `urlPatterns` parameter, `http://localhost:8080/Lesson27/books`.

Because servlets belong to the presentation layer, let's change the presentation a little bit. This code in `doGet()` will show the output in a larger font and in the header <H2> style:

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

If you want to create a war file deployable on any servlet container, find the Deployment Descriptor section in your Lesson27 project, right-click it, and select Export to WAR. In a couple of seconds you'll have `Lesson27.war`, a file that you can deploy in any Java EE 6–compliant container.



FIGURE 27-4

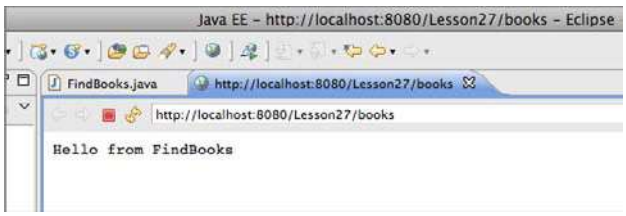


FIGURE 27-5

BROWSER-SERVLET DATA FLOW

One servlet can serve multiple users, so let's review the entire process of client-servlet communication. Java Servlets run in a container, which automatically spawns a new thread for every client's request without requiring you to do any thread programming.

The web browser can use an HTML form, a link, or another program that can send, say, a `Get` or `Post` request to the server. In the very beginning, when the first user's request hits the `FindBooks` servlet, the container will check if this servlet is up-and-running. If not, the container will start it and call the servlet's method `init()` — even if you didn't override this method, it exists in the superclass `HttpServlet`.

Then the container calls the method `service()` of the servlet's superclass, which redirects the request to `doGet()`, `doPost()`, or similar `doXXX()`, passing the arguments `HttpServletRequest` and `HTTPServletResponse`. Your code can get the data entered by the user from the `HttpServletRequest` object with the method `getParameter()`.

After you get the parameter(s), process it in the business layer, which can be implemented either as POJOs talking to some data store or as an EJB. Return the result to the client by getting the reference to the `PrintWriter` object — it knows how to send text data to the user. For non-textual results use the class `OutputStream` instead of `PrintWriter`. Don't forget to set the content type by calling `setContentType()`. For example, if you are sending an object containing PDF content and want the browser to automatically open Acrobat Reader, call the function `Response .setContentType("application/pdf");`.

The servlet container controls when the servlet is loaded, and when its `init()`, `service()`, and `destroy()` methods are called. The method `destroy()` is called when a server administrator decides to unload the servlet, or the server is gracefully shutting down, or the server needs to free some resources. The following code retrieves the name of the book entered by the user and responds with the price of 65 dollars:

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
                    throws ServletException, IOException  {

 String title = req.getParameter("booktitle");;
 PrintWriter out = res.getWriter();
 res.setContentType("text/html");
 out.println("<html><body>");
 out.println("<h2>the book "+title+" costs only $65");
 out.println("<p>Please enter your credit card number");
 out.println("</body></html>");
}
```

HTTP GET AND POST REQUESTS

HTTP specification defines several methods for data exchange on the Web, but `Get` and `Post` are the most widely used ones. If you don't specify the method, `Get` is used by default. Because the HTML tag `<form>` in Listing 27-1 used `method=Get`, the `FindBooks` method `doGet()` will be invoked by the container. In this case the web browser will append the values entered in the form to the end of the URL

after the question mark. For example, if the user enters the word `Apollo` as a book title, the URL may look like this:

```
http://www.mybooks.com?booktitle=Apollo
```

If a form submits multiple values, the URL will include several key/value pairs after the question mark. With `Get` it's easy to copy and paste or bookmark the URL with parameters. On the other hand, with `Get` the data is not protected — you can see it in clear text.

The method `Post` is typically used to send larger amounts of data to the server. It's always used to send binary data to the server. Of course, the log-in forms should use `Post` because you don't want the user's ID and password to be shown in the URL. To process `Post` requests, servlets have to override the method `doPost()`.

It's common to use `Get` for data retrieval and `Post` for both data retrieval and modification.

SESSION TRACKING

HTTP is a *stateless* protocol. If a user retrieves a web page with a list of books from the `FindBooks` servlet or any other server-side program as a result of an HTTP request, and then goes to another web page, this second page does not know what was shown on the first one. To preserve data to more than one web page, *session tracking* has to be implemented.

A session is a logical task, which the user is trying to complete by visiting a website. For example, the process of buying a book may involve several steps: book selection, input of billing and shipping information, and so on. All of these steps combined to form an example of a session. When a purchase order is placed, the session is over. To create a session, call the method `request.getsSession(true)`. Calling this method without an argument will give you either a reference to the existing session or `null`.

The session information can be stored either on the client or on the server side. On the client side the user's session data can be stored using *cookies* (defined in the next section) or URL rewriting. The server-side alternative for storing session data is a session tracking API that implements the interface `javax.servlet.http.HTTPSession`.

Cookies

A cookie is a small piece of data that a container sends to the web client to be stored on the disk. On every subsequent request from that client, the cookie is returned to the server, uniquely associating the request with a given session. The cookies are persistent, but the user may disable them by selecting the appropriate option in the web browser. When the session is created, a special cookie, `JSESSIONID`, is sent to the client. Here's how the servlet can send a business-related cookie (`javax.servlet.http.Cookie`) to the client:

```
Cookie myCookie = new Cookie("bookName","Java Programming 24-hour trainer");
// Set the lifetime of the cookie for 24 hours
myCookie.setMaxAge(60*60*24);
response.addCookie(myCookie);
```

This is how a servlet can retrieve a client's cookies that arrive with `HttpServletRequest`:

```
Cookie[] cookies = request.getCookies();
```

```
for (i=0; i < cookies.length; i++){
  Cookie currentCookie = cookie[i];
  String name = currentCookie.getName();
  String value = currentCookie.getValue();
}
```

Even though you can store multiple cookies on the client, as in the preceding code, it's not a good idea to send the application data back and forth over the network. Typically the session data is stored in the `HTTPSession` object described later in the section "Server-Side HTTPSession."

URL Rewriting

If a client disables cookies, the URL rewriting may be used for session tracking. In this case the session ID and other required session data are attached to the URL string. If you've noticed something such as `jsessionid=12345` in the URL string of any website, it means that URL rewriting is being used.

Server-Side HttpSession

You should keep the data that belong to a user's session (such as the shopping cart) inside the `javax.servlet.http.HttpSession` object in the servlet container, which creates one such object per client. The servlet can store there any `Serializable` objects. The following line creates a session object (or finds a previously created one):

```
HttpSession mySession = request.getSession(true);
```

The call `getSession(true)` means "find this client's session object or create a new one if no object is found." Usually the shopping process consists of a number of subsequent servlet calls (list an inventory, add an item to the shopping cart, enter shipping information, etc.). The method call `getSession(true)` should be used in the very first servlet that opens the business process that can be called a user session. At this moment the application server generates a unique session ID and sends it to the user's web browser using either a cookie or URL rewriting.

The call `getSession(false)` means "find my session object," assuming that it has been created in the previous steps of the session. If this call returns `null`, the session object has been destroyed and you might want to display a message saying that the session has expired, and the user has to start the process from scratch. Let's assume that the following class represents a book on the server side:

```
class Book  {
   String title;
   double price;
}
```

This is what you can do in the method `doGet()` of the servlet to create a session and a shopping cart:

```
HttpSession session = request.getSession(true);

// This sample uses Vector object here to store selected books.
// Try to get the shopping cart that might have been
// created during previous calls to this servlet.

Vector myShoppingCart=session.getAttribute("shoppingCart");

if (myShoppingCart == null){
```

```
    // This is the first call - instantiate the shopping cart
     myShoppingCart = new Vector();
}

// create an instance of a book object
Book selectedBook = new Book();

selectedBook.title=request.getParameter("booktitle");
selectedBook.price= Double.parseDouble(request.getParameter("price"));

// Add the book to our shopping cart
myShoppingCart.addElement(selectedBook);

// Put the shopping cart back into the session object
session.setAttribute("shoppingCart", myShoppingCart);
```

When the book order has been placed the program should close the session by making the following call:

```
session.invalidate();
```

If the session has not been invalidated explicitly, the application server will do it automatically after a specific period. This period is called *timeout* and is a configurable parameter.

Filters

Even after a servlet is deployed you still can change the way it processes requests and responses without modifying the servlet's code. You can create *filters* — code that can be configured to process HTTP requests before they are passed to the servlet, or right after the servlet returns the request. Filters are good for adding such functionality as authentication, logging, encryption, data compression, and audit reporting.

None of these actions depend on the business logic implemented by servlets. Besides, why write code with compressed logic over and over again in each servlet if you can create one filter and apply it to all of them before the data is sent to the client? What if you need to compress and encrypt? Write two filters and chain them so both are applied to the same servlet.

To create a filter, write a class that implements the interface `javax.servlet.Filter` annotated with `@WebFilter`. There are three methods in this interface: `doFilter()`, `init()`, and `destroy()`. Here's a sample of a filter class to be used with two servlets, `FindBooks` and `RemoveBook`:

```
@WebFilter(servletNames={"/FindBooks", "/RemoveBook"})
public class MyAuthenticationFilter implements Filter {

      @Override
      public void doFilter(ServletRequest request, ServletResponse response,
                  FilterChain chain)throws IOException, ServletException {

        // user authentication code goes here

       //Call the next filter, if any
       chain.doFilter(request, response);

      }
```

```
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {}

    @Override
    public void destroy() {}

}
```

The container gives the filter both request and response objects. You can check passed parameters by the client (such as by checking ID/password), perform authentication, and, if the user is not valid, call `response.getWriter()` and send the user the message "You don't belong here" without even passing control to the servlet.

The method `destroy()` is called once before the container removes the filter; if the filter has created some resources such as DBMS connections, close them in the filter's code. The method `init()` is invoked on the filter only once during its instantiation. The filter gets the `FilterConfig` object, which gives you access to the servlet context and initialization parameters if these are specified in the `@WebFilter` annotation or in the file *web.xml*:

```
@WebFilter(servletNames={"/FindBooks", "/RemoveBook"},
           initParams={name="community", value="adults"})
```

EVENT LISTENERS

Several important events happen to a servlet during its life cycle. You can write code reacting to these events in custom event listener classes. For example, the container fires events when the servlet context is created, and if you are interested in processing this event, create a class that implements the `ServletContextListener` interface. To catch the moment when the session has been created, invalidated, or timed out, use `HttpSessionListener`. To catch the beginning of the processing of the servlet request, create a class implementing `ServletRequestListener`.

Imagine that you need to gather statistics on the usage of the bookstore servlet and log the date and time when each user's session was created and destroyed. Listing 27-4 prints the session ID on creation and invalidation of every user's session.



LISTING 27-4: HttpSession listener

```
import javax.servlet.http.HttpSessionListener;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSession;
import javax.servlet.ServletConfig;
import javax.servlet.annotation.WebListener;

@WebListener
public class BookStoreSessionListener implements HttpSessionListener{

    public void init(ServletConfig config){
```

continues

LISTING 27-4 *(continued)*

```
    }

    public void sessionCreated(HttpSessionEvent event){
      HttpSession    session = event.getSession();
      System.out.println("BookStore session created; id:"+ session.getId());
    }

    public void sessionDestroyed(HttpSessionEvent event){
      HttpSession    session = event.getSession();
      System.out.println("BookStore session destroyed; id:"+ session.getId());
    }
}
```

To test this listener, modify the code of `doGet()` in the servlet `FindBooks` from Listing 27-4 to create and destroy the session. (In more realistic scenarios the session creation and destruction would be performed in different methods.)

```
      protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
            // Get or create the session
            request.getSession(true);

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

            //Destroy the session
            request.getSession(true).invalidate();
      }
```

To emulate the multiple-users scenario, make the first request to the `FindBooks` servlet from Eclipse IDE, and another one from the web browser by copying the URL `http://localhost:8080/Lesson27/books` into the address bar. You should see the creation and destruction of two sessions with different IDs in the Eclipse Console view, which runs the server:

```
INFO: BookStore session created; id:104903c1dd6a8306cb33a116257a
INFO: BookStore session destroyed; id:104903c1dd6a8306cb33a116257a

INFO: BookStore session created; id:104f16ba2e714bdd8981c7ea2e87
INFO: BookStore session destroyed; id:104f16ba2e714bdd8981c7ea2e87
```

ASYNCHRONOUS SERVLETS

It's great that servlets automatically create and allocate a separate thread for each user's request, but each thread takes up system resources (both memory bytes and CPU cycles), and after a certain number of concurrent requests the server will simply stop responding. Imagine if thousands of users simultaneously hit the `FindBooks` servlet that has to perform a three-second DBMS search for each request. During these three seconds the container will hold the lock on each thread, doing nothing but waiting for the result of the DBMS query (running on another server!).

The idea of asynchronous servlets is to minimize the time of thread locking. If User A makes a request that takes three seconds on a DBMS server, the container's thread will be given to the request of User B, and, when User A's result comes back from the DBMS, the container will allocate this (or another) thread to return the result to User A. This architecture can substantially increase the number of concurrent requests that can be processed on the same server.

In the past, vendors of servlet containers offered their proprietary solutions for asynchronous processing. Jetty implemented its version of reusing thread via a concept called *continuations*, whereby the thread given to a request is suspended for the time required for the business logic to complete, and then is resumed when the result is ready. Apache Tomcat and GlassFish implemented technology based on the event-based model called Comet, whereby an HTTP connection to the servlet container is maintained, and the server *pushes* the data to the client via this connection.

The Servlets 3.0 specification has introduced a standard, Comet-based way of creating asynchronous Java EE web applications. In `doGet()` or `doPost()` you can instantiate the object `javax.servlet.AsyncContext`, which creates an asynchronous worker thread and doesn't lock the client's thread while preserving the client's request and response objects. Asynchronous processing is out of the scope of this tutorial, but you can find a detailed example of programming an asynchronous servlet at `http://bit.ly/b0iBcr`.

TRY IT

Write a simple HTML client with one text input field that has a Submit button. The user will enter the stock symbol she wants to get a price quote for. Generate a random quote and return a web page with the quote. Reuse the code from `StockServerImpl` from Listing 25-2 to generate the price quotes.

Lesson Requirements

You should have Java 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 Lesson27 folder in the download.*

Hints

You can find several examples of Servlet 3.0 web applications at `www.oracle.com/technetwork/java/javaee/documentation/code-139018.html`.

Step-by-Step

1. Create an HTML client similar to the one from Listing 27-1 to allow the user to enter the stock symbol.
2. Create a servlet called `StockServerServlet` that will take one parameter, the stock symbol, and instantiate a class called `StockQuoteGenerator` that should have the code similar to Listing 25-2. There's no need to implement Remote interface though.

3. Pass the stock symbol received from the client to `StockQuoteGenerator` and get the price.
4. Return the dynamically created HTML page to the client via the request object.
5. Test the servlet in Eclipse IDE first, and then create the war file and deploy it in the document root of GlassFish Server, installed in Lesson 26.
6. Start GlassFish Server.
7. Open the HTML file you created in Step 1 in your web browser and test the servlet by entering various stock symbols.



Please select Lesson 27 on the DVD with the print book, or watch online at `www.wrox.com/go/fainjava` *to view the video that accompanies this lesson.*

28

JavaServer Pages

JavaServer Pages (JSP) was created as a next step in servlets' evolution. JSP 2.2 is part of the Java EE 6 specification, and with it you can do everything that you can do with servlets, but more easily. Let's say you've created and deployed a servlet, which displays "Hello World." The servlet gets a hold of the output stream of the response object and executes the following line of code:

```
out.println("<html><body>Hello World </body></html>");
```

Let's say you run a small software company that employs Alex, an expensive Java developer, and Matilda, a junior Web designer who doesn't know Java but does know HTML. What if you need to change the layout of this HTML page, such as by adding several empty lines on top? It's not a big problem — Alex can modify the preceding line of code, recompile it, and redeploy the servlet. But for making small changes in the HTML-based UI it's more efficient to use Matilda. This is where JSP becomes very handy. Ask Matilda to create the following text file, *HelloWorld.jsp*:

```
<html>
 <body>
   Hello World
 </body>
</html>
```

Place this file into the document root directory (see Lesson 27) in your servlet container running at, say, `MyBooks.com`. Now the users can access this JSP by entering the following URL in their web browsers:

```
http://www.MyBooks.com/HelloWorld.jsp
```

Upon the first request to this page, the JSP container (vendors of servlet containers support JSP, too) will automatically generate, compile, and deploy a servlet based on the content of the file *HelloWorld.jsp*. All subsequent calls to this JSP will be processed a lot faster, because the servlet `HelloWorld` will already be deployed, loaded in memory, and running. As a matter of fact, JSP, as well as servlets, can be preloaded so that even the first user's request is responsive.

You might think that you could make a simple web page by creating *HelloWorld.html* without all this extra code generation. This is true, as long as your page is static and does not use any external dynamically changed data. Remember that HTML is not a programming language but a markup language — it can't even add two and two, but JSP can (see *MyCalculator.jsp* in Listing 28-1).

EMBEDDING JAVA CODE INTO HTML

JSP defines tags that enable you to embed Java code into an HTML page. When the servlet is automatically generated behind the scenes, this Java code will also be included and executed as part of this servlet. JSP tags are included in angle brackets: for example, the `<%=. . .%>` tag displays the value of the variable or expression:`<%=2+2%>`. During the servlet generation process performed by the JSP engine, these tags will be replaced with the regular Java code. For example, the tag `<%=2+2%>` will automatically be replaced by a JSP container with the following Java statement:

```
out.println(2+2);
```

Listing 28-1 shows the content of the two-plus-two calculator called *MyCalculator.jsp*. `<BR>` is an HTML tag for inserting line breaks. Note that Alex, the programmer, had to write only the expression inside the JSP tag; the rest was done by Matilda. Consider this task separation an example of the designer-developer workflow.



LISTING 28-1: MyCalculator.jsp

```
<html>
<body>
   HTML created by Matilda goes here...
   <br>
   You may not know that 2 + 2 is  <%= 2 + 2%>
   <br>
   More   HTML created by Matilda goes here...
 </body>
</html>
```

Create a Dynamic Web Project in Eclipse called Lesson 28. It'll automatically create a file called *index.jsp* — a Hello World JSP under the *WebContent* directory with the content shown in Figure 28-1. Deploying any JSP is a simple matter of copying the JSP file into the document root directory of your JSP container. Of course, if your JSPs are part of a multi-file project, most likely you'll be deploying them in a war file, as described in Lesson 27.

Right-click the name *index.jsp* and select Run on Server and you'll see the Hello World web page. Make a copy of `index.jsp` named *MyCalculator.jsp* and replace the content in the `<body>` part with the content from the `<body>` section from Listing 28-1. Run it and you'll see the web page in Eclipse, as shown in Figure 28-2. The expression `<%2 + 2%>` has been precompiled and replaced with `4`. A JSP is nothing more than a servlet that is automatically generated from a file containing valid HTML and JSP tags.

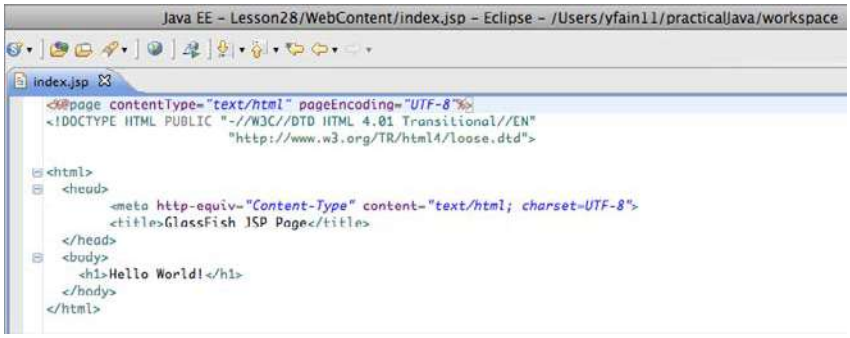


FIGURE 28-1

If you need to change the appearance of the page (colors, fonts, data allocation) without changing the expression (2+2), Matilda can do it easily! After the changes are applied, the JSP will be automatically regenerated into a new servlet and redeployed. Usually you do not even have to restart the server. The only exceptions are preloaded JSPs that are configured to be initialized on server startup. Any business logic changes inside the JSP tags will be programmed by Alex.

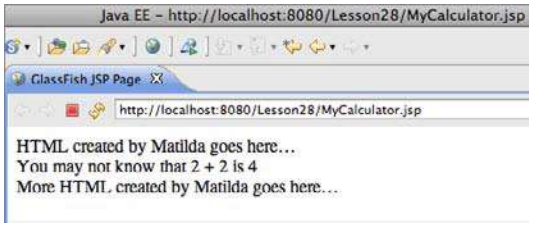


FIGURE 28-2

IMPLICIT JSP OBJECTS

Because JSPs are built on top of servlets, the main concepts remain the same. Following is the list of predefined variables that you can use in JSP pages. These variables are initialized by the JSP container, and you can use them without explicit declarations.

- `request` has the same use as `HttpServletRequest`.
- `response` has the same use as `HttpServletResponse`.
- `out` represents the output write stream `JspWriter`. This variable points at the same object as `HttpServletResponse.getWriter()` in servlets. For example, a simplest JSP that returns a result from a Java class called `CurrencyConverter` might look like this:

```
<html>
  <body>
    <% out.println(CurrencyConverter.getDollarRate()); %>
  </body>
</html>
```

- `session` represents an instance of the user's `HTTPSession` object.
- `exception` represents an instance of the `Throwable` object and contains error information. This variable is available only from the JSP error page (described later in the section "Error Pages").

- `page` represents the instance of the JSP's servlet.
- `pageContext` represents the JSP context and is used with Tag Libraries (described later in this lesson in the section of that name).
- `application` provides access to web context. Its use is similar to that of `ServletContext`.
- `config` provides initialization information used by the JSP container. Its use is similar to that of the class `ServletConfig`.

OVERVIEW OF THE JSP TAGS

Each JSP tag starts and ends with an angle bracket and can be included in any HTML file, but you need to save them as `.jsp` files for proper identification and processing by JSP containers. This section is a brief overview of JSP tags. For more detailed coverage refer to the JSP documentation at `http://java.sun.com/products/jsp/docs.html`.

Directives

Directives do not generate screen output, but instruct the JSP container about the rules that have to be applied to the JSP. Some of the JSP directives are `page`, `include`, `attribute`, and `taglib`.

Page directives start with `<%@ page` and are only in effect within the current page. Directives are used with such attributes as `import`, `extends`, `session`, `errorPage`, `contentType`, and some others. For example, to import the `java.io` package use this directive:

```
<%@ page import="java.io.*" %>
```

Include directives to allow the inclusion of any text from a file or any code from another JSP, at the time when the page is compiled into a servlet, as in the following example:

```
<%@ jsp:include page="calcBankRates.jsp" %>
<%@ include file="bankRates.txt" %>
```

To use a third-party library of custom tags you need to specify where this library is located, as in the following example:

```
<%@ taglib uri="my_taglib.tld" prefix="test" %>
```

The `attribute` directive enables you to define attributes of custom tags, like this:

```
<%@ attribute name="corporate_logo_file_name" %>
```

Declarations

Declarations are used to declare variables before they are used. You can declare a variable salary like this:

```
<%! double salary; %>
```

The variable salary is visible only inside this page. You can declare Java methods in the JSP page the same way:

```
<%! private  void myMethod(){
        ...
 }%>
```

The code contained in the declaration block turns into the Java code in the generated servlet.

Expressions

Expressions start with `<%=` and can contain any Java expression, which will be evaluated. The result will be displayed in the HTML page, replacing the tag itself, like this:

```
<%= salary*1.2 %>
```

Scriptlets

Initially scriptlets were created to give JSP developers a place to put any valid Java code to be included in the generated servlet's `_jspService()` method, which is an equivalent of the servlet's method `service()`, for example a scriptlet can look like this:

```
<% lastName = "Smith"; %>
```

With the creation of JSP Standard Tag Libraries (JSTL) and Expression Language (EL) there is now no need to use scriptlets, but this syntax still works.

Comments

Comments that start with `<%--` and end with `--%>` will be visible in the JSP source code, but will not be included in the resulting HTML page.

```
<%-- Some comments --%>
```

If you'd like to keep the comments in the resulting web page, use regular HTML comment notation:

```
<!-- Some comments -->
```

Standard Actions

Although the directive `include` adds the content of the included page during compile time, the element `jsp:include` does it during run time.

```
<jsp:include page "header.jsp" />
```

The element `forward` enables you to redirect the program flow from the current JSP to another one while preserving the `request` and `response` objects. The other way of redirecting the flow is to use `response.sendRedirect(someURL)`, but in this case new `request` and `response` objects are created, which results in a client request to the server.

```
<jsp:forward page = "someOther.jsp" />
```

Page 349 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

The `plugin` element ensures that your JSP includes an applet or a Java bean (described later in the "JavaBeans" section). During run time the web browser replaces this tag with one of the HTML tags: `<object>` or `<embed>`.

```
<jsp:plugin type=applet code="PriceQuotes.class" >
```

The nested tag `<jsp:param>` is used to pass parameters to an applet or a bean:

```
<jsp:plugin type=applet code="Login.class">
  <jsp:params>
    <jsp:param name="userID" value="SCOTT" />
    <jsp:param name="password" value="TIGER" />
  </jsp:params>
</jsp:plugin>
```

ERROR PAGES

Let's say we have a file called *`calcTax.jsp`* containing code that may throw Java exceptions. Instead of scaring users with stack trace output screens, prepare a friendly *`taxErrors.jsp`* explaining the problem in plain English.

`calcTax.jsp` may have an HTML `<form>` in which the user enters gross income and the number of dependents. The request for tax calculations is sent to the server's `CalcTax` Java class, which might throw an exception during its processing. Include the name of the error page that has to be shown in case of exceptions:

```
<html>
  Some code to calculate tax and other HTML stuff goes here
        ...
     <%@ page errorPage=taxErrors.jsp %>

</html>
```

Next comes the error page *`taxErrors.jsp`*, which illustrates how to use the JSP variable `exception`, which displays the error message in a user-friendly way, and also contains more technical error description for the technical support team:

```
<html>
 <body>
  Unfortunately there was a problem during your tax calculations. We are
  working on this issue - please try again in 10 minutes.
  If the problem persists, please contact our award winning technical support
  team at (212) 555-2222 and provide them with the following information:
 <br>
 <%=exception.toString()>

 </body>
</html>
```

JAVABEANS

JavaBeans specification defines a bean as a Java class that implements the `Serializable` interface and that has a public no-argument constructor, private fields, and public setter and getter methods. The similar concept of Data Transfer Objects (DTOs) was introduced in Lesson 22 (see Listing 22-2). Java beans are used mainly for data storing and exchange. In JSP they are used to avoid mixing Java code and HTML (see also the section "Tag Libraries" later in this lesson; they are also helping to avoid mixing Java and HTML).

Think of the MVC pattern in JSP-based web applications. The JSP belongs to the view tier, the servlet can play a role of a controller, and the Java bean can represent a model. Although you can mix HTML with business logic coded in scriptlets, declarations, and expressions, you should avoid doing this and separate presentation from business logic processing and data storage. First, this will allow you to split the work more easily between Alex and Matilda, and second, you'll be able to have more than one presentation solution (e.g., a different UI for mobile devices) reusing the same Java code.

Using beans is the first step in separating processing logic and presentation. Listing 28-2 shows an example of a bean, called `Student`.



LISTING 28-2: Student bean

```
import java.io.Serializable;

class Student implements Serializable{
          private String lastName;
          private String firstName;
          private boolean undergraduate;

          Student(){
              // constructor's code goes here
          }

          public String getLastName(){
                return lastName;
          }
          public String getFirstName(){
                return firstName;
          }
          public void setLastName(String value){
                   lastName = value;
          }
          public void setFirstName (String value){
                   firstName = value;
          }
          public void setUndergraduate(boolean value){
                   undergraduate = value;
```

continues

LISTING 28-2 *(continued)*

```
        }

         public boolean isUndergraduate (){
                return undergraduate;
        }
}
```

Don't confuse JavaBeans with Enterprise Java Beans (EJB), which is a different concept and will be covered in Lesson 32.

Using JavaBeans in JSP

To use a bean with JSP, first you need to specify its name and location, and after that you can set or get its properties. Following are some examples of bean usage:

```
<jsp:useBean  id="Student" class="com.harward.Student" />
<jsp:getProperty name="Student" property="LastName" />
<jsp:setProperty name="Student" property="LastName" value="Smith"/>
```

The next code snippet populates the `Student` bean's properties `LastName` and `FirstName` based on the data from the HTML tag `<form>`, which has two HTML text input fields called `LName` and `FName`:

```
<jsp:setProperty name="Student" property="LastName" value=
                                    "<%= request.getParameter("LName") %>" />

<jsp:setProperty name="Student" property="FirstName" value=
                                     "<%=request.getParameter("FName") %>" />
```

If property names are the same in the HTML form and in the bean, mapping the HTML form's and the bean's fields becomes even simpler with the asterisk notation:

```
<jsp:setProperty name="Student" property="*" />
```

How Long Does a Bean Live?

If a JSP variable is declared inside a scriptlet, it has a local scope. To give it an instance scope, declare the variable using the declaration `tag`. You can define a bean's scope using the `scope` attribute of the tag `jsp:useBean`. The following list defines the various scopes.

- `page`: The bean is available only within the current page and will be destroyed as soon as the user exits the page. This is a default scope. For example:

  ```
  <jsp:useBean  id="Student" class="com.harward.Student" scope="page" />
  ```

- `request`: The bean is alive for as long as the `request` object is alive. Even if the control will be redirected to a different JSP by means of the tag `jsp:forward`, the bean will remain available on the new page because it'll be using the same request object, like this:

  ```
  <jsp:useBean  id="Student" class="com.harward.Student" scope="request" />
  ```

- `session`: The bean is available for all pages until the user's session ends (see the section "Session Tracking" in Lesson 27).

```
<jsp:useBean  id="Student" class="com.harward.Student" scope="session" />
```

- `application`: The bean is available for all users and all pages — this is a global bean.

```
<jsp:useBean  id="Student" class="com.harward.Student" scope="application" />
```

LOADING JSP FROM SERVLETS

In line with the separation of presentation and processing, JSP should have a bare minimum of any processing. When the servlet receives the data to be sent to the user, instead of sending to the client hard-coded HTML tags it should load and send the JSP page to the client. Ideally, the JSP should be laid out by a Web designer.

Let's say you have a servlet that needs to load a JSP based on the user's selection in the HTML window. If you don't need to get new copies of the `request` and `response` objects you'll need to create an instance of the `RequestDispatcher` class and call its method `forward()`, providing `HttpServletRequest` and `HttpServletResponse` as arguments, as shown in Listing 28-3. The servlet `MyServlet` returns to the Web browser either the JSP showing the data of the Toyota dealership or the JSP showing data about Nissan vehicles.



LISTING 28-3: Servlet loading JSP

```
public class MyServlet extends HttpServlet{
   public void doGet(HttpServletRequest req, HttpServletResponse res)
                                  throws ServletException {

     ServletContext context = getServletContext();
     RequestDispatcher requestDisp = null;

     String make = req.getParameter("carMake");

        if (make.equals("Toyota") {
          requestDisp = context.getRequestDispatcher("Toyota.jsp");
          requestDisp.forward(req,res);
        }
        else if (make.equals("Nissan") {
          requestDisp = context.getRequestDispatcher("Nissan.jsp");
          requestDisp.forward(req,res);
        }
  }
}
```

In some cases the current servlet performs all interactions with the user, and just needs to load the code of another servlet or JSP. For this purpose use the method `include()` instead of `forward()`:

```
requestDisp.include(req,res);
```

Because this redirection happens on the server side, the initial URL is still displayed in the web browser's address bar. To provide the new URL (to allow the user to bookmark the resulting page, for example), use `response.sendRedirect("/new_URL")`.

TAG LIBRARIES

Yet another way of minimizing the amount of code in JSP is to use tag libraries containing custom and reusable tags — either your own original library or a library created by someone else. Each custom tag looks similar to a regular one, but under the hood is always supported by a Java class (or classes) written by a programmer to provide the required functionality.

If you want to create your own custom tags to be used with JSP you have to do the following:

- Create a tag library descriptor — an XML file with the extension *.tld*. It has to be deployed in the directory *WEB-INF/tags*.
- Create Java classes that provide business logic supporting the tags. Such classes are usually deployed as jars in the *WEB-INF/lib* directory.
- Register the tag library with the web application.

Listing 28-4 shows a sample tag library descriptor file. The tag `DowJones` should display a Dow Jones index value. The empty value in `<bodycontent>` means that this is a simple JSP tag with no content and could be used like this: `<sts:DowJones/>`.



LISTING 28-4: Sample .tld file

```
<?xml version=" 1. 0" encoding="UTF-8" ?>
<taglib xmlns=" http: / / j ava. sun. com/xml/ns/ j 2ee"
  xmlns: xsi=" http: //www. w3. org/ 2 001/XMLSchema-instance"
  xsi: schemaLocation=" http: //java. sun. com/ xml/ns/j2 ee
  http: //java. sun. com/ xml/ns/j2 ee/web-jsptaglibrary_2 _0. xsd"
  version="2 . 0" >
  <tlib-version>1.0</ tlib-version>
  <shortname>sts</shortname>
  <uri>http://www.mystockserver.com:8080/taglib</uri>
  <info>Wall Street tag library</info>

  <tag>
    <name>DowJones</name>
    <tagclass>DowJonesTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info>Displays the Dow Jones index</info>
  </tag>

</taglib>
```

The class supporting a JSP tag (e.g. `DowJonesHandler`) has to implement the interface `javax.servlet.jsp.tagext.SimpleTag` or extend `SimpleTagSupport`. The JSP container will call

`DowJonesHandler`'s methods to set the JSP context — `setPageContext()`— start the execution of the tag's code — `doStartTag()`, etc. This class gives you a default implementation of the `SimpleTag` interface and initialized references to the `pageContext` and parent. Place required logic for the tag in the `doTag()` method, which is called by the container at request time.

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class DowJonesHandler extends SimpleTagSupport{

  public int doTag() throws JspException, IOException{

      String dowQuote;
     //  Obtain the DowJones quote by accessing
     //  http://finance.yahoo.com/q?d=t&s=^DJI or similar
        dowQuote=...;

     // and write it to the client
     JspWriter out = getJspContext().getOut();
     out.print("The last price is " + dowQuote);

 }
}
```

To make a tag library recognizable by your JSP container, register it by inserting the following fragment into the file *web.xml*:

```
<taglib>
     <taglib-uri>
           http://www.mystockserver.com/taglib
     </taglib-uri>
      <taglib-location>
            /WEB-INF/taglib.tld
      </taglib-location>
</taglib>
```

When you've done all this, create a simple file called *test.jsp* and start using your tag library. The sample JSP in Listing 28-5 uses the tag `<DowJones>`.



LISTING 28-5: Using a custom tag in a JSP

```
<html>
<head>
  <%@ taglib uri=http://www.mystockserver.com/taglib prefix="sts" %>
</head>
<body>
   Today's Dow Jones index: <sts:DowJones/>
</body>
</html>
```

If a tag requires some parameters, they should be specified in the *.tld* file with the tag `<attribute>`, for example:

```
<tag>
  ...
  <attribute>
    <name>tradeDate</name>
    <required>false</required>
  </attribute>
</tag>
```

The setter method has to be provided in the tag handler class for each parameter. Setters have to be named according to the same naming convention as Java beans:

```
public void setTradeDate(String tradeDate){
...
}
```

Custom tag libraries are created by application developers to fit the needs of a specific project(s). Third parties can also provide non-standards-based tag libraries, as with the open source implementation of JSP tag libraries in the project Apache Taglibs (`http://tomcat.apache.org/taglibs/`).

JSTL

JSP Standard Tag Libraries (JSTL) is a standardized specification for library components that includes actions that are reusable for many JSP-based applications. Standard JSTL guarantees that any Java EE–compliant JSP container will include and support these components. There are five JSTL libraries. They contain an iterator, `if` statements, tags for XML processing, tags for executing SQL, tags for internationalization and commonly used functions.

While the JSP in Listing 28-3 had to specify the location of the tag library on your server, standardized libraries have predefined URLs. For example, to use the `forEach` iterator you'd need to specify the following URI: `http://java.sun.com/jsp/jstl/core`. XML processing tags are located at the following URI: `http://java.sun.com/jsp/jstl/xml`. Accordingly, the code fragment that uses the iterator can look like this:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<c:forEach var="item" items="${sessionScope.cart.items}">
    ...
</c:forEach>
```

Learning programming with JSTL and Expression Language is out of the scope of this tutorial; please refer to the Oracle tutorial at the following URL: `http://download.oracle.com/javaee/5/tutorial/doc/bnake.html`.

TRY IT

Rewrite the sample stock server application that you created in the "Try It" section of Lesson 27, but this time do it using JSP and Java beans. Write a simple HTML client with one text input field and a Submit button. The user will enter the stock symbol she wants to get a price quote for. Generate

a random quote and return a web page with the quote. Reuse the code of `StockServerImpl` from Listing 25-2 for generating the price quotes.

Lesson Requirements

You should have Java 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 Lesson28 folder in the download.*

Step-by-Step

1. In the Eclipse project Lesson28 select the option File ➪ New. Create a new JSP file named *`StockQuote.jsp`* to allow the user to enter a stock symbol and request a price quote. Eclipse will create a file that looks like this:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>Insert title here</title>
  </head>
  <body>

  </body>
</html>
```

2. Create a Java class called `StockServer` to contain a method with one parameter — the stock symbol — and will calculate the price quote for the specified stock with code similar to what is shown in Listing 25-2. There's no need to implement Remote interface though.
3. Use required JSP tags to include the `StockServer` Java class in *`StockQuote.jsp`*, and display the price quote generated by its method `getQuote()` as well as the list of all available stock symbols by calling `getNasdaqSymbols()`.
4. Add an HTML `<form>` tag to *`StockQuote.jsp`* with one text input field and a Submit button so the user can enter the stock symbol and request the price quote.
5. Run and test *`StockQuote.jsp`* from Eclipse.



Please select Lesson 28 on the DVD with the print book, or watch online at `www.wrox.com/go/fainjava` *to view the video that accompanies this lesson.*

29

Developing Web Applications with JSF

In the previous lesson I mentioned that the JSP specification was created on top of servlets to simplify Web application development and as a step toward the separation of responsibilities between web designers and developers. I also said that you should minimize the use of Java code in a JSP page. But JSP doesn't enforce this practice. JavaServer Faces (JSF) does, and using it is the best way of developing Java EE web applications with a thin client.

The current version, JSF 2.0, is based on the Facelets framework that was originally an open-source product. You'll be creating JSF web pages in extensible HTML (`.xhtml` files), which means that the HTML has to be well-formed — every open tag should have a closing one, the values for the tags' attributes must be placed in quotes, the elements tags must respect case-sensitivity, and the document must have a single root tag.

Like JSP, the JSF framework is tag-based. But JavaServer Faces has several important advantages over JSP:

- The JSF framework comes with a set of UI components, while JSP uses standard HTML components.
- JSF is event-driven.
- You can't add Java to `.xhtml` files, which enforces clean code separation between Java and HTML.
- JSF offers easy navigation between web pages.
- JSF introduces configurable managed beans, which contain processing logic and navigation.

I'll give you more details on each of these bullet points later in this lesson.

THE BIG PICTURE

The JSF framework is a part of the presentation tier and it enforces the separation of presentation and processing logic. Reiterating the MVC paradigm, the library of JSF components is responsible for the UI, the servlet `FacesServlet` plays the role of a controller, and managed beans are the model. The processing logic can be placed in POJO classes, in EJB, or in third-party systems, as shown in Figure 29-1.

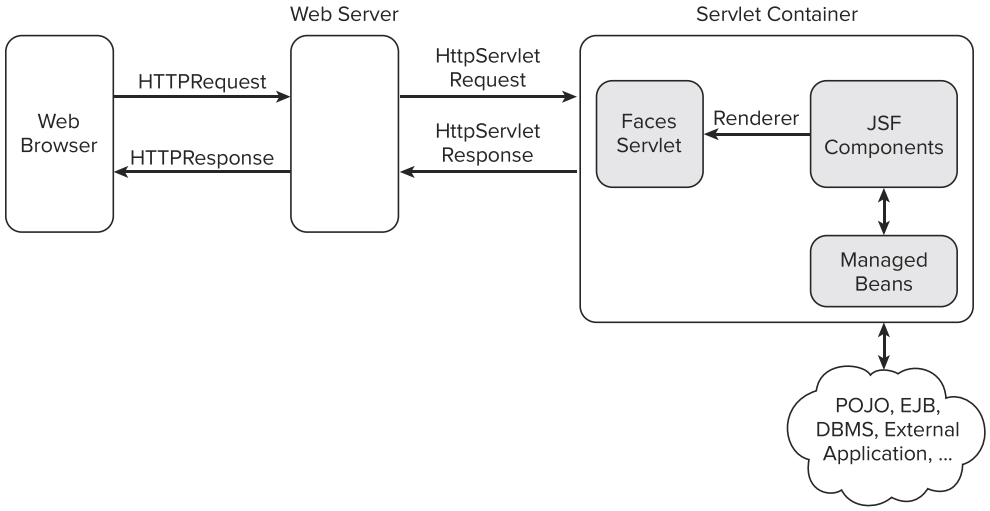


FIGURE 29-1

The managed beans can be linked to the UI JSF components so that when, for example, you fill out a form on a web page, the entered values are automatically assigned to the corresponding field in your Java bean. Sometimes beans that support UI components are called baking beans.

Without further ado, let's start working on a two-page web application to enroll students into classes. The first page will enable users to enter student ID, name, and course, and the second page will confirm the enrollment. The UI will be supported by managed beans, which will serve as a data model, but the data won't be saved (persisted) anywhere (you'll learn about persistence in Lesson 33).

MANAGED BEANS

Managed JSF beans are similar to JavaBeans, introduced in Lesson 28, but the word *managed* means that they can be linked directly to JSF components. Besides storing data they may also contain information about page navigation. `FacesServlet` controls the managed beans.

Let's start working on the student enrollment example by creating a managed bean called `Student`. But first create a new dynamic web project called Lesson29 in Eclipse IDE.

Then create the Java class `Student` in the folder `com.practicaljava.Lesson29.beans` and add the annotation `@ManagedBean` right above the class name. Eclipse will mark this line with a red *x*, complaining about the missing import statement. Right-click this *x*, select the option Quick Fix, and import this annotation class from the `javax.faces.bean` package. The annotation registers the class `Student` as a JSF resource. Add the `@RequestScoped` annotation to ensure that the container creates a new instance of this bean for each user's request.

Declare three private variables: `int studentId`, `String name`, and `long classId`. Then select Source ➪ Generate Getter and Setters and the rest of the `Student` code will be auto-generated. It will look like Listing 29-1.



LISTING 29-1: Managed bean Student

```
package com.practicaljava.lesson29.beans;

import java.io.Serializable;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class Student implements Serializable{

      private long studentId;
      private String name;
       private int classId;

       public long getStudentId() {
            return studentId;
      }
      public void setStudentId(long studentId) {
            this.studentId = studentId;
      }
      public String getName() {
            return name;
      }
      public void setName(String name) {
            this.name = name;
      }
      public int getClassId() {
            return classId;
      }
      public void setClassId(int classId) {
            this.classId = classId;
      }
}
```

Configuration of the managed beans is done using annotations — no additional configuration files are required. But if you specify the file *`faces-config.xml`* in the `WEB-INF` directory, its setting will override the annotation values.

CREATING A JSF WEBSITE

The content of the baking bean `Student` will have to be populated from the UI. With JSF you can either select the components that you want to put on each web page á la carte, or use a more structured and efficient approach called templating. I'll explain the creation of JSF sites with templating.

Templating

The layout of most websites is similar: They have headers, footers, left and/or right sidebars, and content areas. This was taken into consideration by the designers of Facelets, who introduced templating. For example, you can define one template for a site that contains a header and a footer, and just change the views displayed in the content area based on the user's actions. This is what templating is all about.

You start developing the website by creating an `.xhtml` file containing the main layout, and then add a number of views, or *composition pages*. Each of these composition pages will declare which template it has to use, define the required components for the content area, and either reuse the headers and footers from the template or override them. You'll find the methodology of creating pages easy to understand when you create a sample template and composite page.

In Eclipse select File ➪ New ➪ XHTML page, and accept the default location, the *`WebContent`* directory. Enter `StudentTemplate` as the name of the file and click Next. Then select New Facelet Template in the pop-up window shown in Figure 29-2.

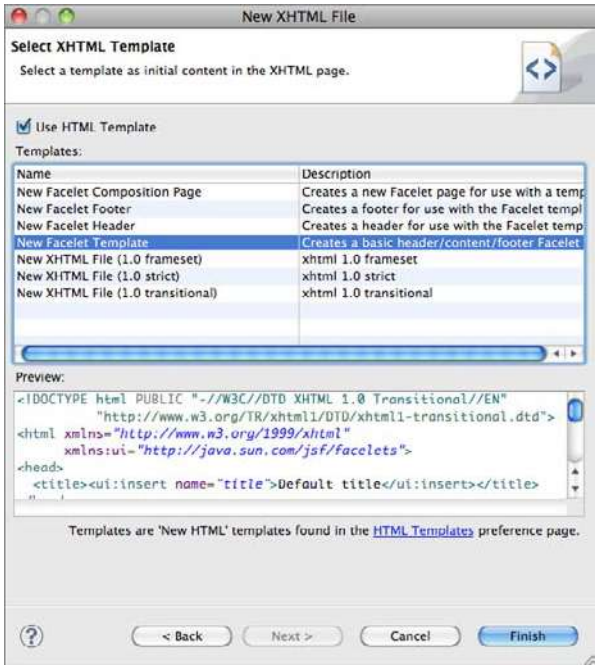


FIGURE 29-2

This will create a basic webpage template with three sections — header, content, and footer — as shown in Listing 29-2.



LISTING 29-2: Facelet template StudentTemplate.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
  <title><ui:insert name="title">Default title</ui:insert></title>
</head>

<body>

<div id="header">
    <ui:insert name="header">
          Header area.  See comments below this line in the source.
          <!--  include your header file or uncomment the include below
                and create header.xhtml in this directory -->
          <!-- <ui:include src="header.xhtml"/> -->
    </ui:insert>
</div>

<div id="content">
  <ui:insert name="content">
          Content area.  See comments below this line in the source.
          <!--  include your content file or uncomment the include below
                 and create content.xhtml in this directory -->
<!-- <div> -->
          <!-- <ui:include src="content.xhtml"/> -->
          <!-- </div> -->
  </ui:insert>
</div>

<div id="footer">
  <ui:insert name="footer">
          Footer area.  See comments below this line in the source.
          <!--  include your header file or uncomment the include below and create
footer.xhtml in this directory -->
      <!--<ui:include src="footer.xhtml"/>  -->
  </ui:insert>
</div>

</body>

</html>
```

The three sections enclosed in the HTML `<div>` tags suggest that the web pages that will be using this template might have a header, a content area, and the footer. If you don't need this page layout just delete the appropriate `<div>`. If you need a sidebar create a `<div>` section with the ID `sidebarleft` or `sidebarright`. Notice the `ui` prefix inside the tags: It points at the *namespace* where all the Facelets

components reside. The namespace concept (see `xmlns` in the code) comes from XML. You can use any prefix instead of `ui`, but it should uniquely identify the location of the components referred to by the prefix. Namespaces are defined in the top portion of the template in Listing 29-2. You can have more than one namespace defined in your JSF page. The `h` prefix is typically used for HTML JSF components, and `f` for the core ones:

```
xmlns:ui=http://java.sun.com/jsf/facelets
xmlns:h=http://java.sun.com/jsf/html
xmlns:f="http://java.sun.com/jsf/core"
```

I'll make a slight text edit to the code in order to have the header and the footer displayed in a large font — note the HTML tag `<h1>`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
  <title><ui:insert name="title">Student Portal</ui:insert></title>
</head>

<body>

<div id="header">
    <ui:insert name="header">
          <h1>Welcome to my Students Portal</h1>
    </ui:insert>
</div>

<div id="content">
  <ui:insert name="content">
          The content of my student portal goes here  </ui:insert>
</div>

<div id="footer">
  <ui:insert name="footer">
          <h1>The footer of my portal is here</h1>
  </ui:insert>
</div>

</body>
</html>
```

You can run and test this template in Eclipse: Right-click *`StudentTemplate.xhtml`* and select Run as ➪ Run on Server. If the GlassFish server is not running, Eclipse will start it for you and deploy the Lesson29 web application. Figure 29-3 shows how the internal Eclipse web browser will show this page. You can also enter the URL `http://localhost:8080/Lesson29/StudentTemplate.xhtml` into a regular web browser — the output will be the same.

I want to keep this well-designed header and footer on each page of my student portal that has views with changing content in the middle. Let's create a new file called *`EnrollStudent.xhtml`*, but this time select New Facelets Composition Page in the pop-up window shown in Figure 29-2. Eclipse will create the file shown in Listing 29-3.

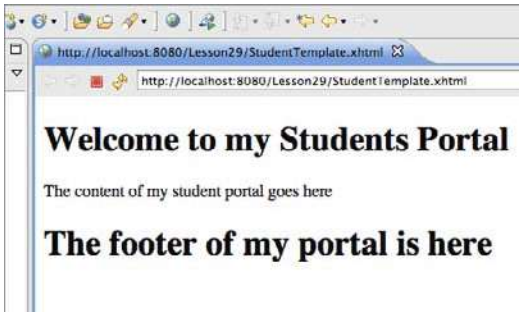


FIGURE 29-3

Available for download on Wrox.com

LISTING 29-3: Default composition page in Eclipse

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

<ui:composition template="">
      <ui:define name="header">
          Add your header here or delete to use the default
      </ui:define>
      <ui:define name="content">
          Add your content here or delete to use the default
      </ui:define>
      <ui:define name="footer">
          Add your footer here or delete to use the default
      </ui:define>
</ui:composition>
</html>
```

Note that this Facelet also has three sections, like the template in Listing 29-2. To link this page to a template, add *`StudentTemplate.xhtml`* to the `template` attribute. To reuse the header and the footer from the template, just remove the Facelets tag `<ui:define>` for the header and footer, as in Listing 29-4. When the JSF implementation can't find the definitions for these sections, it'll use the ones from the template.

Available for download on Wrox.com

LISTING 29-4: EnrollStudent.xhtml, first version

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
```

continues

LISTING 29-4 *(continued)*

```
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

<ui:composition template="StudentTemplate.xhtml">
      <ui:define name="content">
          Add your content here or delete to use the default
      </ui:define>
</ui:composition>
</html>
```

Now run this page in Eclipse or in your web browser, but add `/faces` to the URL, like this: `http://localhost:8080/Lesson29/faces/EnrollStudent.xhtml`. By default this will engage `FacesServlet`, which knows how to properly interpret all JFS tags. Figure 29-4 shows the first version of the Enroll Student page.



FIGURE 29-4

To complete the Enroll Student page, let's add several JSF tags: `form`, `inputText`, `outputText`, and `commandButton`. Listing 29-5 shows the updated version of *`EnrollStudent.xhtml`*. Note that the `value` attribute of each `inputText` tag is bound to the corresponding field of the `student` bean. The first letter of the bean name starts with a lowercase letter (`student`, not `Student`). The `<br/>` tag is just an HTML tag indicating a line break.



LISTING 29-5: EnrollStudent.xhtml, second version

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

<ui:composition template="StudentTemplate.xhtml">
 <ui:define name="content">
  <h:form>

    <h:outputText value="Student Name:"/>
```

```
        <h:inputText  value="#{student.name}" title="name" id="name" /> <br/>

        <h:outputText value="Student ID:"/>
        <h:inputText value="#{student.studentId}" title="id" id="studentId"/> <br/>

        <h:outputText value="Class ID:"/>
        <h:inputText  value="#{student.Id}" title="classId" id="classId" /> <br/>
        <h:commandButton action="enrolled" value="submit" />

      </h:form>
    </ui:define>
  </ui:composition>
  </html>
```

Run `EnrollStudent.html` and you'll see the great-looking portal in Figure 29-5. Just to prove that the UI is really bound to the bean `Student`, I've hard-coded my name in this class. That's why you see it in the form. The Student ID and Class ID fields show zeros — default values for numbers.



FIGURE 29-5

Page Navigation

The command button in Listing 29-5 has the attribute `action="enrolled"`. This means that after this form is processed, the web application has to navigate to another JSP called `Enrolled`. Listing 29-6 shows this page.



LISTING 29-6: Enrolled.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
```

continues

LISTING 29-6 *(continued)*

```
      xmlns:f="http://java.sun.com/jsf/core">

<ui:composition template="StudentTemplate.xhtml">
  <ui:define name="content">

   <h:outputText value="#{student.name}"/>, thank you for enrolling in class
   <h:outputText value="#{student.classId}" />

  </ui:define>

</ui:composition>
</html>
```

It's a pretty simple way of arranging page navigation, isn't it? I've enrolled myself into class 123 in *EnrollStudent.xhtml*. After pressing Submit, I got the confirmation shown in Figure 29-6.

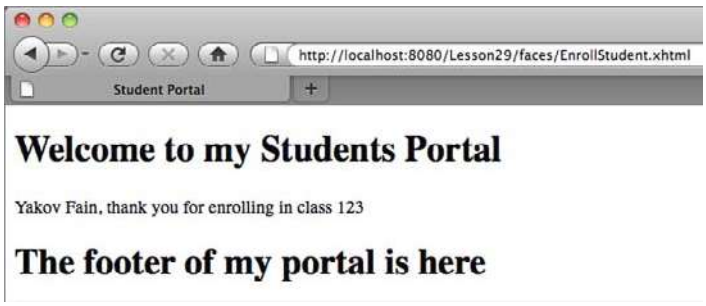


FIGURE 29-6

The `commandButton` button can be used for a different purpose: to call a method on a bean. You can write a bean just to serve as a data model, but nothing prevents you from writing a method that performs some business logic or performs other actions, such as getting data from a web service or saving data using EJBs. In the next code snippet the click on the button will call the method `doSomething()` on the bean `Student`.

```
<h:commandButton value="Do Something" action="#{student.doSomething}" />
```

The method `doSomething()` can return a `String` containing the name of the next JSF page to be called, just like this:

```
class Student{
...
  public String doSomething(){
       // Do something and navigate to the next page
      return "TheNextPage.xhtml";
  }
}
```

To become familiar with other tags available in Facelets, refer to Chapter 5 of the Java EE 6 tutorial, available at `http://download.oracle.com/javaee/6/tutorial/doc/giepx.html`.

CONVERTERS AND VALIDATORS

Managed beans can have *converters* and *validators* assigned to them. A Java class has strongly typed fields, but when the data arrives at the web page it has to be converted into `String`s. JSF performs this conversion automatically for Java data types, but you can write your own converter and add some application-specific logic.

Validators enable you to put some constraints on the data in the managed beans. You can either use predefined annotations such as `@NotNull`, `@Min`, and `@Max` from `javax.validation.constraints` to specify the data validation constraints on the bean's fields, or create your own custom annotations that implement constraints specific to your application. The standard validators on the fields of the `Student` bean may look like this:

```
@ManagedBean
@RequestScoped
public class Student implements Serializable{

      @NotNull(message="you must enter Student ID"
       @Size(min=8, max=10,
                 message="Student ID must include between 8 and 10 symbols")
       private long studentId;

       @Size(min=2, message="Name can have less than two characters")
       private String name;

       @NotNull(message="Please enter Class ID")
       private int classId;
       ...
}
```

The core JSF library has standard validators that can validate length, and ranges for long and double values. You encapsulate these validators right in the JSF file, like this:

```
<h:inputText  value="#{student.name}" title="name" id="name" >
   <f:validateLength minimum="2"/>
</h:inputText>
```

To create a custom validator (such as `CreditCardValidator`) you need to write a Java class that implements the `javax.faces.validator.Validator` interface and performs the validation in the method `validate()`. If the value is not valid, the method has to throw a `ValidatorException`. Custom validators must be registered in *`faces-config.xml`*.

```
<validator>
    <validator-id>CreditCardValidator</validator-id>
    <validator-class>myvalidators.CreditCardValidator</validator-class>
</validator>
```

Custom validators can be used in a JSF page with the `<f: validator>` tag, as in the following example:

```
<h:inputText  value="#{student.creditcad}" title="card" id="card" >
   <f:validator id="CreditCardValidator"/>
</h:inputText>
```

TRY IT

Rewrite the sample stock server application that you created in the "Try It" section of Lesson 28, but this time do it using JSF. The JSF client should have one text input field with a Submit button. The user will enter the stock symbol she wants to get a price quote for. Generate a random quote and return a web page with the quote. Reuse the code for `StockServerImpl` from Listing 25-2 to generate the price quotes. The web page should show the list of available stocks.

The managed bean will have a method called `getQuotes()` that should be called when the user clicks the Submit button.

Lesson Requirements

You should have Java 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 Lesson29 folder in the download.*

Hints

NetBeans 7 IDE supports JSF better than Eclipse does. It has a number of wizards, and in particular you can easily define the navigation of your web pages in a graphical interface. (See `http://netbeans.org/kb/docs/web/jsf20-support.html` for details.) The other popular IDE, IntelliJ IDEA 9, also offers excellent support for JSF. (Refer to `www.jetbrains.com/idea/features/javaserver_faces.html` for details.)

Step-by-Step

1. Create a new `StockServer` bean, which will include the methods `getQuote()` and `getNasdaqSymbols()`.
2. Define a JSF template.
3. Create two JSF pages: one for making a request for a price quote for a specific stock, and the other one for showing the result. The Submit button in the first web page should call the method `getQuote()` on the managed bean.
4. Run and test this JSF application.



Please select Lesson 29 on the DVD with the print book, or watch online at `www.wrox.com/go/fainjava` *to view the video that accompanies this lesson.*

30

Introducing JMS and MOM

People send messages to each other via e-mail, instant messages, Twitter, Facebook, and so on. People can also communicate using more traditional methods such as the post. You just need to drop a letter in a mailbox, and the rest will be taken care of by postal service providers and logistics companies such as USPS, FedEx, UPS, and DHL.

Applications can send messages to each other using message-oriented middleware (MOM), which plays a role similar to that of the delivery services. A program "drops the message" into a message queue (think *mailbox*) using the Java Messaging Service (JMS) API, and the message is delivered to another application that reads messages off of this queue.

Although JMS is a part of the Java EE specification, you can use it with Java SE applications without needing to have any Java application server — just make a `.jar` containing JMS classes available to your program. In this lesson you'll learn how to write simple Java clients to send and receive applications with the JMS API via a MOM provider. In Lesson 31 I'll show you how to bring a Java EE application server to the messaging architecture, and why it's beneficial. Finally, in Lesson 32 you'll learn about the value that message-driven beans bring to the table.

MESSAGING CONCEPTS AND TERMINOLOGY

You have already learned several methods of data exchange in distributed Java applications — direct socket communication, RMI, and HTTP-based interactions. But all of them were based on remote procedure calls (RPC) or the request/response model. MOM enables you to build distributed systems that communicate *asynchronously*.

JMS is an API for working with MOM. JMS itself isn't the transport for messages. JMS is to MOM what JDBC is to a relational DBMS. Java applications can use the same JMS classes with any MOM provider. Here's a list of some popular MOM software:

- WebSphere MQ (IBM)
- EMS (Tibco Software)

- SonicMQ (Progress Software)
- ActiveMQ (open source, Apache)
- Open MQ (open source, Oracle)

If you place an order to buy some stocks by calling the method `placeOrder()` on a remote machine, that's an example of a *synchronous* or blocking call. The calling program can't continue until the code in the `placeOrder()` method is finished or has thrown an error.

With *asynchronous* communications it's different — you can place an order but don't have to wait for its execution. Similarly, when you drop a letter in a mailbox you don't have to wait for a mail truck. The same applies to e-mail — press the Send button and continue working on other things without waiting until your message has been received. Recipients of your e-mails also don't have to be online when you send a message; they can read it later.

The process of placing a trading order comes down to putting a Java object that describes your order into a certain *message queue* of your MOM provider. After placing an order, the program may continue its execution without waiting until the processing of the order is finished. Multiple users can place orders into the same queue. Another program (not necessarily Java) should be de-queueing and processing messages. Figure 30-1 shows how a trading application can place orders (and receive executions) with another application running on a stock exchange.

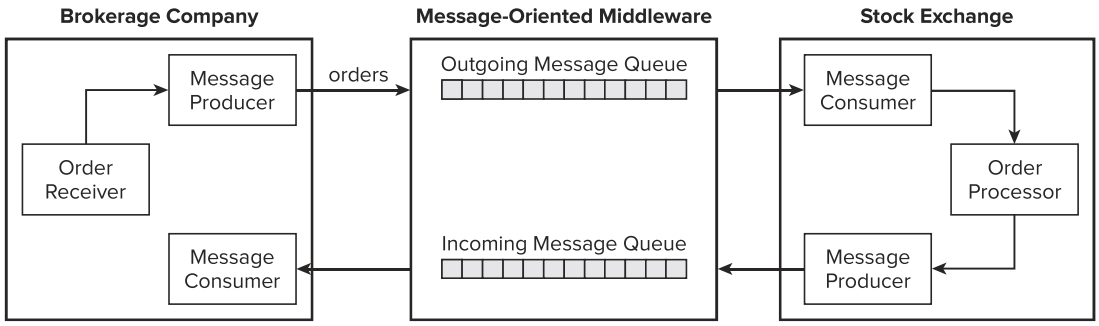


FIGURE 30-1

To increase the throughput of your messaging-based application, add multiple consumers reading messages off of the same queue. You can create a consumer Java program that starts multiple threads, each of them de-queuing messages from the same queue. In Lesson 32 you'll see how easy it is to configure multiple consumers using message-driven beans.

After your order has been processed, the appropriate message object will be placed into another queue and, if your application is active at that time, it will de-queue the message immediately upon its arrival. If your application is not running, but you've opted for guaranteed delivery (explained later in this lesson), the messages will be preserved in the queue by the MOM provider.

Even if the application in a brokerage company is down when the order execution arrives from the stock exchange, it'll be saved in the MOM server until the receiving program is up again.

TWO MODES OF MESSAGE DELIVERY

A program can *send* or *publish* a message. If it sends a message to a particular queue, and another program receives the message from this 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 `TopicSubscribers` 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 message 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());
         }
```

continues

LISTING 30-2 *(continued)*

```
        }

    }

     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.

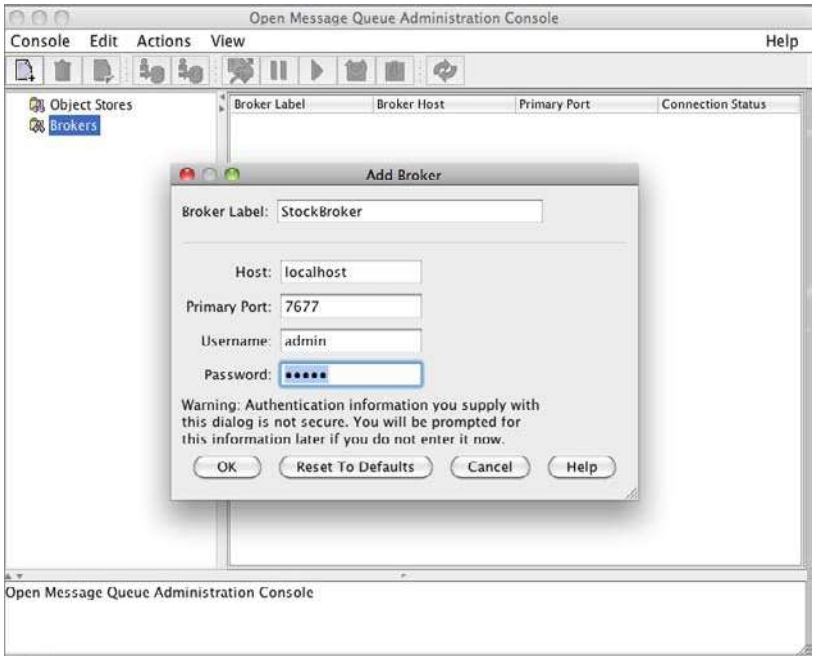


FIGURE 30-2

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.

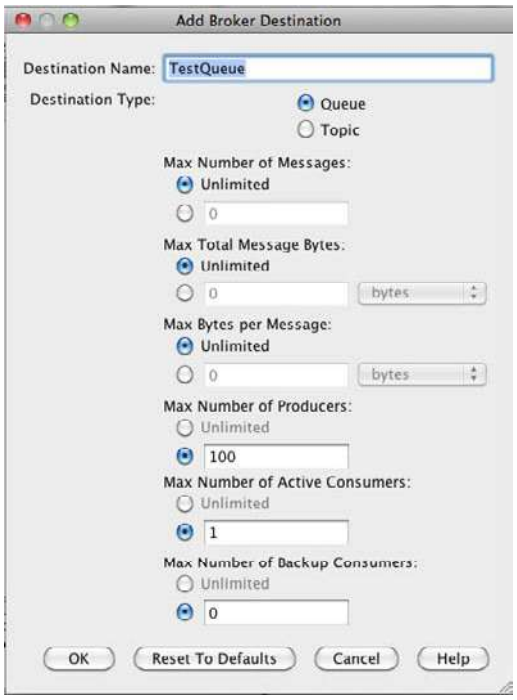


FIGURE 30-3

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.*