Including Files and Applets in JSP Pages

Topics in This Chapter

- Using `jsp:include` to include pages at request time
- Using `<%@ include ... %>` (the `include` directive) to include files at page translation time
- Understanding why `jsp:include` is usually better than the `include` directive
- Using `jsp:plugin` to include applets for the Java Plug-in

Chapter 13

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

- **The `jsp:include` action.** The `jsp:include` action lets you include the output of a page at request time. Its main advantage is that it saves you from changing the main page when the included pages change. Its main disadvantage is that since it includes the *output* of the secondary page, not the secondary page's actual *code* as with the `include` directive, the included pages cannot use any JSP constructs that affect the main page as a whole. The advantages generally far outweigh the disadvantages, and you will almost certainly use it much more than the other inclusion mechanisms. Use of `jsp:include` is discussed in Section 13.1.
- **The `include` directive.** This construct lets you insert JSP code into the main page before that main page is translated into a servlet. Its main advantage is that it is powerful: the included code can contain JSP constructs such as field definitions and content-type settings that affect the main page as a whole. Its main disadvantage is that it is hard to maintain: you have to update the main page whenever any of the included pages change. Use of the `include` directive is discussed in Section 13.2.
- **The `jsp:plugin` action.** Although this book 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. Its main advantage is that it saves you from

writing long, tedious, and error-prone `OBJECT` and `EMBED` tags in your HTML. Its main disadvantage is that it applies to applets, and applets are relatively infrequently used. Use of `jsp:plugin` is discussed in Section 13.4.

13.1 Including Pages at Request Time: The jsp:include Action

Suppose you have a series of pages, all of which have the same navigation bar, contact information, or footer. What can you do? Well, one common "solution" is to cut and paste the same HTML snippets into all the pages. This is a bad idea because when you change the common piece, you have to change every page that uses it. Another common solution is to use some sort of server-side include mechanism whereby the common piece gets inserted as the page is requested. This general approach is a good one, but the typical mechanisms are server specific. Enter `jsp:include`, a portable mechanism that lets you insert any of the following into the JSP output:

- The content of an HTML page.
- The content of a plain text document.
- The output of JSP page.
- The output of a servlet.

The `jsp:include` action includes the output of a secondary page at the time the main page is requested. Although the *output* of the included pages cannot contain JSP, the pages can be the result of resources that use servlets or 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. The server runs the included page in the usual way and places the output into the main page. This is precisely the behavior of the `include` method of the `RequestDispatcher` class (see Chapter 15, "Integrating Servlets and JSP: The Model View Controller (MVC) Architecture"), which is what servlets use if they want to do this type of file inclusion.

The page Attribute: Specifying the Included Page

You designate the included page with the `page` attribute, as shown below. This attribute is required; it should be a relative URL referencing the resource whose output should be included.

```
<jsp:include page="relative-path-to-resource" />
```

Relative URLs that do not start with a slash are interpreted relative to the location of the main page. Relative URLs that start with a slash are interpreted relative to the base Web application directory, *not* relative to the server root. For example, consider a JSP page in the `headlines` Web application that is accessed by the URL http://*host*/headlines/sports/table-tennis.jsp. The table-tennis.jsp file is in the sports subdirectory of whatever directory is used by the `headlines` Web application. Now, consider the following two include statements.

```
<jsp:include page="bios/cheng-yinghua.jsp" />
<jsp:include page="/templates/footer.jsp" />
```

In the first case, the system would look for cheng-yinghua.jsp in the bios subdirectory of sports (i.e., in the sports/bios sub-subdirectory of the main directory of the `headlines` application). In the second case, the system would look for footer.jsp in the templates subdirectory of the `headlines` application, *not* in the templates subdirectory of the server root. The `jsp:include` action *never* causes the system to look at files outside of the current Web application. If you have trouble remembering how the system interprets URLs that begin with slashes, remember this rule: they are interpreted relative to the current Web application whenever the server handles them; they are interpreted relative to the server root only when the client (browser) handles them. For example, the URL in

```
<jsp:include page="/path/file" />
```

is interpreted within the context of the current Web application because the server resolves the URL; the browser never sees it. But, the URL in

```
<IMG SRC="/path/file" ...>
```

is interpreted relative to the server's base directory because the browser resolves the URL; the browser knows nothing about Web applications. For information on Web applications, see Section 2.11.

Core Note

URLs that start with slashes are interpreted differently by the server than by the browser. The server always interprets them relative to the current Web application. The browser always interprets them relative to the server root.

Finally, note that you are permitted to place your pages in the WEB-INF directory. Although the client is prohibited from directly accessing files in this directory, it is the server, not the client, that accesses files referenced by the `page` attribute of

jsp:include. In fact, placing the included pages in WEB-INF is a recommended practice; doing so will prevent them from being accidentally accessed by the client (which would be bad, since they are usually incomplete HTML documents).

Core Approach

To prevent the included files from being accessed separately, place them in WEB-INF or a subdirectory thereof.

XML Syntax and jsp:include

The jsp:include action is one of the first JSP constructs we have seen that has only XML syntax, with no equivalent "classic" syntax. If you are unfamiliar with XML, note three things:

- **XML element names can contain colons.** So, do not be thrown off by the fact that the element name is jsp:include. In fact, the XML-compatible version of all standard JSP elements starts with the jsp prefix (or namespace).
- **XML tags are case sensitive.** In standard HTML, it does not matter if you say BODY, body, or Body. In XML, it matters. So, be sure to use jsp:include in all lower case.
- **XML tags must be explicitly closed.** In HTML, there are container elements such as H1 that have both start and end tags (<H1> ... </H1>) as well as standalone elements such as IMG or HR that have no end tags (<HR>). In addition, the HTML specification defines the end tags of some container elements (e.g., TR, P) to be optional. In XML, all elements are container elements, and end tags are never optional. However, as a convenience, you can replace bodyless snippets such as <blah></blah> with <blah/>. So when using jsp:include, be sure to include that trailing slash.

The flush Attribute

In addition to the required page attribute, jsp:include has a second attribute: flush, as shown below. This attribute is optional; it specifies whether the output stream of the main page should flushed before the inclusion of the page (the default is false). Note, however, that in JSP 1.1, flush was a required attribute and the only legal value was true.

```
<jsp:include page="relative-path-to-resource" flush="true" />
```

A News Headline Page

As an example of a typical use of `jsp:include`, consider the simple news summary page shown in Listing 13.1. Page developers can change the news items in the files Item1.html through Item3.html (Listings 13.2 through 13.4) without having to update the main news page. Figure 13–1 shows the result.

Notice that the included pieces are not complete Web pages. The included pages can be HTML files, plain text files, JSP pages, or servlets (but with JSP pages and servlets, only the output of the page is included, not the actual code). In all cases, however, the client sees only the *composite* result. So, if both the main page and the included pieces contain tags such as `DOCTYPE`, `BODY`, etc., the result will be illegal HTML because these tags will appear twice in the result that the client sees. With servlets and JSP, it is always a good habit to view the HTML source and submit the URL to an HTML validator (see Section 3.5, "Simple HTML-Building Utilities"). When `jsp:include` is used, this advice is even more important because beginners often erroneously design both the main page and the included page as complete HTML documents.

Core Approach

Do not use complete HTML documents for your included pages. Include only the HTML tags appropriate to the place where the included files will be inserted.

Listing 13.1 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>
```

Listing 13.1 WhatsNew.jsp *(continued)*

```
<P>
Here is a summary of our three most recent news stories:
<OL>
  <LI><jsp:include page="/WEB-INF/Item1.html" />
  <LI><jsp:include page="/WEB-INF/Item2.html" />
  <LI><jsp:include page="/WEB-INF/Item3.html" />
</OL>
</BODY></HTML>
```

Listing 13.2 /WEB-INF/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 13.3 /WEB-INF/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 13.4 /WEB-INF/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>
```

Figure 13–1 Including files at request time lets you update the individual files without changing the main page.

The jsp:param Element: Augmenting Request Parameters

The included page uses the same request object as the originally requested page. As a result, the included page normally sees the same request parameters as the main page. If, however, you want to add to or replace those parameters, you can use the `jsp:param` element (which has `name` and `value` attributes) to do so. For example, consider the following snippet.

```
<jsp:include page="/fragments/StandardHeading.jsp">
  <jsp:param name="bgColor" value="YELLOW" />
</jsp:include>
```

Now, suppose that the main page is invoked by means of http://host/path/MainPage.jsp?fgColor=RED. In such a case, the following list summarizes the results of various `getParameter` calls.

- **In main page (MainPage.jsp).** (Regardless of whether the `getParameter` calls are before or after the file inclusion.)
  - `request.getParameter("fgColor")` returns `"RED"`.
  - `request.getParameter("bgColor")` returns `null`.

- **In included page (StandardHeading.jsp).**
  - `request.getParameter("fgColor")` returns `"RED"`.
  - `request.getParameter("bgColor")` returns `"YELLOW"`.

If the main page receives a request parameter that is also specified with the `jsp:param` element, the value from `jsp:param` takes precedence only in the included page.

13.2 Including Files at Page Translation Time: The include Directive

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" %>
```

Think of the `include` directive as a preprocessor: the included file is inserted character for character into the main page, then the resultant page is treated as a single JSP page. So, the fundamental difference between `jsp:include` and the `include` directive is the time at which they are invoked: `jsp:include` is invoked at request time, whereas the `include` directive is invoked at page translation time. However, there are more implications of this difference than you might first think. We summarize them in Table 13.1.

Table 13.1 Differences Between `jsp:include` and the `include` Directive

	`jsp:include` Action	**`include` Directive**
What does basic syntax look like?	`<jsp:include page="..." />`	`<%@ include file="..." %>`
When does inclusion occur?	Request time	Page translation time
What is included?	Output of page	Actual content of file
How many servlets result?	Two (main page and included page each become a separate servlet)	One (included file is inserted into main page, then that page is translated into a servlet)

Table 13.1 Differences Between `jsp:include` and the `include` Directive *(continued)*

	`jsp:include` Action	**`include` Directive**
Can included page set response headers that affect the main page?	No	Yes
Can included page define fields or methods that main page uses?	No	Yes
Does main page need to be updated when included page changes?	No	Yes
What is the equivalent servlet code?	`include` method of `RequestDispatcher`	None
Where is it discussed?	Section 13.1	Section 13.2 (this section)

There are many ramifications of the fact that the included file is inserted at page translation time with the `include` directive (`<%@ include ... %>`), not at request time as with `jsp:include`. However, there are two really important implications: maintenance and power. We discuss these two items in the following two subsections.

Maintenance Problems with the include Directive

The first ramification of the inclusion occurring at page translation time is that it is much more difficult to maintain pages that use the `include` directive than is the case with `jsp:include`. With the `include` directive (`<%@ include ... %>`), if the included file changes, all the JSP files that use it may need to be updated. Servers are required to detect when a JSP page changes and to translate it into a new servlet before handling the next request. Unfortunately, however, they are not required to detect when the included file changes, only when the main page changes. Servers are *allowed* to support a mechanism for detecting that an included file has changed (and then recompiling the servlet), but they are not *required* to do so. In practice, few do. So, with most servers, whenever an included file changes, *you* have to update the modification dates of each JSP page that uses the file.

This is a significant inconvenience; it results in such serious maintenance problems that the `include` directive should be used only in situations in which `jsp:include` would not suffice. Some developers have argued that using the

include directive results in code that executes faster than it would with the jsp:include action. Although this may be true in principle, the performance difference is so small that it is difficult to measure, and the maintenance advantages of jsp:include are so great that it is virtually always preferred when both options are available. In fact, some developers find the maintenance burden of the include directive so onerous that they avoid it altogether. Perhaps this is an overreaction, but, at the very least, reserve the include directive for situations for which you really need the extra power it affords.

Core Approach

For file inclusion, use jsp:include whenever possible. Reserve the include directive (<%@ include ... %>) for cases in which the included file defines fields or methods that the main page uses or when the included file sets response headers of the main page.

Additional Power from the include Directive

If the include directive results in hard-to-maintain code, why would anyone want to use it? Well, that brings up the second difference between jsp:include and the include directive. The include directive is more powerful. With the include directive, 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 line of code:

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

In such a case, you could do the following in the main page:

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

With jsp:include, of course, this would be impossible because of the undefined accessCount variable; the main page would not translate successfully into a servlet. Besides, even if it could be translated without error, there would be no point; jsp:include includes the output of the auxiliary page, and snippet.jsp has no output.

Updating the Main Page

With most servers, if you use the `include` directive and change the included file, you also have to update the modification date of the main 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 9/1/03 --%>
<%@ include file="Navbar.jsp" %>
```

Core Warning

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

XML Syntax for the include Directive

The XML-compatible equivalent of

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

is

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

When this form is used, both the main page and the included file must use XML-compatible syntax throughout.

Example: Reusing Footers

As an example of a situation in which you would use the `include` directive instead of `jsp:include`, suppose that you have a JSP page that generates an HTML snippet containing a small footer that includes access counts and information about the most recent accesses to the current page. Listing 13.5 shows just such a page.

Now suppose you have several pages that want to have footers of that type. You could put the footer in WEB-INF (where it is protected from direct client access) and then have the pages that want to use it do so with the following.

```
<%@ include file="/WEB-INF/ContactSection.jsp" %>
```

Listing 13.6 shows a page that uses this approach; Figure 13–2 shows the result.

"Hold on!" you say, "Yes, ContactSection.jsp defines some instance variables (fields). And, if the main page *used* those instance variables, I would agree that you would have to use the `include` directive. But, in this particular case, the main page does not use the instance variables, so `jsp:include` should be used instead. Right?" Wrong. If you used `jsp:include` here, then all the pages that used the footer would see the same access count. You want each page that uses the footer to maintain a different access count. You do not want ContactSection.jsp to be its own servlet, you want ContactSection.jsp to provide code that will be part of each *separate* servlet that results from a JSP page that uses ContactSection.jsp. You need the `include` directive.

Listing 13.5 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; 2003
<A HREF="http//www.my-company.com/">my-company.com</A>.
This page has been accessed <%= ++accessCount %>
times since server reboot. It was most recently accessed from
<%= accessHost %> at <%= accessDate %>.
<% accessHost = request.getRemoteHost(); %>
<% accessDate = new Date(); %>
```

Listing 13.6 SomeRandomPage.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some Random Page</TITLE>
<META NAME="author" CONTENT="J. Random Hacker">
<META NAME="keywords"
      CONTENT="foo,bar,baz,quux">
```

Listing 13.6 SomeRandomPage.jsp *(continued)*

```
<META NAME="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="/WEB-INF/ContactSection.jsp" %>
</BODY></HTML>
```

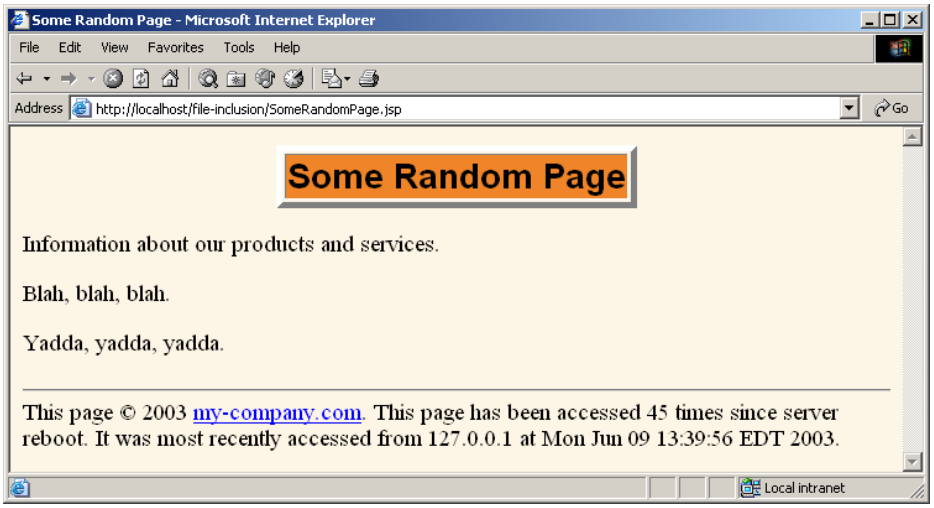

Figure 13–2 Result of SomeRandomPage.jsp. It uses the `include` directive so that it maintains access counts and most-recent-hosts entries separately from any other pages that use ContactSection.jsp.

13.3 Forwarding Requests with jsp:forward

You use `jsp:include` to combine output from the main page and the auxiliary page. Instead, you can use `jsp:forward` to obtain the complete output from the auxiliary page. For example, here is a page that randomly selects either `page1.jsp` or `page2.jsp` to output.

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

To use `jsp:forward`, the main page must not have any output. This brings up the question, what benefit does JSP provide, then? The answer is, none! In fact, use of JSP is a hindrance in this type of situation because a real situation would be more complex, and complex code is easier to develop and test in a servlet than it is in a JSP page. We recommend that you completely avoid the use of `jsp:forward`. If you want to perform a task similar to this example, use a servlet and have it call the `forward` method of `RequestDispatcher`. See Chapter 15 for details.

13.4 Including Applets for the Java Plug-In

Early in the evolution of the Java programming language, the main application area was applets (Java programs embedded in Web pages and executed by Web browsers). Furthermore, most browsers supported the most up-to-date Java version. Now, however, applets are a very small part of the Java world, and the only major browser that supports the Java 2 platform (i.e., JDK 1.2–1.4) is Netscape 6 and later. This leaves applet developers with three choices:

- Develop the applets with JDK 1.1 or even 1.02 (to support *really* old browsers).
- Have users install version 1.4 of the Java Runtime Environment (JRE), then use JDK 1.4 for the applets.

- Have users install any version of the Java 2 Plug-in, then use Java 2 for the applets.

The first option is the one generally chosen for applets that will be deployed to the general public, because that option does not require users to install any special software. You need no special JSP syntax to use this option: just use the normal HTML `APPLET` tag. Just remember that `.class` files for applets need to go in the Web-accessible directories, not `WEB-INF/classes`, because it is the browser, not the server, that executes them. However, the lack of support for the Java 2 Platform imposes several restrictions on these applets:

- 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.

So, developers of complex applets for corporate intranets often choose one of the second two options.

The second option is best if the users all have Internet Explorer 6 (or later) or Netscape 6 (or later). With those browsers, version 1.4 of the JRE will replace the Java Virtual Machine (JVM) that comes bundled with the browser. Again, you need no special JSP syntax to use this option: just use the normal HTML `APPLET` tag. And again, remember that `.class` files for applets need to go in the Web-accessible directories, not `WEB-INF/classes`, because it is the browser, not the server, that executes them.

Core Approach

No matter what approach you use for applets, the applet `.class` files must go in the Web-accessible directories, not in `WEB-INF/classes`. The browser, not the server, uses them.

In large organizations, however, many users have earlier browser versions, and the second choice is not a viable option. So, to address this problem, Sun developed a browser plug-in for Netscape and Internet Explorer that lets you use the Java 2 platform in a variety of browser versions. 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, in some browsers, the normal `APPLET` tag will not work with the plug-in, since these older 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. This element does not add any Java capabilities to the client. How could it? JSP runs entirely on the server; the client knows nothing about JSP. The `jsp:plugin` element merely simplifies the generation of the `OBJECT` or `EMBED` tags.

Core Note

The `jsp:plugin` element does not add any Java capability to the browser. It merely simplifies the creation of the cumbersome `OBJECT` and `EMBED` tags needed by the Java 2 Plug-in.

Servers are permitted some leeway in exactly how they implement `jsp:plugin` but most simply include both `OBJECT` and `EMBED`. To see exactly how your server translates `jsp:plugin`, insert into a page a simple `jsp:plugin` element with `type`, `code`, `width`, and `height` attributes as in the following example. Then, access the page from your browser and view the HTML source. For example, Listing 13.7 shows the HTML code generated by Tomcat for the following `jsp:plugin` element.

```
<jsp:plugin type="applet"
            code="SomeApplet.class"
            width="300" height="200">
</jsp:plugin>
```

Listing 13.7 Code Generated by Tomcat for `jsp:plugin`

```
<object classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
        width="300" height="200"
        codebase="http://java.sun.com/products/plugin/1.2.2/jinst
all-1_2_2-win.cab#Version=1,2,2,0">
  <param name="java_code" value="SomeApplet.class">
  <param name="type" value="application/x-java-applet;">
<COMMENT>
<embed type="application/x-java-applet;" width="300" height="200"
       pluginspage="http://java.sun.com/products/plugin/"
       java_code="SomeApplet.class"
>
<noembed>
</COMMENT>
</noembed></embed>
</object>
```

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 parallel the 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 components 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 in which 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 interapplet 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 1.2.
- **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 NAME="PARAM1" VALUE="VALUE1">
  <PARAM NAME="PARAM2" VALUE="VALUE2">
</APPLET>
```

with

```
<jsp:plugin type="applet"
            code="MyApplet.class"
            width="475" height="350">
  <jsp:params>
    <jsp:param name="PARAM1" value="VALUE1" />
    <jsp:param name="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 13.8 shows a JSP page that uses the `jsp:plugin` element to generate an entry for the Java 2 Plug-in. Listing 13.9 shows the code for the applet itself (which uses Swing, Java 2D, and the auxiliary classes of Listings 13.10 through 13.12). Figure 13–3 shows the result.

Listing 13.8 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>
<CENTER>
<TABLE BORDER=5>
  <TR><TH CLASS="TITLE">
      Using jsp:plugin</TABLE>
```

Listing 13.8 PluginApplet.jsp *(continued)*

```
<P>
<jsp:plugin type="applet"
            code="PluginApplet.class"
            width="370" height="420">
</jsp:plugin>
</CENTER></BODY></HTML>
```

Listing 13.9 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 13.10 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();
```

Listing 13.10 TextPanel.java *(continued)*

```
    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 13.11 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 };
```

Listing 13.11 DrawingPanel.java *(continued)*

```
  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));
  }
}
```

Listing 13.12 WindowUtilities.java

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

/** A few utilities that simplify using windows in Swing. */

public class WindowUtilities {

  /** Tell system to use native look and feel, as in previous
   *  releases. Metal (Java) LAF is the default otherwise.
   */

  public static void setNativeLookAndFeel() {
    try {
      UIManager.setLookAndFeel
        (UIManager.getSystemLookAndFeelClassName());
    } catch(Exception e) {
      System.out.println("Error setting native LAF: " + e);
    }
  }

  ... // See www.coreservlets.com for remaining code.
}
```

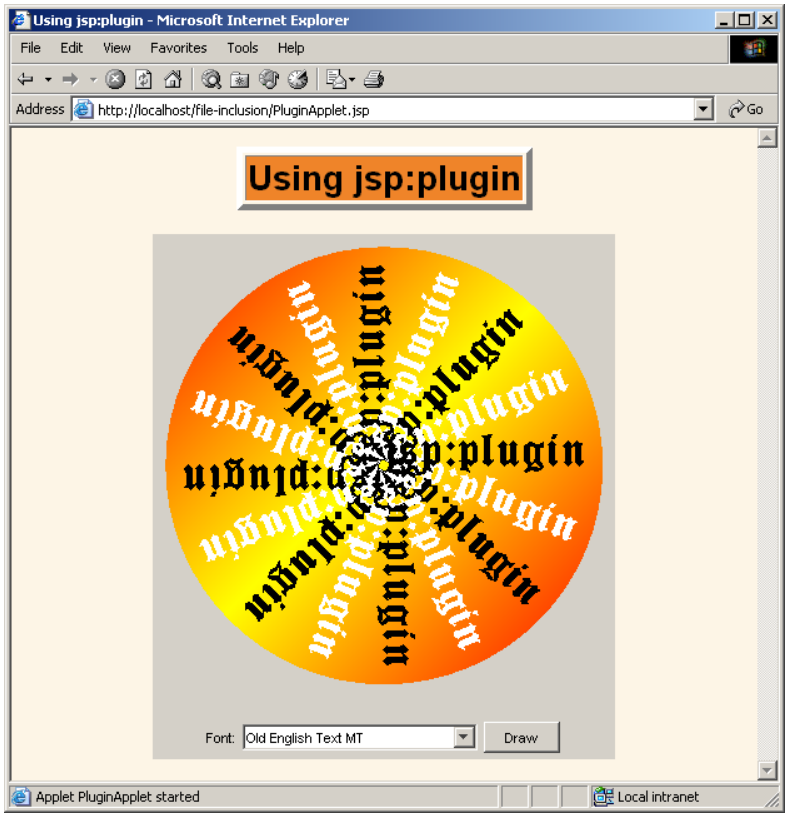

Figure 13–3 Result of PluginApplet.jsp in Internet Explorer with the JDK 1.4 plug-in.