Chapter 20. JavaServer Pages

Topics in This Chapter

- The benefits of JSP
- JSP expressions, scriptlets, and declarations
- Controlling the structure of the servlet that results from a JSP page
- Including files and applets in JSP documents
- Using JavaBeans with JSP
- Creating custom JSP tag libraries
- Combining servlets and JSP: the Model View Controller (Model 2) architecture

20.1 JSP Overview

JavaServer Pages (JSP) technology enables you to mix regular, static HTML with dynamically generated content from servlets. You simply write the regular HTML in the normal manner, using familiar Web-page-building tools. You then enclose the code for the dynamic parts in special tags, most of which start with <% and end with %>. For example, here is a section of a JSP page that results in "Thanks for ordering *Core Web Programming*" for a URL of *http://host/OrderConfirmation.jsp?title=Core+Web+Programming:*

```
Thanks for ordering <I><%= request.getParameter("title") %></I>
```

Separating the static HTML from the dynamic content provides a number of benefits over servlets alone, and the approach used in JavaServer Pages offers several advantages over competing technologies such as ASP, PHP, or ColdFusion. Section 20.2 gives some details on these advantages, but they basically boil down to two facts: that JSP is widely supported and thus doesn't lock you into a particular operating system or Web server and that JSP gives you full access to Java servlet technology for the dynamic part, rather than requiring you to use an unfamiliar and weaker special-purpose language.

The process of making JavaServer Pages accessible on the Web is much simpler than that for servlets. Assuming you have a Web server that supports JSP, you give your file a `.jsp` extension and simply install it in any of the designated JSP locations (which, on many servers, is any place you could put a normal Web page): no compiling, no packages, and no user `CLASSPATH` settings. However, although your *personal* environment doesn't need any special settings, the *server* still has to be set up with access to the servlet and JSP class files and the Java compiler. For details, see your server's documentation or Section 19.2 (Server Installation and Setup).

Although what you write often looks more like a regular HTML file than a servlet, behind the scenes the JSP page is automatically converted to a normal servlet, with the static HTML simply being printed to the output stream associated with the servlet. This translation is normally done the first time the page is requested. To ensure that the first real user doesn't get a momentary delay when the JSP page is translated into a servlet and compiled, developers can simply request the page themselves after first installing it.

One warning about the automatic translation process is in order. If you make an error in the dynamic portion of your JSP page, the system may not be able to properly translate it into a servlet. If your page has such a fatal translation-time error, the server will present an HTML error page describing the problem to the client. Internet Explorer 5, however, typically replaces server-generated error messages with a canned page that it considers friendlier. You will need to turn off this "feature" when debugging JSP pages. To do so with Internet Explorer 5, go to the Tools menu, select Internet Options, choose the Advanced tab, and make sure "Show friendly HTTP error messages" box is not checked.

Core Approach

When debugging JSP pages, be sure to turn off Internet Explorer's "friendly" HTTP error messages.

Aside from the regular HTML, there are three main types of JSP constructs that you embed in a page: *scripting elements, directives,* and *actions.* Scripting elements let you specify Java code that will become part of the resultant servlet, directives let you control the overall structure of the servlet, and actions let you specify existing components that should be used and otherwise control the behavior of the JSP engine. To simplify the scripting elements, you have access to a number of predefined variables, such as `request` in the code snippet just shown.

This chapter covers versions 1.0 and 1.1 of the JavaServer Pages specification. JSP changed dramatically from version 0.92 to version 1.0, and although these changes are very much for the better, you should note that newer JSP pages are almost totally incompatible with the early 0.92 JSP engines and that older JSP pages are equally incompatible with 1.0 JSP engines. The changes from version 1.0 to 1.1 are much less dramatic: the main additions in version 1.1 are the ability to portably define new tags and the use of the servlet 2.2 specification for the underlying servlets. JSP 1.1 pages that do not use custom tags or explicitly call 2.2-specific statements are compatible with JSP 1.0 engines; JSP 1.0 pages are totally upward compatible with JSP 1.1 engines.

20.2 Advantages of JSP

JSP has a number of advantages over many of its alternatives. Here are a few of them.

Versus Active Server Pages (ASP) or ColdFusion

ASP is a competing technology from Microsoft. The advantages of JSP are twofold. First, the dynamic part is written in Java, not VBScript or another ASP-specific language, so it is more powerful and better suited to complex applications that require reusable components. Second, JSP is portable to other operating systems and Web servers; you aren't locked into Windows NT/2000 and IIS. You could make the same argument when comparing JSP to ColdFusion; with JSP you can use Java for the "real code" and are not tied to a particular server product.

Versus PHP

PHP is a free, open-source HTML-embedded scripting language that is somewhat similar to both ASP and JSP. One advantage of JSP is that the dynamic part is written in Java, which already has an extensive API for networking, database access, distributed objects, and the like, whereas PHP requires learning an entirely new, less widely used language. A second advantage is that JSP is much more widely supported by tool and server vendors than is PHP.

Versus Pure Servlets

JSP doesn't provide any capabilities that couldn't, in principle, be accomplished with a servlet. In fact, JSP documents are automatically translated into servlets behind the scenes. But it is more convenient to write (and to modify!) regular HTML than to have a zillion `println` statements that generate the HTML. Plus, by separating the presentation from the content, you can put different people on different tasks: your Web page design experts can build the HTML using familiar tools and leave places for your servlet programmers to insert the dynamic content.

Does this mean that you can just learn JSP and forget about servlets? By no means! JSP developers need to know servlets for four reasons:

1. JSP pages get translated into servlets. You can't understand how JSP works without understanding servlets.

2. JSP consists of static HTML, special-purpose JSP tags, and Java code. What kind of Java code? Servlet code!

3. Some tasks are better accomplished by servlets than by JSP. JSP is good at generating pages that consist of large sections of fairly well structured HTML or other character data. Servlets are better for generating binary data, building pages with highly variable structure, and performing tasks (such as redirection) that involve little or no output.

4. Some tasks are better accomplished by a *combination* of servlets and JSP than by *either* servlets or JSP alone. See Section 20.8 (Integrating Servlets and JSP) for details.

Versus Server-Side Includes (SSI)

SSI is a widely supported technology for inserting externally defined pieces into a static Web page. JSP is better because you have a richer set of tools for building that external piece and have more options regarding the stage of the HTTP response at which the piece actually gets inserted. Besides, SSI is really intended only for simple inclusions, not for "real" programs that use form data, make database connections, and the like.

Versus JavaScript

JavaScript, which is completely distinct from the Java programming language, is normally used to generate HTML dynamically on the *client,* building parts of the Web page as the browser loads the document. This is a useful capability and does not normally overlap with the capabilities of JSP (which runs only on the server). Although JavaScript can also be used on the server, most notably on Netscape, IIS, and BroadVision, Java is more powerful, flexible, reliable, and portable.

20.3 JSP Scripting Elements

JSP scripting elements let you insert code into the servlet that will be generated from the JSP page. There are three forms:

1. *Expressions* of the form `<%= expression %>`, which are evaluated and inserted into the servlet's output

2. *Scriptlets* of the form `<% code %>`, which are inserted into the servlet's `_jspService` method (called by `service`)

3. *Declarations* of the form `<%! code %>`, which are inserted into the body of the servlet class, outside of any existing methods

Each of these scripting elements is described in more detail in the following sections.

In many cases, a large percentage of your JSP page just consists of static HTML, known as *template text.* In almost all respects, this HTML looks just like normal HTML, follows all the same syntax rules, and is simply "passed through" to the client by the servlet created to handle the page. Not only does the HTML look normal, it can be created by whatever tools you already are using for building Web pages. For example, we used Allaire's HomeSite for most of the JSP pages in this book.

There are two minor exceptions to the "template text is passed straight through" rule. First, if you want to have `<%` in the output, you need to put `<\%` in the template text. Second, if you want a comment to appear in the JSP page but not in the resultant document, use

```
<%-- JSP Comment --%>
```

HTML comments of the form

```
<!-- HTML Comment -->
```

are passed through to the resultant HTML normally.

Expressions

A JSP expression is used to insert values directly into the output. It has the following form:

```
<%= Java Expression %>
```

The expression is evaluated, converted to a string, and inserted in the page. That is, this evaluation is performed at run time (when the page is requested) and thus has full access to information about the request. For example, the following shows the date/time that the page was requested:

```
Current time: <%= new java.util.Date() %>
```

It would result in code in the `_jspService` method (called by `service` in servlets that result from JSP pages) similar to the following.

```
out.print("Current time: ");
out.println(new Java.util.Date());
```

Predefined Variables

To simplify these expressions, you can use a number of predefined variables. These implicit objects are discussed in more detail later in this section, but for the purpose of expressions, the most important ones are:

- **request**, the `HttpServletRequest`
- **response**, the `HttpServletResponse`
- **session**, the `HttpSession` associated with the request (unless disabled with the `session` attribute of the `page` directive—see Section 20.4)
- **out**, the `PrintWriter` (a buffered version called `JspWriter`) used to send output to the

client

Here is an example:

```
Your hostname: <%= request.getRemoteHost() %>
```

XML Syntax for Expressions

On some servers, XML authors can use the following alternative syntax for JSP expressions:

```
<jsp:expression>Java Expression</jsp:expression>
```

However, in JSP 1.1 and earlier, servers are not required to support this alternative syntax, and in practice few do. In JSP 1.2, servers are required to support this syntax as long as authors don't mix the XML version and the ASP-like version (`<%= ... %>`) in the same page. Note that XML elements, unlike HTML ones, are case sensitive, so be sure to use `jsp:expression` in lower case.

Installing JSP Pages

Servlets require you to set your `CLASSPATH`, use packages to avoid name conflicts, install the class files in servlet-specific locations, and use special-purpose URLs. Not so with JSP pages. JSP pages can be placed in the same directories as normal HTML pages, images, and style sheets; they can also be accessed through URLs of the same form as those HTML pages, images, and style sheets. Here are a few examples of installation locations and associated URLs:

- **Tomcat default installation directory**

  ```
  install_dir\webapps\ROOT
  install_dir\webapps\ROOT\anyDir
  ```

- **Tomcat URL**

  ```
  http://host/filename.jsp
  http://host/anyDir/filename.jsp
  ```

- **JRun installation directory**

  ```
  install_dir\servers\default\default-app
  install_dir\servers\default\default-app\anyDir
  ```

- **JRun URL**

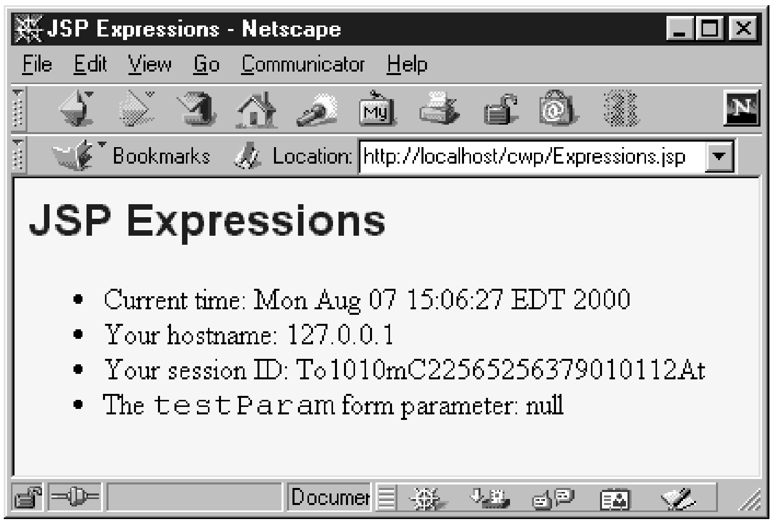
  ```
  http://host/filename.jsp
  http://host/anyDir/filename.jsp
  ```

Note that, although JSP pages *themselves* need no special installation directories, any Java classes called *from* JSP pages still need to go in the standard locations used by servlet classes (e.g., `.../WEB-INF/classes`).

Example: JSP Expressions

Listing 20.1 gives an example JSP page and Figure 20-1 shows the result. With Tomcat we installed the page in `install_dir\webapps\ROOT\cwp\Expressions.jsp` and used a URL of `http://host/cwp/Expressions.jsp`. Notice that we included `META` tags and a style sheet link in the `HEAD` section of the HTML page. It is good practice to include these elements, but there are two reasons why they are often omitted from pages generated by normal servlets. First, with servlets, it is tedious to generate the required `println` statements. With JSP, however, the format is simpler and you can make use of the code reuse options in your usual HTML building tool. Second, servlets cannot use the simplest form of relative URLs (ones that refer to files in the same directory as the current page) since the servlet directories are not mapped to URLs in the same manner as are URLs for normal Web pages. JSP pages, on the other hand, are installed in the normal Web page hierarchy on the server, and relative URLs are resolved properly. Thus, style sheets and JSP pages can be kept together in the same directory. The source code for the style sheet, like all code shown or referenced in the book, can be downloaded from http://www.corewebprogramming.com/.

Figure 20-1. Typical result of `Expressions.jsp`.


JSP Expressions - Netscape
File Edit View Go Communicator Help
Bookmarks
Location: http://localhost/cwp/Expressions.jsp
Documer

JSP Expressions

- Current time: Mon Aug 07 15:06:27 EDT 2000
- Your hostname: 127.0.0.1
- Your session ID: To1010mC2256525637901O112At
- The `testParam` form parameter: null

Listing 20.1 `Expressions.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>JSP Expressions</TITLE>
<META id="keywords"
      CONTENT="JSP,expressions,JavaServer,Pages,servlets">
<META id="description"
      CONTENT="A quick example of JSP expressions.">
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H2>JSP Expressions</H2>
<UL>
  <LI>Current time: <%= new java.util.Date() %>
  <LI>Your hostname: <%= request.getRemoteHost() %>
  <LI>Your session ID: <%= session.getId() %>
  <LI>The <CODE>testParam</CODE> form parameter:
      <%= request.getParameter("testParam") %>
</UL>
</BODY>
</HTML>
```

Scriptlets

If you want to do something more complex than insert a simple expression, JSP scriptlets let you insert arbitrary code into the servlet's `_jspService` method (which is called by `service`). Scriptlets have the following form:

```
<% Java Code %>
```

Scriptlets have access to the same automatically defined variables as expressions (`request`, `response`, `session`, `out`, etc.). So, for example, if you want output to appear in the resultant page, you would use the `out` variable, as in the following example.

```
<%
String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData);
%>
```

In this particular instance, you could have accomplished the same effect more easily by using the following JSP expression:

```
Attached GET data: <%= request.getQueryString() %>
```

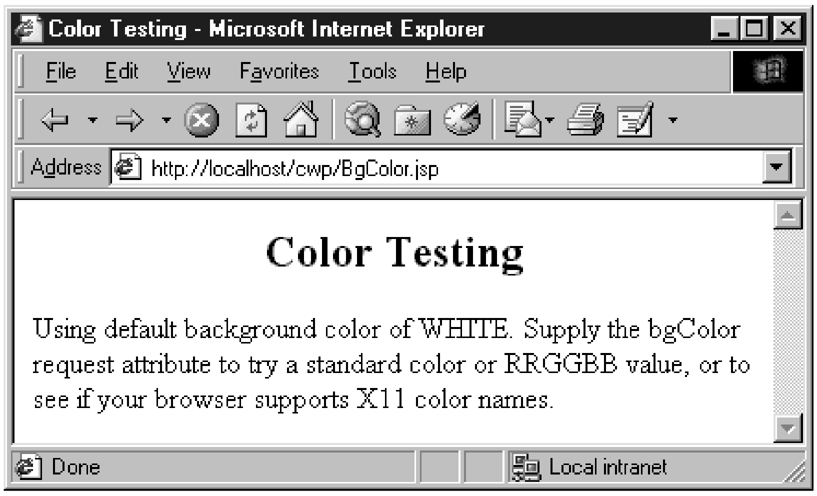
In general, however, scriptlets can perform a number of tasks that cannot be accomplished with expressions alone. These tasks include setting response headers and status codes, invoking side effects such as writing to the server log or updating a database, or executing code that contains loops, conditionals, or other complex constructs. For instance, the following snippet specifies that the current page is sent to the client as plain text, not as HTML (which is the default).

```
<% response.setContentType("text/plain"); %>
```

It is important to note that you can set response headers or status codes at various places within a JSP page, even though this capability appears to violate the rule that this type of response data needs to be specified before any document content is sent to the client. Setting headers and status codes is permitted because servlets that result from JSP pages use a special variety of `Writer` (of type `JspWriter`) that partially buffers the document. This buffering behavior can be changed, however; see Section 20.4 for a discussion of the `buffer` and `autoflush` attributes of the `page` directive.

As an example of executing code that is too complex for a JSP expression, Listing 20.2 presents a JSP page that uses the `bgColor` request parameter to set the background color of the page. Some results are shown in Figures 20-2 and 20-3.

Figure 20-2. Default result of `BGColor.jsp`.

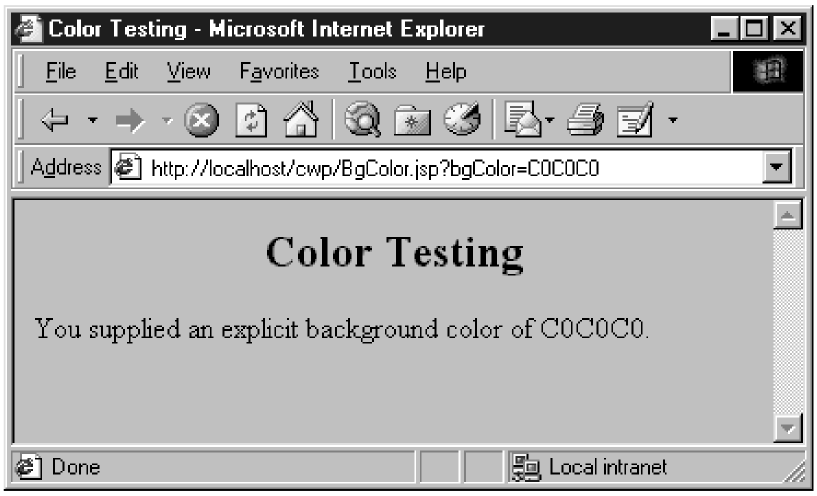

Color Testing - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
Done
Local intranet

http://localhost/cwp/BgColor.jsp

Color Testing

Using default background color of WHITE. Supply the bgColor request attribute to try a standard color or RRGGBB value, or to see if your browser supports X11 color names.

Figure 20-3. Result of `BGColor.jsp` when accessed with a `bgColor` parameter having the RGB value `C0C0C0`.


Color Testing - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
Done
Local intranet

http://localhost/cwp/BgColor.jsp?bgColor=C0C0C0

Color Testing

You supplied an explicit background color of C0C0C0.

Listing 20.2 `BGColor.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Color Testing</TITLE>
</HEAD>
<%
String bgColor = request.getParameter("bgColor");
boolean hasExplicitColor = true;
if (bgColor == null) {
  hasExplicitColor = false;
  bgColor = "WHITE";
}
%>
<BODY BGCOLOR="<%= bgColor %>">
<H2 ALIGN="CENTER">Color Testing</H2>
<%
if (hasExplicitColor) {
  out.println("You supplied an explicit background color of " +
              bgColor + ".");
} else {
  out.println("Using default background color of WHITE. " +
              "Supply the bgColor request attribute to try " +
              "a standard color or RRGGBB value, or to see " +
              "if your browser supports X11 color names.");
}
%>
</BODY>
</HTML>
```

Using Scriptlets to Make Parts of the JSP File Conditional

Another use of scriptlets is to conditionally include standard HTML and JSP constructs. The key to this approach is the fact that code inside a scriptlet gets inserted into the resultant servlet's `_jspService` method (called by `service`) *exactly* as written and that any static HTML (template text) before or after a scriptlet gets converted to `print` statements. This means that scriptlets need not contain complete Java statements and that blocks left open can affect the static HTML or JSP outside of the scriptlets. For example, consider the following JSP fragment containing mixed template text and scriptlets.

```
<% if (Math.random() < 0.5) { %>
Have a <B>nice</B> day!
<% } else { %>
Have a <B>lousy</B> day!
<% } %>
```

When converted to a servlet by the JSP engine, this fragment will result in something similar to the following.

```
if (Math.random() < 0.5) {
  out.println("Have a <B>nice</B> day!");
} else {
  out.println("Have a <B>lousy</B> day!");
}
```

Special Scriptlet Syntax

There are two special constructs you should take note of. First, if you want to use the characters `%>` inside a scriptlet, enter `%\>` instead. Second, the XML equivalent of `<%` *Java Code* `%>` is

```
<jsp:scriptlet>Java Code</jsp:scriptlet>
```

The two forms are treated identically by some JSP engines, but `jsp:scriptlet` is not required to be supported until JSP 1.2.

Declarations

A JSP declaration lets you define methods or fields that get inserted into the main body of the servlet class (*outside* of the `_jspService` method that is called by `service` to process the request). A declaration has the following form:

```
<%! Java Code %>
```

Since declarations do not generate any output, they are normally used in conjunction with JSP expressions or scriptlets. The declarations define methods or fields that are later used by expressions or scriptlets. One caution is warranted however: do not use JSP declarations to override the standard servlet lifecycle methods (`service`, `doGet`, `init`, etc.). The servlet into which the JSP page gets translated already makes use of these methods. There is no need for declarations to gain access to `service`, `doGet`, or `doPost`, since calls to `service` are automatically dispatched to `_jspService`, which is where code resulting from expressions and scriptlets gets put. However, for initialization and cleanup, you can use `jspInit` and `jspDestroy`—the standard `init` and `destroy` methods are guaranteed to call these two methods when in servlets that come from JSP.

Core Note

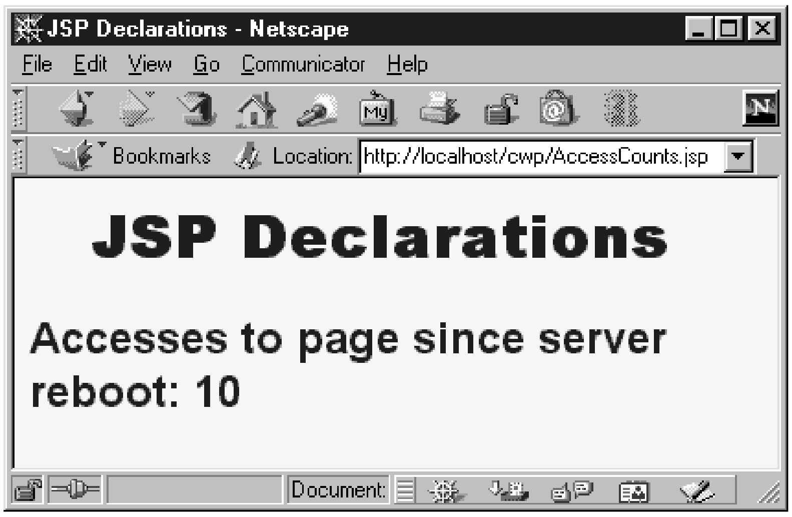
For initialization and cleanup in JSP pages, use JSP declarations to override `jspInit` *and/or* `jspDestroy`*.*

For example, here is a JSP fragment that prints the number of times the current page has been requested since the server was booted (or the servlet class was changed and reloaded). Recall that multiple client requests to the same servlet result only in multiple threads calling the `service` method of a single servlet instance. They do *not* result in the creation of multiple servlet instances except possibly when the servlet implements `SingleThreadModel`. For a discussion of `SingleThreadModel`, see the `isThreadSafe` attribute of the `page` directive (Section 20.4) and Section 19.4 (The Servlet Life Cycle). Thus, instance variables (fields) of a servlet are shared by multiple requests and `accessCount` does not have to be declared `static` below.

```
<%! private int accessCount = 0; %>
Accesses to page since server reboot:
<%= ++accessCount %>
```

Listing 20.3 shows the full JSP page; Figure 20-4 shows a representative result.

Figure 20-4. Visiting `AccessCounts.jsp` after it has been requested nine previous times by the same or different clients.


JSP Declarations - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost/cwp/AccessCounts.jsp
Document:

JSP Declarations

Accesses to page since server reboot: 10

Listing 20.3 `AccessCounts.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
```

```
<HTML>
<HEAD>
<TITLE>JSP Declarations</TITLE>
<META id="keywords"
      CONTENT="JSP,declarations,JavaServer,Pages,servlets">
<META id="description"
      CONTENT="A quick example of JSP declarations.">
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>JSP Declarations</H1>
<%! private int accessCount = 0; %>
<H2>Accesses to page since server reboot:
<%= ++accessCount %></H2>
</BODY>
</HTML>
```

Special Declaration Syntax

As with scriptlets, if you want to use the characters `%>`, enter `%\>` instead. Finally, note that the XML equivalent of `<%!` *Java Code* `%>` is

```
<jsp:declaration>Java Code</jsp:declaration>
```

Predefined Variables

To simplify code in JSP expressions and scriptlets, you are supplied with eight automatically defined variables, sometimes called *implicit objects.* Since JSP declarations result in code that appears outside of the `_jspService` method, these variables are not accessible in declarations. The available variables are `request`, `response`, `out`, `session`, `application`, `config`, `pageContext`, and `page`. Details for each are given below.

request This variable is the `HttpServletRequest` associated with the request; it gives you access to the request parameters, the request type (e.g., `GET` or `POST`), and the incoming HTTP headers (e.g., cookies).

response This variable is the `HttpServletResponse` associated with the response to the client. Note that since the output stream (see `out`) is normally buffered, it is legal to set HTTP status codes and response headers in JSP pages, even though the setting of headers or status codes is not permitted in servlets once any output has been sent to the client.

out This variable is the `PrintWriter` used to send output to the client. However, to make the `response` object useful, this is a buffered version of `PrintWriter` called `JspWriter`. You can adjust the buffer size through use of the `buffer` attribute of the `page` directive. The `out` variable is used almost exclusively in scriptlets, since JSP expressions are automatically placed in the output stream and thus rarely need to refer to `out` explicitly.

session This variable is the `HttpSession` object associated with the request. Recall that sessions are created automatically, so this variable is bound even if there is no incoming session reference. The one exception is if you use the `session` attribute of the `page` directive to turn sessions off. In that case, attempts to reference the `session` variable cause errors at the time the JSP page is translated into a servlet.

application This variable is the `ServletContext` as obtained by `getServletContext`. Servlets and JSP pages can store persistent data in the `ServletContext` object rather than in instance variables. `ServletContext` has `setAttribute` and `getAttribute` methods that let you store arbitrary data associated with specified keys. The difference between storing data in instance variables and storing it in the `ServletContext` is that the `ServletContext` is shared by all servlets in the servlet engine (or in the Web application).

config This variable is the `ServletConfig` object for this page.

pageContext JSP introduced a new class called `PageContext` to give a single point of access to many of the page attributes and to provide a convenient place to store shared data. The `pageContext` variable stores the value of the `PageContext` object associated with the current page.

page This variable is simply a synonym for `this` and is not very useful. It was created as a placeholder for the time when the scripting language could be something other than Java.

20.4 The JSP page Directive

A JSP *directive* affects the overall structure of the servlet that results from the JSP page. The following templates show the two possible forms for directives. Single quotes can be substituted for the double quotes around the attribute values, but the quotation marks cannot be omitted altogether. To obtain quote marks within an attribute value, precede them with a back slash, using `\'`; for `'` and `\"` for `"`.

```
<%@ directive attribute="value" %>

<%@ directive attribute1="value1"
              attribute2="value2"
              ...
              attributeN="valueN" %>
```

In JSP, there are three types of directives: `page`, `include`, and `taglib`. The `page` directive lets you control the structure of the servlet by importing classes, customizing the servlet superclass, setting the content type, and the like. A `page` directive can be placed anywhere within the document; its use is the topic of this section. The second directive, `include`, lets you insert a file into the servlet class at the time the JSP file is translated into a servlet. An `include` directive should be placed in the document at the point at which you want the file to be inserted; it is discussed in Section 20.5. JSP 1.1 introduces a third directive, `taglib`, which can be used to define custom markup tags; it is discussed in Section 20.7.

The `page` directive lets you define one or more of the following case-sensitive attributes: `import`, `contentType`, `isThreadSafe`, `session`, `buffer`, `autoflush`, `extends`, `info`, `errorPage`, `isErrorPage`, and `language`. These attributes are explained in the following subsections.

The import Attribute

The `import` attribute of the `page` directive lets you specify the packages that should be imported by the servlet into which the JSP page gets translated. If you don't explicitly specify any classes to import, the servlet imports `java.lang.*`, `javax.servlet.*`, `javax.servlet.jsp.*`, `javax.servlet.http.*`, and possibly some number of server-specific entries. Never write JSP code that relies on any server-specific classes being imported automatically. Use of the `import` attribute takes one of the following two forms:

```
<%@ page import="package.class" %>
<%@ page import="package.class1,...,package.classN" %>
```

For example, the following directive signifies that all classes in the `java.util` package should be available to use without explicit package identifiers.

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

The `import` attribute is the only `page` attribute that is allowed to appear multiple times within the same document. Although `page` directives can appear anywhere within the document, it is traditional to place `import` statements either near the top of the document or just before the first place that the referenced package is used.

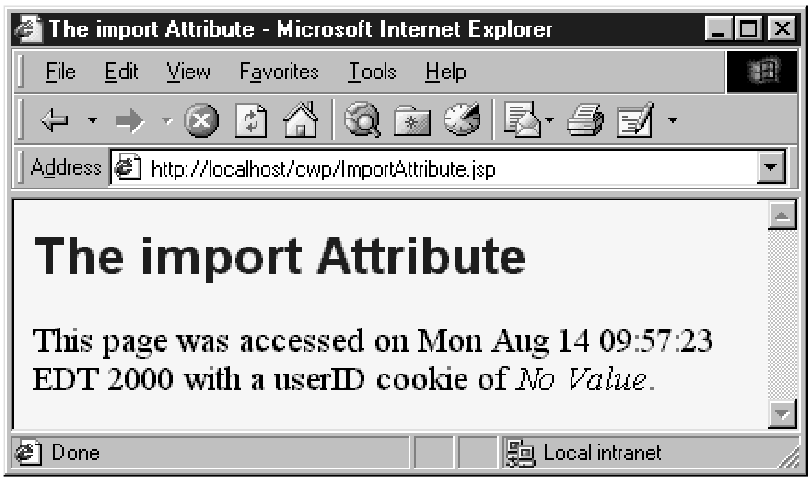
Note that some servers have different rules about where to put different types of class files. For example, the Java Web Server 2.0 lets you put the actual servlet classes in the `servlets` directory but requires you to put classes used by servlets or JSP pages in the `classes` directory. The JSWDK and Tomcat have no such restrictions; check your server's documentation for definitive guidance.

For example, Listing 20.4 presents a page that uses two classes not in the standard JSP import list: `java.util.Date`, and `cwp.ServletUtilities` (see Listing 19.21). To simplify references to these classes, the JSP page uses

```
<%@ page import="java.util.*,cwp.*" %>
```

Figures 20-5 and 20-6 show some typical results.

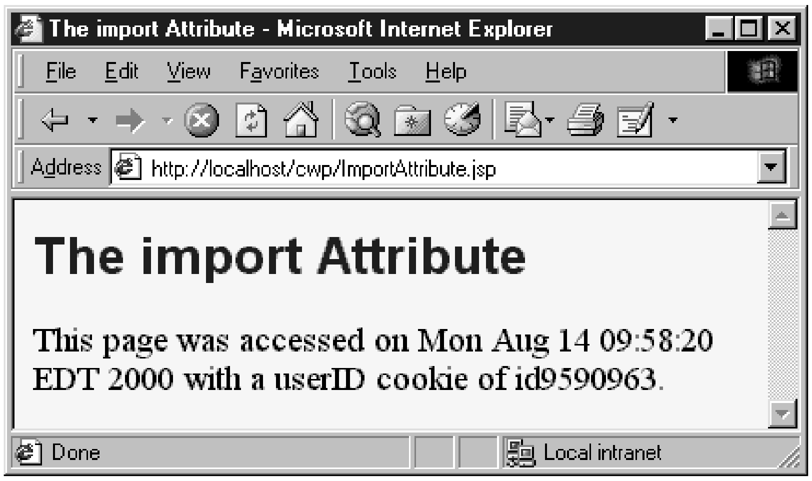
Figure 20-5. `ImportAttribute.jsp` when first accessed.


The import Attribute - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/cwp/ImportAttribute.jsp
Done
Local intranet

The import Attribute

This page was accessed on Mon Aug 14 09:57:23 EDT 2000 with a userID cookie of *No Value*.

Figure 20-6. `ImportAttribute.jsp` when accessed in a subsequent request.


The import Attribute - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
Done
Local intranet

http://localhost/cwp/ImportAttribute.jsp

The import Attribute

This page was accessed on Mon Aug 14 09:58:20 EDT 2000 with a userID cookie of id9590963.

Listing 20.4 ImportAttribute.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>The import Attribute</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H2>The import Attribute</H2>
<%-- JSP page directive --%>
<%@ page import="java.util.*,cwp.*" %>
<%-- JSP Declaration --%>
<%!
private String randomID() {
  int num = (int)(Math.random()*10000000.0);
  return("id" + num);
}
private final String NO_VALUE = "<I>No Value</I>";
%>
<%-- JSP Scriptlet --%>
<%
Cookie[] cookies = request.getCookies();
String oldID =
  ServletUtilities.getCookieValue(cookies, "userID", NO_VALUE);
if (oldID.equals(NO_VALUE)) {
  String newID = randomID();
  Cookie cookie = new Cookie("userID", newID);
  response.addCookie(cookie);
}
%>
<%-- JSP Expressions --%>
This page was accessed on <%= new Date() %> with a userID
cookie of <%= oldID %>.
```

```
</BODY>
</HTML>
```

The contentType Attribute

The `contentType` attribute sets the `Content-Type` response header, indicating the MIME type of the document being sent to the client. For more information on MIME types, see Table 19.1 (Common MIME Types) in Section 19.10 (The Server Response: HTTP Response Headers).

Use of the `contentType` attribute takes one of the following two forms:

```
<%@ page contentType="MIME-Type" %>
<%@ page contentType="MIME-Type; charset=Character-Set" %>
```

For example, the directive

```
<%@ page contentType="application/vnd.ms-excel" %>
```

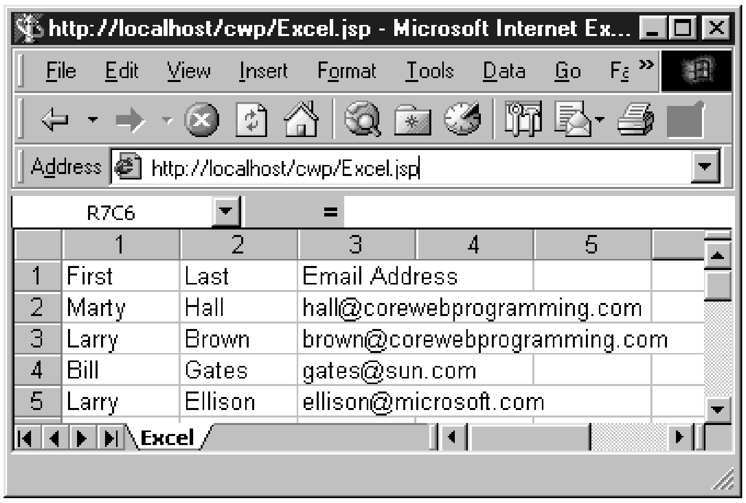
has the same effect as the scriptlet

```
<% response.setContentType("application/vnd.ms-excel"); %>
```

Unlike regular servlets, where the default MIME type is `text/plain`, the default for JSP pages is `text/html` (with a default character set of `ISO-8859-1`).

Listing 20.5 shows a JSP page that generates tab-separated Excel output. Figure 20-7 shows the result in Internet Explorer on a system that has Microsoft Office installed.

Figure 20-7. Excel document (`Excel.jsp`) in Internet Explorer.


http://localhost/cwp/Excel.jsp - Microsoft Internet Ex...
File Edit View Insert Format Tools Data Go Fa
Address http://localhost/cwp/Excel.jsp
R7C6 =
Excel

	1	2	3	4	5
1	First	Last	Email Address		
2	Marty	Hall	hall@corewebprogramming.com		
3	Larry	Brown	brown@corewebprogramming.com		
4	Bill	Gates	gates@sun.com		
5	Larry	Ellison	ellison@microsoft.com		

Listing 20.5 `Excel.jsp`

```
First   Last    Email Address
Marty   Hall    hall@corewebprogramming.com
Larry   Brown   brown@corewebprogramming.com
Bill    Gates   gates@sun.com
Larry   Ellison ellison@microsoft.com
<%@ page contentType="application/vnd.ms-excel" %>
```

```
<%-- There are tabs, not spaces, between columns. --%>
```

The isThreadSafe Attribute

The `isThreadSafe` attribute controls whether or not the servlet that results from the JSP page will implement the `SingleThreadModel` interface. Use of the `isThreadSafe` attribute takes one of the following two forms:

```
<%@ page isThreadSafe="true" %> <%-- Default --%>
<%@ page isThreadSafe="false" %>
```

With normal servlets, simultaneous user requests result in multiple threads concurrently accessing the `service` method of the same servlet instance. This behavior assumes that the servlet is *thread safe;* that is, that the servlet synchronizes access to data in its fields so that inconsistent values will not result from an unexpected ordering of thread execution. In some cases (such as page access counts), you may not care if two visitors occasionally get the same value, but in other cases (such as user IDs), identical values can spell disaster. For example, the following snippet is not thread safe since a thread could be preempted after reading `idNum` but before updating it, yielding two users with the same user ID.

```
<%! private int idNum = 0; %>
<%
String userID = "userID" + idNum;
out.println("Your ID is " + userID + ".");
idNum = idNum + 1;
%>
```

The code should have used a `synchronized` block. This construct is written

```
synchronized(someObject) { ... }
```

and means that once a thread enters the block of code, no other thread can enter the same block (or any other block marked with the same object reference) until the first thread exits. So, the previous snippet should have been written in the following manner.

```
<%! private int idNum = 0; %>
<%
synchronized(this) {
  String userID = "userID" + idNum;
  out.println("Your ID is " + userID + ".");
  idNum = idNum + 1;
}
%>
```

That's the normal servlet behavior: multiple simultaneous requests are dispatched to multiple threads concurrently accessing the same servlet instance. However, if a servlet implements the `SingleThreadModel` interface, the system guarantees that there will not be simultaneous access to the same servlet instance. The system can satisfy this guarantee either by queuing up all requests and passing them to the same servlet instance or by creating a pool of instances, each of which handles a single request at a time.

You use `<%@ page isThreadSafe="false" %>` to indicate that your code is *not* thread safe and thus that the resulting servlet should implement `SingleThreadModel`. The default value is `true`, which means that the system assumes you made your code thread safe and it can consequently use the higher-performance approach of multiple simultaneous threads accessing a single servlet instance.

The session Attribute

The `session` attribute controls whether or not the page participates in HTTP sessions. Use of this attribute takes one of the following two forms:

```
<%@ page session="true" %> <%-- Default --%>
<%@ page session="false" %>
```

A value of `true` (the default) indicates that the predefined variable `session` (of type `HttpSession`) should be bound to the existing session if one exists; otherwise, a new session should be created and bound to `session`. A value of `false` means that no sessions will be used automatically and attempts to access the variable `session` will result in errors at the time the JSP page is translated into a servlet. Turning off session tracking may save significant amounts of server memory on high traffic sites. Just remember that sessions are *user-specific,* not *page-specific.* Thus, it doesn't do any good to turn off session tracking for one page unless you also turn it off for related pages that are likely to be visited in the same client session.

The buffer Attribute

The `buffer` attribute specifies the size of the buffer used by the `out` variable, which is of type `JspWriter` (a subclass of `PrintWriter`). Use of this attribute takes one of two forms:

```
<%@ page buffer="sizekb" %>
<%@ page buffer="none" %>
```

Servers can use a larger buffer than you specify, but not a smaller one. For example, `<%@ page buffer="32kb" %>` means the document content should be buffered and not sent to the client until at least 32 kilobytes have been accumulated or the page is completed. The default buffer size is server specific, but must be at least 8 kilobytes. Be cautious about turning off buffering; doing so requires JSP entries that set headers or status codes to appear at the top of the file, before any HTML content.

The autoflush Attribute

The `autoflush` attribute controls whether the output buffer should be automatically flushed when it is full or whether an exception should be raised when the buffer overflows. Use of this attribute takes one of the following two forms:

```
<%@ page autoflush="true" %> <%-- Default --%>
<%@ page autoflush="false" %>
```

A value of `false` is illegal when `buffer="none"` is also used.

The extends Attribute

The `extends` attribute designates the superclass of the servlet that will be generated for the JSP page and takes the following form:

```
<%@ page extends="package.class" %>
```

Use this attribute with extreme caution since the server will almost certainly be using a custom superclass already.

The info Attribute

The `info` attribute defines a string that can be retrieved from the servlet by means of the `getServletInfo` method. Use of `info` takes the following form:

```
<%@ page info="Some Message" %>
```

The errorPage Attribute

The `errorPage` attribute specifies a JSP page that should process any exceptions (i.e., something of type `Throwable`) thrown but not caught in the current page. It is used as follows:

```
<%@ page errorPage="Relative URL" %>
```

The exception thrown will be automatically available to the designated error page by means of the `exception` variable.

The isErrorPage Attribute

The `isErrorPage` attribute indicates whether or not the current page can act as the error page for another JSP page. Use of `isErrorPage` takes one of the following two forms:

```
<%@ page isErrorPage="true" %>
<%@ page isErrorPage="false" %> <%-- Default --%>
```

The language Attribute

At some point, the `language` attribute is intended to specify the underlying programming language being used, as below.

```
<%@ page language="cobol" %>
```

For now, don't bother with this attribute since `java` is both the default and the only legal choice.

XML Syntax for Directives

Some servers permit you to use an alternative XML-compatible syntax for directives. These constructs take the following form:

```
<jsp:directive.directiveType attribute="value" />
```

For example, the XML equivalent of

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

is

```
<jsp:directive.page import="java.util.*" />
```

20.5 Including Files and Applets in JSP Documents

JSP has three main capabilities for including external pieces into a JSP document.

1. **The include directive.** The construct lets you insert JSP code into the main page before that

main page is translated into a servlet. The included code can contain JSP constructs such as field definitions and content-type settings *that affect the main page as a whole.* This capability is discussed in the first following subsection.

2. **The `jsp:include` action.** Although reusing chunks of JSP code is a powerful capability, most times you would rather sacrifice a small amount of power for the convenience of being able to change the included documents without updating the main JSP page. The `jsp:include` action lets you include the output of a page at request time. Note that `jsp:include` only lets you include the *result* of the secondary page, not the code itself as with the `include` directive. Consequently, the secondary page cannot use any JSP constructs that affect the main page as a whole. Use of `jsp:include` is discussed in the second subsection.

3. **The `jsp:plugin` action.** Although this chapter is primarily about server-side Java, client-side Java in the form of Web-embedded applets continues to play a role, especially within corporate intranets. The `jsp:plugin` element is used to insert applets that use the Java Plug-In into JSP pages. This capability is discussed in the third subsection.

The include Directive: Including Files at Page Translation Time

You use the `include` directive to include a file in the main JSP document at the time the document is translated into a servlet (which is typically the first time it is accessed). The syntax is as follows:

```
<%@ include file="Relative URL" %>
```

There are two ramifications of the fact that the included file is inserted at page translation time, not at request time as with `jsp:include` (see the next subsection).

First, the included file is permitted to contain JSP code such as response header settings and field definitions that affect the main page. For example, suppose `snippet.jsp` contained the following code:

```
<%! int accessCount = 0; %>
```

In such a case, you could do the following:

```
<%@ include file="snippet.jsp" %> <%-- Defines accessCount --%>
<%= accessCount++ %>              <%-- Uses accessCount --%>
```

Second, if the included file changes, all the JSP files that use it need to be updated. Unfortunately, although servers are *allowed* to support a mechanism for detecting when an included file has changed (and then recompiling the servlet), they are not *required* to do so. In practice, few servers support this capability. JSP 1.1 lets you supply a `jsp_precompile` request parameter to instruct the JSP engine to translate the JSP page into a servlet. However, this fails in JSP 1.0. So, the simplest and most portable approach is to update the modification date of the JSP page. Some operating systems have commands that update the modification date without your actually editing the file (e.g., the Unix `touch` command), but a simple portable alternative is to include a JSP comment in the top-level page. Update the comment whenever the included file changes. For example, you might put the modification date of the included file in the comment, as below.

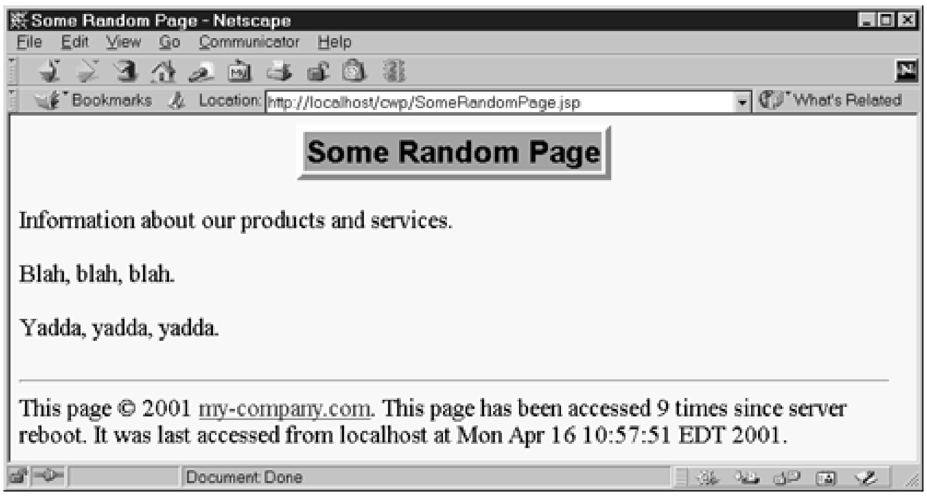
```
<%-- Navbar.jsp modified 3/1/00 --%>
<%@ include file="Navbar.jsp" %>
```

Core Warning

If you change an included JSP file, you must update the modification dates of all JSP files that use it.

For example, Listing 20.6 shows a page fragment that gives corporate contact information and some per-page access statistics appropriate to be included at the bottom of multiple pages within a site. Listing 20.7 shows a page that makes use of it, and Figure 20-8 shows the result.

Figure 20-8. Ninth access to `SomeRandomPage.jsp`.


Some Random Page - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost/cwp/SomeRandomPage.jsp
What's Related
Document: Done

Some Random Page

Information about our products and services.

Blah, blah, blah.

Yadda, yadda, yadda.

This page © 2001 my-company.com. This page has been accessed 9 times since server reboot. It was last accessed from localhost at Mon Apr 16 10:57:51 EDT 2001.

Listing 20.6 `ContactSection.jsp`

```
<%@ page import="java.util.Date" %>
<%-- The following become fields in each servlet that
     results from a JSP page that includes this file. --%>
<%!
private int accessCount = 0;
private Date accessDate = new Date();
private String accessHost = "<I>No previous access</I>";
%>
<P>
<HR>
This page &copy; 2001
<A HREF="http//www.my-company.com/">my-company.com</A>.
This page has been accessed <%= ++accessCount %>
times since server reboot. It was last accessed from
<%= accessHost %> at <%= accessDate %>.
<% accessHost = request.getRemoteHost(); %>
<% accessDate = new Date(); %>
```

Listing 20.7 `SomeRandomPage.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some Random Page</TITLE>
<META id="author" CONTENT="J. Random Hacker">
```

```
<META id="keywords"
      CONTENT="foo,bar,baz,quux">
<META id="description"
      CONTENT="Some random Web page.">
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Some Random Page</TABLE>
<P>
Information about our products and services.
<P>
Blah, blah, blah.
<P>
Yadda, yadda, yadda.
<%@ include file="ContactSection.jsp" %>
</BODY>
</HTML>
```

XML Syntax for the include Directive

The XML-compatible equivalent of

```
<%@ include file="..." %>
```

is

```
<jsp:directive.include file="..." />
```

Including Files at Request Time

The `include` directive (see the previous subsection) lets you include actual JSP code into multiple different pages. Including the code itself is sometimes a useful capability, but the `include` directive requires you to update the modification date of the page whenever the included file changes. This is a significant inconvenience. The `jsp:include` action includes the output of a secondary page at the time the main page is requested. Thus, `jsp:include` does not require you to update the main file when an included file changes. On the other hand, the page has already been translated into a servlet by request time, so the included pages cannot contain JSP that affects the main page as a whole. This is a minor restriction, and `jsp:include` is almost always preferred.

Core Approach

For file inclusion, use `jsp:include` whenever possible. Reserve the `include` directive for cases when the included file defines fields or methods that the main page uses or when the included file sets response headers of the main page.

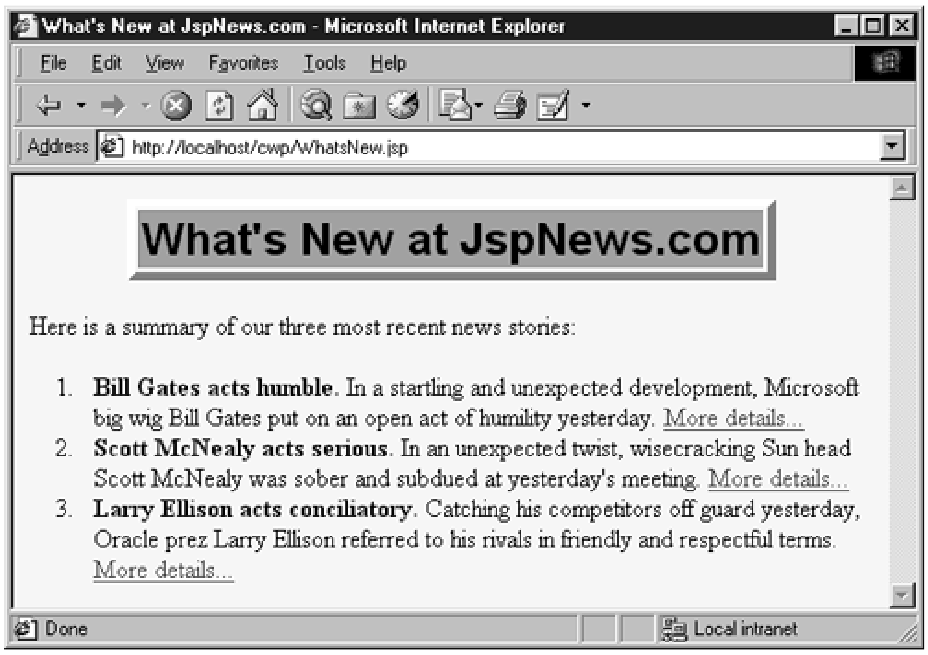
Although the *output* of the included files cannot contain JSP, they can be the result of resources that use JSP to *create* the output. That is, the URL that refers to the included resource is interpreted in the normal manner by the server and thus can be a servlet or JSP page. This is precisely the behavior of the `include` method of the `RequestDispatcher` class, which is what servlets use if they want to do this type of file inclusion.

The `jsp:include` element has two required attributes, as shown in the sample below: `page` (a relative URL referencing the file to be included) and `flush` (which can be omitted in JSP 1.1, and *must* have the value `true` if present).

```
<jsp:include page="Relative URL" flush="true" />
```

As an example, consider the simple news summary page shown in Listing 20.8. Page developers can change the news items in the files `Item1.html` through `Item3.html` (Listings 20.9 through 20.11) without having to update the main news page. Figure 20-9 shows the result.

Figure 20-9. Including files at request time makes it easier to update the individual files.


What's New at JspNews.com - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/cwp/WhatsNew.jsp
Done
Local intranet

What's New at JspNews.com

Here is a summary of our three most recent news stories:

1. **Bill Gates acts humble.** In a startling and unexpected development, Microsoft big wig Bill Gates put on an open act of humility yesterday. More details...
2. **Scott McNealy acts serious.** In an unexpected twist, wisecracking Sun head Scott McNealy was sober and subdued at yesterday's meeting. More details...
3. **Larry Ellison acts conciliatory.** Catching his competitors off guard yesterday, Oracle prez Larry Ellison referred to his rivals in friendly and respectful terms. More details...

Listing 20.8 `WhatsNew.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>What's New at JspNews.com</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      What's New at JspNews.com</TABLE>
<P>
Here is a summary of our three most recent news stories:
<OL>
  <LI><jsp:include page="news/Item1.html" flush="true" />
  <LI><jsp:include page="news/Item2.html" flush="true" />
  <LI><jsp:include page="news/Item3.html" flush="true" />
</OL>
</BODY>
</HTML>
```

Listing 20.9 `Item1.html`

```
<B>Bill Gates acts humble.</B> In a startling and unexpected
development, Microsoft big wig Bill Gates put on an open act of
humility yesterday.
<A HREF="http://www.microsoft.com/Never.html">More details...</A>
```

Listing 20.10 `Item2.html`

```
<B>Scott McNealy acts serious.</B> In an unexpected twist,
wisecracking Sun head Scott McNealy was sober and subdued at
yesterday's meeting.
<A HREF="http://www.sun.com/Imposter.html">More details...</A>
```

Listing 20.11 `Item3.html`

```
<B>Larry Ellison acts conciliatory.</B> Catching his competitors
off guard yesterday, Oracle prez Larry Ellison referred to his
rivals in friendly and respectful terms.
<A HREF="http://www.oracle.com/Mistake.html">More details...</A>
```

Including Applets for the Java Plug-In

With JSP, you don't need any special syntax to include ordinary applets: just use the normal HTML APPLET tag. However, except for intranets that use Netscape 6 exclusively, these applets must use JDK 1.1 or JDK 1.02 since neither Netscape 4.x nor Internet Explorer 5.x supports the Java 2 platform (i.e., JDK 1.2 or 1.3). This lack of support imposes several restrictions on applets:

- In order to use Swing, you must send the Swing files over the network. This process is time consuming and fails in Internet Explorer 3 and Netscape 3.x and 4.01-4.05 (which only support JDK 1.02), since Swing depends on JDK 1.1.
- You cannot use Java 2D.
- You cannot use the Java 2 collections package.
- Your code runs more slowly, since most compilers for the Java 2 platform are significantly improved over their 1.1 predecessors.

To address these problems, Sun developed a browser plug-in for Netscape and Internet Explorer that lets you use the Java 2 platform in a variety of browsers. This plug-in is available at http://java.sun.com/products/plugin/ and also comes bundled with JDK 1.2.2 and later. Since the plug-in is quite large (several megabytes), it is not reasonable to expect users on the WWW at large to download and install it just to run your applets. On the other hand, it is a reasonable alternative for fast corporate intranets, especially since applets can automatically prompt browsers that lack the plug-in to download it.

Unfortunately, however, the normal APPLET tag will not work with the plug-in, since browsers are specifically designed to use only their built-in virtual machine when they see APPLET. Instead, you have to use a long and messy `OBJECT` tag for Internet Explorer and an equally long `EMBED` tag for Netscape. Furthermore, since you typically don't know which browser type will be accessing your page, you have to either include both `OBJECT` and `EMBED` (placing the `EMBED` within the `COMMENT` section of `OBJECT`) or identify the browser type at the time of the request and conditionally build the right tag. This process is straightforward but tedious and time consuming.

The `jsp:plugin` element instructs the server to build a tag appropriate for applets that use the plug-in. Servers are permitted some leeway in exactly how they implement this support, but most simply include both `OBJECT` and `EMBED`.

The jsp:plugin Element

The simplest way to use `jsp:plugin` is to supply four attributes: `type`, `code`, `width`, and `height`. You supply a value of `applet` for the `type` attribute and use the other three attributes in exactly the same way as with the `APPLET` element, with two exceptions: the attribute names are case sensitive, and single or double quotes are always required around the attribute values. So, for example, you could replace

```
<APPLET CODE="MyApplet.class"
        WIDTH=475 HEIGHT=350>
</APPLET>
```

with

```
<jsp:plugin type="applet"
            code="MyApplet.class"
            width="475" height="350">
</jsp:plugin>
```

The `jsp:plugin` element has a number of other optional attributes. Most, but not all, parallel attributes of the `APPLET` element. Here is a full list.

- **type** For applets, this attribute should have a value of `applet`. However, the Java Plug-In also permits you to embed JavaBeans elements in Web pages. Use a value of `bean` in such a case.
- **code** This attribute is used identically to the `CODE` attribute of `APPLET`, specifying the top-level applet class file that extends `Applet` or `JApplet`.
- **width** This attribute is used identically to the `WIDTH` attribute of `APPLET`, specifying the width in pixels to be reserved for the applet.
- **height** This attribute is used identically to the `HEIGHT` attribute of `APPLET`, specifying the height in pixels to be reserved for the applet.
- **codebase** This attribute is used identically to the `CODEBASE` attribute of `APPLET`, specifying the base directory for the applets. The `code` attribute is interpreted relative to this directory. As with the `APPLET` element, if you omit this attribute, the directory of the current page is used as the default. In the case of JSP, this default location is the directory where the original JSP file resided, not the system-specific location of the servlet that results from the JSP file.
- **align** This attribute is used identically to the `ALIGN` attribute of `APPLET` and `IMG`, specifying the alignment of the applet within the Web page. Legal values are `left`, `right`, `top`, `bottom`, and `middle`.

- **hspace** This attribute is used identically to the HSPACE attribute of APPLET, specifying empty space in pixels reserved on the left and right of the applet.
- **vspace** This attribute is used identically to the VSPACE attribute of APPLET, specifying empty space in pixels reserved on the top and bottom of the applet.
- **archive** This attribute is used identically to the ARCHIVE attribute of APPLET, specifying a JAR file from which classes and images should be loaded.
- **name** This attribute is used identically to the NAME attribute of APPLET, specifying a name to use for inter-applet communication or for identifying the applet to scripting languages like JavaScript.
- **title** This attribute is used identically to the very rarely used TITLE attribute of APPLET (and virtually all other HTML elements in HTML 4.0), specifying a title that could be used for a tool-tip or for indexing.
- **jreversion** This attribute identifies the version of the Java Runtime Environment (JRE) that is required. The default is 1.1.
- **iepluginurl** This attribute designates a URL from which the plug-in for Internet Explorer can be downloaded. Users who don't already have the plug-in installed will be prompted to download it from this location. The default value will direct the user to the Sun site, but for intranet use you might want to direct the user to a local copy.
- **nspluginurl** This attribute designates a URL from which the plug-in for Netscape can be downloaded. The default value will direct the user to the Sun site, but for intranet use you might want to direct the user to a local copy.

The jsp:param and jsp:params Elements

The `jsp:param` element is used with `jsp:plugin` in a manner similar to the way that `PARAM` is used with `APPLET`, specifying a name and value that are accessed from within the applet by `getParameter`. There are two main differences, however. First, since `jsp:param` follows XML syntax, attribute names must be lower case, attribute values must be enclosed in single or double quotes, and the element must end with `/>`, not just `>`. Second, all `jsp:param` entries must be enclosed within a `jsp:params` element.

So, for example, you would replace

```
<APPLET CODE="MyApplet.class"
        WIDTH=475 HEIGHT=350>
  <PARAM id="PARAM1" VALUE="VALUE1">
  <PARAM id="PARAM2" VALUE="VALUE2">
</APPLET>
```

with

```
<jsp:plugin type="applet"
            code="MyApplet.class"
            width="475" height="350">
  <jsp:params>
    <jsp:param id="PARAM1" value="VALUE1" />
    <jsp:param id="PARAM2" value="VALUE2" />
  </jsp:params>
```

```
</jsp:plugin>
```

The jsp:fallback Element

The `jsp:fallback` element provides alternative text to browsers that do not support `OBJECT` or `EMBED`. You use this element in almost the same way as you would use alternative text placed within an `APPLET` element. So, for example, you would replace

```
<APPLET CODE="MyApplet.class"
        WIDTH=475 HEIGHT=350>
  <B>Error: this example requires Java.</B>
</APPLET>
```

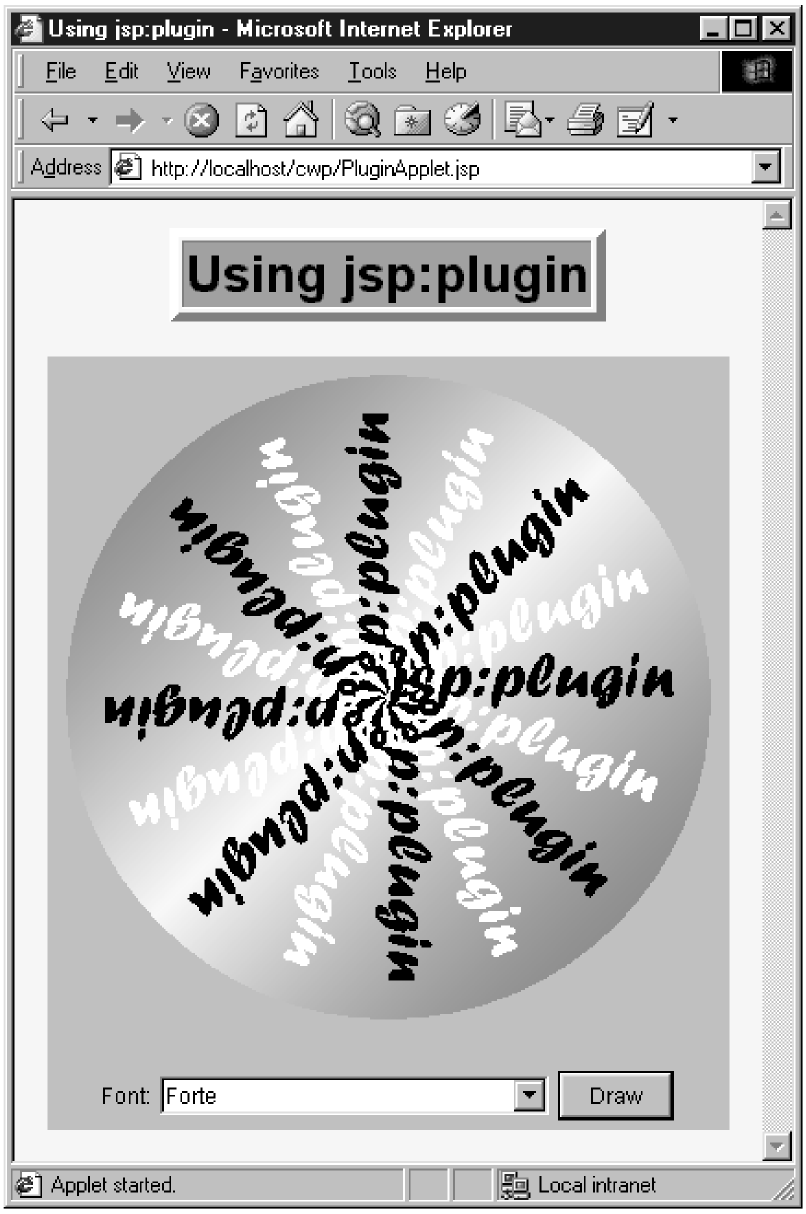
with

```
<jsp:plugin type="applet"
            code="MyApplet.class"
            width="475" height="350">
  <jsp:fallback>
    <B>Error: this example requires Java.</B>
  </jsp:fallback>
</jsp:plugin>
```

A jsp:plugin Example

Listing 20.12 shows a JSP page that uses the `jsp:plugin` element to generate an entry for the Java 2 Plug-In. Listings 20.13 through 20.15 show the code for the applet itself (which uses Swing and Java 2D), and Figure 20-10 shows the result.

Figure 20-10. Result of `PluginApplet.jsp` in Internet Explorer when the Java 2 Plug-In is installed.


Using jsp:plugin - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/cwp/PluginApplet.jsp
Using jsp:plugin
jsp:plugin
Font: Forte
Draw
Applet started.
Local intranet

Listing 20.12 `PluginApplet.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Using jsp:plugin</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Using jsp:plugin</TABLE>
<P>
<CENTER>
<jsp:plugin type="applet"
            code="PluginApplet.class"
            width="370" height="420">
</jsp:plugin>
</CENTER>
</BODY>
</HTML>
```

Listing 20.13 **`PluginApplet.java`**

```
import javax.swing.*;

/** An applet that uses Swing and Java 2D and thus requires
 *  the Java Plug-In.
 */

public class PluginApplet extends JApplet {
  public void init() {
    WindowUtilities.setNativeLookAndFeel();
    setContentPane(new TextPanel());
  }
}
```

Listing 20.14 **`TextPanel.java`**

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/** JPanel that places a panel with text drawn at various angles
 *  in the top part of the window and a JComboBox containing
 *  font choices in the bottom part.
 */

public class TextPanel extends JPanel
                       implements ActionListener {
  private JComboBox fontBox;
  private DrawingPanel drawingPanel;

  public TextPanel() {
    GraphicsEnvironment env =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
    String[] fontNames = env.getAvailableFontFamilyNames();
    fontBox = new JComboBox(fontNames);
    setLayout(new BorderLayout());
    JPanel fontPanel = new JPanel();
    fontPanel.add(new JLabel("Font:"));
    fontPanel.add(fontBox);
    JButton drawButton = new JButton("Draw");
    drawButton.addActionListener(this);
    fontPanel.add(drawButton);
    add(fontPanel, BorderLayout.SOUTH);
    drawingPanel = new DrawingPanel();
    fontBox.setSelectedItem("Serif");
    drawingPanel.setFontName("Serif");
    add(drawingPanel, BorderLayout.CENTER);
  }

  public void actionPerformed(ActionEvent e) {
    drawingPanel.setFontName((String)fontBox.getSelectedItem());
    drawingPanel.repaint();
  }
}
```

Listing 20.15 `DrawingPanel.java`

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/** A window with text drawn at an angle. The font is
 *  set by means of the setFontName method.
 */

class DrawingPanel extends JPanel {
  private Ellipse2D.Double circle =
    new Ellipse2D.Double(10, 10, 350, 350);
  private GradientPaint gradient =
    new GradientPaint(0, 0, Color.red, 180, 180, Color.yellow,
                      true); // true means to repeat pattern
  private Color[] colors = { Color.white, Color.black };

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D)g;
    g2d.setPaint(gradient);
    g2d.fill(circle);
    g2d.translate(185, 185);
    for (int i=0; i<16; i++) {
      g2d.rotate(Math.PI/8.0);
      g2d.setPaint(colors[i%2]);
      g2d.drawString("jsp:plugin", 0, 0);
    }
  }

  public void setFontName(String fontName) {
    setFont(new Font(fontName, Font.BOLD, 35));
  }
}
```

20.6 Using JavaBeans with JSP

The JavaBeans API provides a standard format for Java classes. Visual manipulation tools and other programs can automatically discover information about classes that follow this format and can then create and manipulate the classes without the user having to explicitly write any code.

Full coverage of JavaBeans is beyond the scope of this book. If you want details, pick up one of the many books on the subject or see the documentation and tutorials at http://java.sun.com/beans/docs/. For the purposes of this chapter, all you need to know about beans are three simple points:

1. **A bean class must have a zero-argument (empty) constructor.** You can satisfy this requirement either by explicitly defining such a constructor or by omitting all constructors, which results in an empty constructor being created automatically. The empty constructor will be called when JSP elements create beans.

2. **A bean class should have no public instance variables (fields).** We hope you already follow this practice and use accessor methods instead of allowing direct access to the instance variables. Use of accessor methods lets you do three things: (1) impose constraints on variable values (e.g., have the `setSpeed` method of your `Car` class disallow negative speeds); (2) change your internal data structures without changing the class interface (e.g., change from English units to metric units internally, but still have `getSpeedInMPH` and `getSpeedInKPH` methods); (3) perform side effects automatically when values change (e.g., update the user interface when `setPosition` is called).

3. **Persistent values should be accessed through methods called getXxx and setXxx.** For example, if your `Car` class stores the current number of passengers, you might have methods named `getNumPassengers` (which takes no arguments and returns an `int`) and `setNumPassengers` (which takes an `int` and has a `void` return type). In such a case, the `Car` class is said to have a *property* named `numPassengers` (notice the lowercase n in the property name, but the uppercase N in the method names). If the class has a `get`*Xxx* method but no corresponding `set`*Xxx,* the class is said to have a read-only property named *xxx.*

   The one exception to this naming convention is with boolean properties: they use a method called `is`*Xxx* to look up their values. So, for example, your `Car` class might have methods called `isLeased` (which takes no arguments and returns a `boolean`) and `setLeased` (which takes a `boolean` and has a `void` return type), and would be said to have a `boolean` property named `leased` (again, notice the lowercase leading letter in the property name).

   Although you can use JSP scriptlets or expressions to access arbitrary methods of a class, standard JSP actions for accessing beans can only make use of methods that use the `get`*Xxx*/`set`*Xxx* or `is`*Xxx*/`set`*Xxx* design pattern.

Basic Bean Use

The `jsp:useBean` action lets you load a bean to be used in the JSP page. Beans provide a very useful capability because they let you exploit the reusability of Java classes without sacrificing the convenience that JSP adds over servlets alone.

The simplest syntax for specifying that a bean should be used is:

```
<jsp:useBean id="name" class="package.Class" />
```

This usually means "instantiate an object of the class specified by `Class`, and bind it to a variable with the name specified by `id`." The bean class definition should be in the server's class path (generally, in the same directories where servlets can be installed), *not* in the directory that contains the JSP file. So, for example, the JSP action

```
<jsp:useBean id="book1" class="cwp.Book" />
```

can normally be thought of as equivalent to the scriptlet

```
<% cwp.Book book1 = new cwp.Book(); %>
```

Although it is convenient to think of `jsp:useBean` as being equivalent to building an object, `jsp:useBean` has additional options that make it more powerful. As we'll see later, you can specify a `scope` attribute that makes the bean associated with more than just the current page. If beans can be shared, it is useful to obtain references to existing beans, so the `jsp:useBean` action specifies that a new object is instantiated only if there is no existing one with the same `id` and `scope`.

Rather than using the `class` attribute, you are permitted to use `beanName` instead. The difference is that `beanName` can refer either to a class or to a file containing a serialized bean object. The value of the `beanName` attribute is passed to the `instantiate` method of `java.beans.Bean`.

In most cases, you want the local variable to have the same type as the object being created. In a few cases, however, you might want the variable to be declared to have a type that is a superclass of the actual bean type or is an interface that the bean implements. Use the `type` attribute to control this, as in the following example:

```
<jsp:useBean id="thread1" class="MyClass" type="Runnable" />
```

This use results in code similar to the following being inserted into the `_jspService` method:

```
Runnable thread1 = new MyClass();
```

Note that since `jsp:useBean` uses XML syntax, the format differs in three ways from HTML syntax: the attribute names are case sensitive, either single or double quotes can be used (but one or the other *must* be used), and the end of the tag is marked with `/>`, not just `>`. The first two syntactic differences apply to all JSP elements that look like `jsp`:*xxx.* The third difference applies unless the element is a container with a separate start and end tag.

There are also a few character sequences that require special handling in order to appear inside attribute values. To get `'` within an attribute value, use `\'`. Similarly, to get `"`, use `\"`; to get `\` use `\\`; to get `%>`, use `%\>`; and to get `<%`, use `<\%`.

Accessing Bean Properties

Once you have a bean, you can access its properties with `jsp:getProperty`, which takes a `name` attribute that should match the `id` given in `jsp:useBean` and a `property` attribute that names the property of interest. Alternatively, you could use a JSP expression and explicitly call a method on the object that has the variable name specified with the `id` attribute. For example, assuming that the `Book` class has a `String` property called `title` and that you've created an instance called `book1` by using the `jsp:useBean` example just given, you could insert the value of the `title` property into the JSP page in either of the following two ways:

```
<jsp:getProperty id="book1" property="title" />
<%= book1.getTitle() %>
```

The first approach is preferable in this case, since the syntax is more accessible to Web page designers who are not familiar with the Java programming language. However, direct access to the variable is useful when you are using loops, conditional statements, and methods not represented as properties.

If you are not familiar with the concept of bean properties, the standard interpretation of the statement "this bean has a property of type `T` called `foo`" is "this class has a method called `getFoo` that returns something of type `T` and has another method called `setFoo` that takes a `T` as an argument and stores it for later access by `getFoo`."

Setting Bean Properties: Simple Case

To modify bean properties, you normally use `jsp:setProperty`. This action has several different forms, but with the simplest form you just supply three attributes: `name` (which should match the `id` given by `jsp:useBean`), `property` (the name of the property to change), and `value` (the new value). Later in this section we will present some alternate forms of `jsp:setProperty` that let you automatically associate a property with a request parameter. That section also explains how to supply values that are computed at request time (rather than fixed strings) and discusses the type conversion conventions that let you supply string values for parameters that expect numbers, characters, or boolean values.

An alternative to using the `jsp:setProperty` action is to use a scriptlet that explicitly calls methods on the bean object. For example, given the `book1` object shown earlier in this section, you could use either of the following two forms to modify the `title` property:

```
<jsp:setProperty id="book1"
                 property="title"
                 value="Core Web Programming" />
<% book1.setTitle("Core Web Programming"); %>
```

Using `jsp:setProperty` has the advantage that it is more accessible to the nonprogrammer, but direct access to the object lets you perform more complex operations such as setting the value conditionally or calling methods other than `get`*Xxx* or `set`*Xxx* on the object.

Example: StringBean

Listing 20.16 presents a simple class called `StringBean` that is in the `cwp` package. Because the class has no public instance variables (fields) and has a zero-argument constructor since it doesn't declare any explicit constructors, it satisfies the basic criteria for being a bean. Since `StringBean` has a method called `getMessage` that returns a `String` and another method called `setMessage` that takes a `String` as an argument, in beans terminology the class is said to have a `String` parameter called `message`.

Listing 20.17 shows a JSP file that uses the `StringBean` class. First, an instance of `StringBean` is created with the `jsp:useBean` action as follows:

```
<jsp:useBean id="stringBean" class="cwp.StringBean" />
```

After this, the `message` property can be inserted into the page in either of the following two ways:

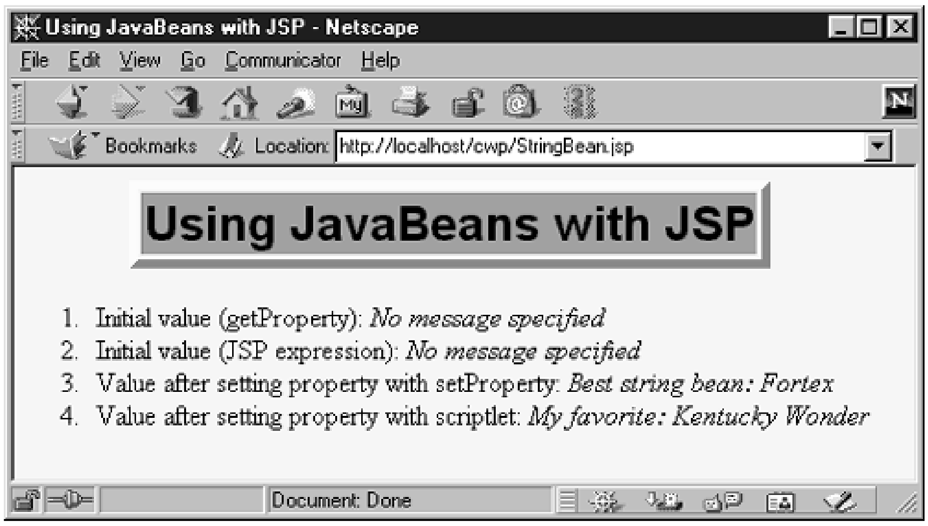
```
<jsp:getProperty id="stringBean" property="message" />
<%= stringBean.getMessage() %>
```

The `message` property can be modified in either of the following two ways:

```
<jsp:setProperty id="stringBean"
                 property="message"
                 value="some message" />
<% stringBean.setMessage("some message"); %>
```

Figure 20-11 shows the result.

Figure 20-11. Result of `StringBean.jsp`.


Using JavaBeans with JSP - Netscape
File Edit View Go Communicator Help
Bookmarks
Location: http://localhost/cwp/StringBean.jsp
Document: Done

Using JavaBeans with JSP

1. Initial value (getProperty): *No message specified*
2. Initial value (JSP expression): *No message specified*
3. Value after setting property with setProperty: *Best string bean: Fortex*
4. Value after setting property with scriptlet: *My favorite: Kentucky Wonder*

Listing 20.16 `StringBean.java`

```
package cwp;

/** A simple bean that has a single String property
 *  called message.
 */

public class StringBean {
  private String message = "No message specified";
  public String getMessage() {
    return(message);
  }

  public void setMessage(String message) {
    this.message = message;
  }
}
```

Listing 20.17 `StringBean.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Using JavaBeans with JSP</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Using JavaBeans with JSP</TABLE>
```

```
<jsp:useBean id="stringBean" class="cwp.StringBean" />
<OL>
<LI>Initial value (getProperty):
    <I><jsp:getProperty id="stringBean"
                        property="message" /></I>
<LI>Initial value (JSP expression):
    <I><%= stringBean.getMessage() %></I>
<LI><jsp:setProperty id="stringBean"
                     property="message"
                     value="Best string bean: Fortex" />
    Value after setting property with setProperty:
    <I><jsp:getProperty id="stringBean"
                        property="message" /></I>
<LI><% stringBean.setMessage("My favorite: Kentucky Wonder"); %>
    Value after setting property with scriptlet:
    <I><%= stringBean.getMessage() %></I>
</OL>
</BODY>
</HTML>
```

Setting Bean Properties

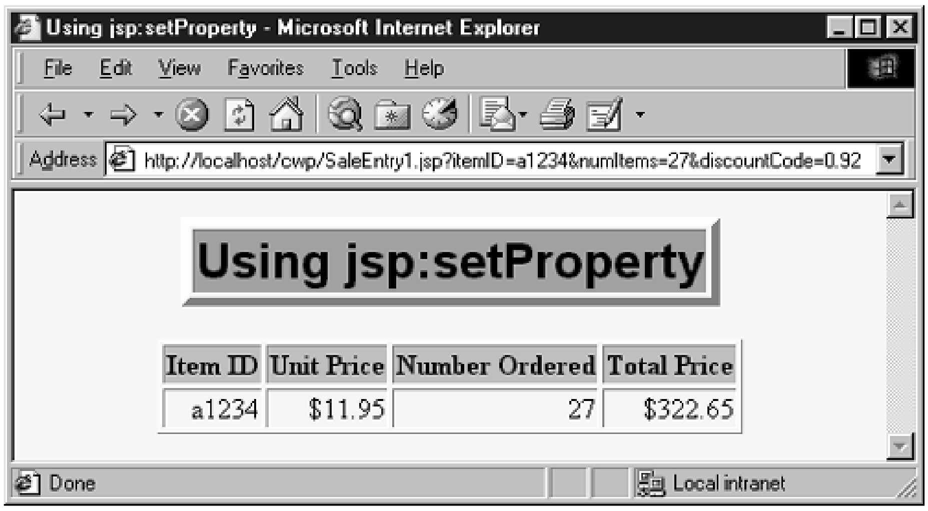
You normally use `jsp:setProperty` to set bean properties. The simplest form of this action takes three attributes: `name` (which should match the `id` given by `jsp:useBean`), `property` (the name of the property to change), and `value` (the new value).

For example, the `SaleEntry` class shown in Listing 20.18 has an `itemID` property (a `String`), a `numItems` property (an `int`), a `discountCode` property (a `double`), and two read-only properties `itemCost` and `totalCost` (each of type `double`). Listing 20.19 shows a JSP file that builds an instance of the `SaleEntry` class by means of:

```
<jsp:useBean id="entry" class="cwp.SaleEntry" />
```

The results are shown in Figure 20-12.

Figure 20-12. Result of `SaleEntry1.jsp`.


Using jsp:setProperty - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
http://localhost/cwp/SaleEntry1.jsp?itemID=a1234&numItems=27&discountCode=0.92
Done
Local intranet

Using jsp:setProperty

Item ID	Unit Price	Number Ordered	Total Price
a1234	$11.95	27	$322.65

Once the bean is instantiated, using an input parameter to set the `itemID` is straightforward, as shown below:

```
<jsp:setProperty
    id="entry"
    property="itemID"
    value='<%= request.getParameter("itemID") %>' />
```

Notice that we used a JSP expression for the `value` parameter. Most JSP attribute values have to be fixed strings, but the `value` and `name` attributes of `jsp:setProperty` are permitted to be request-time expressions. If the expression uses double quotes internally, recall that single quotes can be used instead of double quotes around attribute values and that `\'` and `\"` can be used to represent single or double quotes within an attribute value.

Listing 20.18 `SaleEntry.java`

```
package cwp;

/** Simple bean to illustrate the various forms
 *  of jsp:setProperty.
 */

public class SaleEntry {
  private String itemID = "unknown";
  private double discountCode = 1.0;
  private int numItems = 0;

  public String getItemID() {
    return(itemID);
  }

  public void setItemID(String itemID) {
    if (itemID != null) {
      this.itemID = itemID;
    } else {
      this.itemID = "unknown";
    }
  }

  public double getDiscountCode() {
    return(discountCode);
  }

  public void setDiscountCode(double discountCode) {
    this.discountCode = discountCode;
  }

  public int getNumItems() {
    return(numItems);
  }

  public void setNumItems(int numItems) {
    this.numItems = numItems;
  }
  // In real life, replace this with database lookup.

  public double getItemCost() {
    double cost;
    if (itemID.equals("a1234")) {
      cost = 12.99*getDiscountCode();
    } else {
      cost = -9999;
    }
    return(roundToPennies(cost));
  }

  private double roundToPennies(double cost) {
    return(Math.floor(cost*100)/100.0);
  }

  public double getTotalCost() {
    return(getItemCost() * getNumItems());
  }
}
```

Listing 20.19 `SaleEntry1.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Using jsp:setProperty</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Using jsp:setProperty</TABLE>
<jsp:useBean id="entry" class="cwp.SaleEntry" />
<jsp:setProperty
    id="entry"
    property="itemID"
    value='<%= request.getParameter("itemID") %>' />
<%
int numItemsOrdered = 1;
try {
  numItemsOrdered =
    Integer.parseInt(request.getParameter("numItems"));
} catch(NumberFormatException nfe) {}
%>
<jsp:setProperty
    id="entry"
    property="numItems"
    value="<%= numItemsOrdered %>" />
<%
double discountCode = 1.0;
try {
  String discountString =
    request.getParameter("discountCode");
  // Double.parseDouble not available in JDK 1.1.
  discountCode =
    Double.valueOf(discountString).doubleValue();
} catch(NumberFormatException nfe) {}
%>
<jsp:setProperty
    id="entry"
    property="discountCode"
    value="<%= discountCode %>" />
<BR>
<TABLE ALIGN="CENTER" BORDER=1>
<TR CLASS="COLORED">
  <TH>Item ID<TH>Unit Price<TH>Number Ordered<TH>Total Price
<TR ALIGN="RIGHT">
  <TD><jsp:getProperty id="entry" property="itemID" />
  <TD>$<jsp:getProperty id="entry" property="itemCost" />
  <TD><jsp:getProperty id="entry" property="numItems" />
  <TD>$<jsp:getProperty id="entry" property="totalCost" />
</TABLE>
</BODY>
</HTML>
```

Associating Individual Properties with Input Parameters

Setting the `itemID` property is easy since its value is a `String`. Setting the `numItems` and `discountCode` properties is a bit more problematic since their values must be numbers and `getParameter` returns a `String`. Here is the somewhat cumbersome code required to set `numItems`:

```
<%
int numItemsOrdered = 1;
try {
  numItemsOrdered =
    Integer.parseInt(request.getParameter("numItems"));
} catch(NumberFormatException nfe) {}
%>
<jsp:setProperty
    id="entry"
    property="numItems"
    value="<%= numItemsOrdered %>" />
```

Fortunately, JSP has a nice solution to this problem. It lets you associate a property with a request parameter and automatically perform type conversion from strings to numbers, characters, and boolean values. Instead of using the `value` attribute, you use `param` to name an input parameter. The value of this parameter is automatically used as the value of the property, and simple type conversions are performed automatically. If the specified input parameter is missing from the request, no action is taken (the system does not pass `null` to the associated property).

So, for example, setting the `numItems` property can be simplified to:

```
<jsp:setProperty
    id="entry"
    property="numItems"
    param="numItems" />
```

Listing 20.20 shows the entire JSP page reworked in this manner.

Listing 20.20 SaleEntry2.jsp

```
...
<jsp:useBean id="entry" class="cwp.SaleEntry" />
<jsp:setProperty
    id="entry"
    property="itemID"
    param="itemID" />
<jsp:setProperty
    id="entry"
    property="numItems"
    param="numItems" />
<jsp:setProperty
    id="entry"
    property="discountCode"
    param="discountCode" />
...
```

Automatic Type Conversions

When bean properties are associated with input parameters, the system automatically performs simple type conversions for properties that expect primitive types (byte, int, double, etc.) or the corresponding wrapper types (Byte, Integer, Double, etc.). One warning is in order, however: both JSWDK 1.0.1 and the Java Web Server 2.0 have a bug that causes them to crash at page translation time for pages that try to perform automatic type conversions for properties that expect `double` values. Tomcat and most recent servers work as expected.

Associating All Properties with Input Parameters

Associating a property with an input parameter saves you the bother of performing conversions for many of the simple built-in types. JSP lets you take the process one step further by associating *all* properties with identically named input parameters. All you have to do is to supply `"*"` for the `property` parameter. So, for example, all three of the `jsp:setProperty` statements of Listing 20.20 can be replaced by the following simple line. Listing 20.21 shows the complete page.

```
<jsp:setProperty id="entry" property="*" />
```

Although this approach is simple, four small warnings are in order. First, as with individually associated properties, no action is taken when an input parameter is missing. In particular, the system does not supply `null` as the property value. Second, the JSWDK and the Java Web Server both fail for conversions to properties that expect `double` values. Third, automatic type conversion does not guard against illegal values as effectively as does manual type conversion. So, you might consider error pages when using automatic type conversion. Fourth, since both property names and input parameters are case sensitive, the property name and input parameter must match exactly.

Listing 20.21 `SaleEntry3.jsp`

```
...
<jsp:useBean id="entry" class="cwp.SaleEntry" />
<jsp:setProperty id="entry" property="*" />
...
```

Sharing Beans

Up to this point, we have treated the objects that were created with `jsp:useBean` as though they were simply bound to local variables in the `_jspService` method (which is called by the `service` method of the servlet that is generated from the page). Although the beans are indeed bound to local variables, that is not the only behavior. They are also stored in one of four different locations, depending on the value of the optional `scope` attribute of `jsp:useBean`. The `scope` attribute has the following possible values:

- **page** This is the default value. It indicates that, in addition to being bound to a local variable, the bean object should be placed in the `PageContext` object for the duration of the current request. Storing the object there means that servlet code can access it by calling `getAttribute` on the predefined `pageContext` variable.

- **application** This very useful value means that, in addition to being bound to a local variable, the bean will be stored in the shared `ServletContext` available through the predefined `application` variable or by a call to `getServletContext()`. The `ServletContext` is shared by all servlets in the same Web application (or all servlets in the same server or servlet engine if no explicit Web applications are defined). Values in the `ServletContext` can be retrieved by the `getAttribute` method. This sharing has a couple of ramifications.

  First, it provides a simple mechanism for multiple servlets and JSP pages to access the same object. See the following subsection (Conditional Bean Creation) for details and an example.

  Second, it lets a servlet *create* a bean that will be used in JSP pages, not just *access* one that was previously created. This approach lets a servlet handle complex user requests by setting up beans, storing them in the `ServletContext`, then forwarding the request to one of several possible JSP pages to present results appropriate to the request data. For details on this approach, see Section 20.8 (Integrating Servlets and JSP).

- **session** This value means that, in addition to being bound to a local variable, the bean will be stored in the `HttpSession` object associated with the current request, where it can be retrieved with `getAttribute`.

- **request** This value signifies that, in addition to being bound to a local variable, the bean object should be placed in the `ServletRequest` object for the duration of the current request, where it is available by means of the `getAttribute` method. Storing values in the request object is common when using the MVC (Model 2) architecture. For details, see Section 20.8 (Integrating Servlets and JSP)

Conditional Bean Creation

To make bean sharing more convenient, there are two situations where bean-related elements are evaluated conditionally.

First, a `jsp:useBean` element results in a new bean being instantiated only if no bean with the same `id` and `scope` can be found. If a bean with the same `id` and `scope` *is* found, the preexisting bean is simply bound to the variable referenced by `id`. A typecast is performed if the preexisting bean is of a more specific type than the bean being declared, and a `ClassCastException` results if this typecast is illegal.

Second, instead of

```
<jsp:useBean ... />
```

you can use

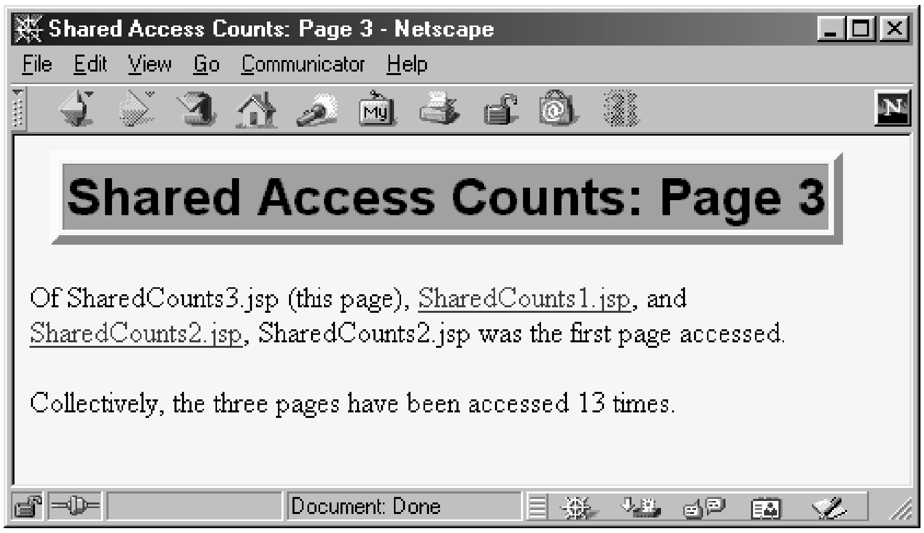
```
<jsp:useBean ...>statements</jsp:useBean>
```

The point of using the second form is that the statements between the `jsp:useBean` start and end tags are executed *only* if a new bean is created, *not* if an existing bean is used. This conditional execution is convenient for setting initial bean properties for beans that are shared by multiple pages. Since you don't know which page will be accessed first, you don't know which page should contain the initialization code. No problem: they can all contain the code, but only the page first accessed actually executes it. For example, Listing 20.22 shows a simple bean that can be used to record cumulative access counts to any of a set of related pages. It also stores the name of the first page that was accessed. Since there is no way to predict which page in a set will be accessed first, each page that uses the shared counter has statements like the following:

```
<jsp:useBean id="counter"
             class="cwp.AccessCountBean"
             scope="application">
  <jsp:setProperty id="counter"
                   property="firstPage"
                   value="Current Page Name" />
</jsp:useBean>
Collectively, the pages using the counter have been accessed
<jsp:getProperty id="counter" property="accessCount" />
times.
```

Listing 20.23 shows the first of three pages that use this approach. The source code archive at http://www.corewebprogramming.com/ contains the other two nearly identical pages. Figure 20-13 shows a typical result.

Figure 20-13. Result of a user visiting `SharedCounts3.jsp`. The first page visited by any user was `SharedCounts2.jsp`. `SharedCounts1.jsp`, `SharedCounts2.jsp`, and `SharedCounts3.jsp` were collectively visited a total of twelve times after the server was last started but prior to the visit shown in this figure.


Shared Access Counts: Page 3 - Netscape
File Edit View Go Communicator Help
Document: Done

Shared Access Counts: Page 3

Of SharedCounts3.jsp (this page), SharedCounts1.jsp, and SharedCounts2.jsp, SharedCounts2.jsp was the first page accessed.

Collectively, the three pages have been accessed 13 times.

Listing 20.22 **`AccessCountBean.java`**

```
package cwp;

/** Simple bean to illustrate sharing beans through
 *  use of the scope attribute of jsp:useBean.
 */

public class AccessCountBean {
  private String firstPage;
  private int accessCount = 1;

  public String getFirstPage() {
    return(firstPage);
  }

  public void setFirstPage(String firstPage) {
    this.firstPage = firstPage;
  }

  public int getAccessCount() {
    return(accessCount++);
  }
}
```

Listing 20.23 **`SharedCounts1.jsp`**

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Shared Access Counts: Page 1</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
```

```
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Shared Access Counts: Page 1</TABLE>
<P>
<jsp:useBean id="counter"
             class="cwp.AccessCountBean"
             scope="application">
  <jsp:setProperty id="counter"
                   property="firstPage"
                   value="SharedCounts1.jsp" />
</jsp:useBean>
Of SharedCounts1.jsp (this page),
<A HREF="SharedCounts2.jsp">SharedCounts2.jsp</A>, and
<A HREF="SharedCounts3.jsp">SharedCounts3.jsp</A>,
<jsp:getProperty id="counter" property="firstPage" />
was the first page accessed.
<P>
Collectively, the three pages have been accessed
<jsp:getProperty id="counter" property="accessCount" />
times.
</BODY>
</HTML>
```

20.7 Defining Custom JSP Tags

JSP 1.1 introduced an extremely valuable new capability: the ability to create your own JSP tags. You define how a tag, its attributes, and its body are interpreted, then group your tags into collections called *tag libraries* that can be used in any number of JSP files. The ability to define tag libraries in this way permits Java developers to boil down complex server-side behaviors into simple and easy-to-use elements that content developers can easily incorporate into their JSP pages.

Custom tags accomplish some of the same goals as beans that are accessed with `jsp:useBean` (see Section 20.6)—encapsulating complex behaviors into simple and accessible forms. There are several differences, however. First, beans cannot manipulate JSP content; custom tags can. Second, complex operations can be reduced to a significantly simpler form with custom tags than with beans. Third, custom tags require quite a bit more work to set up than do beans. Fourth, beans are often defined in one servlet and then used in a different servlet or JSP page (see the following section on integrating servlets and JSP), whereas custom tags usually define more self-contained behavior. Finally, custom tags are available only in JSP 1.1, but beans can be used in both JSP 1.0 and 1.1.

The Components That Make Up a Tag Library

To use custom JSP tags, you need to define three separate components: the tag handler class that defines the tag's behavior, the tag library descriptor file that maps the XML element names to the tag implementations, and the JSP file that uses the tag library. The rest of this subsection gives an overview of each of these components, and the following subsections give details on how to build these components for various styles of tags.

The Tag Handler Class

When defining a new tag, your first task is to define a Java class that tells the system what to do when it sees the tag. This class must implement the `javax.servlet.jsp.tagext.Tag` interface. This is usually accomplished by extending the `TagSupport` or `BodyTagSupport` class. Listing 20.24 is an example of a simple tag that just inserts "`Custom tag example (cwp.tags.ExampleTag)`" into the JSP page wherever the corresponding tag is used. Don't worry about understanding the exact behavior of this class; that will be made clear in the next subsection. For now, just note that it is in the `cwp.tags` class and is called `ExampleTag`. Thus, with Tomcat 3, the class file would be in *`install_dir`*`/webapps/ROOT/WEB-INF/classes/cwp/tags/ExampleTag.class`.

Listing 20.24 `ExampleTag.java`

```
package cwp.tags;

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

/** Very simple JSP tag that just inserts a string
 *  ("Custom tag example...") into the output.
 *  The actual name of the tag is not defined here;
 *  that is given by the Tag Library Descriptor (TLD)
 *  file that is referenced by the taglib directive
 *  in the JSP file.
 */

public class ExampleTag extends TagSupport {
  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
      out.print("Custom tag example " +
                "(cwp.tags.ExampleTag)");
    } catch(IOException ioe) {
      System.out.println("Error in ExampleTag: " + ioe);
    }
    return(SKIP_BODY);
  }
}
```

The Tag Library Descriptor File

Once you have defined a tag handler, your next task is to identify the class to the server and to associate it with a particular XML tag name. This task is accomplished by means of a tag library descriptor file (in XML format) like the one shown in Listing 20.25. This file contains some fixed information, an arbitrary short name for your library, a short description, and a series of tag descriptions. The nonbold part of the listing is the same in virtually all tag library descriptors and can be copied verbatim from the source code archive at http://www.corewebprogramming.com/ or from the Tomcat 3 standard examples (*`install_dir`*`/webapps/examples/ WEB-INF/jsp`).

The format of tag descriptions is described in later sections. For now, just note that the `tag` element defines the main name of the tag (really tag suffix, as will be seen shortly) and identifies the class that handles the tag. Since the tag handler class is in the `cwp.tags` package, the fully qualified class name of `cwp.tags.ExampleTag` is used. Note that this is a class name, not a URL or relative path name. The class can be installed anywhere on the server that beans or other supporting classes can be put. With Tomcat 3, the standard base location is *install_dir*/webapps/ROOT/WEB-INF/classes, so ExampleTag would be in install_dir/webapps/ROOT/WEB-INF/classes/cwp/tags. Although it is always a good idea to put your servlet classes in packages, a surprising feature of Tomcat 3.1 is that tag handlers are *required* to be in packages.

Listing 20.25 cwp-taglib.tld

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>cwp</shortname>
  <info>
    A tag library from Core Web Programming Java 2 Edition,
    http://www.corewebprogramming.com/.
  </info>
  <tag>
    <name>example</name>
    <tagclass>cwp.tags.ExampleTag</tagclass>
    <info>Simplest example: inserts one line of output</info>
  </tag>
</taglib>
```

The JSP File

Once you have a tag handler implementation and a tag library description, you are ready to write a JSP file that makes use of the tag. Listing 20.26 gives an example. Somewhere before the first use of your tag, you need to use the taglib directive. This directive has the following form:

```
<%@ taglib uri="..." prefix="..." %>
```

The required uri attribute can be either an absolute or relative URL referring to a tag library descriptor file like the one shown in Listing 20.25.

The prefix attribute, also required, specifies a prefix that will be used in front of whatever tag name the tag library descriptor defined. For example, if the TLD file defines a tag named tag1 and the prefix attribute has a value of test, the actual tag name would be test:tag1. This tag could be used in either of the following two ways, depending on whether it is defined to be a container that makes use of the tag body:

```
<test:tag1>Arbitrary JSP</test:tag1>
```

or just

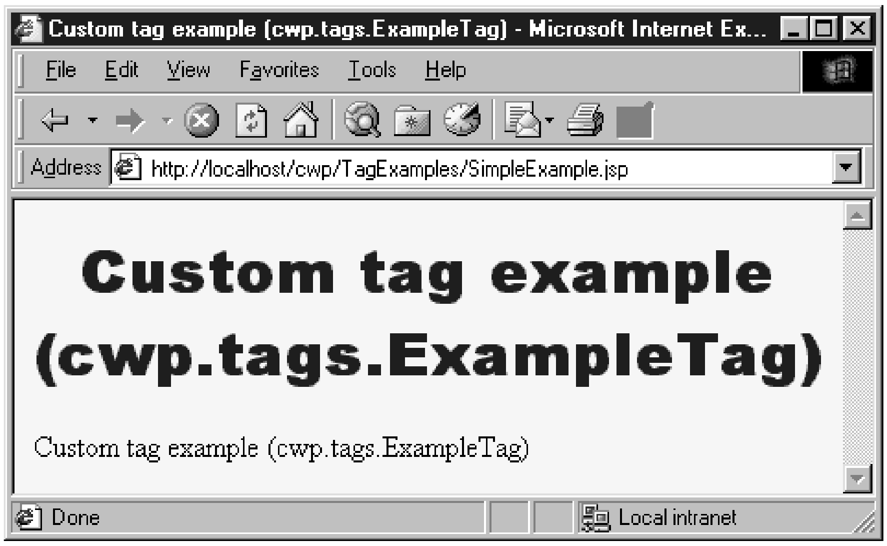
```
<test:tag1 />
```

To illustrate, the descriptor file of Listing 20.25 is called cwp-taglib.tld, and resides in the same directory as the JSP file shown in Listing 20.26. Thus, the taglib directive in the JSP file uses a simple relative URL giving just the filename, as shown below.

```
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
```

Furthermore, since the `prefix` attribute is `cwp` (for *Core Web Programming*), the rest of the JSP page uses `cwp:example` to refer to the `example` tag defined in the descriptor file. Figure 20-14 shows the result.

Figure 20-14. Result of `SimpleExample.jsp`.


Custom tag example (cwp.tags.ExampleTag) - Microsoft Internet Ex...
File Edit View Favorites Tools Help
Address http://localhost/cwp/TagExamples/SimpleExample.jsp
Done
Local intranet

Custom tag example (cwp.tags.ExampleTag)

Custom tag example (cwp.tags.ExampleTag)

Listing 20.26 `SimpleExample.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
<TITLE><cwp:example /></TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1><cwp:example /></H1>
<cwp:example />
</BODY>
</HTML>
```

Defining a Basic Tag

This subsection gives details on defining simple tags without attributes or tag bodies; the tags are thus of the form `<prefix:tagname />`.

A Basic Tag: Tag Handler Class

Tags that either have no body or that merely include the body verbatim should extend the `TagSupport` class. This is a built-in class in the `javax.servlet.jsp.tagext` package that implements the `Tag` interface and contains much of the standard functionality basic tags need. Because of other classes you will use, your tag should normally import classes in the `javax.servlet.jsp` and `java.io` packages as well. So, most tag implementations contain the following `import` statements after the package declaration:

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

We recommend that you download an example from http://www.corewebprogramming.com/ and use it as the starting point for your own implementations.

For a tag without attributes or body, all you need to do is override the `doStartTag` method, which defines code that gets called *at request time* where the element's start tag is found. To generate output, the method should obtain the `JspWriter` (the specialized `PrintWriter` available in JSP pages through use of the predefined `out` variable) from the `pageContext` field by means of `getOut`. In addition to the `getOut` method, the `pageContext` field (of type `PageContext`) has methods for obtaining other data structures associated with the request. The most important ones are `getRequest`, `getResponse`, `getServletContext`, and `getSession`.

Since the `print` method of `JspWriter` throws `IOException`, the `print` statements should be inside a `try`/`catch` block. To report other types of errors to the client, you can declare that your `doStartTag` method throws a `JspException` and then throw one when the error occurs.

If your tag does not have a body, your `doStartTag` should return the `SKIP_BODY` constant. This instructs the system to ignore any content between the tag's start and end tags. As we will see shortly, `SKIP_BODY` is sometimes useful even when there is a tag body (e.g., if you sometimes include it and other times omit it), but the simple tag we're developing here will be used as a stand-alone tag (`<prefix:tagname />`) and thus does not have body content.

Listing 20.27 shows a tag implementation that uses this approach to generate a random 50-digit prime through use of the `Primes` class shown in the previous chapter.

Listing 20.27 `SimplePrimeTag.java`

```
package cwp.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.math.*;
import cwp.*;

/** Generates a prime of approximately 50 digits. */

public class SimplePrimeTag extends TagSupport {
  protected int len = 50;

  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
      BigInteger prime = Primes.nextPrime(Primes.random(len));
      out.print(prime);
    } catch(IOException ioe) {
      System.out.println("Error generating prime: " + ioe);
    }
    return(SKIP_BODY);
  }
}
```

A Basic Tag: Tag Library Descriptor File

The general format of a descriptor file is almost always the same: it should contain an XML version identifier followed by a `DOCTYPE` declaration followed by a `taglib` container element. To get started, just download a sample from http://www.corewebprogramming.com/. The important part to understand is what goes *in* the `taglib` element: the `tag` element. For tags without attributes, the `tag` element should contain four elements between `<tag>` and `</tag>`:

1. **name**, whose body defines the base tag name to which the prefix of the `taglib` directive will be attached. In this case, we use

   ```
   <name>simplePrime</name>
   ```

   to assign a base tag name of `simplePrime`.

2. **tagclass**, which gives the fully qualified class name of the tag handler. In this case, we use

   ```
   <tagclass>cwp.tags.SimplePrimeTag</tagclass>
   ```

3. **info**, which gives a short description. Here, we use

   ```
   <info>Outputs a random 50-digit prime.</info>
   ```

4. **bodycontent**, which can be omitted, but if present should have the value `empty` for tags without bodies. Tags with normal bodies that might be interpreted as normal JSP use a value of `JSP`, and the rare tags whose handlers completely process the body themselves use a value of `tagdependent`. For the `SimplePrimeTag` discussed here, we would use `empty` as below:

   ```
   <bodycontent>empty</bodycontent>
   ```

   Unfortunately, however, Tomcat 3.1 does not support the `bodycontent` element, and TLD files that contain it will not work. Tomcat 3.2, JRun, and most other servers support `bodycontent` properly.

Core Warning

Do not use the `bodycontent` element with Tomcat 3.1.

Listing 20.28 shows the relevant part of the TLD file.

Listing 20.28 `cwp-taglib.tld` (Excerpt)

```
<tag>
   <name>simplePrime</name>
   <tagclass>cwp.tags.SimplePrimeTag</tagclass>
   <info>Outputs a random 50-digit prime.</info>
```

```
  </tag>
```

A Basic Tag: JSP File

JSP documents that make use of custom tags need to use the `taglib` directive, supplying a `uri` attribute that gives the location of the tag library descriptor file and a `prefix` attribute that specifies a short string that will be attached (along with a colon) to the main tag name. Listing 20.29 shows a JSP document that uses

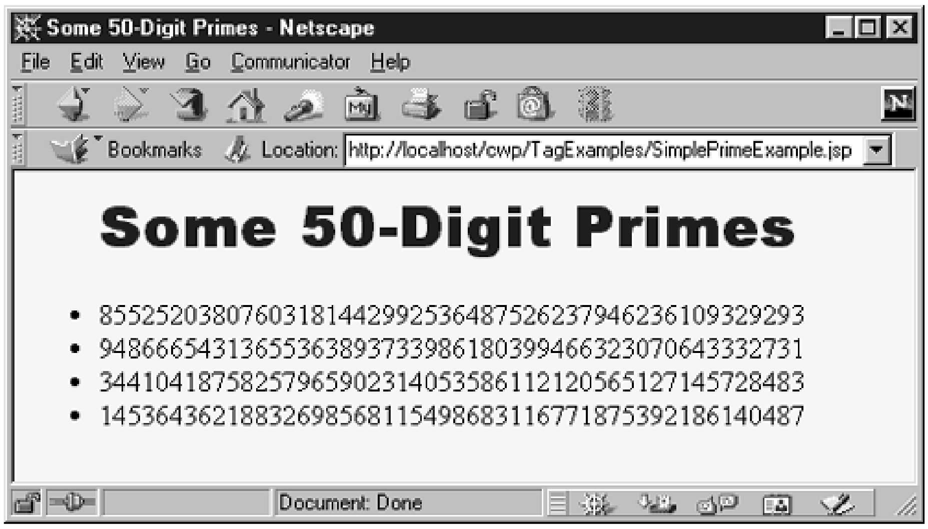
```
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
```

to use the TLD file just shown in Listing 20.28 with a prefix of `cwp`. Since the base tag name is `simplePrime`, the full tag used is

```
<cwp:simplePrime />
```

Figure 20-15 shows the result.

Figure 20-15. Result of `SimplePrimeExample.jsp`.



Some 50-Digit Primes - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost/cwp/TagExamples/SimplePrimeExample.jsp
Document: Done

Some 50-Digit Primes

- 85525203807603181442992536487526237946236109329293
- 94866654313655363893733986180399466323070643332731
- 34410418758257965902314053586112120565127145728483
- 14536436218832698568115498683116771875392186140487

Listing 20.29 `SimplePrimeExample.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some 50-Digit Primes</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Some 50-Digit Primes</H1>
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
<UL>
  <LI><cwp:simplePrime />
  <LI><cwp:simplePrime />
  <LI><cwp:simplePrime />
  <LI><cwp:simplePrime />
</UL>
</BODY>
</HTML>
```

Assigning Attributes to Tags

Allowing tags like

```
<prefix:name attribute1="value1" attribute2="value2" ... />
```

adds significant flexibility to your tag library. This subsection explains how to add attribute support to your tags.

Tag Attributes: Tag Handler Class

Providing support for attributes is straightforward. Use of an attribute called `attribute1` simply results in a call to a method called `setAttribute1` in your class that extends `TagSupport` (or that otherwise implements the `Tag` interface). The attribute value is supplied to the method as a `String`. Consequently, adding support for an attribute named `attribute1` is merely a matter of implementing the following method:

```
public void setAttribute1(String value1) {
  doSomethingWith(value1);
}
```

Note that an attribute of `attributeName` (lowercase `a`) corresponds to a method called `setAttributeName` (uppercase `A`).

One of the most common things to do in the attribute handler is to simply store the attribute in a field that will later be used by `doStartTag` or a similar method. For example, the following is a section of a tag implementation that adds support for the `message` attribute.

```
private String message = "Default Message";

public void setMessage(String message) {
  this.message = message;
}
```

If the tag handler will be accessed from other classes, it is a good idea to provide a `getAttributeName` method in addition to the `setAttributeName` method. Only `setAttributeName` is required, however.

Listing 20.30 shows a subclass of `SimplePrimeTag` that adds support for the `length` attribute. When such an attribute is supplied, it results in a call to `setLength`, which converts the input `String` to an `int` and stores it in the `len` field already used by the `doStartTag` method in the parent class.

Listing 20.30 `PrimeTag.java`

```
package cwp.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
/** Generates an N-digit random prime (default N = 50).
 *  Extends SimplePrimeTag, adding a length attribute
 *  to set the size of the prime. The doStartTag
 *  method of the parent class uses the len field
 *  to determine the approximate length of the prime.
 */

public class PrimeTag extends SimplePrimeTag {
  public void setLength(String length) {
    try {
      len = Integer.parseInt(length);
    } catch(NumberFormatException nfe) {
      len = 50;
    }
  }
}
```

Tag Attributes: Tag Library Descriptor File

Tag attributes must be declared inside the `tag` element by means of an `attribute` element. The `attribute` element has three nested elements that can appear between `<attribute>` and `</attribute>`.

1. **name**, a required element that defines the case-sensitive attribute name. In this case, we use `<name>length</name>`
2. **required**, a required element that stipulates whether the attribute must always be supplied (`true`) or is optional (`false`). In this case, to indicate that `length` is optional, we use `<required>false</required>` If you omit the attribute, no call is made to the `setAttributeName` method. So, be sure to give default values to the fields that the method sets.
3. **rtexprvalue**, an optional attribute that indicates whether the attribute value can be a JSP expression like `<%= expression %>` (`true`) or whether it must be a fixed string (`false`). The default value is `false`, so this element is usually omitted except when you want to allow attributes to have values determined at request time.

Listing 20.31 shows the relevant `tag` element within the tag library descriptor file. In addition to supplying an `attribute` element to describe the `length` attribute, the `tag` element also contains the standard `name` (`prime`), `tagclass` (`cwp.tags.PrimeTag`), and `info` (short description) elements.

Listing 20.31 `cwp-taglib.tld` (Excerpt)

```
<tag>
  <name>prime</name>
  <tagclass>cwp.tags.PrimeTag</tagclass>
  <info>Outputs a random N-digit prime.</info>
  <attribute>
    <name>length</name>
    <required>false</required>
  </attribute>
```

```
</tag>
```

Tag Attributes: JSP File

Listing 20.32 shows a JSP document that uses the `taglib` directive to load the tag library descriptor file and to specify a prefix of `cwp`. Since the `prime` tag is defined to permit a `length` attribute, Listing 20.32 uses

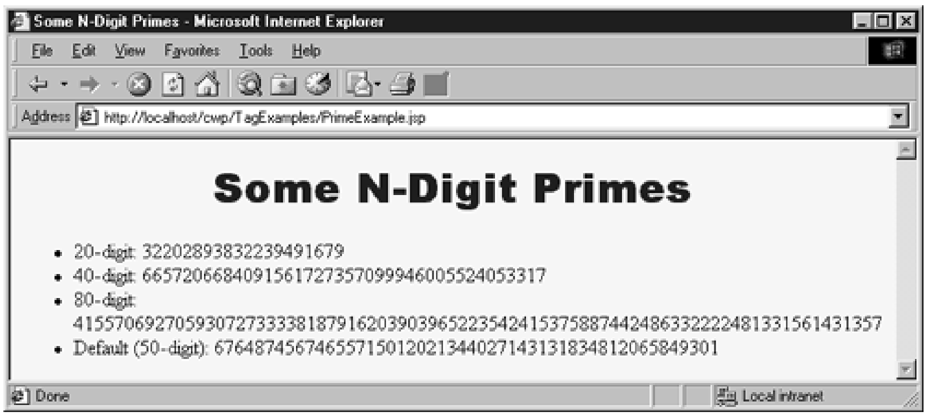
```
<cwp:prime length="xxx" />
```

Remember that custom tags follow XML syntax, which requires attribute values to be enclosed in either single or double quotes. Also, since the `length` attribute is not required, it is permissible to just use

```
<cwp:prime />
```

The tag handler is responsible for using a reasonable default value in such a case. Figure 20-16 shows the result of Listing 20.32.

Figure 20-16. Result of `PrimeExample.jsp`.


Some N-Digit Primes - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/cwp/TagExamples/PrimeExample.jsp
Done
Local intranet

Some N-Digit Primes

- 20-digit: 32202893832239491679
- 40-digit: 6657206684091561727357099946005524053317
- 80-digit: 41557069270593072733338187916203903965223542415375887442486332222481331561431357
- Default (50-digit): 67648745674655715012021344027143131834812065849301

Listing 20.32 `PrimeExample.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some N-Digit Primes</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Some N-Digit Primes</H1>
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
<UL>
  <LI>20-digit: <cwp:prime length="20" />
  <LI>40-digit: <cwp:prime length="40" />
  <LI>80-digit: <cwp:prime length="80" />
  <LI>Default (50-digit): <cwp:prime />
</UL>
</BODY>
```

```
</HTML>
```

Including the Tag Body

Up to this point, all of the custom tags you have seen ignore the tag body and thus are used as stand-alone tags of the form

```
<prefix:tagname />
```

In this section, we see how to define tags that use their body content and are thus used in the following manner:

```
<prefix:tagname>body</prefix:tagname>
```

Tag Bodies: Tag Handler Class

In the previous examples, the tag handlers defined a `doStartTag` method that returned `SKIP_BODY`. To instruct the system to make use of the body that occurs between the new element's start and end tags, your `doStartTag` method should return `EVAL_BODY_INCLUDE` instead. The body content can contain JSP scripting elements, directives, and actions, just like the rest of the page. The JSP constructs are translated into servlet code at page translation time, and that code is invoked at request time.

If you make use of a tag body, then you might want to take some action *after* the body as well as before it. Use the `doEndTag` method to specify this action. In almost all cases, you want to continue with the rest of the page after finishing with your tag, so the `doEndTag` method should return `EVAL_PAGE`. If you want to abort the processing of the rest of the page, you can return `SKIP_PAGE` instead.

Listing 20.33 defines a tag for a heading element that is more flexible than the standard HTML `H1` through `H6` elements. This new element allows a precise font size, a list of preferred font names (the first entry that is available on the client system will be used), a foreground color, a background color, a border, and an alignment (`LEFT`, `CENTER`, `RIGHT`). Only the alignment capability is available with the `H1` through `H6` elements. The heading is implemented through use of a one-cell table enclosing a `SPAN` element that has embedded style sheet attributes. The `doStartTag` method generates the `TABLE` and `SPAN` start tags, then returns `EVAL_BODY_INCLUDE` to instruct the system to include the tag body. The `doEndTag` method generates the `</SPAN>` and `</TABLE>` tags, then returns `EVAL_PAGE` to continue with normal page processing. Various `setAttributeName` methods are used to handle the attributes like `bgColor` and `fontSize`.

Listing 20.33 **`HeadingTag.java`**

```
package cwp.tags;

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

/** Generates an HTML heading with the specified background
 *  color, foreground color, alignment, font, and font size.
 *  You can also turn on a border around it, which normally
 *  just barely encloses the heading, but which can also
 *  stretch wider. All attributes except the background
 *  color are optional.
 */

public class HeadingTag extends TagSupport {
  private String bgColor; // The one required attribute
  private String color = null;
  private String align="CENTER";
  private String fontSize="36";
  private String fontList="Arial, Helvetica, sans-serif";
  private String border="0";
  private String width=null;

  public void setBgColor(String bgColor) {
    this.bgColor = bgColor;
  }

  public void setColor(String color) {
    this.color = color;
  }

  public void setAlign(String align) {
    this.align = align;
  }
    public void setFontSize(String fontSize) {
    this.fontSize = fontSize;
  }

  public void setFontList(String fontList) {
    this.fontList = fontList;
  }

  public void setBorder(String border) {
    this.border = border;
  }

  public void setWidth(String width) {
    this.width = width;
  }

  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
      out.print("<TABLE BORDER=" + border +
                " BGCOLOR=\"" + bgColor + "\"" +
                " ALIGN=\"" + align + "\"");
      if (width != null) {
        out.print(" WIDTH=\"" + width + "\"");
      }
      out.print("><TR><TH>");
      out.print("<SPAN STYLE=\"" +
                "font-size: " + fontSize + "px; " +
                "font-family: " + fontList + "; ");
      if (color != null) {
        out.println("color: " + color + ";");
      }
      out.print("\"> "); // End of <SPAN ...>
    } catch(IOException ioe) {
      System.out.println("Error in HeadingTag: " + ioe);
    }
    return(EVAL_BODY_INCLUDE); // Include tag body
  }
  public int doEndTag() {
    try {
      JspWriter out = pageContext.getOut();
      out.print("</SPAN></TABLE>");
    } catch(IOException ioe) {
      System.out.println("Error in HeadingTag: " + ioe);
    }
    return(EVAL_PAGE); // Continue with rest of JSP page
  }
}
```

Tag Bodies: Tag Library Descriptor File

There is only one new feature in the use of the `tag` element for tags that use body content: the `bodycontent` element should contain the value `JSP` as below.

```
<bodycontent>JSP</bodycontent>
```

Remember, however, that Tomcat 3.1 does not support `bodycontent`, and since `bodycontent` is intended primarily as a hint to development environments, we will omit it in our examples. The `name`, `tagclass`, `info`, and `attribute` elements are used in the same manner as described previously. Listing 20.34 gives the relevant part of the code.

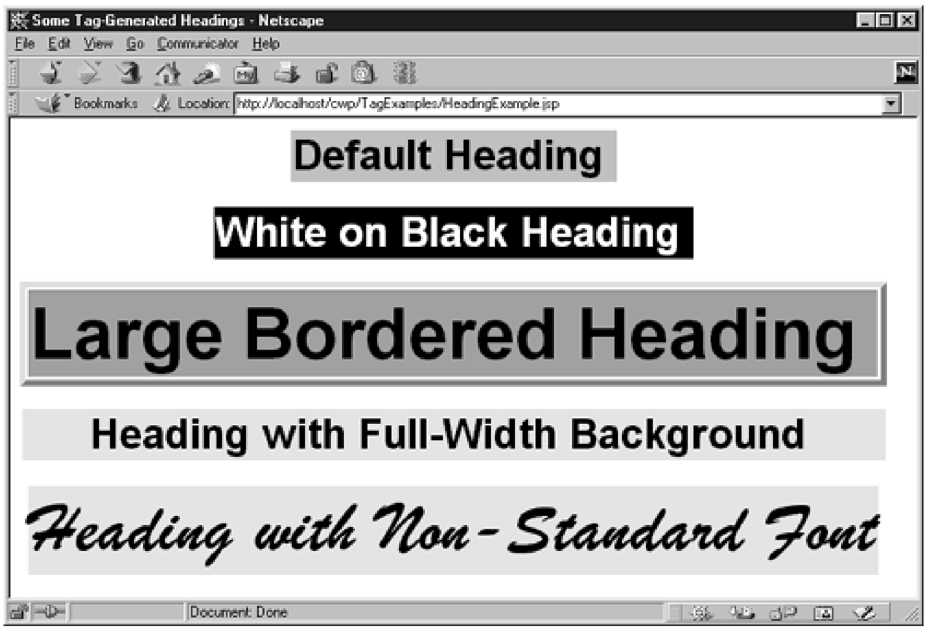
Listing 20.34 `cwp-taglib.tld` (Excerpt)

```
<tag>
  <name>heading</name>
  <tagclass>cwp.tags.HeadingTag</tagclass>
  <info>Outputs a 1-cell table used as a heading.</info>
  <attribute>
    <name>bgColor</name>
    <required>true</required>  <!-- bgColor is required -->
  </attribute>
  <attribute>
    <name>color</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>align</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>fontSize</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>fontList</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>border</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>width</name>
    <required>false</required>
  </attribute>
</tag>
```

Tag Bodies: JSP File

Listing 20.35 shows a document that uses the `heading` tag just defined. Since the `bgColor` attribute was defined to be required, all uses of the tag include it. Figure 20-17 shows the result.

Figure 20-17. The custom `cwp:heading` element gives you much more control over heading format than do the standard `H1` through `H6` elements in HTML.


Some Tag-Generated Headings - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost/cwp/TagExamples/HeadingExample.jsp
Document: Done

Default Heading

White on Black Heading

Large Bordered Heading

Heading with Full-Width Background

Heading with Non-Standard Font

Listing 20.35 `HeadingExample.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some Tag-Generated Headings</TITLE>
</HEAD>
<BODY>
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
<cwp:heading bgColor="#C0C0C0">
```

```
Default Heading
</cwp:heading>
<P>
<cwp:heading bgColor="BLACK" color="WHITE">
White on Black Heading
</cwp:heading>
<P>
<cwp:heading bgColor="#EF8429" fontSize="60" border="5">
Large Bordered Heading
</cwp:heading>
<P>
<cwp:heading bgColor="CYAN" width="100%">
Heading with Full-Width Background
</cwp:heading>
<P>
<cwp:heading bgColor="CYAN" fontSize="60"
                fontList="Brush Script MT, Times, serif">
Heading with Non-Standard Font
</cwp:heading>
</BODY>
</HTML>
```

Optionally Including the Tag Body

Most tags either *never* make use of body content or *always* do so. This section shows you how to use request time information to decide whether or not to include the tag body. Although the body can contain JSP that is interpreted at page translation time, the result of that translation is servlet code that can be invoked or ignored at request time.

Optional Body Inclusion: Tag Handler Class

Optionally including the tag body is a trivial exercise: just return `EVAL_BODY_INCLUDE` or `SKIP_BODY` depending on the value of some request time expression. The important thing to know is how to discover that request time information, since `doStartTag` does not have `HttpServletRequest` and `HttpServletResponse` arguments as do `service`, `_jspService`, `doGet`, and `doPost`. The solution to this dilemma is to use `getRequest` to obtain the `HttpServletRequest` from the automatically defined `pageContext` field of `TagSupport`. Strictly speaking, the return type of `getRequest` is `ServletRequest`, so you have to do a typecast to `HttpServletRequest` if you want to call a method that is not inherited from `ServletRequest`. However, in this case we just use `getParameter`, so no typecast is required.

Listing 20.36 defines a tag that ignores its body unless a request time `debug` parameter is supplied. Such a tag provides a useful capability whereby you embed debugging information directly in the JSP page during development but activate it only when a problem occurs.

Listing 20.36 `DebugTag.java`

```
package cwp.tags;

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

/** A tag that includes the body content only if
 *  the "debug" request parameter is set.
 */

public class DebugTag extends TagSupport {
  public int doStartTag() {
    ServletRequest request = pageContext.getRequest();
    String debugFlag = request.getParameter("debug");
    if ((debugFlag != null) &&
        (!debugFlag.equalsIgnoreCase("false"))) {
      return(EVAL_BODY_INCLUDE);
    } else {
      return(SKIP_BODY);
    }
  }
}
```

Optional Body Inclusion: Tag Library Descriptor File

If your tag *ever* makes use of its body, you should provide the value `JSP` inside the `bodycontent` element (if you use `bodycontent` at all). Other than that, all the elements within `tag` are used in the same way as described previously. Listing 20.37 shows the entries needed for `DebugTag`.

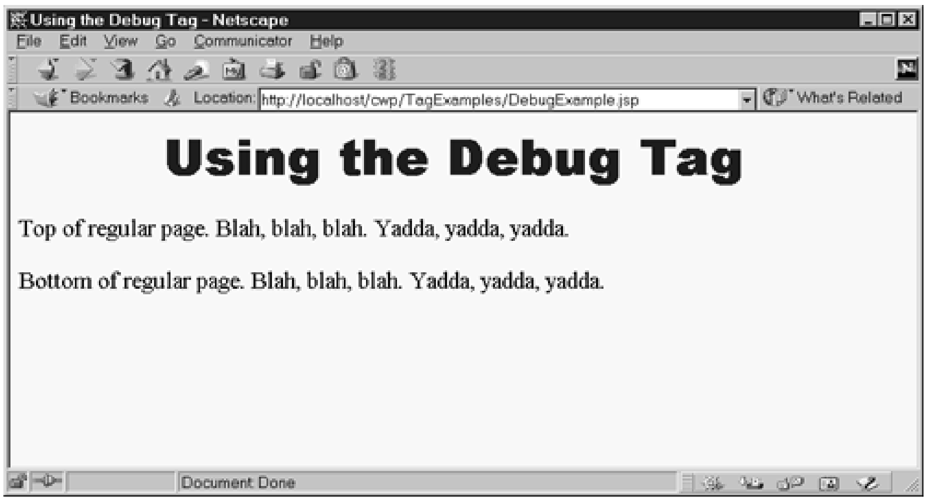
Listing 20.37 `cwp-taglib.tld` (Excerpt)

```
<tag>
  <name>debug</name>
  <tagclass>cwp.tags.DebugTag</tagclass>
  <info>Includes body only if debug param is set.</info>
</tag>
```

Optional Body Inclusion: JSP File

Listing 20.38 shows a page that encloses debugging information between `<cwp:debug>` and `</cwp:debug>`. Figures20-18 and 20-19 show the normal result and the result when a request time `debug` parameter is supplied, respectively.

Figure 20-18. The body of the `cwp:debug` element is normally ignored.


Using the Debug Tag - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost/cwp/TagExamples/DebugExample.jsp
What's Related
Document Done

Using the Debug Tag

Top of regular page. Blah, blah, blah. Yadda, yadda, yadda.

Bottom of regular page. Blah, blah, blah. Yadda, yadda, yadda.

Figure 20-19. The body of the `cwp:debug` element is included when a `debug` request parameter is supplied.

graphics/20fig19.gif

Listing 20.38 `DebugExample.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Using the Debug Tag</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Using the Debug Tag</H1>
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
Top of regular page. Blah, blah, blah. Yadda, yadda, yadda.
<P>
<cwp:debug>
<B>Debug:</B>
<UL>
  <LI>Current time: <%= new java.util.Date() %>
  <LI>Requesting hostname: <%= request.getRemoteHost() %>
  <LI>Session ID: <%= session.getId() %>
</UL>
</cwp:debug>
<P>
Bottom of regular page. Blah, blah, blah. Yadda, yadda, yadda.
</BODY>
</HTML>
```

Manipulating the Tag Body

The `cwp:prime` element ignored any body content, the `cwp:heading` element used body content, and the `cwp:debug` element ignored or used it depending on a request time parameter. The common thread among these elements is that the tag body was never modified; it was either ignored or included verbatim (after JSP translation). This section shows you how to process the tag body.

Manipulating Body: Tag Handler Class

Up to this point, all of the tag handlers have extended the `TagSupport` class. This is a good standard starting point, as it implements the required `Tag` interface and performs a number of useful setup operations like storing the `PageContext` reference in the `pageContext` field. However, `TagSupport` is not powerful enough for tag implementations that need to manipulate their body content, and `BodyTagSupport` should be used instead.

`BodyTagSupport` extends `TagSupport`, so the `doStartTag` and `doEndTag` methods are used in the same way as before. The two important new methods defined by `BodyTagSupport` are:

1. **doAfterBody**, a method that you should override to handle the manipulation of the tag body. This method should normally return `SKIP_BODY` when it is done, indicating that no further body processing should be performed.
2. **getBodyContent**, a method that returns an object of type `BodyContent` that encapsulates information about the tag body.

The `BodyContent` class has three important methods:

1. **getEnclosingWriter**, a method that returns the `JspWriter` being used by `doStartTag` and `doEndTag`.
2. **getReader**, a method that returns a `Reader` that can read the tag's body.
3. **getString**, a method that returns a `String` containing the entire tag body.

The `ServletUtilities` class (see Listing 19.11) contains a static `filter` method that takes a string and replaces `<`, `>`, `"`, and `&` with `&<`, `&gt;`, `&"`, and `&&`, respectively. This method is useful when servlets output strings that might contain characters that would interfere with the HTML structure of the page in which the strings are embedded. Listing 20.39 shows a tag implementation that gives this filtering functionality to a custom JSP tag.

Listing 20.39 `FilterTag.java`

```
package cwp.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import cwp.*;
/** A tag that replaces <, >, ", and & with their HTML
 *  character entities (&lt;, &gt;, &quot;, and &amp;).
 *  After filtering, arbitrary strings can be placed
 *  in either the page body or in HTML attributes.
 */

public class FilterTag extends BodyTagSupport {
  public int doAfterBody() {
    BodyContent body = getBodyContent();
    String filteredBody =
      ServletUtilities.filter(body.getString());
    try {
      JspWriter out = body.getEnclosingWriter();
      out.print(filteredBody);
    } catch(IOException ioe) {
      System.out.println("Error in FilterTag: " + ioe);
    }
    // SKIP_BODY means we're done. If we wanted to evaluate
    // and handle the body again, we'd return EVAL_BODY_TAG.
    return(SKIP_BODY);
  }
}
```

Manipulating Body: Tag Library Descriptor File

Tags that manipulate their body content should use the `bodycontent` element the same way as tags that simply include it verbatim; they should supply a value of `JSP`. Again, however, remember that Tomcat 3.1 does not properly support `bodycontent`, and omitting `bodycontent` is perfectly acceptable on any server. Other than that, nothing new is required in the descriptor file, as you can see by examining Listing 20.40, which shows the relevant portion of the TLD file.

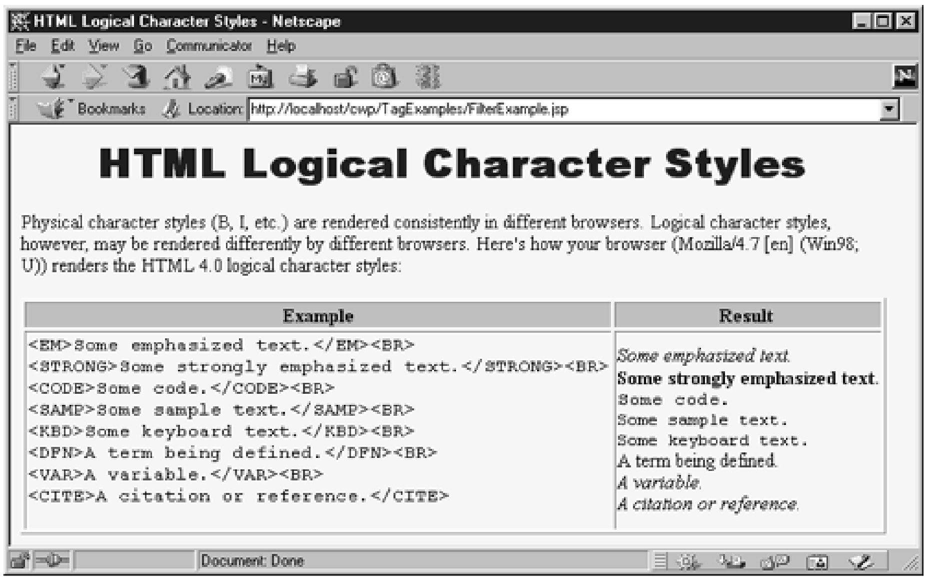
Listing 20.40 `cwp-taglib.tld` (Excerpt)

```
<tag>
  <name>filter</name>
  <tagclass>cwp.tags.FilterTag</tagclass>
  <info>Replaces HTML-specific characters in body.</info>
</tag>
```

Manipulating Body: JSP File

Listing 20.41 shows a page that uses a table to show some sample HTML and its result. Creating this table would be tedious in regular HTML since the table cell that shows the original HTML would have to change all the < and > characters to `&lt`; and `&gt`;. Doing so is particularly onerous during development when the sample HTML is frequently changing. Use of the `<cwp:filter>` tag greatly simplifies the process, as Listing 20.41 illustrates. Figure 20-20 shows the result.

Figure 20-20. The `cwp:filter` element lets you insert text without worrying about it containing special HTML characters.


HTML Logical Character Styles - Netscape
File Edit View Go Communicator Help
Bookmarks
Location: http://localhost/cwp/TagExamples/FilterExample.jsp
Document: Done

HTML Logical Character Styles

Physical character styles (B, I, etc.) are rendered consistently in different browsers. Logical character styles, however, may be rendered differently by different browsers. Here's how your browser (Mozilla/4.7 [en] (Win98; U)) renders the HTML 4.0 logical character styles:

Example	Result
`<EM>Some emphasized text.</EM><BR>`	*Some emphasized text.*
`<STRONG>Some strongly emphasized text.</STRONG><BR>`	**Some strongly emphasized text.**
`<CODE>Some code.</CODE><BR>`	`Some code.`
`<SAMP>Some sample text.</SAMP><BR>`	`Some sample text.`
`<KBD>Some keyboard text.</KBD><BR>`	`Some keyboard text.`
`<DFN>A term being defined.</DFN><BR>`	A term being defined.
`<VAR>A variable.</VAR><BR>`	*A variable.*
`<CITE>A citation or reference.</CITE>`	*A citation or reference.*

Listing 20.41 `FilterExample.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>HTML Logical Character Styles</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>HTML Logical Character Styles</H1>
Physical character styles (B, I, etc.) are rendered consistently
in different browsers. Logical character styles, however,
may be rendered differently by different browsers.
Here's how your browser
(<%= request.getHeader("User-Agent") %>)
renders the HTML 4.0 logical character styles:
<P>
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
<TABLE BORDER=1 ALIGN="CENTER">
<TR CLASS="COLORED"><TH>Example<TH>Result
<TR>
<TD><PRE><cwp:filter>
<EM>Some emphasized text.</EM><BR>
<STRONG>Some strongly emphasized text.</STRONG><BR>
<CODE>Some code.</CODE><BR>
<SAMP>Some sample text.</SAMP><BR>
<KBD>Some keyboard text.</KBD><BR>
<DFN>A term being defined.</DFN><BR>
<VAR>A variable.</VAR><BR>
<CITE>A citation or reference.</CITE>
</cwp:filter></PRE>
<TD>
<EM>Some emphasized text.</EM><BR>
```

```
<STRONG>Some strongly emphasized text.</STRONG><BR>
<CODE>Some code.</CODE><BR>
<SAMP>Some sample text.</SAMP><BR>
<KBD>Some keyboard text.</KBD><BR>
<DFN>A term being defined.</DFN><BR>
<VAR>A variable.</VAR><BR>
<CITE>A citation or reference.</CITE>
</TABLE>
</BODY>
</HTML>
```

Including or Manipulating the Tag Body Multiple Times

Rather than just including or processing the body of the tag a single time, you sometimes want to do so more than once. The ability to support multiple body inclusion lets you define a variety of iteration tags that repeat JSP fragments a variable number of times, repeat them until a certain condition occurs, and so forth. This subsection shows you how to build such tags.

The Tag Handler Class

Tags that process the body content multiple times should start by extending `BodyTagSupport` and implementing `doStartTag`, `doEndTag`, and, most importantly, `doAfterBody` as before. The difference lies in the return value of `doAfterBody`. If this method returns `EVAL_BODY_TAG`, the tag body is evaluated again, resulting in a new call to `doAfterBody`. This process continues until `doAfterBody` returns `SKIP_BODY`.

Listing 20.42 defines a tag that repeats the body content the number of times specified by the `reps` attribute. Since the body content can contain JSP (which is converted into servlet code at page translation time but is invoked at request time), each repetition does not necessarily result in the same output to the client.

Listing 20.42 `RepeatTag.java`

```
package cwp.tags;

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

/** A tag that repeats the body the specified
 *  number of times.
 */

public class RepeatTag extends BodyTagSupport {
  private int reps;

  public void setReps(String repeats) {
    try {
      reps = Integer.parseInt(repeats);
    } catch(NumberFormatException nfe) {
      reps = 1;
    }
  }
  public int doAfterBody() {
    if (reps-- >= 1) {
      BodyContent body = getBodyContent();
      try {
        JspWriter out = body.getEnclosingWriter();
        out.println(body.getString());
        body.clearBody(); // Clear for next evaluation
      } catch(IOException ioe) {
        System.out.println("Error in RepeatTag: " + ioe);
      }
      return(EVAL_BODY_TAG);
    } else {
      return(SKIP_BODY);
    }
  }
}
```

The Tag Library Descriptor File

Listing 20.43 shows the relevant section of the TLD file that gives the name `cwp:repeat` to the tag just defined. To accommodate request time values in the `reps` attribute, the file uses an `rtexprvalue` element (enclosing a value of `true`) within the `attribute` element.

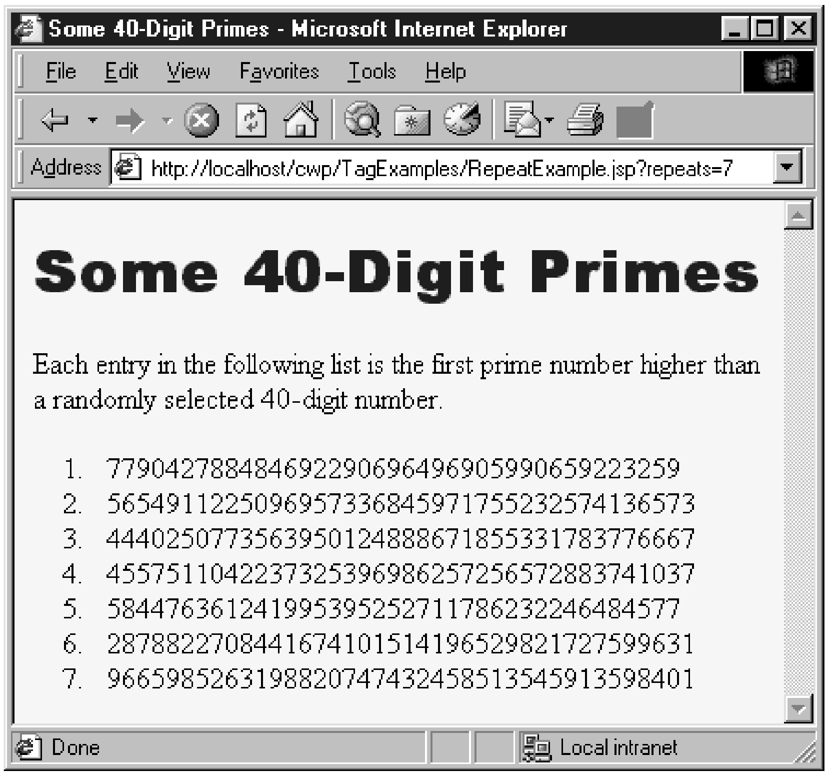
Listing 20.43 `cwp-taglib.tld` (Excerpt)

```
<tag>
  <name>repeat</name>
  <tagclass>cwp.tags.RepeatTag</tagclass>
  <info>Repeats body the specified number of times.</info>
  <attribute>
    <name>reps</name>
    <required>true</required>
    <!-- rtexprvalue indicates whether attribute
         can be a JSP expression. -->
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

The JSP File

Listing 20.44 shows a JSP document that creates a numbered list of prime numbers. The number of primes in the list is taken from the request time `repeats` parameter. Figure 20-21 shows one possible result.

Figure 20-21. Result of `RepeatExample.jsp` when accessed with a `repeats` parameter of 20.


Some 40-Digit Primes - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/cwp/TagExamples/RepeatExample.jsp?repeats=7
Done
Local intranet

Some 40-Digit Primes

Each entry in the following list is the first prime number higher than a randomly selected 40-digit number.

1. 7790427884846922906964969059906592232593
2. 5654911225096957336845971755232574136573
3. 4440250773563950124888671855331783776667
4. 4557511042237325396986257256572883741037
5. 5844763612419953952527117862322464845770
6. 2878822708441674101514196529821727599631
7. 9665985263198820747432458513545913598401

Listing 20.44 **`RepeatExample.jsp`**

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some 40-Digit Primes</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Some 40-Digit Primes</H1>
Each entry in the following list is the first prime number
higher than a randomly selected 40-digit number.
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
<OL>
<!-- Repeats N times. A null reps value means repeat once. -->
<cwp:repeat reps='<%= request.getParameter("repeats") %>'>
  <LI><cwp:prime length="40" />
</cwp:repeat>
</OL>
</BODY>
</HTML>
```

Using Nested Tags

Although Listing 20.44 places the `cwp:prime` element within the `cwp:repeat` element, the two elements are independent of each other. The first generates a prime number regardless of where it is used, and the second repeats the enclosed content regardless of whether that content uses a `cwp:prime` element.

Some tags, however, depend on a particular nesting. For example, in standard HTML, the TD and TH elements can only appear within TR, which in turn can only appear within TABLE. The color and alignment settings of TABLE are inherited by TR, and the values of TR affect how TD and TH behave. So, the nested elements cannot act in isolation even when nested properly. Similarly, the tag library descriptor file makes use of a number of elements like taglib, tag, attribute, and required where a strict nesting hierarchy is imposed.

This subsection shows you how to define tags that depend on a particular nesting order and where the behavior of certain tags depends on values supplied by earlier ones.

The Tag Handler Classes

Class definitions for nested tags can extend *either* TagSupport or BodyTagSupport, depending on whether they need to manipulate their body content (these extend BodyTagSupport) or, more commonly, just ignore it or include it verbatim (these extend TagSupport).

There are two key new approaches for nested tags, however. First, nested tags can use findAncestorWithClass to find the tag in which they are nested. This method takes a reference to the current class (e.g., this) and the Class object of the enclosing class (e.g., EnclosingTag.class) as arguments. If no enclosing class is found, the method in the nested class can throw a JspTagException that reports the problem. Second, if one tag wants to store data that a later tag will use, it can place that data in the instance of the enclosing tag. The definition of the enclosing tag should provide methods for storing and accessing this data.

Suppose that we want to define a set of tags that would be used like this:

```
<cwp:if>
  <cwp:condition><%= someExpression %></cwp:condition>
  <cwp:then>JSP to include if condition is true</cwp:then>
  <cwp:else>JSP to include if condition is false</cwp:else>
</cwp:if>
```

To accomplish this task, the first step is to define an IfTag class to handle the cwp:if tag. This handler should have methods to specify and check whether the condition is true or false (setCondition and getCondition) as well as methods to designate and check whether the condition has ever been explicitly set (setHasCondition and getHasCondition), since we want to disallow cwp:if tags that contain no cwp:condition entry. Listing 20.45 shows the code for IfTag.

The second step is to define a tag handler for cwp:condition. This class, called IfConditionTag, defines a doStartTag method that merely checks whether the tag appears within IfTag. It returns EVAL_BODY_TAG if so and throws an exception if not. The handler's doAfterBody method looks up the body content (getBodyContent), converts it to a String (getString), and compares that to "true". This approach means that an explicit value of true can be substituted for a JSP expression like <%= expression %> if, during initial page development, you want to temporarily designate that the then portion should always be used. Using a comparison to "true" also means that *any* other value will be considered false. Once this comparison is performed, the result is stored in the enclosing tag by means of the setCondition method of IfTag. The code for IfConditionTag is shown in Listing 20.46.

The third step is to define a class to handle the cwp:then tag. The doStartTag method of this class verifies that it is inside IfTag and also checks that an explicit condition has been set (i.e., that the IfConditionTag has already appeared within the IfTag). The doAfterBody method checks for the condition in the `IfTag` class, and, if it is true, looks up the body content and prints it. Listing 20.47 shows the code.

The final step in defining tag handlers is to define a class for `cwp:else`. This class is very similar to the one to handle the `then` part of the tag, except that this handler only prints the tag body from `doAfterBody` if the condition from the surrounding `IfTag` is false. The code is shown in Listing 20.48.

Listing 20.45 `IfTag.java`

```
package cwp.tags;

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

/** A tag that acts like an if/then/else. */

public class IfTag extends TagSupport {
  private boolean condition;
  private boolean hasCondition = false;

  public void setCondition(boolean condition) {
    this.condition = condition;
    hasCondition = true;
  }

  public boolean getCondition() {
    return(condition);
  }
  public void setHasCondition(boolean flag) {
    this.hasCondition = flag;
  }

  /** Has the condition field been explicitly set? */

  public boolean hasCondition() {
    return(hasCondition);
  }

  public int doStartTag() {
    return(EVAL_BODY_INCLUDE);
  }
}
```

Listing 20.46 `IfConditionTag.java`

```
package cwp.tags;

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

/** The condition part of an if tag. */

public class IfConditionTag extends BodyTagSupport {
  public int doStartTag() throws JspTagException {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (parent == null) {
      throw new JspTagException("condition not inside if");
    }
    return(EVAL_BODY_TAG);
  }
  public int doAfterBody() {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    String bodyString = getBodyContent().getString();
    if (bodyString.trim().equals("true")) {
      parent.setCondition(true);
    } else {
      parent.setCondition(false);
    }
    return(SKIP_BODY);
  }
}
```

Listing 20.47 IfThenTag.java

```
package cwp.tags;

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

/** The then part of an if tag. */

public class IfThenTag extends BodyTagSupport {
  public int doStartTag() throws JspTagException {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (parent == null) {
      throw new JspTagException("then not inside if");
    } else if (!parent.hasCondition()) {
      String warning =
        "condition tag must come before then tag";
      throw new JspTagException(warning);
    }
    return(EVAL_BODY_TAG);
  }
  public int doAfterBody() {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (parent.getCondition()) {
      try {
        BodyContent body = getBodyContent();
        JspWriter out = body.getEnclosingWriter();
        out.print(body.getString());
      } catch(IOException ioe) {
        System.out.println("Error in IfThenTag: " + ioe);
      }
    }
    return(SKIP_BODY);
  }
}
```

Listing 20.48 `IfElseTag.java`

```
package cwp.tags;

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

/** The else part of an if tag. */

public class IfElseTag extends BodyTagSupport {
  public int doStartTag() throws JspTagException {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (parent == null) {
      throw new JspTagException("else not inside if");
    } else if (!parent.hasCondition()) {
      String warning =
        "condition tag must come before else tag";
      throw new JspTagException(warning);
    }
    return(EVAL_BODY_TAG);
  }

  public int doAfterBody() {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (!parent.getCondition()) {
      try {
        BodyContent body = getBodyContent();
        JspWriter out = body.getEnclosingWriter();
        out.print(body.getString());
      } catch(IOException ioe) {
        System.out.println("Error in IfElseTag: " + ioe);
      }
    }
    return(SKIP_BODY);
  }
}
```

The Tag Library Descriptor File

Even though there is an explicit required nesting structure for the tags just defined, the tags must be declared separately in the TLD file. This means that nesting validation is performed only at request time, not at page translation time. In principle, you could instruct the system to do some validation at page translation time by using a `TagExtraInfo` class. This class has a `getVariableInfo` method that you can use to check that attributes exist and where they are used. Once you have defined a subclass of `TagExtraInfo`, you associate it with your tag in the tag library descriptor file by means of the `teiclass` element, which is used just like `tagclass`. In practice, however, `TagExtraInfo` is a bit cumbersome to use.

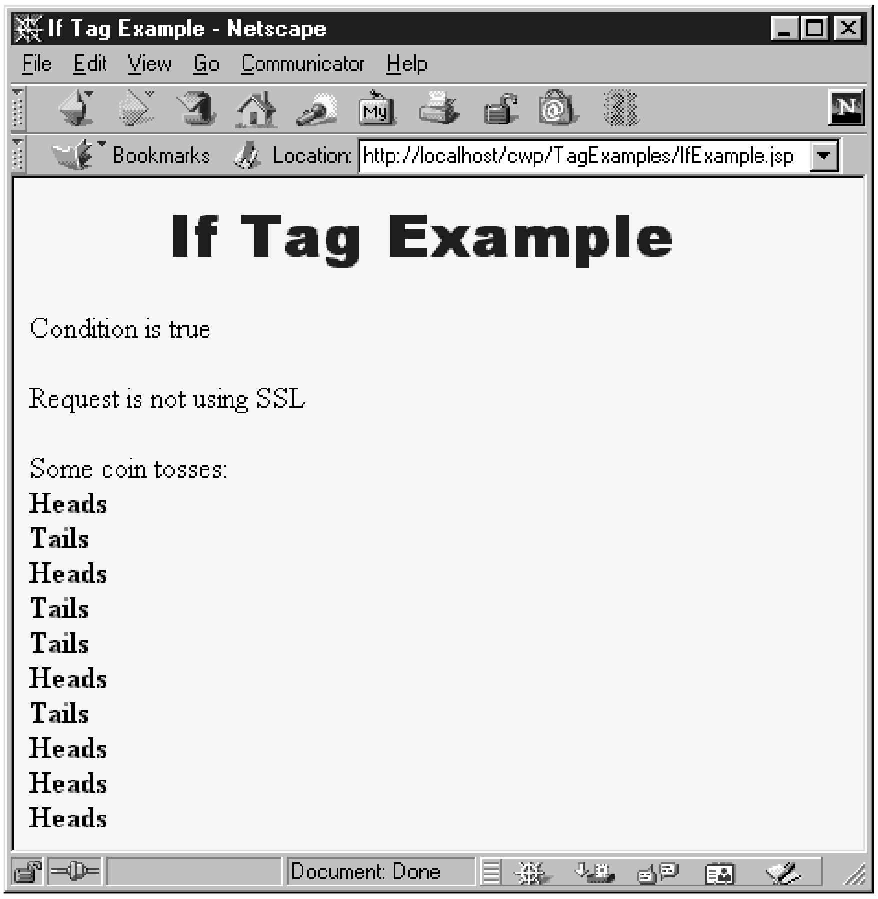
Listing 20.49 `cwp-taglib.tld` (Excerpt)

```
<tag>
  <name>if</name>
  <tagclass>cwp.tags.IfTag</tagclass>
  <info>if/condition/then/else tag.</info>
</tag>
<tag>
  <name>condition</name>
  <tagclass>cwp.tags.IfConditionTag</tagclass>
  <info>condition part of if/condition/then/else tag.</info>
</tag>
<tag>
  <name>then</name>
  <tagclass>cwp.tags.IfThenTag</tagclass>
  <info>then part of if/condition/then/else tag.</info>
</tag>
<tag>
  <name>else</name>
  <tagclass>cwp.tags.IfElseTag</tagclass>
  <info>else part of if/condition/then/else tag.</info>
</tag>
```

The JSP File

Listing 20.50 shows a page that uses the `cwp:if` tag three different ways. In the first instance, a value of `true` is hardcoded for the condition. In the second instance, a parameter from the HTTP request is used for the condition, and in the third case, a random number is generated and compared to a fixed cutoff. Figure 20-22 shows a typical result.

Figure 20-22. Result of `IfExample.jsp`.


If Tag Example - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost/cwp/TagExamples/IfExample.jsp
Document: Done

If Tag Example

Condition is true

Request is not using SSL

Some coin tosses:
Heads
Tails
Heads
Tails
Tails
Heads
Tails
Heads
Heads
Heads

Listing 20.50 `IfExample.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>If Tag Example</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>If Tag Example</H1>
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
<cwp:if>
  <cwp:condition>true</cwp:condition>
  <cwp:then>Condition is true</cwp:then>
  <cwp:else>Condition is false</cwp:else>
</cwp:if>
<P>
<cwp:if>
  <cwp:condition><%= request.isSecure() %></cwp:condition>
  <cwp:then>Request is using SSL (https)</cwp:then>
  <cwp:else>Request is not using SSL</cwp:else>
</cwp:if>
<P>
Some coin tosses:<BR>
```

```
<cwp:repeat reps="10">
  <cwp:if>
    <cwp:condition><%= Math.random() < 0.5 %></cwp:condition>
    <cwp:then><B>Heads</B><BR></cwp:then>
    <cwp:else><B>Tails</B><BR></cwp:else>
  </cwp:if>
</cwp:repeat>
</BODY>
</HTML>
```

20.8 Integrating Servlets and JSP

Servlets are great when your application requires a lot of real programming to accomplish its task. As you've seen in the previous chapter, servlets can manipulate HTTP status codes and headers, use cookies, track sessions, save information between requests, compress pages, access databases, generate GIF images on-the-fly, and perform many other tasks flexibly and efficiently. But, generating HTML with servlets can be tedious and can yield a result that is hard to modify. That's where JSP comes in; it lets you separate much of the presentation from the dynamic content. That way, you can write the HTML in the normal manner, even using HTML-specific tools and putting your Web content developers to work on your JSP documents. JSP expressions, scriptlets, and declarations let you insert simple Java code into the servlet that results from the JSP page, and directives let you control the overall layout of the page. For more complex requirements, you can wrap up Java code inside beans or define your own JSP tags.

Great. We have everything we need, right? Well, no, not quite. The assumption behind a JSP document is that it provides a *single* overall presentation. What if you want to give totally different results depending on the data that you receive? Beans and custom tags, although extremely powerful and flexible, don't overcome the limitation that the JSP page defines a relatively fixed top-level page appearance. The solution is to use *both* servlets and JavaServer Pages. If you have a complicated application that may require several substantially different presentations, a servlet can handle the initial request, partially process the data, set up beans, and then forward the results to one of a number of different JSP pages, depending on the circumstances. This approach is known as the *model view controller architecture* or *model 2* approach to JSP. For code that supports a formalization of this approach, see the Apache Struts Framework at http://jakarta.apache.org/struts/.

Forwarding Requests

The key to letting servlets forward requests or include external content is to use a `RequestDispatcher`. You obtain a `RequestDispatcher` by calling the `getRequestDispatcher` method of `ServletContext`, supplying a URL relative to the server root. For example, to obtain a `RequestDispatcher` associated with `http://yourhost/presentations/presentation1.jsp`, you would do the following:

```
String url = "/presentations/presentation1.jsp";
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher(url);
```

Once you have a `RequestDispatcher`, you use `forward` to completely transfer control to the associated URL and you use `include` to output the associated URL's content. In both cases, you supply the `HttpServletRequest` and `HttpServlet Response` as arguments. Both methods throw `ServletException` and `IOException`. For example, Listing 20.51 shows a portion of a servlet that forwards the request to one of three different JSP pages, depending on the value of the `operation` parameter. To avoid repeating the `getRequestDispatcher` call, we use a utility method called `gotoPage` that takes the URL, the `HttpServletRequest`, and the `HttpServletResponse`; gets a `RequestDispatcher`; and then calls `forward` on it.

Listing 20.51 Request Forwarding Example

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
  String operation = request.getParameter("operation");
  if (operation == null) {
    operation = "unknown";
  }
  if (operation.equals("operation1")) {
    gotoPage("/operations/presentation1.jsp",
             request, response);
  } else if (operation.equals("operation2")) {
    gotoPage("/operations/presentation2.jsp",
             request, response);
  } else {
    gotoPage("/operations/unknownRequestHandler.jsp",
             request, response);
  }
}

private void gotoPage(String address,
                      HttpServletRequest request,
                      HttpServletResponse response)
    throws ServletException, IOException {
  RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher(address);
  dispatcher.forward(request, response);
}
```

Using Static Resources

In most cases, you forward requests to a JSP page or another servlet. In some cases, however, you might want to send the request to a static HTML page. In an e-commerce site, for example, requests that indicate that the user does not have a valid account name might be forwarded to an account application page that uses HTML forms to gather the requisite information. With `GET` requests, forwarding requests to a static HTML page is perfectly legal and requires no special syntax; just supply the address of the HTML page as the argument to `getRequestDispatcher`. However, since forwarded requests use the same request method as the original request, `POST` requests cannot be forwarded to normal HTML pages. The solution to this problem is to simply rename the HTML page to have a `.jsp` extension. Renaming `somefile.html` to `somefile.jsp` does not change its output for `GET` requests, but `somefile.html` cannot handle `POST` requests, whereas `somefile.jsp` gives an identical response for both `GET` and `POST`.

Supplying Information to the Destination Pages

There are three main places for the servlet to store the data that the JSP pages will use: in the `HttpServletRequest`, in the `HttpSession`, and in the `ServletContext`. These storage locations correspond to the three non-default values of the scope attribute `jsp:useBean`: `request`, `session`, and `application`.

1. **Storing data that servlet looked up and that JSP page will use only in this request.** The servlet would create and store data as follows:

   ```
   SomeClass value = new SomeClass(...);
   request.setAttribute("key", value);
   ```

   Then, the servlet would forward to a JSP page that uses the following to retrieve the data:

   ```
   <jsp:useBean id="key" class="SomeClass"
                scope="request" />
   ```

2. **Storing data that servlet looked up and that JSP page will use in this request and in later requests from same client.** The servlet would create and store data as follows:

   ```
   SomeClass value = new SomeClass(...);
   HttpSession session = request.getSession(true);
   session.setAttribute("key", value);
   ```

   Then, the servlet would forward to a JSP page that uses the following to retrieve the data:

   ```
   <jsp:useBean id="key" class="SomeClass"
                scope="session" />
   ```

3. **Storing data that servlet looked up and that JSP page will use in this request and in later requests from any client.** The servlet would create and store data as follows:

   ```
   SomeClass value = new SomeClass(...);
   getServletContext().setAttribute("key", value);
   ```

   Then, the servlet would forward to a JSP page that uses the following to retrieve the data:

   ```
   <jsp:useBean id="key" class="SomeClass"
                scope="application" />
   ```

Interpreting Relative URLs in the Destination Page

Although a servlet can forward the request to arbitrary locations on the same server, the process is quite different from that of using the `sendRedirect` method of `HttpServletResponse`. First, `sendRedirect` requires the client to reconnect to the new resource, whereas the `forward` method of `RequestDispatcher` is handled completely on the server. Second, `sendRedirect` does not automatically preserve all of the request data; `forward` does. Third, `sendRedirect` results in a different final URL, whereas with `forward`, the URL of the original servlet is maintained.

This final point means that, if the destination page uses relative URLs for images or style sheets, it needs to make them relative to the server root, not to the destination page's actual location. For example, consider the following style sheet entry:

```
<LINK REL=STYLESHEET
      HREF="my-styles.css"
      TYPE="text/css">
```

If the JSP page containing this entry is accessed by means of a forwarded request, `my-`

`styles.css` will be interpreted relative to the URL of the *originating* servlet, not relative to the JSP page itself, almost certainly resulting in an error. The solution is to give the full server path to the style sheet file, as follows:

```
<LINK REL=STYLESHEET
      HREF="/path/my-styles.css"
      TYPE="text/css">
```

The same approach is required for addresses used in `<IMG SRC=...>` and `<A HREF=...>`.

Alternative Means of Getting a RequestDispatcher

In servers that support version 2.2 of the servlet specification, there are two additional ways of obtaining a `RequestDispatcher` besides the `getRequestDispatcher` method of `ServletContext`.

First, since most servers let you register explicit names for servlets or JSP pages, it makes sense to access them by name rather than by path. Use the `getNamedDispatcher` method of `ServletContext` for this task.

Second, you might want to access a resource by a path relative to the current servlet's location, rather than relative to the server root. This approach is not common when servlets are accessed in the standard manner (`http://host/servlet/ServletName`), because JSP files would not be accessible by means of `http://host/servlet/...` since that URL is reserved especially for servlets.

However, it is common to register servlets under another path, and in such a case you can use the `getRequestDispatcher` method of `HttpServletRequest` rather than the one from `ServletContext`. For example, if the originating servlet is at `http://host/travel/TopLevel`,

```
getServletContext().getRequestDispatcher("/travel/cruises.jsp")
```

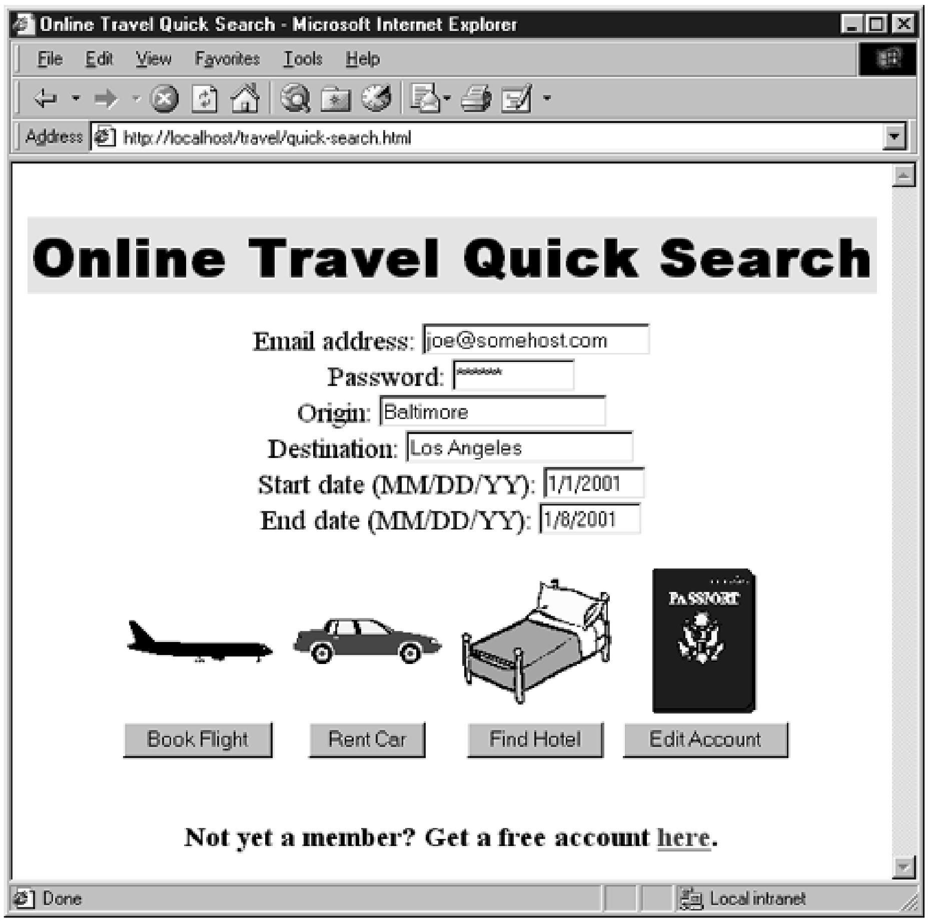
could be replaced by

```
request.getRequestDispatcher("cruises.jsp");
```

Example: An On-Line Travel Agent

Consider the case of an on-line travel agent that has a quick-search page, as shown in Figure 20-23 and Listing 20.52. Users need to enter their e-mail address and password to associate the request with their previously established customer account. Each request also includes a trip origin, trip destination, start date, and end date. However, the action that will result will vary substantially based upon the action requested. For example, pressing the "Book Flights" button should show a list of available flights on the dates specified, ordered by price (see Figure 20-24). The user's real name, frequent flyer information, and credit card number should be used to generate the page. On the other hand, selecting "Edit Account" should show any previously entered customer information, letting the user modify values or add entries. Likewise, the actions resulting from choosing "Rent Cars" or "Find Hotels" will share much of the same customer data but will have a totally different presentation.

Figure 20-23. Front end to travel servlet (see Listing 20.52).

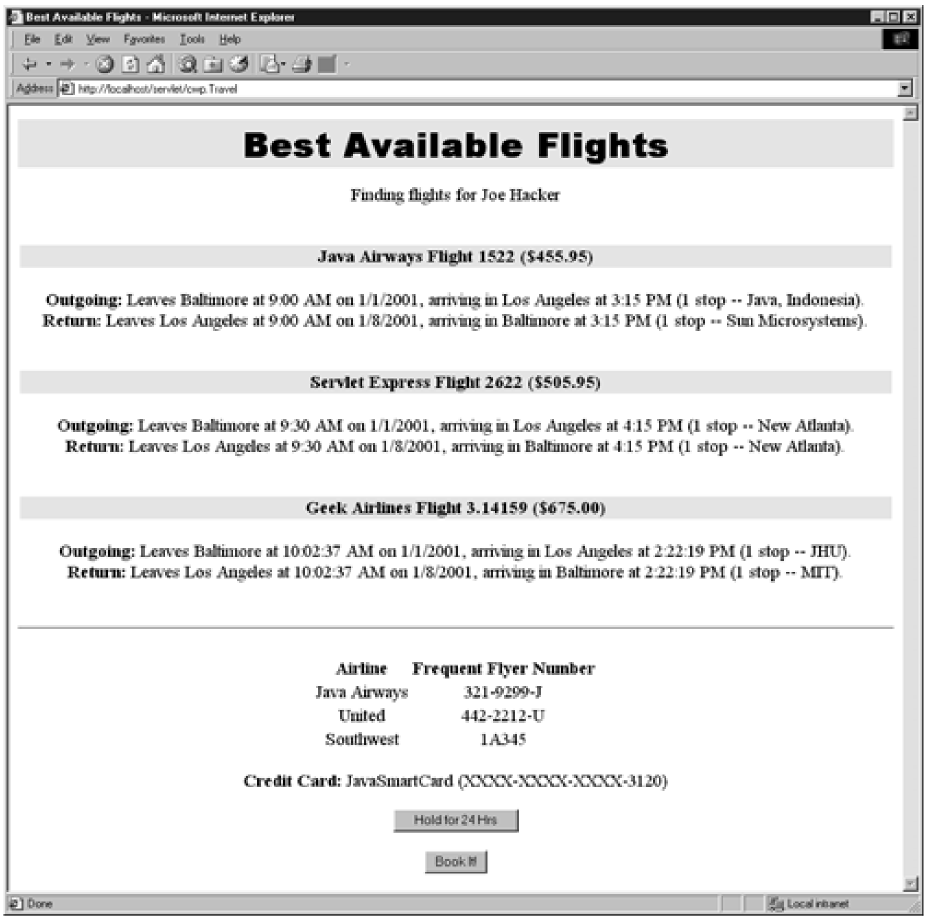

Online Travel Quick Search - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/travel/quick-search.html
Book Flight
Rent Car
Find Hotel
Edit Account
PASSPORT
Done
Local intranet

Online Travel Quick Search

Email address: joe@somehost.com
Password: ******
Origin: Baltimore
Destination: Los Angeles
Start date (MM/DD/YY): 1/1/2001
End date (MM/DD/YY): 1/8/2001

Not yet a member? Get a free account here.

Figure 20-24. Result of travel servlet (Listing 20.53) dispatching request to `BookFlights.jsp` (Listing 20.54).


Best Available Flights - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlet/cwp.Travel
Hold for 24 Hrs
Book It!
Done
Local intranet

Best Available Flights

Finding flights for Joe Hacker

Java Airways Flight 1522 ($455.95)

Outgoing: Leaves Baltimore at 9:00 AM on 1/1/2001, arriving in Los Angeles at 3:15 PM (1 stop -- Java, Indonesia).
Return: Leaves Los Angeles at 9:00 AM on 1/8/2001, arriving in Baltimore at 3:15 PM (1 stop -- Sun Microsystems).

Servlet Express Flight 2622 ($505.95)

Outgoing: Leaves Baltimore at 9:30 AM on 1/1/2001, arriving in Los Angeles at 4:15 PM (1 stop -- New Atlanta).
Return: Leaves Los Angeles at 9:30 AM on 1/8/2001, arriving in Baltimore at 4:15 PM (1 stop -- New Atlanta).

Geek Airlines Flight 3.14159 ($675.00)

Outgoing: Leaves Baltimore at 10:02:37 AM on 1/1/2001, arriving in Los Angeles at 2:22:19 PM (1 stop -- JHU).
Return: Leaves Los Angeles at 10:02:37 AM on 1/8/2001, arriving in Baltimore at 2:22:19 PM (1 stop -- MIT).

Airline	Frequent Flyer Number
Java Airways	321-9299-J
United	442-2212-U
Southwest	1A345

Credit Card: JavaSmartCard (XXXX-XXXX-XXXX-3120)

To accomplish the desired behavior, the front end (Listing 20.52) submits the request to the top-level travel servlet shown in Listing 20.53. This servlet looks up the customer information (see http://www.corewebprogramming.com for the actual code used, but this would be replaced by a database lookup in real life), puts it in the `HttpSession` object associating the value (of type `cwp.TravelCustomer`) with the name `customer`, and then forwards the request to a different JSP page corresponding to each of the possible actions. The destination page (see Listing 20.54 and the result in Figure 20-24) looks up the customer information by means of

```
<jsp:useBean id="customer"
             class="cwp.TravelCustomer"
             scope="session" />
```

then uses `jsp:getProperty` to insert customer information into various parts of the page.

You should pay careful attention to the `TravelCustomer` class (shown partially in Listing 20.55; available in full at www.corewebprogramming.com). In particular, note that the class spends a considerable amount of effort making the customer information accessible as plain strings or even HTML-formatted strings through simple properties. Every task that requires any substantial amount of programming is spun off into the bean, rather than being performed in the JSP page itself. This is typical of servlet/JSP integration—the use of JSP does not *entirely* obviate the need to format data as strings or HTML in Java code. Significant up-front effort to make the data conveniently available to JSP more than pays for itself when multiple JSP pages access the same type of data.

Listing 20.52 `/travel/quick-search.html` (Excerpt)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
```

```
<HEAD>
  <TITLE>Online Travel Quick Search</TITLE>
  <LINK REL=STYLESHEET
        HREF="travel-styles.css"
        TYPE="text/css">
</HEAD>
<BODY>
<BR>
<H1>Online Travel Quick Search</H1>
<FORM ACTION="/servlet/cwp.Travel" METHOD="POST">
<CENTER>
Email address: <INPUT TYPE="TEXT" id="emailAddress"><BR>
Password: <INPUT TYPE="PASSWORD" id="password" SIZE=10><BR>
...
<TABLE CELLSPACING=1>
<TR>
  <TH> <IMG SRC="airplane.gif" WIDTH=100 HEIGHT=29
                 ALIGN="TOP" ALT="Book Flight">
  ...
<TR>
  <TH><SMALL>
      <INPUT TYPE="SUBMIT" id="flights" VALUE="Book Flight">
      </SMALL>
  ...
</TABLE>
</CENTER>
</FORM>
...
</BODY>
</HTML>
```

Listing 20.53 **`Travel.java`**

```
package cwp;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Top-level travel-processing servlet. This servlet sets up
 *  the customer data as a bean, then forwards the request
 *  to the airline booking page, the rental car reservation
 *  page, the hotel page, the existing account modification
 *  page, or the new account page.
 */

public class Travel extends HttpServlet {
  private TravelCustomer[] travelData;

  public void init() {
    travelData = TravelData.getTravelData();
  }
  /** Since password is being sent, use POST only. */

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    String emailAddress = request.getParameter("emailAddress");
    String password = request.getParameter("password");
    TravelCustomer customer =
      TravelCustomer.findCustomer(emailAddress, travelData);
    if ((customer == null) || (password == null) ||
        (!password.equals(customer.getPassword()))) {
      gotoPage("/travel/accounts.jsp", request, response);
    }
    // The methods that use the following parameters will
    // check for missing or malformed values.
    customer.setStartDate(request.getParameter("startDate"));
    customer.setEndDate(request.getParameter("endDate"));
    customer.setOrigin(request.getParameter("origin"));
    customer.setDestination(request.getParameter
                              ("destination"));
    HttpSession session = request.getSession(true);
    session.setAttribute("customer", customer);
    if (request.getParameter("flights") != null) {
      gotoPage("/travel/BookFlights.jsp",
               request, response);
    } else if (request.getParameter("cars") != null) {
      gotoPage("/travel/RentCars.jsp",
               request, response);
    } else if (request.get0.Parameter("hotels") != null) {
      gotoPage("/travel/FindHotels.jsp",
               request, response);
    } else if (request.getParameter("cars") != null) {
      gotoPage("/travel/EditAccounts.jsp",
               request, response);
    } else {
      gotoPage("/travel/IllegalRequest.jsp",
               request, response);
    }
  }

  private void gotoPage(String address,
                        HttpServletRequest request,
                        HttpServletResponse response)
      throws ServletException, IOException {
    RequestDispatcher dispatcher =
      getServletContext().getRequestDispatcher(address);
    dispatcher.forward(request, response);
  }
}
```

Listing 20.54 `BookFlights.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Best Available Flights</TITLE>
  <LINK REL=STYLESHEET
        HREF="/travel/travel-styles.css"
        TYPE="text/css">
</HEAD>
<BODY>
<H1>Best Available Flights</H1>
<CENTER>
<jsp:useBean id="customer"
             class="cwp.TravelCustomer"
             scope="session" />
Finding flights for
<jsp:getProperty id="customer" property="fullName" />
<P>
<jsp:getProperty id="customer" property="flights" />
<P><BR><HR><BR>
<FORM ACTION="/servlet/BookFlight">
<jsp:getProperty id="customer"
                 property="frequentFlyerTable" />
<P>
<B>Credit Card:</B>
<jsp:getProperty id="customer" property="creditCard" />
<P>
<INPUT TYPE="SUBMIT" id="holdButton" VALUE="Hold for 24 Hrs">
<P>
<INPUT TYPE="SUBMIT" id="bookItButton" VALUE="Book It!">
</FORM>
</CENTER>
</BODY>
</HTML>
```

Listing 20.55 `TravelCustomer.java` (Excerpt)

```
package cwp;

import java.util.*;
import java.text.*;

/** Describes a travel services customer. Implemented
 *  as a bean with some methods that return data in HTML
 *  format, suitable for access from JSP.
 */

public class TravelCustomer {
  private String emailAddress, password, firstName, lastName;
  private String creditCardName, creditCardNumber;
  private String phoneNumber, homeAddress;
  private String startDate, endDate;
  private String origin, destination;
  private FrequentFlyerInfo[] frequentFlyerData;
  private RentalCarInfo[] rentalCarData;
  private HotelInfo[] hotelData;

  public TravelCustomer(String emailAddress,
                        String password,
                        String firstName,
                        String lastName,
                        String creditCardName,
                        String creditCardNumber,
                        String phoneNumber,
                        String homeAddress,
                        FrequentFlyerInfo[] frequentFlyerData,
                        RentalCarInfo[] rentalCarData,
                        HotelInfo[] hotelData) {
    setEmailAddress(emailAddress);
    setPassword(password);
    setFirstName(firstName);
    setLastName(lastName);
    setCreditCardName(creditCardName);
    setCreditCardNumber(creditCardNumber);
    setPhoneNumber(phoneNumber);
    setHomeAddress(homeAddress);
    setStartDate(startDate);
    setEndDate(endDate);
    setFrequentFlyerData(frequentFlyerData);
    setRentalCarData(rentalCarData);
    setHotelData(hotelData);
  }
  public String getEmailAddress() {
    return(emailAddress);
  }

  public void setEmailAddress(String emailAddress) {
    this.emailAddress = emailAddress;
  }
  ...
  public String getCreditCard() {
    String cardName = getCreditCardName();
    String cardNum = getCreditCardNumber();
    cardNum = cardNum.substring(cardNum.length() - 4);
    return(cardName + " (XXXX-XXXX-XXXX-" + cardNum + ")");
  }
  ...
  private String getFlightDescription(String airline,
                                      String flightNum,
                                      String price,
                                      String stop1,
                                      String stop2,
                                      String time1,
                                      String time2,
                                      String flightOrigin,
                                            String flightDestination,
                                            String flightStartDate,
                                            String flightEndDate) {
    String flight =
      "<P><BR>\n" +
      "<TABLE WIDTH=\"100%\"><TR><TH CLASS=\"COLORED\">\n" +
      "<B>" + airline + " Flight " + flightNum +
      " ($" + price + ")</B></TABLE><BR>\n" +
      "<B>Outgoing:</B> Leaves " + flightOrigin +
      " at " + time1 + " AM on " + flightStartDate +
      ", arriving in " + flightDestination +
      " at " + time2 + " PM (1 stop -- " + stop1 + ").\n" +
      "<BR>\n" +
      "<B>Return:</B> Leaves " + flightDestination +
      " at " + time1 + " AM on " + flightEndDate +
      ", arriving in " + flightOrigin +
      " at " + time2 + " PM (1 stop -- " + stop2 + ").\n";
    return(flight);
  }
  ...
}
```

Forwarding Requests From JSP Pages

The most common request-forwarding scenario is that the request first comes to a servlet and the servlet forwards the request to a JSP page. The reason a servlet usually handles the original request is that checking request parameters and setting up beans requires a lot of programming, and it is more convenient to do this programming in a servlet than in a JSP document. The reason that the destination page is usually a JSP document is that JSP simplifies the process of creating the HTML content.

However, just because this is the *usual* approach doesn't mean that it is the *only* way of doing things. It is certainly possible for the destination page to be a servlet. Similarly, it is quite possible for a JSP page to forward requests elsewhere. For example, a request might go to a JSP page that normally presents results of a certain type and that forwards the request elsewhere only when it receives unexpected values.

Sending requests to servlets instead of JSP pages requires no changes whatsoever in the use of the `RequestDispatcher`. However, there is special syntactic support for forwarding requests from JSP pages. In JSP, the `jsp:forward` action is simpler and easier to use than wrapping up `RequestDispatcher` code in a scriptlet. This action takes the following form:

```
<jsp:forward page="Relative URL" />
```

The `page` attribute is allowed to contain JSP expressions so that the destination can be computed at request time. For example, the following sends about half the visitors to `http://host/examples/page1.jsp` and the others to `http://host/examples/page2.jsp`.

```
<% String destination;
   if (Math.random() > 0.5) {
     destination = "/examples/page1.jsp";
   } else {
     destination = "/examples/page2.jsp";
   }
%>
<jsp:forward page="<%= destination %>" />
```

20.9 Summary

In principle, everything that can be accomplished with JSP can be accomplished with servlets alone. In practice, JSP makes the generation of HTML content easier and more reliable. In doing so, JSP encourages a separation between the Java code that generates the content and the HTML code that presents it, thus permitting the use of standard HTML development tools, a division of labor on large projects, and the ability to change either the content or the presentation without altering the other piece.

As a result, few real sites use servlets alone. That does not, however, mean that they use JSP entirely in lieu of servlets. Instead, complex sites typically combine servlets and JSP, using servlets for some pages, JSP for others, and a combination for the remainder.

In this chapter we gave you the basic building blocks of JSP and showed you how to control both the specific dynamic content that is invoked and the high-level structure of the servlet that results from each JSP page. We also showed what tools are needed to make complex pages maintainable by non-Java programmers: beans and custom tags. Finally, we showed you how to put all those pieces together in integrated servlet/JSP applications.

Whew. Give yourself a rest, then go buy an answering machine to screen all those calls that will soon be arriving from eager potential employers.

CONTENTS