DAY 21

Writing Java Servlets and JavaServer Pages

The last subject to be explored is one of the most exciting and dynamic areas of growth in Java: the use of a Web server as a platform for application development.

The Java language has moved beyond applications that run on your computer and applets that run on a World Wide Web page. *Servlets*, Web applications run by a server over the Internet and presented by a Web browser, employ Java without the prohibitive security restrictions in place for applets. A servlet can use all the features of the language.

Using servlets, you can collect input from users through Web page forms, present records from a database or another source, and create Web pages dynamically.

This approach can be enhanced by JavaServer Pages (JSP), a way to create Web pages that mix static HTML with the output of servlets and Java expressions.

JavaServer Pages enable nonprogrammers to work on Web sites developed with Java. Today, you learn about each of the following topics:

- How servlets differ from applications and applets
- How to run servlets as part of the Apache Web server and other servers
- How to receive data from a Web page form
- How to store and retrieve cookies
- How to use servlets to dynamically generate Web content
- How to create a JavaServer Page
- How to use Java variables, expressions, and statements on a page

Using Servlets

Servlets are Java classes run by a Web server that has an interpreter that supports the Java Servlet specification. This interpreter, which is often called a *servlet engine*, is optimized to run servlets with a minimum of the server's resources.

Java servlets often serve the same purpose as programs implemented using the Common Gateway Interface, a protocol for writing software that sends and receives information through a Web server. CGI programming has been supported on the Web for most of its existence. Most CGI programs, which also are called CGI *scripts*, have been written using languages such as Perl, Python, and C.

CGI programs are used often for these purposes:

- Collecting user input from a form on a Web page
- Receiving information from arguments specified as part of a URL
- Running programs on the computer that runs the Web server
- Storing and retrieving *cookies*, files that store a user's preferences and other information on his computer
- Sending data back to a user in the form of an HTML document, GIF graphic, or another format

Java servlets can do all these things along with some behavior that's difficult to implement using most CGI scripting languages.

Servlets offer full support for *sessions*, a way to keep track of a particular user over time as a Web site's pages are being viewed. They also can communicate directly with a Web server using a standard interface. As long as the server supports Java servlets, it can exchange information with those programs.

Java servlets have the same portability advantages as the language itself. Although Sun's official implementation of servlets was created with the Apache Software Foundation—the open source developers who created the Apache Web server—many other companies and groups have introduced tools to support Java servlets such as IBM WebSphere, BEA WebLogic, and the developers of the Jetty server.

Servlets also run efficiently in memory. If 10 people are simultaneously using the same CGI script, a Web server will have 10 copies of that script loaded into memory. If 10 people are using a Java servlet, only one copy of the servlet will be loaded, spawning threads to handle each user.

Supporting Servlets

Servlets can be created using the `javax.servlet` and `javax.servlet.http` packages, which are a standard part of the Java 2 Enterprise Edition (J2EE), an expanded version of the Java class library that supports large-scale development projects.

These classes implement the Java Servlet and JavaServer Pages specification. At this time, the most up-to-date versions are Java Servlet 2.4 and JavaServer Pages 2.0.

For users of the Java 2 Standard Edition (J2SE), which has been covered throughout this book, the servlet packages can be downloaded from Sun's Java servlet site at `http://java.sun.com/products/servlet`. Click the Downloads link and under the Specifications heading, choose the Download Class Files link for the Java Servlet specification version 2.3 (class files for version 2.4 also might be available).

After installing the class files, to make them available, add the root folder of the installation to your computer's `Classpath` environmental variable (for example, if the `javax` folder was saved in `c:\java\servlet`, add the `c:\java\servlet` folder to the `Classpath`).

You can create and compile servlets with these packages, but running them also requires a server supporting Java Servlet 2.3 and JavaServer Pages 1.2.

Servlets are supported by several Web servers, each of which has its own installation, security, and administration procedures.

The most popular choice for new servlet developers is Tomcat, an open-source server developed by the Apache Software Foundation in cooperation with Sun Microsystems. Version 4.1 of Tomcat supports Java Servlet 2.3 and JavaServer Pages 1.2. A newer version, 5.0, also supports the Servlet 2.4 and JavaServer Pages 2.0 specifications.

Tomcat can run in conjunction with another Web server (such as Apache's Web server) or as a standalone server. Your current Web server or Web application server may already include support for servlets.

The software is available as a free download from Apache's Web site at the address `http://jakarta.apache.org/tomcat`. Several versions are available: Download a version with 5.*x* numbering (such as 5.0.19).

Full installation instructions for Tomcat are available from the Web site. If you want to run it as a standalone server for testing purposes, in most cases, the following procedure works:

1. View the downloads page for Tomcat and other Apache projects: Visit `http://jakarta.apache.org/tomcat`, and in the Downloads section of the site menu, click the Binaries link.

   A page opens with links to Apache software you can download, which are called *builds*.
2. In the Release Builds section, scroll to the Tomcat 5.0.x heading.

   The software can be downloaded as a `ZIP` archive, an archive compressed with `TAR` and `GZ`, or a self-extracting `EXE` file.
3. Click the link of the version you want to download.
4. Install the software, making note of the folder where it is installed.
5. Create an environmental variable called `JAVA_HOME` that contains the folder where Java is installed on your computer.
6. Create an environmental variable called `CATALINA_HOME` that contains the folder where Tomcat was installed.
7. In the `bin` folder of the Tomcat installation, use either `catalina.sh` or `catalina.bat` to run the server with one of these commands:

   ```
   catalina.sh start
   ```

   ```
   catalina.bat start
   ```

   Tomcat begins running at port 8080 of your computer. (There's also a `stop` command to shut it down.)
8. To verify that it's running, open the address `http://localhost:8080` with a Web browser. If you're testing Tomcat from another computer, replace `localhost` with the domain name or IP address of the server where Tomcat has been installed.

If you don't have a server but you want to begin developing servlets, several companies offer commercial Web hosting with Java servlet support. These companies have already installed Tomcat and configured it to work with their servers, leaving you to focus on writing servlets using the classes of the `javax.servlet` and `javax.servlet.http` packages.

NOTE

For previous editions of the book, I tested servlets using Motivational Marketing Associates (MMA) to host servlets and JavaServer Pages. MMA offers Java servlet hosting on an Apache Web server running Linux. You can find out more about its commercial hosting services by visiting `http://www.mmaweb.com`.

Developing Servlets

Java servlets are created and compiled just like any other Java class. After you install the servlet packages and add them to your computer's `Classpath`, you can compile servlets with the SDK's Java compiler or any other current compiler.

Every servlet is a subclass of the `HttpServlet` class, which is part of the `javax.servlet` package. This class includes methods that represent the life cycle of a servlet and collect information from the Web server running the servlet.

A servlet's life cycle methods function similarly to the life cycle methods of applets.

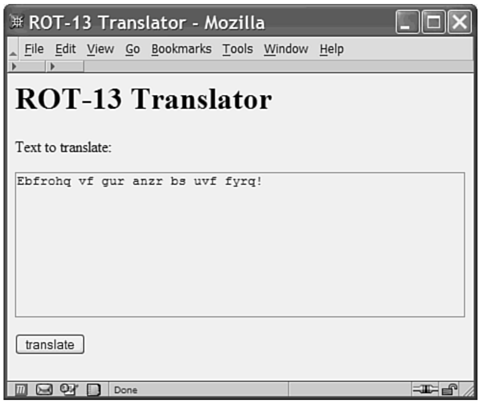
The `init(`*`ServletConfig`*`)` method is called automatically when a Web server first brings a servlet online to handle a user's request. As mentioned earlier, one Java servlet can handle multiple requests from different Web users. The `init()` method is called only once, when a servlet comes online. If a servlet is already online when another request to use the servlet is received, the `init()` method won't be called again.

The `init()` method has one argument—`ServletConfig`, an interface in the `javax.servlet` package that contains methods to find out more about the environment in which a servlet is running.

The `destroy()` method is called when a Web server takes a servlet offline. Like the `init()` method, this is called only once, when all users have finished receiving information from the servlet. If this doesn't take place in a specified amount of time, `destroy()` is called automatically, preventing a servlet from being hung up while it waits for information to be exchanged with a user.

One of the main tasks of a servlet is to collect information from a Web user and present something back in response. You can collect information from a user by using a *form*, which is a group of text boxes, radio buttons, text areas, and other input fields on a Web page.

Figure 21.1 shows a Web form on a page loaded with the Mozilla Web browser.


ROT-13 Translator - Mozilla
File Edit View Go Bookmarks Tools Window Help
translate
Done

ROT-13 Translator

Text to translate:

Ebfrohq vf gur anzr bs uvf fyrq!

FIGURE 21.1
Collecting information with a Web form.

The form displayed in Figure 21.1 contains two fields: a text area and a clickable Translate button. The Hypertext Markup Language (HTML) tags used to display this page are the following:

```
<html>
<body>
<head><title>ROT-13 Translator</title></head>
<h1>ROT-13 Translator</h1>
<p>Text to translate:
<form action="Rot13" method="POST">
<textarea name="text" ROWS=8 COLS=55>
</textarea>
<p><input type="submit" value="translate">
</form>
</body>
</html>
```

The form is contained within the `<form>` and `</form>` HTML tags. Each field on the form is represented by its own tags: `<textarea>` and `</textarea>` for the text area and `<input>` for the Translate button. The text area is given a name, `text`.

TIP

Servlets require you to have a basic familiarity with HTML because the only user interface for a servlet is a Web page running in a browser. Two books that are good for learning HTML are *Sams Teach Yourself HTML in 24 Hours, Sixth Edition*, by Dick Oliver and Michael Morrison (ISBN 0-67232-520-9) and *Sams Teach Yourself Web Publishing with HTML and XHTML in 21 Days, Fourth Edition*, by Laura Lemay and Rafe Colburn (ISBN 0-672-32519-5).

Each field on a form stores information that can be transmitted to a Web server and then sent to a Java servlet. Web browsers communicate with servers by using Hypertext Transfer Protocol (HTTP). Form data can be sent to a server using two kinds of HTTP requests: `GET` and `POST`.

When a Web page calls a server using `GET` or `POST`, the name of the program that handles the request must be specified as a Web address, also called a *uniform resource locator (URL)*.

A `GET` request affixes all data on a form to the end of a URL, as in this example:

```
http://www.java21days.com/servlets/beep?number=5551220&repeat=no
```

A `POST` request includes form data as a header sent separately from the URL. This is generally preferred, and it's required when confidential information is being collected on the form. Also, some Web servers and browsers do not support URLs longer than 255 characters, which limits the amount of information that can be sent in a `GET` request.

Java servlets handle both of these requests through methods inherited from the `HttpServlet` class: `doGet(`*`HttpServletRequest`*`, `*`HttpServletResponse`*`)` and `doPost(`*`HttpServletRequest`*`, `*`HttpServletResponse`*`)`. These methods throw two kinds of exceptions: `ServletException`, which is part of the `javax.servlet` package, and `IOException`, an exception in the standard `java.io` package that involves input and output streams.

The `doGet()` and `doPost()`methods have two arguments: an `HttpServletRequest` object and an `HttpServletResponse` object. One is called when a `GET` request is used to execute the servlet, and the other is called with `POST`. A common technique in Java servlet programming is to use one method to call the other, as in the following example:

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

    doPost(request, response);
}
```

The request and response objects belong to classes in the `javax.servlet.http` package. A servlet receives information about how it was run by calling methods of the `HttpServletRequest` class. For example, when a Web form is submitted to a servlet, each field on the form is stored as a string by the `HttpServletRequest` class.

You can retrieve these fields in a servlet by calling the `getParameter(`*`String`*`)` method with the name of the field as an argument. This method returns `null` if no field of that name exists.

A servlet communicates with the user by sending back an HTML document, a graphics file, or another type of information supported by a Web browser. It sends this information by calling the methods of the `HttpServletResponse` class.

The first thing you must do when preparing a response is to define the kind of content the servlet is sending to a browser. Call the `setContentType(`*`String`*`)` method with the content type as an argument.

The most common form for a response is HTML, which is set by calling `setContentType("text/html")`. You also can send a response as text (`"text/plain"`), graphics files (`"image/gif"`, `"image/jpeg"`), and application-specific formats such as `"application/msword"`.

To send data to a browser, you create a servlet output stream associated with the browser and then call the `println(`*`String`*`)` method on that stream. Servlet output streams are represented by the `ServletOutputStream` class, which is part of the `javax.servlet` package. You can get one of these streams by calling the response object's `getOutputStream()` method.

The following example creates a servlet output stream from an `HttpServletResponse` object called `response` and then sends a short Web page to that stream:

```
ServletOutputStream out = response.getOutputStream();
out.println("<html>");
out.println("<body>");
out.println("<h1>Hello World!</h1>");
out.println("</body>");
out.println("</html>");
```

Listing 21.1 contains a Java servlet that receives data from the form displayed in Figure 21.1.

LISTING 21.1 The Full Text of `Rot13.java`

```
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class Rot13 extends HttpServlet {

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

        String text = req.getParameter("text");
        String translation = translate(text);
        res.setContentType("text/html");
```

LISTING 21.1 continued

```
        ServletOutputStream out = res.getOutputStream();
        out.println("<html>");
        out.println("<body>");
        out.println("<head><title>ROT-13 Translator</title></head>");
        out.println("<h1>ROT-13 Translator</h1>");
        out.println("<p>Text to translate:");
        out.println("<form action=\"Rot13\" method=\"POST\">");
        out.println("<textarea name=\"text\" ROWS=8 COLS=55>");
        out.println(translation);
        out.println("</textarea>");
        out.println("<p><input type=\"submit\" value=\"translate\">");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }

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

        doPost(req, res);
    }

    String translate(String input) {
        StringBuffer output = new StringBuffer();
        if (input != null) {
            for (int i = 0; i < input.length(); i++) {
                char inChar = input.charAt(i);
                if ((inChar >= 'A') & (inChar <= 'Z')) {
                    inChar += 13;
                    if (inChar > 'Z')
                        inChar -= 26;
                }
                if ((inChar >= 'a') & (inChar <= 'z')) {
                    inChar += 13;
                    if (inChar > 'z')
                        inChar -= 26;
                }
                output.append(inChar);
            }
        }
        return output.toString();
    }
}
```

After saving the servlet, compile it with the Java compiler.

The `Rot13` servlet receives text from a Web form, translates it using ROT-13, and then displays the result in a new Web form. ROT-13 is a trivial method of encrypting text through letter substitution. Each letter of the alphabet is replaced with the letter that's 13 places away: A becomes N, N becomes A, B becomes O, O becomes B, C becomes P, P becomes C, and so on.

Because the ROT-13 encryption scheme is easy to decode, it isn't used when information must remain secret. Instead, it's used casually on Internet discussion forums such as Usenet newsgroups. For example, if someone on a movie newsgroup wants to share a spoiler that reveals a plot detail about an upcoming movie, she can encode it in ROT-13 to prevent people from reading it accidentally.

NOTE

Want to know the big secret from the 1973 film *Soylent Green*? Decode this ROT-13 text: Fbba gurl'yy or oerrqvat hf yvxr pnggyr! Lbh'ir tbg gb jnea rirelbar naq gryy gurz! Fblyrag terra vf znqr bs crbcyr! Lbh'ir tbg gb gryy gurz! Fblyrag terra vf crbcyr!

To make the ROT-13 servlet available, you must publish its class files in a folder on your Web server that has been designated for Java servlets.

Tomcat is organized so that servlets, other classes, and JavaServer Pages are placed in subfolders of the software's `webapps` folder. One way to deploy a servlet's class file is to store it in a `WEB-INF\classes` subfolder somewhere in the `webapps` hierarchy of folders.

Tomcat 4 includes several example servlets in the `servlets-examples` and `jsp-examples` folders inside `webapps`. You can deploy the ROT-13 servlet in the `servlet-examples` folder by storing `Rot13.class` in `webapps\servlet-examples\WEB-INF\classes` (Windows) or `webapps/servlet-examples/WEB-INF/classes` (Linux).

After adding the class file, restart Tomcat and run the servlet by loading its address with a Web browser.

The address of the servlet depends on where it was stored in the `webapps` folder. If you put it in the `java` folder, it's in `/java/Rot13`, as in `http://www.java21days.com:8080/java/Rot13`.

TIP

You can try out a working ROT-13 servlet on the book's Web site; visit `http://www.java21days.com` and open the Day 21 page.

Using Cookies

Many Web sites can be customized to keep track of information about you and the features you want the site to display. This customization is possible because of a Web browser feature called *cookies*, small files containing information that a Web site wants to remember about a user, such as a username, the number of visits, and the like. The files are stored on the user's computer, and a Web site can read only the cookies on the user's system that the site has created.

Because of privacy considerations, most Web browsers can be configured to reject all cookies or ask permission before allowing a site to create a cookie. The default behavior for most browsers is to accept all cookies.

With servlets, you can easily create and retrieve cookies as a user runs your application. Cookies are supported by the `Cookie` class in the `javax.servlet.http` package.

To create a cookie, call the `Cookie(`*`String`*`, `*`String`*`)` constructor. The first argument is the name you want to give the cookie, and the second is the cookie's value.

One use for cookies is to count the number of times someone has loaded a servlet. The following statement creates a cookie named `visits` and gives it the initial value of 1:

```
Cookie visitCookie = new Cookie("visits", "1");
```

When you create a cookie, you must decide how long it should remain valid on a user's computer. Cookies can be valid for an hour, a day, a year, or any time in between. When a cookie is no longer valid, the Web browser deletes it automatically.

Call a cookie's `setMaxAge(`*`int`*`)` method to set the amount of time the cookie remains valid, in seconds. If you use a negative value as an argument, the cookie remains valid only while the user's Web browser is open. If you use 0 as a value, the cookie is not stored on a user's computer.

NOTE

The purpose of creating a cookie with a maximum age of 0 is to tell the Web browser to delete the cookie if it already has one.

Cookies are sent to a user's computer along with the data displayed by the Web browser. To send a cookie, call the `addCookie(`*`Cookie`*`)` method of an `HttpServletResponse` object.

You can add more than one cookie to a response. When cookies are stored on a user's computer, they're associated with the URL of the Web page or program that created the cookie. You can associate several cookies with the same URL.

When a Web browser requests a URL, the browser checks to see whether any cookies are associated with that URL. If there are, the cookies are sent along with the request.

In a servlet, call the `getCookies()` method of an `HttpServletRequest` object to receive an array of `Cookie` objects. You can call each cookie's `getName()` and `getValue()` methods to find out about that cookie and do something with the data.

Listing 21.2 contains `ColorServlet`, an extended version of the ROT-13 servlet that enables a user to select the background color of the page. The color is stored as a cookie called `color`, and the servlet requests the cookie from a Web browser every time the servlet is loaded.

LISTING 21.2 The Full Text of `ColorServlet.java`

```
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class ColorServlet extends HttpServlet {

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

        String pageColor;
        String colorParameter = req.getParameter("color");
        if (colorParameter != null) {
            Cookie colorCookie = new Cookie("color", colorParameter);
            colorCookie.setMaxAge(31536000);
            res.addCookie(colorCookie);
            pageColor = colorParameter;
        } else {
            pageColor = retrieveColor(req.getCookies());
        }
        String text = req.getParameter("text");
        String translation = translate(text);
        res.setContentType("text/html");
        ServletOutputStream out = res.getOutputStream();
        out.println("<html>");
        out.println("<body bgcolor=\"" + pageColor + "\">");
        out.println("<head><title>ROT-13 Translator</title></head>");
        out.println("<h1>ROT-13 Translator</h1>");
        out.println("<p>Text to translate:");
        out.println("<form action=\"ColorServlet\" method=\"POST\">");
```

LISTING 21.2 continued

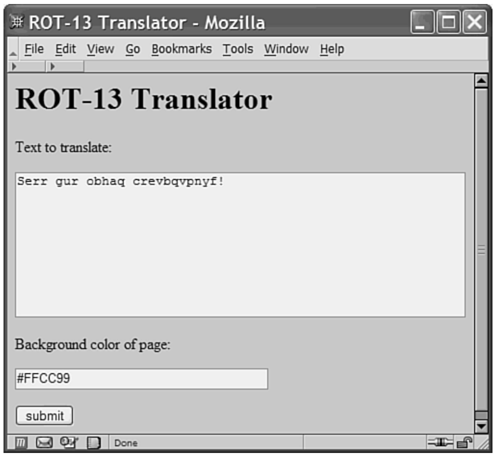
```
        out.println("<textarea name=\"text\" ROWS=8 COLS=55>");
        out.println(translation);
        out.println("</textarea>");
        out.println("<p>Background color of page:");
        out.println("<p><input type=\"text\" name=\"color\" value=\"" +
            pageColor + "\" SIZE=40>");
        out.println("<p><input type=\"submit\" value=\"submit\">");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }

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

        doPost(req, res);
    }

    String translate(String input) {
        StringBuffer output = new StringBuffer();
        if (input != null) {
            for (int i = 0; i < input.length(); i++) {
                char inChar = input.charAt(i);
                if ((inChar >= 'A') & (inChar <= 'Z')) {
                    inChar += 13;
                    if (inChar > 'Z')
                        inChar -= 26;
                }
                if ((inChar >= 'a') & (inChar <= 'z')) {
                    inChar += 13;
                    if (inChar > 'z')
                        inChar -= 26;
                }
                output.append(inChar);
            }
        }
        return output.toString();
    }

    String retrieveColor(Cookie[] cookies) {
        String inColor = "#FFFFFF";
        for (int i = 0; i < cookies.length; i++) {
            String cookieName = cookies[i].getName();
            if (cookieName.equals("color")) {
                inColor = cookies[i].getValue();
            }
        }
        return inColor;
```

LISTING 21.2 continued

```
    }
}
```

Figure 21.2 shows the servlet running in a Web browser.

FIGURE 21.2


ROT-13 Translator - Mozilla
File Edit View Go Bookmarks Tools Window Help
submit
Done

ROT-13 Translator

Text to translate:

```
Serr gur obhaq crevbqvpnyf!
```

Background color of page:

#FFCC99

A Web page generated by the `ColorServlet` *servlet.*

To change the page's color, type a new value into the Background Color of Page text field and click the Submit button.

Colors are expressed as a # sign followed by three two-digit hexadecimal numbers (in Figure 21.2, the numbers are FF, CC, and 99). These numbers represent the amount of red, green, and blue the color contains, ranging from a minimum of 00 to a maximum of FF. If you aren't familiar with hexadecimal colors, you can try these out while testing the servlet:

- `#FF0000`—Bright red
- `#00FF00`—Bright green
- `#0000FF`—Bright blue
- `#FFAAAA`—Light red
- `#AAFFAA`—Light green
- `#AAAAFF`—Light blue
- `#FFCC66`—Butterscotch

Using Sessions

One of the most powerful features offered in servlets is support for *sessions*, a means of monitoring a user over time as a servlet is being used.

Normally, the World Wide Web is a stateless protocol, which means that there's no easy way to follow a user around from page to page. A client Web browser requests a URL from a server, receives the file associated with that URL, and then is completely forgotten by the server. Nothing is done to track what a specific user does over a period of time on a Web site.

This information isn't important if you're just offering a collection of static pages, but it's essential for many Web applications. This is especially true in e-commerce: an online store needs to know which items you've added to your shopping cart when it's time to calculate your bill, the final total needs to be remembered when a charge must be applied to your credit card, and so on.

Servlets can retain the state of a user through the use of `HttpSession`, a class that represents sessions. There can be one session object for each user running your servlet.

Sessions are created, deleted, and maintained behind the scenes by the server running the servlet.

A user's session can be created or retrieved by calling the `getSession(`*`Boolean`*`)` method of the servlet's request object. Use an argument of `true` if a session should be created when one doesn't already exist for the user, as in this example for an `HttpRequest` object named `req`:

```
HttpSession state = req.getSession(true);
```

A session object must be accessed in this manner before any servlet output has been composed by calling a response object's methods.

You can find out whether the session is newly created by calling its `isNew()` method, which returns `true` under that circumstance.

If you need to keep track of something as a user employs your servlet, it can be stored in the session object.

The session object can hold objects in a manner comparable to a `Vector`, one of the data structures described on Day 8, "Data Structures."

Objects held by a session are called its *attributes*. Call the session's `setAttribute(`*`String`*`, `*`Object`*`)` method with two arguments: a name to give the attribute and the object.

To retrieve an attribute, call the `getAttribute(String)` method with its name as the only argument. It returns the object, which must be cast from `Object` to the desired class, or `null` if no attribute of that name exists.

To remove an attribute when it's no longer needed, call `removeAttribute(String)` with its name as the argument. This method does not return `null` if the attribute does not exist; instead, it simply does nothing.

All three methods throw `IllegalStateException` exceptions if the session is no longer valid. This can occur if the session was deleted by the server before the request was made, some kind of error prevented sessions from being maintained, or similar reasons.

Today's next project uses sessions to track whether a user has provided login information to the servlet yet, as shown in Figure 21.3.

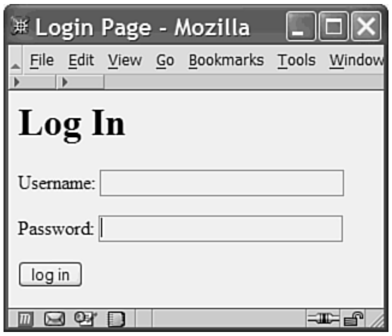


FIGURE 21.3
Using servlets to log in users.

A servlet that's used to log in a user, authenticating that the person has a valid username and password, can be loaded under three different circumstances:

- The servlet is run before the user logs in. A form must be provided so that the user can provide a username and password.
- The servlet is run to log in. The username and password provided by the user must be authenticated in some manner, presumably by checking a database.
- The servlet is run after a user logs in.

To know what has happened before, which is necessary in all these circumstances, sessions are used.

The `LoginServlet` program handles user logins with three session attributes: username, password, and `loggedIn`, a `Boolean` object that is `true` when the user has logged in and `false` otherwise. The source code is shown in Listing 21.3.

LISTING 21.3 The Full Text of `LoginServlet.java`

```java
import java.io.*;
import java.util.Date;
import javax.servlet.*;
import javax.servlet.http.*;

public class LoginServlet extends HttpServlet {

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

        HttpSession session = req.getSession();
        Boolean loggedIn = (Boolean) session.getAttribute("loggedIn");
        if (loggedIn == null) {
            loggedIn = new Boolean(false);
        }
        String username = req.getParameter("username");
        String password = req.getParameter("password");
        Date lastVisit;
        res.setContentType("text/html");
        ServletOutputStream out = res.getOutputStream();
        out.println("<html>");
        out.println("<body>");
        out.println("<head><title>Login Page</title></head>");
        if (loggedIn.booleanValue() == true) {
            // user is already logged in
            username = (String) session.getAttribute("username");
            password = (String) session.getAttribute("password");
            lastVisit = (Date) session.getAttribute("lastVisit");
            out.println("<p>Welcome back, " + username);
            out.println("<p>You last visited on " + lastVisit);
            lastVisit = new Date();
            session.setAttribute("lastVisit", lastVisit);
        } else {
            if (username == null) {
                // user has not submitted the form required to log in
                out.println("<h1>Log In</h1>");
                out.println("<form action=\"LoginServlet\" " +
                    "method=\"POST\">");
                out.println("<p>Username:");
                out.println("<input type=\"text\" name=\"username\" " +
                    "value=\"\" SIZE=30>");
                out.println("<p>Password:");
                out.println("<input type=\"password\" name=\"password\" " +
                    "value=\"\" SIZE=30>");
                out.println("<p><input type=\"submit\" value=\"log in\">");
                out.println("</form>");
            } else {
                // user has submitted the log in form
```

LISTING 21.3 continued

```
                out.println("Logging in " + username);
                session.setAttribute("username", username);
                session.setAttribute("password", password);
                session.setAttribute("loggedIn", new Boolean(true));
                session.setAttribute("lastVisit", new Date());
                out.println("<a href=\"LoginServlet\">Reload Page</a>");
            }
            out.println("</body>");
            out.println("</html>");
        }
    }

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

        doPost(req, res);
    }
}
```

When the servlet is loaded for the first time in a Web browser, it presents a form as shown earlier in Figure 21.3.

Filling out the form and clicking the Submit button displays a page that has the text “Logging in” and a Reload Page hyperlink.

Clicking the hyperlink loads a page with a greeting such as the following:

```
Welcome back, rcade

You last visited on Sun Feb 29 14:38:18 EST 2004
```

The servlet does not contain any code to check whether the provided username and password are valid. It simply stores them in a session so that they’re available when the servlet is run again subsequently.

JavaServer Pages

Java servlets make it easy to generate HTML text dynamically, producing pages that change in response to user input and data.

However, servlets make it difficult to generate HTML text that never changes, because it is cumbersome and tedious to use Java statements to output HTML.

Servlets also require the services of a Java programmer any time the HTML needs to be changed. The servlet must be edited, recompiled, and deployed on the Web, and few organizations would be comfortable handing that task to a nonprogrammer.

JavaServer Pages are a complement to servlets rather than a replacement. They make it easy to separate two kinds of Web content:

- Static content, the portions of a Web page that don't change, such as an online store's description of each product
- Dynamic content, the portions of a Web page generated by a servlet, such as the store's pricing and availability data for each product, which can change as items sell out

When you use only servlets on a project, it becomes difficult to make minor changes, such as correcting a typo in text, rewording a paragraph, or altering some HTML tags to change how the page is presented. Any kind of change requires the servlet to be edited, compiled, tested, and redeployed on the Web server.

With JavaServer Pages, you can put the static content of a Web page in an HTML document and call servlets from within that content. You also can use other parts of the Java language on a page, such as expressions, `if-then` blocks, and variables. A Web server that supports the Tomcat specification knows how to read these pages and execute the Java code they contain, generating an HTML document as if you wrote a servlet to handle the whole task. In actuality, JavaServer Pages use servlets for everything.

You create aJavaServer Page as you would create an HTML document—in a text editor or Web publishing program such as Microsoft FrontPage 2003 or Macromedia Dreamweaver MX. When you save the page, use the `.jsp` file extension to indicate that the file is a JavaServer Page instead of an HTML document. Then the page can be published on a Web server like an HTML document, as long as the server supports servlets and JavaServer Pages.

When a user requests the JavaServer Page for the first time, the Web server compiles a new servlet that presents the page. This servlet combines everything that has been put into the page:

- Text marked up with HTML
- Calls to Java servlets
- Java expressions and statements
- Special JavaServer Pages variables

Writing a JavaServer Page

A JavaServer Page consists of three kinds of elements, each with its own special markup tag that's similar to HTML:

- Scriptlets—Java statements executed when the page is loaded. Each of these statements is surrounded by `<%` and `%>` tags.
- Expressions—Java expressions that are evaluated, producing output displayed on the page. These are surrounded by `<%=` and `%>` tags.
- Declarations—Statements to create instance variables and handle other setup tasks required in the presentation of the page. These are surrounded by `<%!` and `%>` tags.

Using Expressions

Listing 21.4 contains a JavaServer Page that includes one expression, a call to the `java.util.Date()` constructor. This constructor produces a string containing the current time and date. Enter this file with any text editor that can save files as plain text. (The editor you've been using to create Java source code will work for this purpose as well.)

LISTING 21.4 The Full Text of `time.jsp`

```
<html>
<head>
<title>Clock</title>
</head>
<body>
<h1 align="Center">
<%= new java.util.Date() %>
</h1>
</body>
</html>
```

After saving the file, upload it to your Web server in a folder where other Web pages are stored. Unlike Java servlets, which must be in a folder that has been designated for servlets, JavaServer Pages can be placed in any folder that's accessible on the Web.

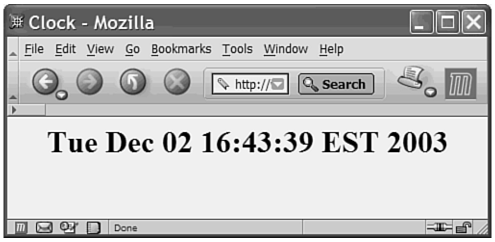
In Tomcat 4, you can place the page in any folder inside the `webapps` folder. If you stored the page in `webapps\java`, it would be available at `/java/time.jsp`, as in `http://www.java21days.com:8080/java/time.jsp`.

When you load the page's URL for the first time with a Web browser, the Web server compiles the JavaServer Page into a servlet automatically. This causes the page to load slowly for the first time, but subsequent requests run much more quickly.

Figure 21.4 shows the output of `time.jsp`.

FIGURE 21.4

Using an expression in a JavaServer Page.



```
Tue Dec 02 16:43:39 EST 2003
```

When a JavaServer Page includes an expression, it's evaluated to produce a value and displayed on the page. If the expression produces different values each time the page is displayed, as `time.jsp` does in line 7 of Listing 21.4, this is reflected in the page when loaded in a Web browser.

There are several servlet objects you can refer to in expressions and other elements of a JavaServer Page using the following variable names:

- `out`—The servlet output stream
- `request`—The HTTP servlet request
- `response`—The HTTP servlet responses
- `session`—The current HTTP session
- `application`—The servlet context used to communicate with the Web server
- `config`—The servlet configuration object used to see how the servlet was initialized

Using these variables, you can call the same methods from within a JavaServer Page that are available in a servlet.

Listing 21.5 contains the text of the next page you'll create, `environment.jsp`, which shows how the `request` variable can be used on a page. This variable represents an object of the `HttpServletRequest` class, and you can call the object's `getHeader(`*`String`*`)` method to retrieve HTTP headers that describe the request in more detail.

LISTING 21.5 The Full Text of `environment.jsp`

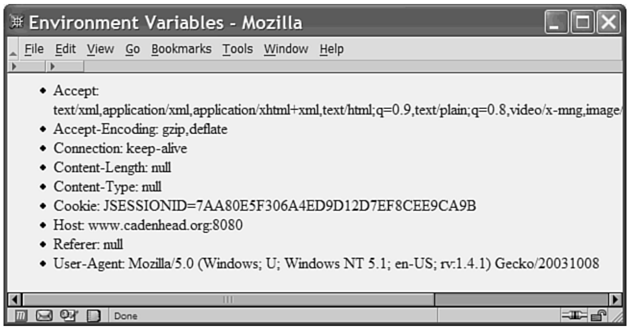
```
<html>
<head>
<title>Environment Variables</title>
</head>
```

LISTING 21.5 continued

```
 5: <body>
 6: <ul>
 7: <li>Accept: <%= request.getHeader("Accept") %>
 8: <li>Accept-Encoding: <%= request.getHeader("Accept-Encoding") %>
 9: <li>Connection: <%= request.getHeader("Connection") %>
10: <li>Content-Length: <%= request.getHeader("Content-Length") %>
11: <li>Content-Type: <%= request.getHeader("Content-Type") %>
12: <li>Cookie: <%= request.getHeader("Cookie") %>
13: <li>Host: <%= request.getHeader("Host") %>
14: <li>Referer: <%= request.getHeader("Referer") %>
15: <li>User-Agent: <%= request.getHeader("User-Agent") %>
16: </ul>
17: </body>
18: </html>
```

In lines 7–15 of the `environment.jsp` page, each line contains a call to `getHeader()` that retrieves a different HTTP request header. Figure 21.5 shows an example of the output. The values reported for each header depend on your Web server and the Web browser you're using, so you won't see the same values for `User-Agent`, `Referer`, and other headers.

FIGURE 21.5
Using servlet variables on a JavaServer Page.


Environment Variables - Mozilla
File Edit View Go Bookmarks Tools Window Help
Done

- Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-mng,image/
- Accept-Encoding: gzip,deflate
- Connection: keep-alive
- Content-Length: null
- Content-Type: null
- Cookie: JSESSIONID=7AA80E5F306A4ED9D12D7EF8CEE9CA9B
- Host: www.cadenhead.org:8080
- Referer: null
- User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.4.1) Gecko/20031008

Using Scriptlets

You also can use Java statements in a JavaServer Page—calling methods, assigning values to variables, creating conditional statements, and so on. These statements begin with the `<%` tag and end with the `%>` tag. More than one statement can be enclosed within these tags.

Statements that appear inside a JavaServer Page are called *scriptlets*. You can use any of the servlet variables that were available for expressions.

Listing 21.6 contains the code for `shopforbooks.jsp`, a Web page that displays a list of books, with hyperlinks to each book's page at an online bookstore.

LISTING 21.6 The Full Text of `shopforbooks.jsp`

```
<html>
<head>
<title>Shop for Books</title>
</head>
<body>
<h2 align="Left">Favorite Books</h2>
<%
String[] bookTitle = { "Catch-22", "Something Happened",
    "Good as Gold" };
String[] isbn = { "0684833395", "0684841215", "0684839741" };
String amazonLink = "http://www.amazon.com/exec/obidos/ASIN/";
String bnLink = "http://shop.bn.com/booksearch/isbnInquiry.asp?isbn=";

String store = request.getParameter("store");
if (store == null) {
    store = "Amazon";
}
for (int i = 0; i < bookTitle.length; i++) {
    if (store.equals("Amazon"))
        out.println("<li><a href=\"" + amazonLink + isbn[i] + "\">" +
            bookTitle[i] + "</a>");
    else
        out.println("<li><a href=\"" + bnLink + isbn[i] + "\">" +
            bookTitle[i] + "</a>");
}
%>
<p>Preferred Bookstore:
<form action="shopforbooks.jsp" method="POST">
<p><input type="radio" value="Amazon"
<%= (store.equals("Amazon") ? " checked" : "") %>
name="store"> Amazon.Com
<p><input type="radio" value="BN"
<%= (store.equals("BN") ? " checked" : "") %>
name="store"> Barnes & Noble
<p><input type="submit" value="Change Store">
</form>
</body>
</html>
```

This JavaServer Page includes a form at the bottom of the page that lets users pick which bookstore they like to use for online shopping.

In line 28, the form is being submitted to the URL of the JavaServer Page. Because pages are actually servlets, they can also receive form data that's sent by `POST` or `GET`.

This page uses the `store` field to hold `"Amazon"` if Amazon.com is the preferred store and `"BN"` if Barnes & Noble is the preferred store.

One thing to note as you test the server page is how the radio buttons on the form always match the store you've chosen. This occurs because of expressions that appear on lines 29 and 31. Here's one of those expressions:

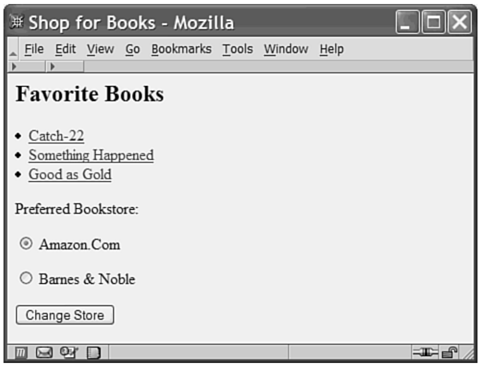
```
<%= (store.equals("Amazon") ? " checked" : "") %>
```

This expression uses the ternary operator with the conditional `store.equals("Amazon")`. If this condition is true, the word `"checked"` is the value of the expression. Otherwise, an empty string (`""`) is the value.

The value of expressions is displayed as part of the JavaServer Page. Figure 21.6 shows what this page looks like in a Web browser.

FIGURE 21.6

Displaying dynamic content using scriptlets.



Favorite Books

- Catch-22
- Something Happened
- Good as Gold

Preferred Bookstore:

[✓ Amazon.Com]

Barnes & Noble

Using Declarations

Another element you can insert into a JavaServer Page is a *declaration*, which is a statement that sets up a variable or method that will be defined in the page when it's compiled into a servlet. This feature is primarily used in conjunction with expressions and servlets.

Declarations are surrounded by `<%!` and `%>` tags, as in the following example:

```
<!% boolean noCookie = true %>
<!% String userName = "New user" %>
```

These declarations create two instance variables: `noCookie` and `userName`. When the JavaServer Page is compiled into a servlet, these variables will be part of the definition of that class.

Listing 21.7 contains a JavaServer Page that uses a declaration to present a counter.

LISTING 21.7 The Full Text of `counter.jsp`

```
 1: <%@ page import="counter.*" %>
 2: <html>
 3: <head>
 4: <title>Counter Example</title>
 5: </head>
 6: <body>
 7: <h1>JSP Stats</h1>
 8: <%! Counter visits; %>
 9: <%! int count; %>
10:
11: <%
12: visits = new Counter(application.getRealPath("counter.dat"));
13: count = visits.getCount() + 1;
14: %>
15:
16: <p>This page has been loaded <%= count %> times.
17:
18: <% visits.setCount(count); %>
19: </body>
20: </html>
```

Before you can try this page, you need to create a helper class that's called by statements in lines 8, 12, 13, and 18 of the page.

The `Counter` class in Listing 21.8 represents a Web counter that tallies each hit to a page.

LISTING 21.8 The Full Text of `Counter.java`

```
package counter;

import java.io.*;
import java.util.*;

public class Counter {
    private int count;
    private String filepath;

    public Counter(String inFilepath) {
```

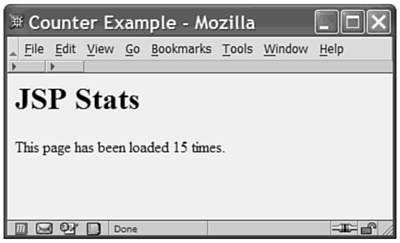
LISTING 21.8 continued

```
        count = 0;
        filepath = inFilepath;
    }

    public int getCount() {
        try {
            File countFile = new File(filepath);
            FileReader file = new FileReader(countFile);
            BufferedReader buff = new BufferedReader(file);
            String current = buff.readLine();
            count = Integer.parseInt(current);
            buff.close();
        } catch (IOException e) {
            // do nothing
        } catch (NumberFormatException nfe) {
            // do nothing
        }
        return count;
    }

    public void setCount(int newCount) {
        count = newCount;
        try {
            File countFile = new File(filepath);
            FileWriter file = new FileWriter(countFile);
            BufferedWriter buff = new BufferedWriter(file);
            String output = "" + newCount;
            buff.write(output, 0, output.length());
            buff.close();
        } catch (IOException e) {
            // do nothing
        }
    }
}
```

After you compile this class successfully, it should be stored in a `WEB-INF\classes\counter` subfolder in the same part of the `webapps` hierarchy as `counter.jsp`. For example, if the JavaServer Page is in `webapps\examples\jsp`, the class file should be saved in `webapps\examples\WEB-INF\classes\counter`.

The `Counter` class loads an integer value from a file called `counter.dat` that is stored on the Web server. The `getCount()` method retrieves the current value of the counter, and the `setCount(`*`int`*`)` method sets the current value. After the value is set, it's saved to the file so that the counter continues to incrementally increase.

Figure 21.7 shows `counter.jsp` loaded in a Web browser.



Counter Example - Mozilla

JSP Stats

This page has been loaded 15 times.

FIGURE 21.7
Using Java ServerPages and Java objects to count visits to a Web page.

Creating a Web Application

By combining Java classes, servlets, and JavaServer Pages, you can create interactive Web applications—sites that dynamically generate content in response to user input in a sophisticated, cohesive way.

Every time you shop on an e-commerce site such as Amazon.com or use an online reference such as the Internet Movie Database (IMDB), you are running a Web application.

To see how several aspects of Java technology can work together on the Web, you create `Guestbook`, a Web application that enables visitors to leave a message for the creator of a site.

The `Guestbook` project is made up of three things:

- `guestbook.jsp`—A JavaServer Page that displays guest book entries from a text file on a Web server and provides a form where a visitor can add an entry
- `guestbookpost.jsp`—A JavaServer Page that saves a new guest book entry to the text file
- `Guestbook.java`—A class used to filter out some characters before they are saved in the guest book

The JavaServer Pages in this project make heavy use of scriptlets and expressions. Listing 21.9 contains the source code for `guestbook.jsp`.

LISTING 21.9 The Full Text of `guestbook.jsp`

```
<%@ page import="java.util.*,java.io.*" %>
<html>
<head>
<title>Visitors Who Signed our Guestbook</title>
</head>
<body>
<h3>Visitors Who Signed our Guestbook</h3>
<%
String id = request.getParameter("id");
```

LISTING 21.9 continued

```
boolean noSignatures = true;
try {
    String filename = application.getRealPath(id + ".gbf");
    FileReader file = new FileReader(filename);
    BufferedReader buff = new BufferedReader(file);
    boolean eof = false;
    while (!eof) {
        String entry = buff.readLine();
        if (entry == null)
            eof = true;
        else {
            StringTokenizer entryData = new StringTokenizer(entry, "^");
            String name = (String) entryData.nextElement();
            String email = (String) entryData.nextElement();
            String url = (String) entryData.nextElement();
            String entryDate = (String) entryData.nextElement();
            String ip = (String) entryData.nextElement();
            String comments = (String) entryData.nextElement();
            out.print("<p>From: " + name);
            if (!email.equals("None"))
                out.println(" <" + email + "><br>");
            else
                out.println("<br>");
            if (!url.equals("None"))
                out.println("Home Page: <a href=\"" + url + "\">" +
                    url + "</a><br>");
            out.println("Date: " + entryDate + "<br>");
            out.println("IP: " + ip);
            out.println("<blockquote>");
            out.println("<p>" + comments);
            out.println("</blockquote>");
            noSignatures = false;
        }
    }
    buff.close();
} catch (IOException e) {
    out.println("<p>This guestbook could not be read because of an error.");
    log("Guestbook Error: " + e.toString());
}
if (noSignatures)
    out.println("<p>No one has signed our guestbook yet.");
%>
<h3>Sign Our Guestbook</h3>
<form method="POST" action="guestbookpost.jsp">
  <table border="0" cellpadding="5" cellspacing="0" width="100%">
    <tr>
      <td width="15%" valign="top" align="right">Your Name:</td>
      <td width="50%"><input type="text" name="name" size="40"></td>
```

LISTING 21.9 continued

```
    </tr>
    <tr>
      <td width="15%" valign="top" align="right">Your E-mail Address:</td>
      <td width="50%"><input type="text" name="email" size="40"></td>
    </tr>
    <tr>
      <td width="15%" valign="top" align="right">Your Home Page:</td>
      <td width="50%"><input type="text" name="url" size="40"></td>
    </tr>
    <tr>
      <td width="15%" valign="top" align="right">Your Comments:</td>
      <td width="50%">
        <textarea rows="6" name="comments" cols="40"></textarea>
      </td>
    </tr>
  </table>
  <p align="center"><input type="submit" value="Submit" name="B1">
  <input type="reset" value="Reset" name="Reset"></p>
<input type="hidden" name="id" value="<%= id %>">
</form>
</body>
</html>
```

After you save this page, store it in any folder on your server where JavaServer Pages can be stored. You can test this even before anything else in the project is done, as long as you have an empty guest book file.

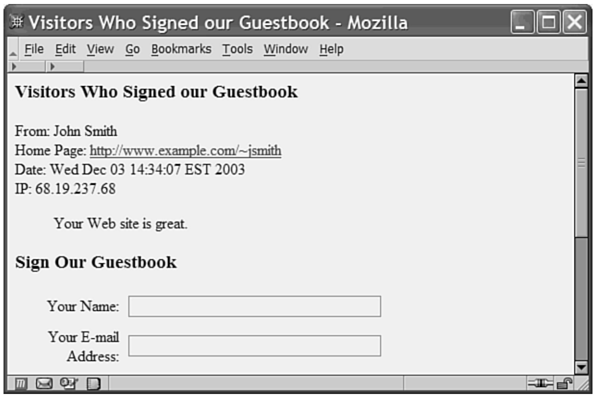
To create this file, save an empty text file on your system and give it the name `cinema.gbf`. Store it on the Web in the `webapps` folder relative to where the guest book page has been stored. For instance, if the page is in `webapps\examples\jsp\guestbook.jsp`, the text file should be saved in `\webapps\examples`.

When you load this JavaServer Page, you must include a parameter that specifies the ID of the guest book to load, as in this URL:

```
http://www.java21days.com:8080/examples/jsp/guestbook.jsp?id=cinema
```

The server name and folder depend on where you have published `guestbook.jsp`.

Figure 21.8 shows what your guest book looks like when your JavaServer Page compiles successfully and tries to display the contents of the `cinema.gbf` file.


Visitors Who Signed our Guestbook - Mozilla
File Edit View Go Bookmarks Tools Window Help

Visitors Who Signed our Guestbook

From: John Smith
Home Page: http://www.example.com/~jsmith
Date: Wed Dec 03 14:34:07 EST 2003
IP: 68.19.237.68

Your Web site is great.

Sign Our Guestbook

Your Name:
Your E-mail Address:

FIGURE 21.8
Testing the `guestbook.jsp` *page.*

The guest book file stores each guest book entry on its own line, with a caret ("^") separating each field in the entry. Visitors can provide their name, email address, home page address, and a comment. Two other things are saved for each entry: the date and time it was written and the IP address of the visitor.

The following text is an example of a guest book file that contains two entries:

```
John Smith^jsmith@example.com^http://www.example.com/~jsmith^Wed Dec 03
➥14:34:07 EST 2003^68.19.237.68^Your Web site is great.
D. James^deejay@example.com^http://www.imdb.com^Wed Dec 03 14:36:38 EST
➥2003^68.19.237.68^Thanks for the information.
```

The next JavaServer Page to create is `guestbookpost.jsp`, the page that updates the guest book with new entries submitted by visitors. Listing 21.10 contains the source code for this JavaServer Page.

LISTING 21.10 The Full Text of `guestbookpost.jsp`

```
<%@ page import="java.util.*,java.io.*,example.*" %>
<html>
<head>
<title>Thank You For Signing Our Guestbook</title>
</head>
<body>
<h3>Thank You For Signing Our Guestbook</h3>
<%
String id = request.getParameter("id");
String[] entryFields = { "name", "email", "url", "comments" };
String[] entry = new String[4];
for (int i = 0; i < entryFields.length; i++) {
    entry[i] = Guestbook.filterString(request.getParameter(entryFields[i]));
}
```

LISTING 21.10 continued

```
Date now = new Date();
String entryDate = now.toString();
String ip = request.getRemoteAddr();
%>

<p>Your entry looks like this:
<p>From: <%= entry[0] %>
<%= (!entry[1].equals("None") ? "<"+entry[1]+">" : "") %><br>
<% if (!entry[2].equals("None")) { %>
Home Page: <a href="<%= entry[2] %>"><%= entry[2] %></a><br>
<% } %>
Date: <%= entryDate %><br>
IP: <%= ip %>
<blockquote>
<p><%= entry[3] %>
</blockquote>

<%
try {
    boolean append = true;
    String filename = application.getRealPath(id + ".gbf");
    FileWriter fw = new FileWriter(filename, append);
    BufferedWriter fileOut = new BufferedWriter(fw);
    String newEntry = entry[0] + "^" + entry[1] + "^" + entry[2] + "^"
        + entryDate + "^" + ip + "^" + entry[3];
    fileOut.write(newEntry, 0, newEntry.length());
    fileOut.newLine();
    fileOut.close();
} catch (IOException e) {
    out.println("<p>This guestbook could not be updated.");
    log("Guestbook Error: " + e.toString());
}
%>

<p><a href="guestbook.jsp?id=<%= id %>">View the Guestbook</a>
</body>

</html>51: </html>
```

The `guestbookpost.jsp` JavaServer Page collects data from a Web form, removes characters from the data that can't be put in the guest book, and stores the result in a text file.

Each guest book has its own file with a name that begins with the ID parameter of the book and ends with the `.gbf` file extension. If the guest book has the ID of `cinema`, the filename is `cinema.gbf`.

Like the other JavaServer Page included in this Web application, `guestbookpost.jsp` can be stored in any folder on your server where JavaServer Pages are kept. For this project, store the page in the same folder as `guestbook.jsp` and `cinema.gbf`.

Before you can try the `Guestbook` application, you must create a Java class that is used to filter some unwanted text from guest book entries before they are posted.

Three characters cannot be included in the guest book because of the way entries are stored in a file:

- Caret characters ("^")
- Return characters, which have the integer value of 13 in Java
- Linefeed characters, which have the integer value of 10

To remove these characters before they are saved in a guest book, a helper class called `Guestbook` is created. This class has a static method called `filterString(`*`String`*`)` that removes those three characters from a string.

Listing 21.11 contains the source code for this class.

LISTING 21.11 The Full Text of `Guestbook.java`

```java
package example;

public class Guestbook {
    public static String filterString(String input) {
        input = replaceText(input, '^', ' ');
        input = replaceText(input, (char)13, ' ');
        input = replaceText(input, (char)10, ' ');
        return input;
    }

    private static String replaceText(String inString, char oldChar,
        char newChar) {

        while (inString.indexOf(oldChar) != -1) {
            int oldPosition = inString.indexOf(oldChar);
            StringBuffer data = new StringBuffer(inString);
            data.setCharAt(oldPosition, newChar);
            inString = data.toString();
        }
        return inString;
    }
}
```

The `replaceText()` method in lines 11–21 of Listing 21.11 does most of the work in the class. It takes three arguments:

- A string that might contain unwanted characters
- A character that should be removed
- A character that should be added in its place

When you compile the `Guestbook` class, it should be stored in a `WEB-INF\classes\guestbook` subfolder in the same part of the `webapps` hierarchy as the project's JavaServer Pages. For example, if the pages are in `webapps\examples\jsp`, the class file should be saved in `webapps\examples\WEB-INF\classes\guestbook`.

To test the `guestbookpost.jsp` server page, open the page that displays guest book entries using an `ID` parameter of `cinema` again, as in this example:

```
http://www.java21days.com:8080/java/guestbook.jsp?id=cinema
```

Add a few guest book entries to see how they are displayed in the guest book.

JSP Standard Tag Library

For Java programmers eager to apply their skills on the World Wide Web, the development of the language for server-side programming has been one of those "walk before you can crawl" situations.

When servlets were introduced in 1997, they made it possible to use Java to create programs similar to Common Gateway Interface scripts that collect input from Web forms and URL parameters, producing HTML output in response. This works great for writing programs to process mail and handle other simple tasks, but because larger Web applications require multiple pages, it becomes clear that producing HTML output using Java statements can be cumbersome. Revisions become more difficult, especially for any non-programmers involved in the work.

Two years later, Sun took another step in the right direction with JavaServer Pages (JSP), which make it easy to combine static HTML output with dynamic output created by Java statements. Using JSP, Java code can be placed on Web pages among HTML markup and edited like any other page. Programmers also can create their own custom JSP markup tags to interact with Java objects. The pages compile into servlets automatically.

Unfortunately, the ease with which Java code could be placed on a JavaServer Page proved to be a misstep, because it encourages the bad habit of placing a lot of mission-critical application code in a place where it's difficult to maintain, insecure, and easily bungled by anyone editing the HTML around the code. For example, statements to create, open, and query a database connection, complete with usernames and passwords to gain access to the data, can be put on a JavaServer Page.

A giant step has been taken with the release of the JSP Standard Tag Library (JSTL), a set of custom JSP tags and a new data-access language that enable JSP-based Web applications to handle presentation without ever resorting to Java code.

Tag libraries, which also are called *taglibs*, consist of tags placed on a JavaServer Page in a format similar to HTML tags. JSTL's tag library offers functionality common to most Web applications. There are tags for loops and conditionals, tags to read XML documents and database results using SQL, tags for accessing JavaBeans, and support for other tag libraries.

JSTL 1.1, which requires a Java servlet container supporting Java Servlet 2.4 and JavaServer Pages 2.0, can be found at the home page `http://java.sun.com/products/jsp/jstl`.

The project will eventually be supported natively by servlet containers. For now, it can be implemented in minutes: Copy the Java archive files `jstl.jar` and `standard.jar` to the `WEB-INF/lib` folder of a Web application.

NOTE

The Java 2 Enterprise Edition (J2EE) also includes support for JSTL 1.1, but it's packaged a bit differently. Look for the file `appserv-jstl.jar` in the `lib` subfolder of your J2EE installation. It includes all the files offered elsewhere as `jstl.jar` and `standard.jar`.

A reference implementation of JSTL created by the Apache Jakarta open source Java project can be downloaded from `http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html`. To differentiate it from other tag libraries, Jakarta calls JSTL the "standard" library.

JSTL consists of two complementary components:

- A set of five custom JSP tag libraries that provide the core functionality required in Web applications
- The JSTL Expression Language, which makes it possible to access data at runtime without using JSP expressions or Java statements

There's also a separate version of each JSTL library that works with Java expressions using existing JSP syntax.

Listing 21.12 contains a demonstration of JSTL and the Expression Language: `hello.jsp`, a JavaServer Page that uses a parameter called `name` as part of a greeting.

LISTING 21.12 The Full Text of `hello.jsp`

```
<html>
<head>
   <title>Hello Example</title>
</head>

<body>
<%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

<c:choose>
   <c:when test='${!empty param.name}'>
      <h2>Hello, <c:out value='${param.name}'/></h2>
   </c:when>
   <c:otherwise>
      <h2>Hello, stranger</h2>
   </c:otherwise>
</c:choose>

</body>
</html>
```

The `hello.jsp` page looks for a parameter called `name` specified in the address (URL) of the page. For example, the URL `http://example.com/hello.jsp?name=Sailor` gives `name` the value `Sailor` and causes the page to display the response “Hello, Sailor.” When the name parameter is omitted, “Hello, Stranger” displays instead.

The first JSP code in the page is the following directive:

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

Like other tag libraries, each JSTL library must be made available using a directive on the page before any of its tags can be employed. The preceding directive makes the core library available on the page. All tags from this library are prefaced with the text “c:” followed by the tag’s name, as in this example:

```
<c:out value='${param.name}'/>
```

JSTL tags are called *actions*, in recognition of the fact that they generate dynamic Web content. Like XML elements, actions can stand alone (`<`*`tagname`*`/>`) or be paired (`<tagname>`…`</`*`tagname`*`>`).

The `c:out` action displays the contents of the local or instance variable indicated by its `value` attribute.

The variable is specified using JSTL's Expression Language, a simple data-access syntax that borrows from ECMAScript (JavaScript) and XPath. Statements that use the language are contained within "${" and "}" characters.

In the preceding example, the variable `param` is one of several standard Expression Language variables that contain information about the page, Web application, and servlet container. The `param` variable is a collection that holds all the page parameters, each represented as a string.

The rest of the page contains three core actions used to create a conditional block:

```
<c:choose>
   <c:when test='${!empty param.name}'>
      <h2>Hello, <c:out value='${param.name}'/></h2>
   </c:when>
   <c:otherwise>
      <h2>Hello, stranger</h2>
   </c:otherwise>
</c:choose>
```

The `c:choose-c:when-c:otherwise` block mirrors the functionality of a `switch-case-default` Java statement, displaying enclosed HTML output only for the first `c:when` action that has a `test` attribute with the value `true`. If none of the actions is `true`, the `c:otherwise` contents are displayed.

JSTL is composed of five tag libraries:

- The core library (prefix `c`, default URI `http://java.sun.com/jstl/core`) contains general features: output, conditional display of content, looping, variable creation in several scopes, JavaBeans access, exception handling, URL imports, and URL redirection.
- The SQL library (prefix `sql`, default URI `http://java.sun.com/jstl/sql`) covers database access: data source selection, queries, updates, transactions, and looping through results.
- The internationalization and formatting library (prefix `fmt`, default URI `http://java.sun.com/jstl/fmt`) offers these actions: locale and resource bundle use, text localization, and number and date formatting.
- The XML processing library (prefix `x`, default URI `http://java.sun.com/jstl/xml`) supports XML: parsing, XPath access, XSLT transformation, looping through nodes, and conditional processing.

- The function library (prefix `fn`, default URI `http://java.sun.com/jstl/functions`) contains useful functions to manipulate strings and collections.

The Expression Language has operators to retrieve information from instance variables (`${object.varName}`), data structures (`${object["name"]}`), and indexed arrays or lists (`${object[1]}`).

There also are operators for arithmetic (`+`, `-`, `*`, `/`, `%`), comparisons (`==`, `!=`, `<`, `<=`, `>`, `>=`), logic (`&&`, `||`, `!`), and `empty`, which detects `null` objects and empty strings or collections. Parentheses can be used to group subexpressions.

The language offers automatic type conversion and five kinds of literals: strings, which are enclosed within single or double quotes; integers, floating-point numbers, boolean values (`true` or `false`), and `null`.

Nine special variables are available in any expression:

- `cookie`—A collection of all request cookies, each as an instance of the `Cookie` class from the `javax.servlet.http` package
- `header`—A collection of all request headers, each as a string
- `headerValues`—A collection of all request headers, each as a string array
- `initParam`—A collection of all application initialization parameters, each as a string
- `pageContext`—An instance of `jspPageContext` from the `javax.servlet` package
- `param`—A collection of all request parameters, each as a string
- `paramValues`—A collection of all request parameters, each as a string array

When using one of these collections or other data structures, the `c:foreach` action makes it easy to loop through each element. The following example displays all the header variables sent with a JSP page:

```
<c:forEach items='${header}' var='head'>
   <ul>
      <li>Name: <c:out value='${head.key}'/></li>
      <li>Value: <c:out value='${head.value}'/></li>
   </ul>
</c:forEach>
```

Four variables can be used to make explicit references to variable scope:

- `applicationScope`—A collection of all application scope objects
- `pageScope`—A collection of all page scope objects
- `requestScope`—A collection of all request scope objects

- `sessionScope`—A collection of all session scope objects

For example, the expression `${sessionScope.price}` retrieves an object named `price` in session scope. If no other scope has an object named `price`, the expression `${price}` also works.

The day's final project shows some of the power and flexibility of JSTL. This page uses XML actions to present data from an RSS newsfeed, an XML format for offering Web content in machine-readable form.

JSTL can import and parse XML, HTML, or any other data available at a URL, even if it isn't on the same server as the Java Server Page using it.

RSS, which stands for Really Simple Syndication, enables Web sites to share headlines, links, and other content with each other (and with readers using software called an RSS aggregator). Listing 21.13 contains an example: a simplified version of the RSS feed for the SportsFilter Web site.

LISTING 21.13 The Full Text of `sportsfilter.rss`

```
<rss version="0.91">
  <channel>
  <title>SportsFilter</title>
  <link>http://www.sportsfilter.com/</link>
  <item>
    <title>Babe Ruth used steriods?</</title>
    <link>http://sports.espn.go.com/mlb/news/story?id=1745899</link>
    <description>Houston Astros second baseman Jeff Kent said the
steroids controversy is an embarrassment to baseball and that the
public needs to rethink whether sports heroes of yore abstained
from illegal performance-enhancing drugs.</description>
  </item>
  <item>
    <title>Phoenix sports fans shells out the bucks</title>
    <link>http://washpost.com/wp-dyn/articles/A10394-2004Feb26.html</link>
    <description>Phoenix-area taxpayers have invested $700 million
in new stadiums for their pro baseball, basketball, football, and
hockey franchises, a world record for governmental sports support,
as described in today's Washington Post.</description>
  </item>
  <item>
    <title>Just buy it</title>
    <link>http://www.nike.com/nikebiz/news/pressrelease.jhtml</link>
    <description>Marion Jones competing in floor gymnastics, Randy
Johnson bowling, Lance Armstrong in the boxing ring, Andre Agassi
fielding 2B for the Red Sox. Nike rolls out their spring campaign
asking in some rather creatively edited commercial spots starting
tonight.</description>
```

LISTING 21.13 continued

```
  </item>
</channel>
</rss>
```

SportsFilter, a weblog at `http://www.sportsfilter.com`, shares the latest news in an RSS feed. The XML data from the feed can be read into a variable using the `c:import` action from the core library and parsed with the `x:parse` action from the XML library.

The parsed data can be examined, filtered, transformed, and displayed. Listing 21.14 contains a JavaServer Page application that uses simple XPath statements to extract and display parts of the XML data.

LISTING 21.14 The Full Text of `rss.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <!-- Declare the two tag libraries used on this page -->
  <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>
  <%@ taglib uri='http://java.sun.com/jstl/xml' prefix='x' %>

<!-- Import the RSS feed at the URL specified by feed -->
<!-- For example -->
<!-- http://sportsfilter.com/rss.cfm -->
<c:import url='${param.feed}' var='feedData'/>

<!-- Parse the RSS feed -->
<x:parse xml='${feedData}' var='feed'/>

<!-- Retrieve the channel element -->
<x:set select='$feed//rss//channel' var='channel'/>

<!-- Display the channel element's title -->
  <title><x:out select='$channel//title'/></title>
</head>

<body>

<!-- Display the channel element's link and title -->
<p>Headlines from <a href="<x:out select='$channel//link'/>">
<x:out select='$channel//title'/></a>

<!-- Loop through the item elements -->
<p><ul>
<x:forEach select='$feed//rss//channel//item'>
```

LISTING 21.14 continued

```
   <!-- Display each element's link, title, description -->
   <!-- Descriptions may contain HTML. -->
   <li><a href="<x:out select='link'/>"><x:out select='title'/></a>:
<x:out select='description' escapeXml='false'/></li>
   <br>
</x:forEach>
</ul>

</body>
</html>
```

Comments on the page describe the JSTL actions that it contains.

The most challenging part of JSTL is the Expression Language, which can be avoided by using versions of the tag libraries that use JSP syntax for Java statements and expressions.

JSTL's Expression Language is distinguished from the other JSP syntax by the limits on what it can do. There are no assignment operators or conditional logic, which forces developers to use external Java classes and JSTL actions for these tasks.

Although some Java programmers may blanch at the thought of learning another language simply for Web applications, the Expression Language is simple enough to be picked up quickly, especially by those comfortable with JavaScript.

By keeping Java code out of JavaServer Pages, the Expression Language offers reusability and reliability. In conjunction with JSTL, it brings JSP much closer to the promise of separating the presentation of a Web application from the code required to make it happen.

JSTL serves as a nice complementary offering to Struts, an open-source Web application framework that's also the work of the Apache Jakarta project.

CAUTION

Programmers who have embraced the model-view-controller philosophy embodied by Struts may question JSTL's inclusion of database actions that don't belong in an application's presentation layer.

For anything but trivial Web applications, programmers should consider putting database access behavior in classes accessed from JSP rather than using JSTL's SQL actions.

Summary

At this point, you now have three different ways to use the Java language on the World Wide Web: applets, servlets, and JavaServer Pages.

The main purpose of the classes in the `javax.servlet` and `javax.servlet.http` packages is to exchange information with a Web server. Java servlets are an alternative to the Common Gateway Interface, the most popular way that programming languages are used to retrieve and present data on the Web.

Because servlets can use all features of the Java language with the exception of a graphical user interface, you can use them to create sophisticated Web applications.

JavaServer Pages are an effective way to separate static content on Web pages from the dynamic content generated by servlets for those pages. By using expressions, statements, and declarations, you can write Java programs on server pages without ever needing to compile a program, lay it out on a Web page, design an interface, or publish class files.

The Java Standard Tag Library (JSTL) and its Expression Language take the separation of content and code a step further, offering a way for Web applications to use Java within the same tag-based context used in HTML and XML.

For the last three weeks, you've had a chance to work with the syntax and the core classes that make up the Java language and the Java 2 class library. You've ventured out into some of the excellent class libraries offered by other companies and developers such as XOM, the XML Object Model library, and JSTL.

You are now ready to tackle the biggest challenge yet: Turning an empty source code file into a robust and reliable program implemented as a set of Java classes.

This book has an official Web site at `http://www.java21days.com` with answers to frequently asked questions, source code for the entire book, error corrections, and supplementary material. There's also a weblog, Workbench, for people who are teaching themselves Java 365 days a year.

Happy compiling!

Q&A

Q Is there a way to make a Java applet communicate with a servlet?

A If you want the applet to continue running after it contacts the servlet, the servlet must be on the same machine as the Web page that contains the applet. For security reasons, applets cannot make a network connection to any machine other than the one that hosts the applet.

If you want an applet to load a servlet in the Web browser, you can call the applet's `getAppletContext()` method to get an `AppletContext` object, and then call that object's `showDocument(`*`URL`*`)` method with the servlet's URL as the argument.

Q Why do servlets and JavaServer Pages require the `getRealPath()` method to determine where a file is located on the Web server? Can't you store the file in the same folder as a servlet and use the filename without referring to a path?

A Tomcat doesn't support relative filenames inside servlets or JavaServer Pages. You must know the exact location of a file on the Web server to read or write data in the file. Because this information isn't always available to you in a live Web-hosting environment, the `ServletContext` interface includes the `getRealPath()` method. This method asks the Web server for the full pathname of a file. One of the biggest advantages of using Tomcat rather than Common Gateway Interface scripts is that you can communicate directly with the server.

In the `counter.jsp` example earlier today, the `counter.dat` file was created in the same folder where `counter.jsp` is stored. Tomcat doesn't store files in the same folder as servlets.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. If a servlet is run at the same time by five Web users, how many times is the servlet's `init()` method called?
   a. 5
   b. 1
   c. 0–1
2. If you see a `request` variable on a JavaServer Page, what class in `javax.servlets.http` is it referring to?
   a. `HttpServletResponse`
   b. `HttpServletRequest`
   c. `ServletContext`
3. Which of the JavaServer Pages elements uses the `<%=` and `%>` tags?
   a. Declarations
   b. Expressions
   c. Statements

Answers

1. c. The `init()` method is called when the Web server first loads the servlet. This may have taken place before all five of these users requested the servlet, so it could call `init()` one time or not at all.
2. b. `HttpServletRequest`
3. b. The expression inside the tags will be evaluated and its value will be displayed on the page at the expression's location.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

```
public class CharCase {
    public static void main(String[] arguments) {
        float x = 9;
        float y = 5;
        char c = '1';
        switch (c) {
            case 1:
                x = x + 2;
            case 2:
                x = x + 3;
            default:
                x = x + 1;
        }
        System.out.println("Value of x: " + x);
    }
}
```

What will be the value of x when it is displayed?

a. 9.0
b. 10.0
c. 11.0
d. The program will not compile.

The answer is available on the book's Web site at `http://www.java21days.com`. Visit the Day 21 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create a servlet that stores the data entered in a form in a file.
2. Write a JavaServer Page that displays one greeting for Internet Explorer users and another greeting for everyone else.

Where applicable, exercise solutions are offered on the book's Web site at `http://www.java21days.com`.