Chapter 3: A Fast Introduction to Basic JSP Programming

Topics in This Chapter

- Understanding the benefits of JSP
- Invoking Java code with JSP expressions, scriptlets, and declarations
- Structuring 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

Chapter 3

JavaServer Pages (JSP) technology enables you to mix regular, static HTML with dynamically generated content. 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 3.2 gives some details on these advantages, but they basically boil down to two facts: JSP is widely supported and thus doesn't lock you into a particular operating system or Web server; and JSP gives you full access to the Java programming language and Java servlet technology for the dynamic part, rather than requiring you to use an unfamiliar and weaker special-purpose language.

3.1 JSP Overview

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 place 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 Chapter 1 (Server Setup and Configuration).

Although what you write often looks more like a regular HTML file than like 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 experience 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. Alternatively, if you deliver your applications on the same server you develop them on, you can deliver the precompiled servlet class files in their server-specific directories (see example locations on page 128). You can even omit the JSP source code in such a case.

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 the "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 book covers versions 1.1 and 1.2 of the JavaServer Pages specification. Basic JSP constructs are backward-compatible with JSP 1.0, but custom tags, Web applications, and use of the deployment descriptor (*web.xml*) are specific to JSP 1.1 and later. Furthermore, JSP 1.1 did not mandate the use of Java 2; JSP 1.2 does. Consequently, if you use constructs specific to Java 2 (e.g., collections), your JSP 1.2 code will not run on JSP 1.1-compatible servers that are running on top of JDK 1.1. Finally, note that all JSP 1.x versions are completely incompatible with the long-obsolete JSP 0.92. If JSP 0.92 was your only exposure to JSP, you have a pleasant surprise in store; JSP technology has been totally revamped (and improved) since then.

3.2 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 JSP 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 and IIS. Even if ASP.NET (not yet available as of fall 2001) succeeds in addressing the problem of developing server-side code with VBScript, you cannot expect to use ASP on multiple servers and operating systems.

You could make the same argument when comparing JSP to the current version of ColdFusion; with JSP you can use Java for the "real code" and are not tied to a particular server product. Note, however, that the next release of ColdFusion (version 5.0) will be within the context of a J2EE server, allowing developers to easily mix ColdFusion and servlet/JSP code.

Versus PHP

PHP (a recursive acronym for "PHP: Hypertext Preprocessor") 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 by using familiar tools and either leave places for your servlet programmers to insert the dynamic content or invoke the dynamic content indirectly by means of XML tags.

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! You can't write that code if you don't understand servlet programming.
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 3.8 (Integrating Servlets and JSP: The MVC Architecture) for details.

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*). JSP pages still include `SCRIPT` tags for JavaScript, just as normal HTML pages do. In fact, JSP can even be used to dynamically generate the JavaScript that will be sent to the client.

It is also possible to use JavaScript on the server, most notably on Netscape, IIS, and BroadVision servers. However, Java is more powerful, flexible, reliable, and portable.

3.3 Invoking Code with JSP Scripting Elements

There are a number of different ways to generate dynamic content from JSP, as illustrated in Figure 3–1. Each of these approaches has a legitimate place; the size and complexity of the project is the most important factor in deciding which approach is appropriate. However, be aware that people err on the side of placing too much code directly in the page much more often than they err on the opposite end of the spectrum. Although putting small amounts of Java code directly in JSP pages works fine for simple applications, using long and complicated blocks of Java code in JSP pages yields a result that is hard to maintain, hard to debug, and hard to divide among different members of the development team. Nevertheless, many pages are quite simple, and the first two approaches of Figure 3–1 (placing explicit Java code directly in the page) work quite well. This section discusses those approaches.

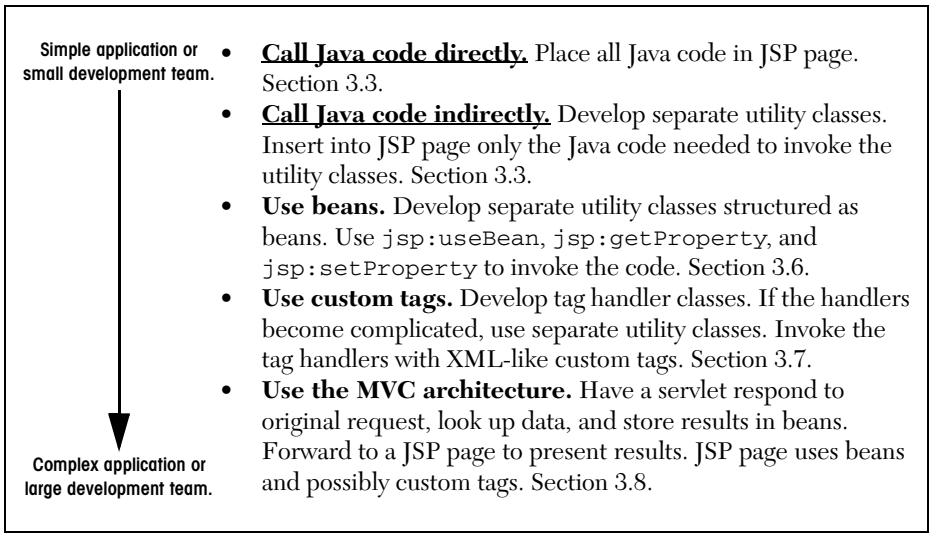

Figure 3–1 Strategies for invoking dynamic code from JSP.

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, I used Macromedia'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() %>
```

Predefined Variables

To simplify these expressions, you can use a number of predefined variables (or "implicit objects"). There is nothing magic about these variables; the system simply tells you what names it will use for the local variables in `_jspService`. 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 3.4)
- **`out`**, the `Writer` (a buffered version called `JspWriter`) used to send output to the client

Here is an example:

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

JSP/Servlet Correspondence

Now, I just stated that a JSP expression is evaluated and inserted into the page output. Although this is true, it is sometimes helpful to understand in a bit more detail what is going on.

It is actually pretty simple: JSP expressions basically become `print` (or `write`) statements in the servlet that results from the JSP page. Whereas regular HTML becomes `print` statements with double quotes around the text, JSP expressions become `print` statements with no double quotes. Instead of being placed in the `doGet` method, these print statements are placed in a new method called `_jspService` that is called by `service` for both `GET` and `POST` requests. For instance, Listing 3.1 shows a small JSP sample that includes some static HTML and a JSP expression. Listing 3.2 shows a `_jspService` method that might result. Of course, different vendors will produce code in slightly different ways, and optimizations such as reading the HTML from a static byte array are quite common.

Also, I oversimplified the definition of the `out` variable; `out` in a JSP page is a `JspWriter`, so you have to modify the slightly simpler `PrintWriter` that directly results from a call to `getWriter`. So, don't expect the code your server generates to look *exactly* like this.

Listing 3.1 Sample JSP Expression: Random Number

```
<H1>A Random Number</H1>
<%= Math.random() %>
```

Listing 3.2 Representative Resulting Servlet Code: Random Number

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html");
  HttpSession session = request.getSession(true);
  JspWriter out = response.getWriter(); // Oversimplified a bit
  out.println("<H1>A Random Number</H1>");
  out.println(Math.random());
  ...
}
```

If you want to see the exact code that your server generates, you'll have to dig around a bit to find it. In fact, some servers delete the source code files once they are successfully compiled. But here is a summary of the locations used by three common, free development servers.

Tomcat 4.0 Autogenerated Servlet Source Code
install_dir/work/localhost/_
(The final directory is an underscore.)

JRun 3.1 Autogenerated Servlet Source Code
install_dir/servers/default/default-app/WEB-INF/jsp
(More generally, in the *WEB-INF/jsp* directory of the Web application to which the JSP page belongs.)

ServletExec 4.0 Autogenerated Servlet Source Code
install_dir/Servlets/pagecompile
(More generally, in *install_dir/ServletExec Data/virtual-server-name/web-app-name/pagecompile*.)

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 (`<jsp:expression> ... </jsp:expression>`) and the standard JSP version that follows ASP syntax

(`<%= ... %>`) 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 for HTML pages, images, and style sheets. Here are a few examples of default installation locations (i.e., locations that apply when you aren't using custom Web applications) and associated URLs. Where I list *SomeDirectory*, you can use any directory name you like. (But you are never allowed to use *WEB-INF* or *META-INF* as directory names. For the default Web application, you also have to avoid a directory name that matches the URL prefix of any other Web application. For information on defining your own Web application, see Chapter 4, "Using and Deploying Web Applications.")

- **Tomcat Directory**
  install_dir/webapps/ROOT
  (or *install_dir/webapps/ROOT/SomeDirectory*)
- **JRun Directory**
  install_dir/servers/default/default-app
  (or *install_dir/servers/default/default-app/SomeDirectory*)
- **ServletExec Directory**
  install_dir/public_html
  (or *install_dir/public_html/SomeDirectory*)
- **Corresponding URLs**
  http://host/Hello.html
  (or *http://host/SomeDirectory/Hello.html*)
  http://host/Hello.jsp
  (or *http://host/SomeDirectory/Hello.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*; see Sections 1.7 and 1.9).

Example: JSP Expressions

Listing 3.3 gives an example JSP page called *Expressions.jsp*. I placed the file in a subdirectory called *jsp-intro*, copied the entire directory from my development directory to the deployment location just discussed, and used a base URL of *http://host/jsp-intro/Expressions.jsp*. Figures 3–2 and 3–3 show some typical results.

Notice that I included `META` tags and a style sheet link in the `HEAD` section of the JSP 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 tools.

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 as long as the JSP page is accessed directly by the client, rather than indirectly by means of a `RequestDispatcher`. Even then, there are some techniques you can use to simplify the use of relative URLs. For details, see Section 4.5 (Handling Relative URLs in Web Applications).

Thus, in most cases 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 found at *http://www.moreservlets.com*.

Listing 3.3 *Expressions.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>JSP Expressions</TITLE>
<META NAME="keywords"
      CONTENT="JSP,expressions,JavaServer Pages,servlets">
<META NAME="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>Server: <%= application.getServerInfo() %>
  <LI>Session ID: <%= session.getId() %>
  <LI>The <CODE>testParam</CODE> form parameter:
      <%= request.getParameter("testParam") %>
</UL>
</BODY>
</HTML>
```

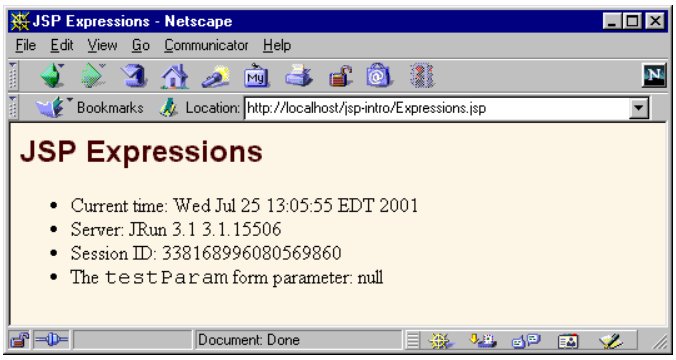

Figure 3–2 Result of *Expressions.jsp* using JRun 3.1 and omitting the `testParam` request parameter.

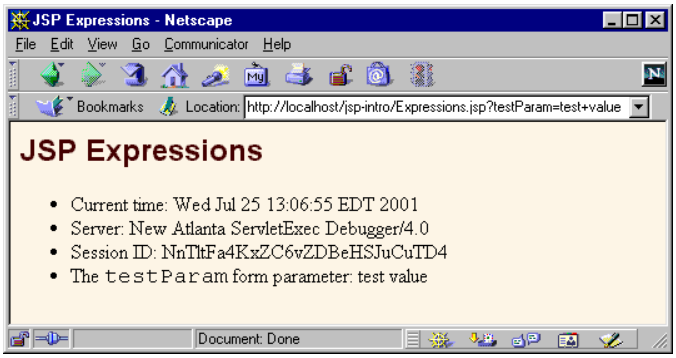

Figure 3–3 Result of *Expressions.jsp* using ServletExec 4.0 and specifying `test+value` as the value of the `testParam` request parameter.

Scriptlets

If you want to do something more complex than output 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 do expressions (`request`, `response`, `session`, `out`, etc.). So, for example, if you want to explicitly send output to the resultant page, you could 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 3.4 for a discussion of the `buffer` and `autoflush` attributes of the `page` directive.

JSP/Servlet Correspondence

It is easy to understand how JSP scriptlets correspond to servlet code: the scriptlet code is just directly inserted into the `_jspService` method: no strings, no `print` statements, no changes whatsoever. For instance, Listing 3.4 shows a small JSP sample that includes some static HTML, a JSP expression, and a JSP scriptlet. Listing 3.5 shows a `_jspService` method that might result. Again, different vendors will produce this code in slightly different ways, and I oversimplified the `out` variable (which is a `JspWriter`, not the slightly simpler `PrintWriter` that results from a call to `getWriter`). So, don't expect the code your server generates to look *exactly* like this.

Listing 3.4 Sample JSP Expression/Scriptlet

```
<H2>foo</H2>
<%= bar() %>
<% baz(); %>
```

Listing 3.5 Representative Resulting Servlet Code: Expression/Scriptlet

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html");
  HttpSession session = request.getSession(true);
  JspWriter out = response.getWriter();
  out.println("<H2>foo</H2>");
  out.println(bar());
  baz();
  ...
}
```

Scriptlet Example

As an example of code that is too complex for a JSP expression alone, Listing 3.6 presents a JSP page that uses the `bgColor` request parameter to set the background color of the page. *JSP-Styles.css* is omitted so that the style sheet does not override the background color. Figures 3–4, 3–5, and 3–6 show the default result, the result for a background of `C0C0C0`, and the result for `papayawhip` (one of the oddball X11 color names still supported for historical reasons), respectively.

Listing 3.6 *BGColor.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Color Testing</TITLE>
</HEAD>
<%
String bgColor = request.getParameter("bgColor");
if (bgColor == null) { bgColor = "WHITE"; }
%>
<BODY BGCOLOR="<%= bgColor %>">
<H2 ALIGN="CENTER">Testing a Background of "<%= bgColor %>"</H2>
</BODY>
</HTML>
```

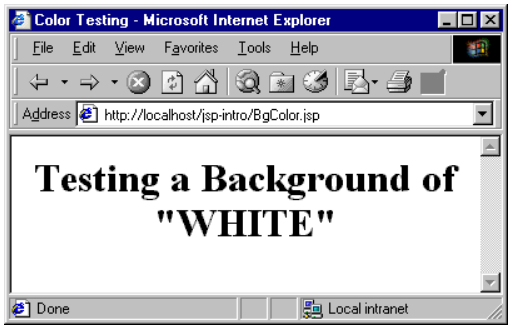

Figure 3–4 Default result of *BGColor.jsp*.

Figure 3–5 Result of *BGColor.jsp* when accessed with a `bgColor` parameter having the RGB value `C0C0C0`.

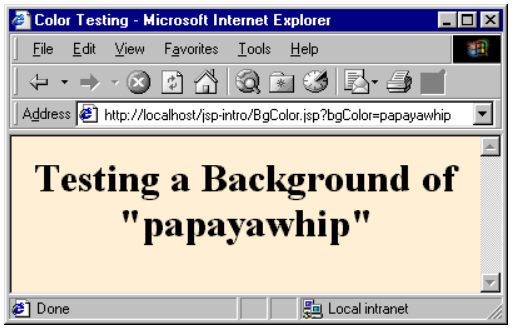

Figure 3–6 Result of *BGColor.jsp* when accessed with a `bgColor` parameter having the X11 color name `papayawhip`.

Using Scriptlets to Make Parts of the JSP Page Conditional

Another use of scriptlets is to conditionally output HTML or other content that is *not* within any JSP tags. 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!
<% } %>
```

You probably find that a bit confusing. I certainly did the first few times. Neither the "have a nice day" nor the "have a lousy day" lines are contained within a JSP tag, so it seems odd that only one of the two becomes part of the output for any given request. But, when you think about how this example will be converted to servlet code by the JSP engine, you get the following easily understandable result.

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

XML and Other 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>
```

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 (`<jsp:scriptlet> ... </jsp:scriptlet>`) and the ASP-like version (`<% ... %>`) in the same page. Remember that XML elements are case sensitive; be sure to use `jsp:scriptlet` in lower case.

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 life-cycle 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 is 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 Approach

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

Aside from overriding standard methods like `jspInit` and `jspDestroy`, the utility of JSP declarations for defining methods is somewhat questionable. Moving the methods to separate classes (possibly as static methods) makes them easier to write (since you are using a Java environment, not an HTML-like one), easier to test (no need to run a server), easier to debug (no tricks are needed to see the standard output), and easier to reuse (many different JSP pages can use the same utility class). However, using JSP declarations to define fields, as we will see shortly, gives you something not easily reproducible with separate utility classes: a place to store data that is persistent between requests.

Core Approach

Consider separate helper classes instead of methods defined by means of JSP declarations.

JSP/Servlet Correspondence

JSP declarations result in code that is placed inside the servlet class definition but outside the `_jspService` method. Since fields and methods can be declared in any order, it does not matter if the code from declarations goes at the top or bottom of the servlet. For instance, Listing 3.7 shows a small JSP snippet that includes some static HTML, a JSP declaration, and a JSP expression. Listing 3.8 shows a servlet that might result. Note that the specific name of the resultant servlet is not defined by the JSP specification, and in fact different servers have different conventions. Besides, as already stated, different vendors will produce this code in slightly different ways, and I oversimplified the `out` variable (which is a `JspWriter`, not the slightly simpler `PrintWriter` that results from a call to `getWriter`). So, don't expect the code your server generates to look *exactly* like this.

Listing 3.7 Sample JSP Declaration

```
<H1>Some Heading</H1>
<%!
  private String randomHeading() {
    return("<H2>" + Math.random() + "</H2>");
  }
%>
<%= randomHeading() %>
```

Listing 3.8 Representative Resulting Servlet Code: Declaration

```
public class xxxx implements HttpJspPage {
 private String randomHeading() {
   return("<H2>" + Math.random() + "</H2>");
 }

 public void _jspService(HttpServletRequest request,
                         HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html");
  HttpSession session = request.getSession(true);
  JspWriter out = response.getWriter();
  out.println("<H1>Some Heading</H1>");
  out.println(randomHeading());
  ...
 }

  ...
}
```

Declaration Example

In this example, a JSP fragment prints the number of times the current page has been requested since the server was booted (or the servlet class was changed and reloaded). A hit counter in one line of code!

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

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` (see Section 2.3, "The Servlet Life Cycle"). Thus, instance variables (fields) of a normal servlet are shared by multiple requests, and `accessCount` does not have to be declared `static`. Now, advanced readers might wonder if the snippet just shown is thread safe; does the code guarantee that each visitor gets a unique count? The answer is no; in unusual situations multiple users could see the same value. For access counts, as long as the count is correct in the long run, it does not matter if two different users occasionally see the same count. But, for values such as session identifiers, it is critical to have unique values. For an example similar to the previous snippet but that guarantees thread safety, see the discussion of the `isThreadSafe` attribute of the `page` directive in Section 3.4.

Listing 3.9 shows the full JSP page; Figure 3–7 shows a representative result. Now, before you rush out and use this approach to track access to all your pages, a couple of cautions are in order. First of all, you couldn't use this for a real hit counter, since the count starts over whenever you restart the server. So, a real hit counter would need to use `jspInit` and `jspDestroy` to read the previous count at startup and store the old count when the server is shut down. Even then, it would be possible for the server to crash unexpectedly (e.g., when a rolling blackout strikes Silicon Valley). So, you would have to periodically write the hit count to disk. Finally, some advanced servers support distributed applications whereby a cluster of servers appears to the client as a single server. If your servlets or JSP pages might need to support distribution in this way, plan ahead and avoid the use of fields for persistent data. Use a database instead.

Listing 3.9 *AccessCounts.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>JSP Declarations</TITLE>
<META NAME="keywords"
      CONTENT="JSP,declarations,JavaServer,Pages,servlets">
<META NAME="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>
```

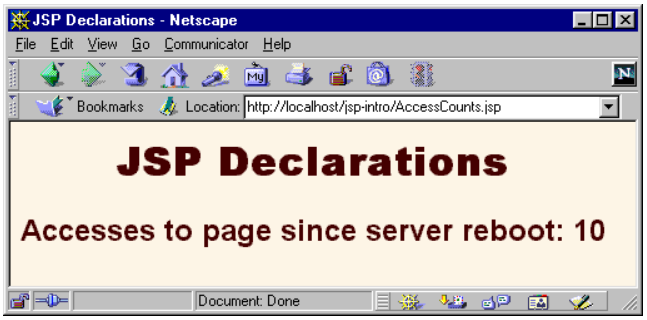

Figure 3–7 Visiting *AccessCounts.jsp* after it has been requested nine previous times by the same or different clients.

XML and Special Declaration Syntax

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

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

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 (`<jsp:declaration> ... </jsp:declaration>`) and the standard ASP-like version (`<%! ... %>`) in the same page. Remember that XML elements are case sensitive; be sure to use `jsp:declaration` in lower case.

Predefined Variables

To simplify code in JSP expressions and scriptlets, you are supplied with eight automatically defined local variables in `_jspService`, 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. Since the output stream (see `out`) is normally buffered, it is usually legal to set HTTP status codes and response headers in the body of 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. If you turn buffering off, however (see the `buffer` attribute in Section 3.4), you must set status codes and headers before supplying any output.
- **`out`**
  This variable is the `Writer` used to send output to the client. However, to make it easy to set response headers at various places in the JSP page, `out` is not the standard `PrintWriter` but rather a buffered version of `Writer` 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 in JSP, so this variable is bound even if there is no incoming session reference. The one exception is when you use the `session` attribute of the `page` directive (Section 3.4) to disable session tracking. 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 Web application, whereas instance variables are available only to the same servlet that stored the data.
- **`config`**
  This variable is the `ServletConfig` object for this page. The `jspInit` method would use it to read initialization parameters.
- **`pageContext`**
  JSP introduced a class called `PageContext` to give a single point of access to many of the page attributes. The `pageContext` variable stores the value of the `PageContext` object associated with the current page. If a method or constructor needs access to multiple page-related objects, passing `pageContext` is easier than passing many separate references to `out`, `request`, `response`, and so forth.
- **`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.

3.4 Structuring Autogenerated Servlets: 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 backslash, 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 3.5. JSP 1.1 introduced a third directive, `taglib`, which is used to define custom markup tags; it is discussed in Section 3.7.

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`, `language`, and `pageEncoding`. 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. As illustrated in Figure 3–8, using separate utility classes makes your dynamic code easier to maintain, debug, and reuse, and your utility classes are sure to use packages.

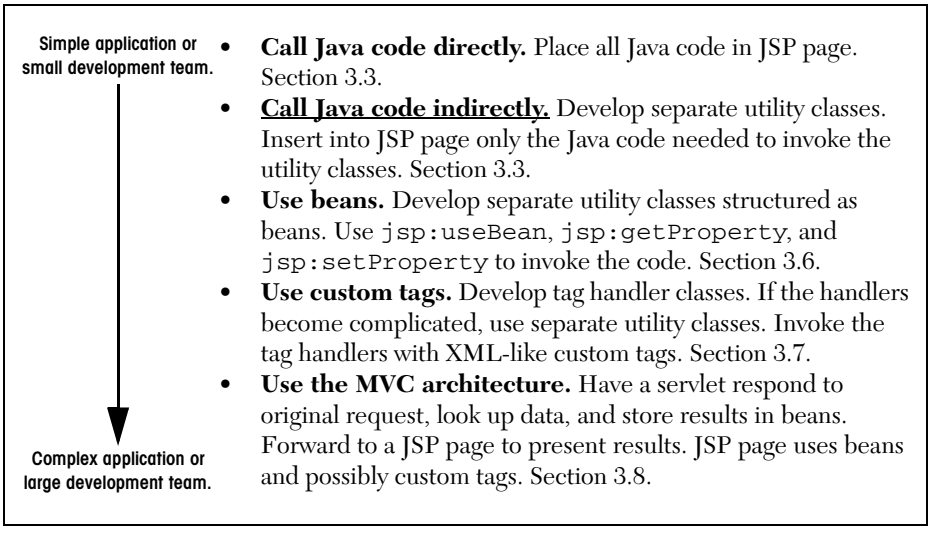

Figure 3–8 Strategies for invoking dynamic code from JSP.

In fact, *all* of your utility classes should be placed in packages. For one thing, packages are a good strategy on any large project because they help protect against

name conflicts. With JSP, however, packages are absolutely required. That's because, in the absence of packages, classes you reference are assumed to be in the same package as the current class. For example, suppose that a JSP page contains the following scriptlet.

```
<% Test t = new Test(); %>
```

Now, if `Test` is in an imported package, there is no ambiguity. But, if `Test` is not in a package, or the package to which `Test` belongs is not explicitly imported, then the system will assume that `Test` is in the same package as the autogenerated servlet. The problem is that the autogenerated servlet's package is not known! It is quite common for servers to create servlets whose package is determined by the directory in which the JSP page is placed. Other servers use different approaches. So, you simply cannot rely on packageless classes to work properly. The same argument applies to beans (Section 3.6), since beans are just classes that follow some simple naming and structure conventions.

Core Approach

Always put your utility classes and beans in packages.

By default, 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, although the JSP pages go in the normal HTML directories of the server, the classes you write that are used by JSP pages must be placed in the special servlet directories (e.g., *.../WEB-INF/classes*; see Sections 1.7 and 1.9).

For example, Listing 3.10 presents a page that uses three classes not in the standard JSP import list: `java.util.Date`, `moreservlets.ServletUtilities` (see Listing 2.17), and `moreservlets.LongLivedCookie` (see Listing 2.18). To simplify references to these classes, the JSP page uses

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

Figures 3–9 and 3–10 show some typical results.

Listing 3.10 *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.*,moreservlets.*" %>
<%-- 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 LongLivedCookie("userID", newID);
  response.addCookie(cookie);
}
%>
<%-- JSP Expressions --%>
This page was accessed on <%= new Date() %> with a userID
cookie of <%= oldID %>.
</BODY>
</HTML>
```

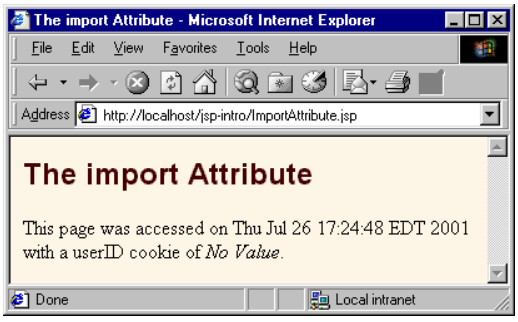

Figure 3–9 *ImportAttribute.jsp* when first accessed.

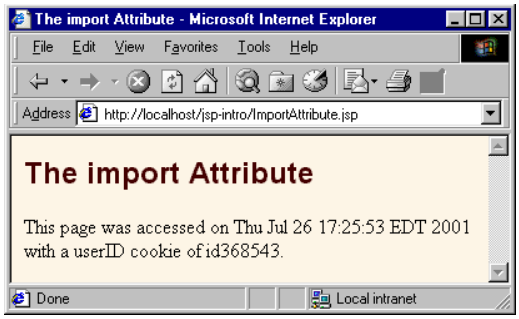

Figure 3–10 *ImportAttribute.jsp* when accessed in a subsequent request.

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 2.1 (Common MIME Types) in Section 2.8 (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"); %>
```

The main difference between the two forms is that `response.setContentType` can be invoked conditionally whereas the `page` directive cannot be. Setting the content type conditionally is occasionally useful when the same content can be displayed in different forms—for an example, see the Section "Generating Excel Spreadsheets" starting on page 254 of *Core Servlets and JavaServer Pages* (available in PDF at *http://www.moreservlets.com*).

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`). Thus, JSP pages that output HTML in a Latin character set need not use `contentType` at all. But, pages in JSP 1.1 and earlier that output other character sets need to use `contentType` even when they generate HTML. For example, Japanese JSP pages might use the following.

```
<%@ page contentType="text/html; charset=Shift_JIS" %>
```

In JSP 1.2, however, the `pageEncoding` attribute (see details later in this section) can be used to directly specify the character set.

Listing 3.11 shows a JSP page that generates tab-separated Excel output. Note that the `page` directive and comment are at the bottom so that the carriage returns at the ends of the lines don't show up in the Excel document (remember: JSP does not ignore white space—JSP usually generates HTML where most white space is ignored by the browser). Figure 3–11 shows the result in Internet Explorer on a system that has Microsoft Office installed.

Listing 3.11 *Excel.jsp*

```
First   Last    Email Address
Marty   Hall    hall@moreservlets.com
Larry   Brown   brown@corewebprogramming.com
Steve   Balmer  balmer@sun.com
Scott   McNealy mcnealy@microsoft.com
<%@ page contentType="application/vnd.ms-excel" %>
<%-- There are tabs, not spaces, between columns. --%>
```

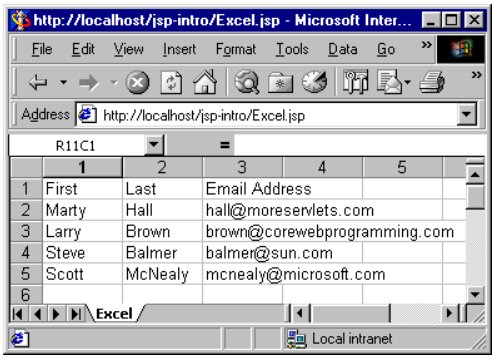

Figure 3–11 Excel document (*Excel.jsp*) in Internet Explorer.

The isThreadSafe Attribute

The `isThreadSafe` attribute controls whether the servlet that results from the JSP page will implement the `SingleThreadModel` interface (Section 2.3). 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 static 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 static 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 that concurrently access 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 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. The possibility of a pool of instances explains the need for the `static` qualifier in the `idNum` field declaration in the previous examples.

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.

Explicitly synchronizing your code as in the previous snippet is preferred whenever possible. In particular, explicit synchronization yields higher performance pages that are accessed frequently. However, using `isThreadSafe="false"` is useful when the problematic code is hard to find (perhaps it is in a class for which you have no source code) and for quick testing to see if a problem stems from race conditions at all.

Core Note

With frequently accessed pages, you get better performance by using explicit synchronization than by using the `isThreadSafe` attribute.

The session Attribute

The `session` attribute controls whether 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`. 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, the page is completed, or the output is explicitly flushed (e.g., with `response.flushBuffer`). The default buffer size is server specific, but must be at least 8 kilobytes. Be cautious about turning off buffering; doing so requires JSP elements 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" %>
```

This attribute is normally reserved for developers or vendors that implement fundamental changes to the way that pages operate (e.g., to add in personalization features). Ordinary mortals should steer clear of this attribute.

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. The designated error page must use `isErrorPage="true"` (see next entry) to indicate that it permits use as an error page. The `errorPage` attribute 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. For an example, see Section 11.10 of *Core Servlets and JavaServer Pages* (available in PDF at *http://www.moreservlets.com*).

Note that the `errorPage` attribute is used to designate *page-specific* error pages. To designate error pages that apply to an entire Web application or to various categories of errors within an application, use the `error-page` element in *web.xml*. For details, see Section 5.8 (Designating Pages to Handle Errors).

The isErrorPage Attribute

The `isErrorPage` attribute indicates whether 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.

The pageEncoding Attribute

The `pageEncoding` attribute, available only in JSP 1.2, defines the character encoding for the page. The default value is ISO-8859-1 unless the `contentType` attribute of the `page` directive is specified, in which case the `charset` entry of `contentType` is the default.

XML Syntax for Directives

All JSP 1.2 servers (containers) and some JSP 1.1 servers permit you to use an alternative XML-compatible syntax for directives as long as you don't mix the XML version and the normal version in the same page. 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.*" />
```

3.5 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 of the following subsections.

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 *output* of the secondary page, not the secondary page's actual code 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.

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" %>
```

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 may 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. So, you may have to update the modification dates of each JSP page that uses the included code. 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 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="..." />
```

Remember that only servlet and JSP containers (servers) that support JSP 1.2 are required to support the XML version.

Including Pages at Request Time: The jsp:include Action

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 main 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. Also, inclusion at page translation time is marginally faster. These are relatively minor considerations, 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 pages cannot contain JSP, the pages 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. The server runs the included page in the usual manner and places the output into the main page. This is precisely the behavior of the `include` method of the `RequestDispatcher` class (Section 3.8), which is what servlets use if they want to do this type of file inclusion.

The `jsp:include` element has two attributes, as shown in the sample below: `page` and `flush`. The `page` attribute is required and designates a relative URL referencing the file to be included. The `flush` attribute specifies whether the output stream of the main page should flushed before the inclusion of the page. In JSP 1.2, `flush` is an optional attribute and the default value is `false`. In JSP 1.1, `flush` is a required attribute and the only legal value is `true`.

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

The included file automatically is given the same request parameters as the originally requested page. If you want to augment those parameters, you can use the `jsp:param` element (which has `name` and `value` attributes). 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 main page receives `"RED"` for calls to `request.getParameter("fgColor")` and `null` for calls to `request.getParameter("bgColor")` (regardless of whether the `bgColor` attribute is accessed before or after the inclusion of the *StandardHeading.jsp* page). The *StandardHeading.jsp* page would receive `"RED"` for calls to `request.getParameter("fgColor")` and `"YELLOW"` for calls to `request.getParameter("bgColor")`. If the main page receives a request parameter that is also specified with the `jsp:param` element, the value from `jsp:param` takes precedence in the included page.

As an example of a typical use of `jsp:include`, consider the simple news summary page shown in Listing 3.12. Page developers can change the news items in the files `Item1.html` through `Item3.html` (Listings 3.13 through 3.15) without having to update the main news page. Figure 3–12 shows the result.

Listing 3.12 *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 3.13 *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 3.14 *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 3.15 *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>
```

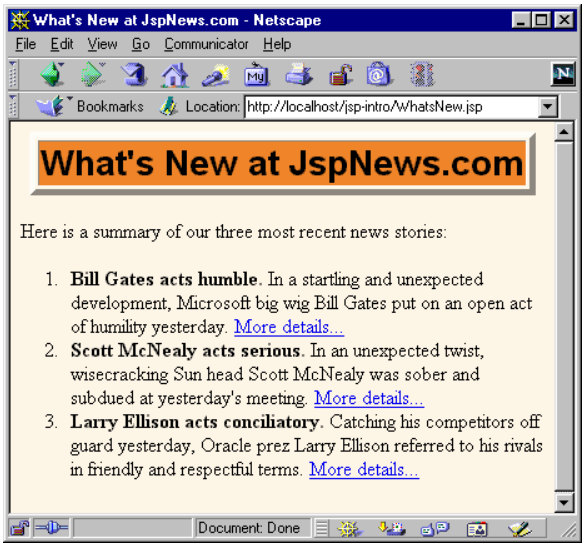

Figure 3–12 Including files at request time makes it easier to update the individual files.

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 1.2–1.4). This lack of support imposes several restrictions on 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.

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. 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.

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. You don't need to create an applet to perform this experiment.

Note that JRun 3.0 SP2 does not support `jsp:plugin`; JRun 3.1 supports it properly.

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 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 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 1.1.
- **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 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 3.16 shows a JSP page that uses the `jsp:plugin` element to generate an entry for the Java 2 Plug-In. Listings 3.17 through 3.20 show the code for the applet itself (which uses Swing and Java 2D), and Figure 3–13 shows the result.

Listing 3.16 *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 3.17 *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 3.18 *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 3.19 *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));
  }
}
```

Listing 3.20 *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.moreservlets.com for remaining code.
}
```

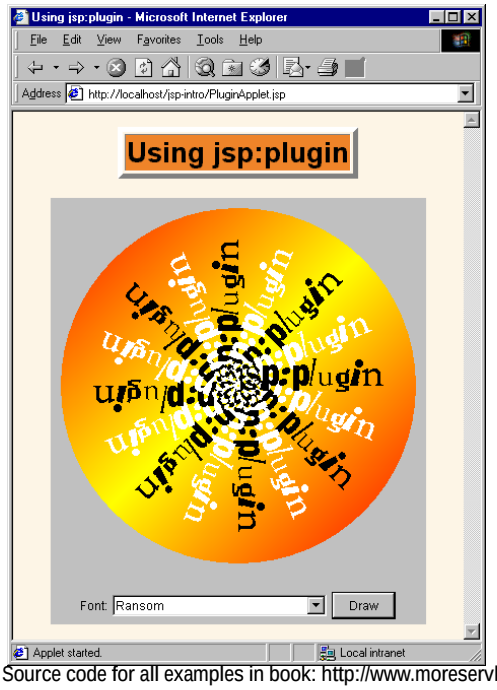

Figure 3–13 Result of *PluginApplet.jsp* in Internet Explorer when the Java 2 Plug-In is installed.

3.6 Using JavaBeans with JSP

This section discusses the third general strategy for inserting dynamic content in JSP pages (see Figure 3–14): by means of JavaBeans components.

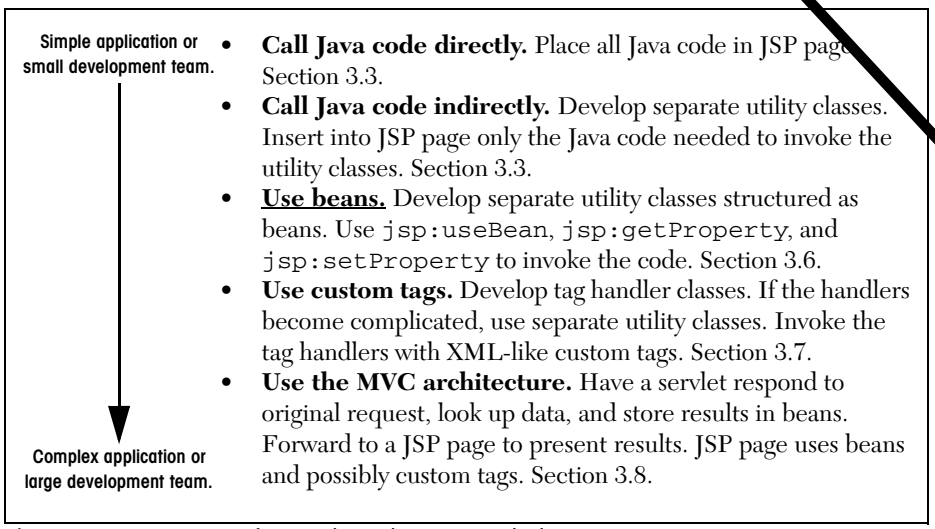

Figure 3–14 Strategies for invoking dynamic code from JSP.

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. Use of JavaBeans components in JSP provides three advantages over scriptlets and JSP expressions.

1. **No Java syntax.** By using beans, page authors can manipulate Java objects, using only XML-compatible syntax: no parentheses, semicolons, or curly braces. This promotes a stronger separation between the content and the presentation and is especially useful in large development teams that have separate Web and Java developers.
2. **Simpler object sharing.** The JSP bean constructs make it much easier to share objects among multiple pages or between requests than if the equivalent explicit Java code were used.
3. **Convenient correspondence between request parameters and object properties.** The JSP bean constructs greatly simplify the process of reading request parameters, converting from strings, and stuffing the results inside objects.

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/products/javabeans/docs/*. For the purposes of this chapter, however, 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. In fact, as we will see in Section 3.8 (Integrating Servlets and JSP: The MVC Architecture), it is quite common for a servlet to create a bean and a JSP page to merely look up data from the existing bean. In that case, the requirement that the bean have a zero-argument constructor is waived.
2. **A bean class should have no public instance variables (fields).** I 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 without users of your class changing their code: (a) impose constraints on variable values (e.g., have the `setSpeed` method of your `Car` class disallow negative speeds); (b) change your internal data structures (e.g., change from English units to metric units internally, but still have `getSpeedInMPH` and `getSpeedInKPH` methods); (c) 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 get*Xxx* and set*Xxx*.** 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* naming convention.

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 the following.

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

This statement usually means "instantiate an object of the class specified by `Class`, and bind it to a variable with the name specified by `id`."

So, for example, the JSP action

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

can normally be thought of as equivalent to the scriptlet

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

The bean class definition should be placed 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. Thus, on most servers, the proper location for bean classes is the *.../WEB-INF/classes* directory discussed in Sections 1.7 and 1.9. With some servers, however (e.g., ServletExec), you have to explicitly add bean classes to the server's `CLASSPATH` *if* you are using the default servlet directories (i.e., not using user-defined Web applications). With user-defined Web applications (see Chapter 4), *all* servers permit individual bean classes to be placed in the application's *WEB-INF/classes* directory and JAR files containing bean classes to be placed in the *WEB-INF/lib* directory.

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 associates the bean with more than just the current page. If beans can be shared, it is useful to obtain references to existing beans, rather than always building a new object. 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 declaration, 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.

A few character sequences also 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 name="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 it 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 I 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 name="book1"
                 property="title"
                 value="Core Servlets and JavaServer Pages" />
<% book1.setTitle("Core Servlets and JavaServer Pages"); %>
```

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 3.21 presents a simple class called `StringBean` that is in the `moreservlets` 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` property called `message`.

Listing 3.22 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="moreservlets.StringBean" />
```

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

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

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

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

Please note that I do not recommend that you really mix the explicit Java syntax and the XML syntax in the same page; this example is just meant to illustrate the equivalent results of the two forms.

Figure 3–15 shows the result.

Listing 3.21 *StringBean.java*

```
package moreservlets;

/** 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 3.22 *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>
```

Listing 3.22 *StringBean.jsp (continued)*

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

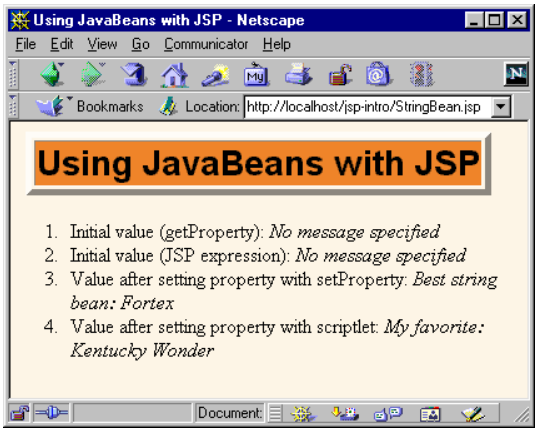

Figure 3–15 Result of *StringBean.jsp*.

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 3.23 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 3.24 shows a JSP file that builds an instance of the `SaleEntry` class by means of:

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

The results are shown in Figure 3–16.

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

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

Notice that I used a JSP expression for the `value` parameter. Most JSP attribute values have to be fixed strings, but the `value` attribute of `jsp:setProperty` is permitted to be a request time expression. 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. In any case, the point is that it is *possible* to use JSP expressions here, but doing so requires the use of explicit Java code. In some applications, avoiding such explicit code is the main reason for using beans in the first place. Besides, as the next examples will show, the situation becomes much more complicated when the bean property is not of type `String`. The next two subsections will discuss how to solve these problems.

Listing 3.23 *SaleEntry.java*

```
package moreservlets;

/** 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);
  }
```

Listing 3.23 *SaleEntry.java (continued)*

```
  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 3.24 *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="moreservlets.SaleEntry" />
<jsp:setProperty
    name="entry"
    property="itemID"
    value='<%= request.getParameter("itemID") %>' />
<%
int numItemsOrdered = 1;
try {
  numItemsOrdered =
    Integer.parseInt(request.getParameter("numItems"));
} catch(NumberFormatException nfe) {}
%>
<jsp:setProperty
    name="entry"
    property="numItems"
    value="<%= numItemsOrdered %>" />
<%
double discountCode = 1.0;
try {
  String discountString =
    request.getParameter("discountCode");
  // In JDK 1.1 use Double.valueOf(discountString).doubleValue()
  discountCode =
    Double.parseDouble(discountString);
} catch(NumberFormatException nfe) {}
%>
<jsp:setProperty
    name="entry"
    property="discountCode"
    value="<%= discountCode %>" />
<BR>
```

Listing 3.24 *SaleEntry1.jsp (continued)*

```
<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 name="entry" property="itemID" />
  <TD>$<jsp:getProperty name="entry" property="itemCost" />
  <TD><jsp:getProperty name="entry" property="numItems" />
  <TD>$<jsp:getProperty name="entry" property="totalCost" />
</TABLE>
</BODY>
</HTML>
```

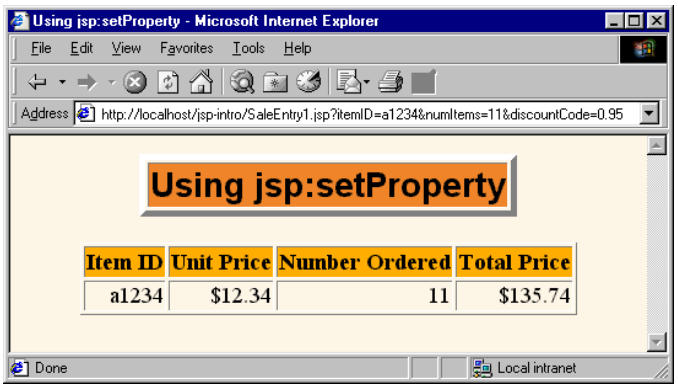

Figure 3–16 Result of *SaleEntry1.jsp*.

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
    name="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 the named request parameter is automatically used as the value of the bean property, and simple type conversions are performed automatically. If the specified 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
    name="entry"
    property="numItems"
    param="numItems" />
```

Listing 3.25 shows the relevant part of the JSP page reworked in this manner.

Listing 3.25 *SaleEntry2.jsp*

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

Converting Types Automatically

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.).

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 3.25 can be replaced by the following simple line. Listing 3.26 shows the relevant part of the page.

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

Although this approach is simple, three 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, 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. Third, since both bean property names and request parameters are case sensitive, the property name and request parameter name must match exactly.

Listing 3.26 *SaleEntry3.jsp*

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

Sharing Beans

Up to this point, I 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. 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 (Creating Beans Conditionally) 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 3.8 (Integrating Servlets and JSP: The MVC Architecture).
- **`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 3.8 (Integrating Servlets and JSP: The MVC Architecture).

Creating Beans Conditionally

To make bean sharing more convenient, you can conditionally evaluate bean-related elements in two situations.

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 3.27 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 to ensure that only the first page that is accessed sets the `firstPage` attribute.

```
<jsp:useBean id="counter"
             class="moreservlets.AccessCountBean"
             scope="application">
  <jsp:setProperty name="counter"
                   property="firstPage"
                   value="Current Page Name" />
</jsp:useBean>
```

Listing 3.28 shows the first of three pages that use this approach. The source code archive at *http://www.moreservlets.com* contains the other two nearly identical pages. Figure 3–17 shows a typical result.

Listing 3.27 *AccessCountBean.java*

```
package moreservlets;

/** 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;
  }
```

Listing 3.27 *AccessCountBean.java (continued)*

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

  public void setAccessCountIncrement(int increment) {
    accessCount = accessCount + increment;
  }
}
```

Listing 3.28 *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="moreservlets.AccessCountBean"
             scope="application">
  <jsp:setProperty name="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 name="counter" property="firstPage" />
was the first page accessed.
<P>
Collectively, the three pages have been accessed
<jsp:getProperty name="counter" property="accessCount" />
times.
<jsp:setProperty name="counter" property="accessCountIncrement"
                 value="1" />
</BODY>
</HTML>
```

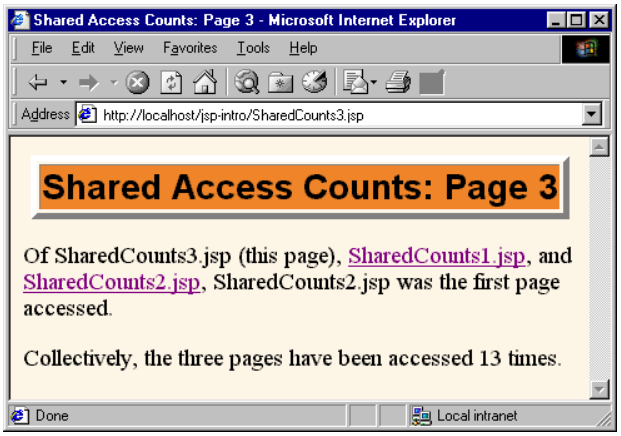

Figure 3–17 Result of a user visiting *SharedCounts3.jsp*. The first page visited by any user was *SharedCounts2.jsp*. *SharedCounts1.jsp*, *SharedCounts2.jsp*, and *SharedCounts3.jsp* were collectively visited a total of twelve times after the server was last started but before the visit shown in this figure.

3.7 Defining Custom JSP Tag Libraries

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. This section introduces the basic capabilities of custom tags. New features introduced in JSP 1.2 are covered in Chapter 11 (New Tag Library Features in JSP 1.2).

Custom tags accomplish some of the same goals as beans that are accessed with `jsp:useBean` (see Figure 3–18)—encapsulating complex behaviors into simple and accessible forms. There are several differences, however:

1. Custom tags can manipulate JSP content; beans cannot.
2. Complex operations can be reduced to a significantly simpler form with custom tags than with beans.
3. Custom tags require quite a bit more work to set up than do beans.

4. Custom tags usually define relatively self-contained behavior, whereas 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).
5. Custom tags are available only in JSP 1.1 and later, but beans can be used in all JSP 1.x versions.

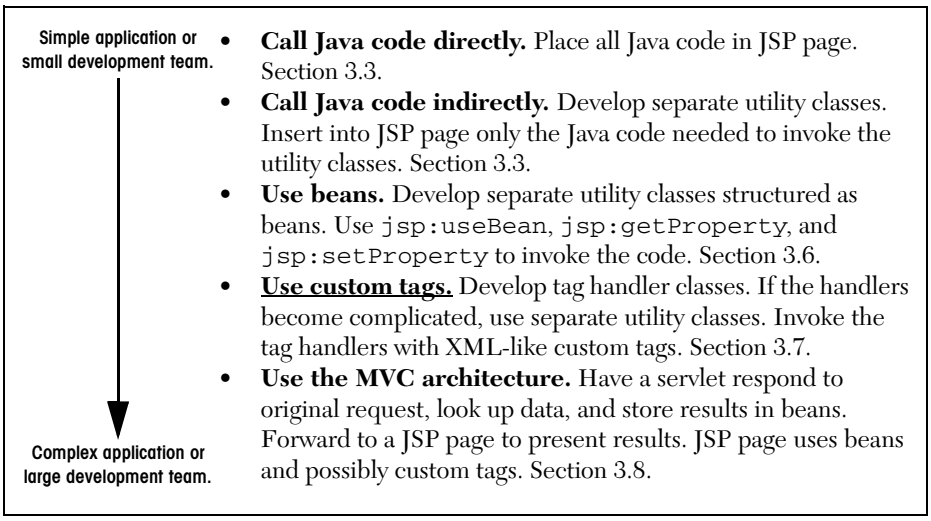

Figure 3–18 Strategies for invoking dynamic code from JSP.

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. Most people find that the first tag they write is the hardest—the difficulty being in knowing where each component should go, not in writing the components. So, I suggest that you start by just downloading the examples of this subsection and getting the example tag working. After that, you can move on to the following subsections and try some of your own 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. You usually accomplish this by extending the `TagSupport` or `BodyTagSupport` class.

Listing 3.29 is an example of a simple tag that just inserts "`Custom tag example (msajsp.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 the class is in the `moreservlets.tags` package and is called `ExampleTag`. Consequently, the class file needs to be placed in *tags* subdirectory of the *moreservlets* subdirectory of whatever directory the current Web application is using for Java class files (i.e., *.../WEB-INF/classes*—see Sections 1.7 and 1.9). With Tomcat, for example, the class file would be in *install_dir/webapps/ROOT/WEB-INF/classes/moreservlets/tags/ExampleTag.class*.

Listing 3.29 *ExampleTag.java*

```
package moreservlets.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 " +
                "(moreservlets.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 3.30. 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.moreservlets.com*.

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 `moreservlets.tags` package, the fully qualified class name of `moreservlets.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, the base location for classes in the default Web application is *install_dir/webapps/ROOT/WEB-INF/classes*, so *ExampleTag.class* would be in *install_dir/webapps/ROOT/WEB-INF/classes/moreservlets/tags*. Although it is always a good idea to put your servlet classes in packages, a surprising feature of Tomcat is that tag handlers are *required* to be in packages.

Listing 3.30 *msajsp-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>msajsp-tags</shortname>
  <info>
    A tag library from More Servlets and JavaServer Pages,
    http://www.moreservlets.com/.
  </info>
  <tag>
    <name>example</name>
    <tagclass>moreservlets.tags.ExampleTag</tagclass>
    <bodycontent>empty</bodycontent>
    <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 3.31 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 3.30. For now, we will use a simple relative URL corresponding to a TLD file that is in the same directory as the JSP page that uses it. When we get to Web applications, however (see Chapter 4), we will see that it makes more sense for larger applications to put the TLD files in a subdirectory inside the *WEB-INF* directory. This configuration makes it easier to reuse the same TLD file from JSP pages in multiple directories, and it prevents end users from retrieving the TLD file. Furthermore, as we will see in Section 5.13 (Locating Tag Library Descriptors), you can use the Web application deployment descriptor (i.e., *web.xml*) to change the meaning of strings supplied to the `uri` attribute of the `taglib` directive. When starting out, however, you will probably find it easiest to put the TLD file in the same directory as the JSP page that uses it and then use a simple filename as the value of the `uri` attribute.

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 3.30 is called `msajsp-taglib.tld` and resides in the same directory as the JSP file shown in Listing 3.31 (i.e., any of the standard locations for JSP files described in Section 3.3, *not* the directory where Java class files are placed). Thus, the `taglib` directive in the JSP file uses a simple relative URL giving just the filename, as shown below.

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

Furthermore, since the `prefix` attribute is `msajsp` (for *More Servlets and JavaServer Pages*), the rest of the JSP page uses `msajsp:example` to refer to the `example` tag defined in the descriptor file. Figure 3–19 shows the result.

Listing 3.31 *SimpleExample.jsp*

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

Figure 3–19 Result of *SimpleExample.jsp*.

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.*;
```

I recommend that you grab an example from *http://www.moreservlets.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* at the place where the element's start tag is found. To generate output, the method should obtain the `JspWriter` (the specialized `Writer` available in JSP pages through use of the predefined `out` variable) from the automatically defined `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 3.32 shows a tag implementation that uses this approach to generate a random 50-digit prime number through use of the `Primes` class (Listing 3.33), which is adapted from Section 7.3 (Persistent Servlet State and Auto-Reloading Pages) of *Core Servlets and JavaServer Pages*. Remember that the full text of *Core Servlets and JavaServer Pages* is available in PDF at *http://www.moreservlets.com*.

Listing 3.32 *SimplePrimeTag.java*

```
package moreservlets.tags;

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

/** Generates a prime of approximately 50 digits.
 *  (50 is actually the length of the random number
 *  generated -- the first prime above that number will
 *  be returned.)
 */

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);
  }
}
```

Listing 3.33 *Primes.java*

```
package moreservlets;

import java.math.BigInteger;

/** A few utilities to generate a large random BigInteger,
 *  and find the next prime number above a given BigInteger.
 */
```

Listing 3.33 *Primes.java (continued)*

```
public class Primes {
  // Note that BigInteger.ZERO and BigInteger.ONE are
  // unavailable in JDK 1.1.
  private static final BigInteger ZERO = BigInteger.ZERO;
  private static final BigInteger ONE = BigInteger.ONE;
  private static final BigInteger TWO = new BigInteger("2");

  // Likelihood of false prime is less than 1/2^ERR_VAL
  // Assumedly BigInteger uses the Miller-Rabin test or
  // equivalent, and thus is NOT fooled by Carmichael numbers.
  // See section 33.8 of Cormen et al.'s Introduction to
  // Algorithms for details.
  private static final int ERR_VAL = 100;

  public static BigInteger nextPrime(BigInteger start) {
    if (isEven(start))
      start = start.add(ONE);
    else
      start = start.add(TWO);
    if (start.isProbablePrime(ERR_VAL))
      return(start);
    else
      return(nextPrime(start));
  }

  private static boolean isEven(BigInteger n) {
    return(n.mod(TWO).equals(ZERO));
  }

  private static StringBuffer[] digits =
    { new StringBuffer("0"), new StringBuffer("1"),
      new StringBuffer("2"), new StringBuffer("3"),
      new StringBuffer("4"), new StringBuffer("5"),
      new StringBuffer("6"), new StringBuffer("7"),
      new StringBuffer("8"), new StringBuffer("9") };

  private static StringBuffer randomDigit(boolean isZeroOK) {
    int index;
    if (isZeroOK) {
      index = (int)Math.floor(Math.random() * 10);
    } else {
      index = 1 + (int)Math.floor(Math.random() * 9);
    }
    return(digits[index]);
  }
```

Listing 3.33 *Primes.java (continued)*

```
  /** Create a random big integer where every digit is
   *  selected randomly (except that the first digit
   *  cannot be a zero).
   */

  public static BigInteger random(int numDigits) {
    StringBuffer s = new StringBuffer("");
    for(int i=0; i<numDigits; i++) {
      if (i == 0) {
        // First digit must be non-zero.
        s.append(randomDigit(false));
      } else {
        s.append(randomDigit(true));
      }
    }
    return(new BigInteger(s.toString()));
  }

  /** Simple command-line program to test. Enter number
   *  of digits, and it picks a random number of that
   *  length and then prints the first 50 prime numbers
   *  above that.
   */

  public static void main(String[] args) {
    int numDigits;
    try {
      numDigits = Integer.parseInt(args[0]);
    } catch (Exception e) { // No args or illegal arg.
      numDigits = 150;
    }
    BigInteger start = random(numDigits);
    for(int i=0; i<50; i++) {
      start = nextPrime(start);
      System.out.println("Prime " + i + " = " + start);
    }
  }
}
```

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, as shown earlier in Listing 3.30. To get started, just download a

sample from the source code archive at *http://www.moreservlets.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, I 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, I use

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

   Note that `tagclass` was renamed `tag-class` in JSP 1.2. So, if you use features specific to JSP 1.2 and use the JSP 1.2 `DOCTYPE`, you should use `tag-class`, not `tagclass`.
3. **`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` (the default value), and the rare tags whose handlers completely process the body themselves use a value of `tagdependent`. For the `SimplePrimeTag` discussed here, I use `empty` as below:

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

   Note that `bodycontent` was renamed `body-content` in JSP 1.2. However, as with the other new element names, you are only required to make the change if you use the JSP 1.2 `DOCTYPE`.
4. **`info`**, which gives a short description. Here, I use

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

   Note that `info` was renamed `description` in JSP 1.2.

Core Note

In JSP 1.2, `tagclass` was renamed `tag-class`, `bodycontent` was renamed `body-content`, and `info` was renamed `description`. However, the old element names still work in JSP 1.2 servers as long as the TLD file uses the JSP 1.1 `DOCTYPE`.

Listing 3.34 shows the relevant part of the TLD file.

Listing 3.34 *msajsp-taglib.tld* (Excerpt 1)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE ...>
<taglib>
  ...
  <tag>
    <name>simplePrime</name>
    <tagclass>moreservlets.tags.SimplePrimeTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info>Outputs a random 50-digit prime.</info>
  </tag>
  ...
</taglib>
```

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. Remember that the `uri` attribute can be an absolute or relative URL. When first learning, it is easiest to use a simple relative URL corresponding to a TLD file that is in the same directory as the JSP page that uses it. When we get to Web applications, however (see Chapter 4), we will see that it makes more sense for larger applications to put the TLD files in a subdirectory inside the *WEB-INF* directory. This configuration makes it easier to reuse the same TLD file from JSP pages in multiple directories, and it prevents end users from retrieving the TLD file.

Furthermore, as we will see in Section 5.13 (Locating Tag Library Descriptors), you can use the Web application deployment descriptor (i.e., *web.xml*) to change the meaning of strings supplied to the `uri` attribute of the `taglib` directive. For now, however, you will probably find it easiest to put the TLD file in the same directory as the JSP page that uses it and then use a simple filename as the value of the `uri` attribute.

Listing 3.35 shows a JSP document that uses

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

to use the TLD file just shown in Listing 3.34 with a prefix of `msajsp`. Since the base tag name is `simplePrime`, the full tag used is

```
<msajsp:simplePrime />
```

Figure 3–20 shows the result.

Listing 3.35 *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="msajsp-taglib.tld" prefix="msajsp" %>
<UL>
  <LI><msajsp:simplePrime />
  <LI><msajsp:simplePrime />
  <LI><msajsp:simplePrime />
  <LI><msajsp:simplePrime />
</UL>
</BODY>
</HTML>
```

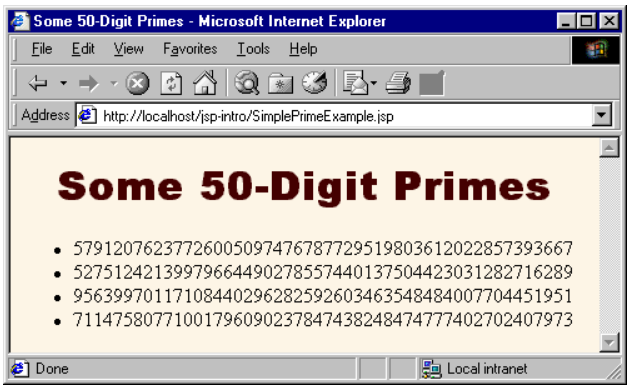

Figure 3–20 Result of *SimplePrimeExample.jsp*.

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). 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`).

Static values (i.e., those determined at page translation time) are always supplied to the method as type `String`. However, you can use `rtexprvalue` and `type` elements in the TLD file to permit attributes of other types to be dynamically calculated. See the following subsection for details.

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 3.36 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 3.36 *PrimeTag.java*

```
package moreservlets.tags;

/** 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 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 five nested elements that can appear between `<attribute>` and `</attribute>`.

1. **`name`**, a required element that defines the case-sensitive attribute name. In this case, I 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, I use

   ```
   <required>false</required>
   ```

   If `required` is `false` and the JSP page omits the attribute, no call is made to the `setAttributeName` method. So, be sure to give default values to the fields that the method sets. Omitting a required attribute results in an error at page translation time.
3. **`rtexprvalue`**, an optional element 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.

4. **`type`**, an optional element that designates the class to which the value should be typecast. Designating a `type` is only legal when `rtexprvalue` is `true`.
5. **`example`**, an optional element that gives an example of how to use the tag. This element is intended for development environments and has no effect on execution; it is available only in JSP 1.2.

Listing 3.37 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` (`moreservlets.tags.PrimeTag`), `bodycontent` (`empty`), and `info` (short description) elements. Note that if you use features specific to JSP 1.2 and the JSP 1.2 `DOCTYPE` (see Chapter 11, "New Tag Library Features in JSP 1.2"), you should change `tagclass`, `bodycontent`, and `info` to `tag-class`, `body-content`, and `description`, respectively.

Listing 3.37 *msajsp-taglib.tld* (Excerpt 2)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE ...>
<taglib>
  ...
  <tag>
    <name>prime</name>
    <tagclass>moreservlets.tags.PrimeTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info>Outputs a random N-digit prime.</info>
    <attribute>
      <name>length</name>
      <required>false</required>
    </attribute>
  </tag>
  ...
</taglib>
```

Tag Attributes: JSP File

Listing 3.38 shows a JSP document that uses the `taglib` directive to load the tag library descriptor file and to specify a prefix of `msajsp`. Since the `prime` tag is defined to permit a `length` attribute, Listing 3.38 uses

```
<msajsp: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

```
<msajsp:prime />
```

The tag handler is responsible for using a reasonable default value in such a case. Figure 3–21 shows the result of Listing 3.38.

Listing 3.38 *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="msajsp-taglib.tld" prefix="msajsp" %>
<UL>
  <LI>20-digit: <msajsp:prime length="20" />
  <LI>40-digit: <msajsp:prime length="40" />
  <LI>80-digit: <msajsp:prime length="80" />
  <LI>Default (50-digit): <msajsp:prime />
</UL>
</BODY>
</HTML>
```

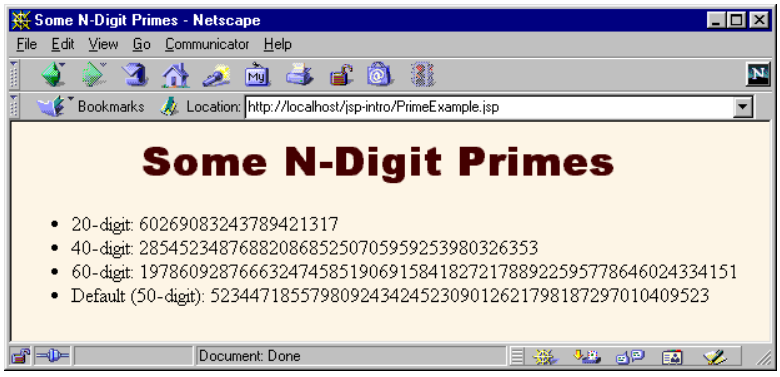

Figure 3–21 Result of *PrimeExample.jsp*.

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 written 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 3.39 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 3.39 *HeadingTag.java*

```
package moreservlets.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;
  }
```

Listing 3.39 *HeadingTag.java (continued)*

```
  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 `bodycontent` is optional (`JSP` is the default value) and is mainly intended for IDEs. The `name`, `tagclass`, `info`, and `attribute` elements are used in the same manner as described previously. Listing 3.40 gives the relevant part of the code.

Listing 3.40 *msajsp-taglib.tld* (Excerpt 3)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE ...>
<taglib>
  ...
  <tag>
    <name>heading</name>
    <tagclass>moreservlets.tags.HeadingTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <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>
```

Listing 3.40 *msajsp-taglib.tld* (Excerpt 3) *(continued)*

```
    <attribute>
      <name>width</name>
      <required>false</required>
    </attribute>
  </tag>

  ...
</taglib>
```

Tag Bodies: JSP File

Listing 3.41 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 3–22 shows the result.

Listing 3.41 *HeadingExample.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some Tag-Generated Headings</TITLE>
</HEAD>
<BODY>
<%@ taglib uri="msajsp-taglib.tld" prefix="msajsp" %>
<msajsp:heading bgColor="#C0C0C0">
Default Heading
</msajsp:heading>
<P>
<msajsp:heading bgColor="BLACK" color="WHITE">
White on Black Heading
</msajsp:heading>
<P>
<msajsp:heading bgColor="#EF8429" fontSize="60" border="5">
Large Bordered Heading
</msajsp:heading>
<P>
<msajsp:heading bgColor="CYAN" width="100%">
Heading with Full-Width Background
</msajsp:heading>
<P>
<msajsp:heading bgColor="CYAN" fontSize="60"
                fontList="Brush Script MT, Times, serif">
Heading with Non-Standard Font
</msajsp:heading>
</BODY>
</HTML>
```

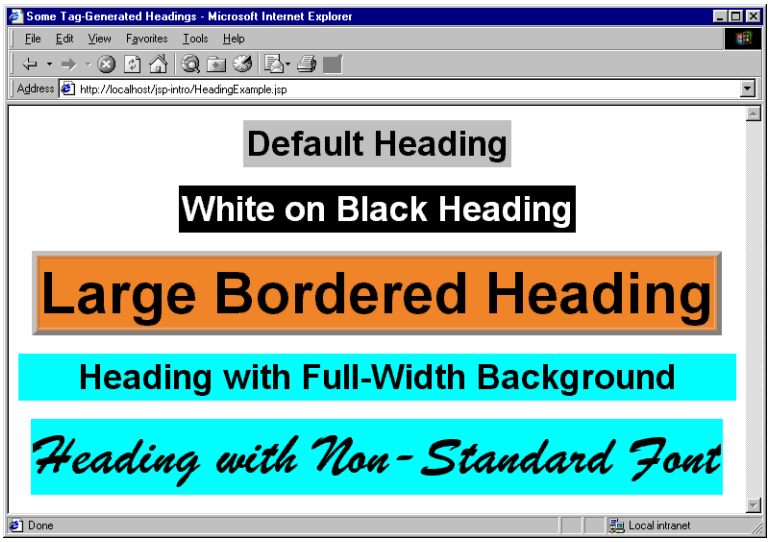

Figure 3–22 The custom `msajsp:heading` element gives you much more succinct control over heading format than do the standard `H1` through `H6` elements in HTML.

Optionally Including the Tag Body

Most tags either *never* make use of body content or *always* do so. In either case, you decide in advance whether the body content is used. However, you are also permitted to make this decision at request time. This subsection shows you how to use request time information to decide whether to include the tag body.

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 I just use `getParameter`, so no typecast is required.

Listing 3.42 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 3.42 *DebugTag.java*

```
package moreservlets.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 3.43 shows the entries needed for `DebugTag`.

Listing 3.43 *msajsp-taglib.tld* (Excerpt 4)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE ...>
<taglib>
  ...
  <tag>
    <name>debug</name>
    <tagclass>moreservlets.tags.DebugTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>Includes body only if debug param is set.</info>
  </tag>
  ...
</taglib>
```

Optional Body Inclusion: JSP File

Suppose that you have an application where most of the problems that occur are due to requests occurring close together in time, the host making the request, or session tracking. In such a case, the time, requesting host, and session ID would be useful information to track. Listing 3.43 shows a page that encloses debugging information between `<msajsp:debug>` and `</msajsp:debug>`. Figures 3–23 and 3–24 show the normal result and the result when a request time `debug` parameter is supplied, respectively.

Listing 3.44 *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="msajsp-taglib.tld" prefix="msajsp" %>
Top of regular page. Blah, blah, blah. Yadda, yadda, yadda.
<P>
<msajsp:debug>
<B>Debug:</B>
```

Listing 3.44 *DebugExample.jsp (continued)*

```
<UL>
  <LI>Current time: <%= new java.util.Date() %>
  <LI>Requesting hostname: <%= request.getRemoteHost() %>
  <LI>Session ID: <%= session.getId() %>
</UL>
</msajsp:debug>
<P>
Bottom of regular page. Blah, blah, blah. Yadda, yadda, yadda.
</BODY>
</HTML>
```

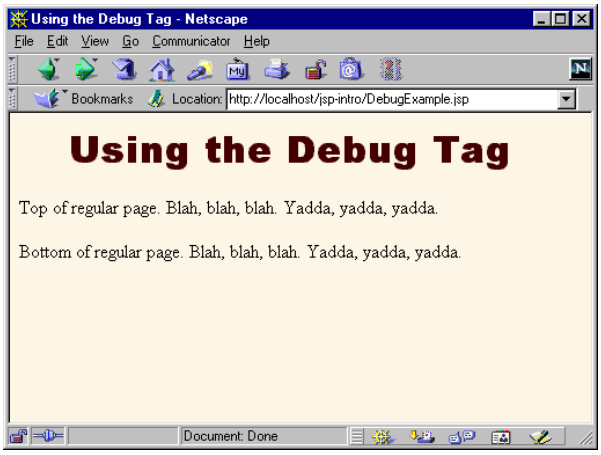

Figure 3–23 The body of the `msajsp:debug` element is normally ignored.

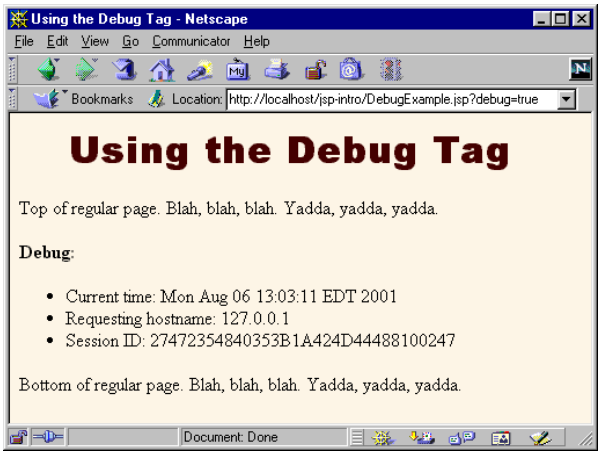

Figure 3–24 The body of the `msajsp:debug` element is included when a `debug` request parameter is supplied.

Manipulating the Tag Body

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

Tag Body Processing: Tag Handler Class

Up to this point, all of the tag handlers have extended the `TagSupport` class. This is a good standard starting point, since 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. Two important new methods are defined by `BodyTagSupport`:

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. In tag libraries that are intended only for JSP 1.2, you can use the `bodyContent` field of `BodyTagSupport` instead of calling `getBodyContent`. Most libraries, however, are intended to run in either JSP version.

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 2.10) contains a static `filter` method that takes a string and replaces <, >, ", and & with `<`, `>`, `&quot;`, 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 3.45 shows a tag implementation that gives this filtering functionality to a custom JSP tag.

Listing 3.45 *FilterTag.java*

```
package moreservlets.tags;

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

/** 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
    // (JSP 1.1/1.2) or EVAL_BODY_AGAIN (JSP 1.2 only)
    return(SKIP_BODY);
  }
}
```

Tag Body Processing: 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`. Other than that, nothing new is required in the descriptor file, as you can see by examining Listing 3.46, which shows the relevant portion of the TLD file.

Listing 3.46 *msajsp-taglib.tld* (Excerpt 5)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE ...>
<taglib>
  ...
  <tag>
    <name>filter</name>
    <tagclass>moreservlets.tags.FilterTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>Replaces HTML-specific characters in body.</info>
  </tag>
  ...
</taglib>
```

Tag Body Processing: JSP File

Listing 3.47 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 `<` and `>`. This necessity is particularly onerous during development when the sample HTML is frequently changing. Use of the `<msajsp:filter>` tag greatly simplifies the process, as Listing 3.47 illustrates. Figure 3–25 shows the result.

Listing 3.47 *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="msajsp-taglib.tld" prefix="msajsp" %>
```

Listing 3.47 *FilterExample.jsp (continued)*

```
<TABLE BORDER=1 ALIGN="CENTER">
<TR CLASS="COLORED"><TH>Example<TH>Result
<TR>
<TD><PRE><msajsp: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>
</msajsp: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>
```

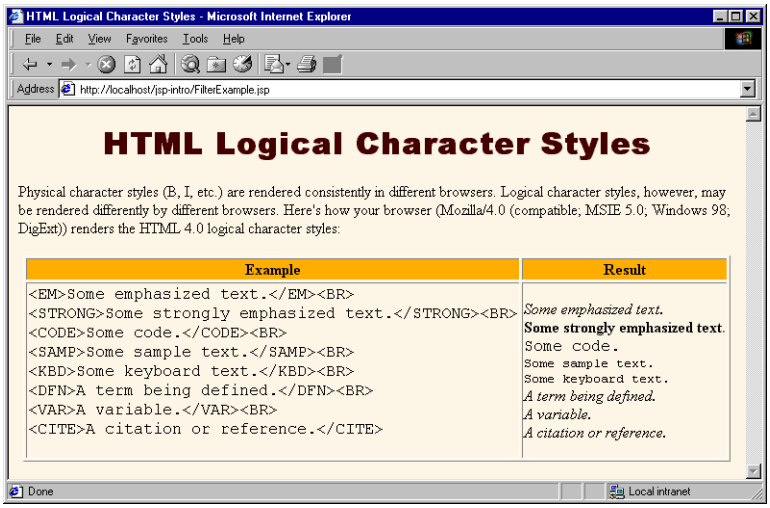

Figure 3–25 The `msajsp:filter` element lets you insert text without worrying about it containing special HTML characters.

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.

Multiple Body Actions: 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`, then the tag body is evaluated again, resulting in a new call to `doAfterBody`. This process continues until `doAfterBody` returns `SKIP_BODY`. In JSP 1.2, the `EVAL_BODY_TAG` constant is deprecated and replaced with `EVAL_BODY_AGAIN`. The two constants have the same value, but `EVAL_BODY_AGAIN` is a clearer name. So, if your tag library is designed to be used only in JSP 1.2 containers (e.g., it uses some features specific to JSP 1.2 as described in Chapter 11), you should use `EVAL_BODY_AGAIN`. Most tag libraries, however, are designed to run in either JSP version and thus use `EVAL_BODY_TAG`.

Core Note

`EVAL_BODY_TAG` is renamed `EVAL_BODY_AGAIN` in JSP 1.2.

Listing 3.48 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 3.48 *RepeatTag.java*

```
package moreservlets.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);
      }
      // Replace EVAL_BODY_TAG with EVAL_BODY_AGAIN in JSP 1.2.
      return(EVAL_BODY_TAG);
    } else {
      return(SKIP_BODY);
    }
  }
}
```

Multiple Body Actions: the Tag Library Descriptor File

Listing 3.49 shows the relevant section of the TLD file that gives the name `msa-jsp: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 3.49 *msajsp-taglib.tld* (Excerpt 6)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE ...>
<taglib>
  ...
  <tag>
    <name>repeat</name>
    <tagclass>moreservlets.tags.RepeatTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <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>
  ...
</taglib>
```

Multiple Body Actions: the JSP File

Listing 3.50 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 3–26 shows one possible result.

Listing 3.50 *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="msajsp-taglib.tld" prefix="msajsp" %>
<OL>
<!-- Repeats N times. A null reps value means repeat once. -->
```

Listing 3.50 *RepeatExample.jsp (continued)*

```
<msajsp:repeat reps='<%= request.getParameter("repeats") %>'>
  <LI><msajsp:prime length="40" />
</msajsp:repeat>
</OL>
</BODY>
</HTML>
```

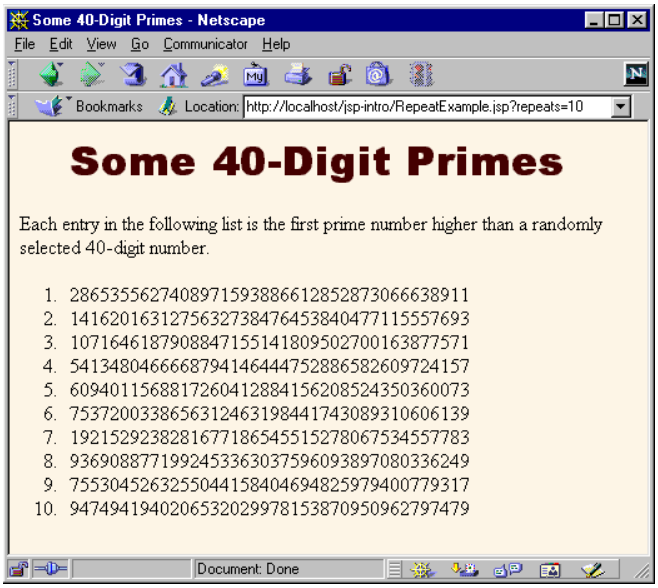

Figure 3–26 Result of *RepeatExample.jsp* when accessed with a `repeats` parameter of 10.

Using Nested Tags

Although Listing 3.50 places the `msajsp:prime` element within the `msajsp: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 an `msajsp: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 can-

not 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.

Nested Tags: 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`).

Although nested tags use the standard tag handler classes, they use two new techniques within those classes. 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:

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

To accomplish this task, the first step is to define an `IfTag` class to handle the `msajsp:if` tag. This handler should have methods to specify and check whether the condition is `true` or `false` (`setCondition` and `getCondition`). The handler should also have methods to designate and check whether the condition has ever been explicitly set (`setHasCondition` and `getHasCondition`), since we want to disallow `msajsp:if` tags that contain no `msajsp:condition` entry. Listing 3.51 shows the code for `IfTag`.

The second step is to define a tag handler for `msajsp:condition`. This class, called `IfConditionTag`, defines a `doStartTag` method that merely checks whether the tag appears within `IfTag`. It returns `EVAL_BODY_TAG` (`EVAL_BODY_BUFFERED` in tag libraries that are specific to JSP 1.2) 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 3.52.

Listing 3.51 *IfTag.java*

```
package moreservlets.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 3.52 *IfConditionTag.java*

```
package moreservlets.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");
    }
    // If your tag library is intended to be used ONLY
    // in JSP 1.2, replace EVAL_BODY_TAG with
    // EVAL_BODY_BUFFERED.
    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);
  }
}
```

The third step is to define a class to handle the `msajsp: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 3.53 shows the code.

The final step in defining tag handlers is to define a class for `msajsp:else`. This class is very similar to the one that handles 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 3.54.

Listing 3.53 *IfThenTag.java*

```
package moreservlets.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);
    }
    // If your tag library is intended to be used ONLY
    // in JSP 1.2, replace EVAL_BODY_TAG with
    // EVAL_BODY_BUFFERED.
    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 3.54 *IfElseTag.java*

```
package moreservlets.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);
    }
    // If your tag library is intended to be used ONLY
    // in JSP 1.2, replace EVAL_BODY_TAG with
    // EVAL_BODY_BUFFERED.
    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);
  }
}
```

Nested Tags: 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 JSP 1.1, 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 whether 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 (`tei-class` in JSP 1.2), which is used just like `tagclass`. In practice, however, `TagExtraInfo` is a bit cumbersome to use. Fortunately, JSP 1.2 introduced a very useful new class for this purpose: `TagLibraryValidator`. See Chapter 11 (New Tag Library Features in JSP 1.2) for information on using this class.

Listing 3.55 *msajsp-taglib.tld* (Excerpt 7)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE ...>
<taglib>
  ...
  <tag>
    <name>if</name>
    <tagclass>moreservlets.tags.IfTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>if/condition/then/else tag.</info>
  </tag>
  <tag>
    <name>condition</name>
    <tagclass>moreservlets.tags.IfConditionTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>condition part of if/condition/then/else tag.</info>
  </tag>
  <tag>
    <name>then</name>
    <tagclass>moreservlets.tags.IfThenTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>then part of if/condition/then/else tag.</info>
  </tag>
  <tag>
    <name>else</name>
    <tagclass>moreservlets.tags.IfElseTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>else part of if/condition/then/else tag.</info>
  </tag>
  ...
</taglib>
```

Nested Tags: the JSP File

Listing 3.56 shows a page that uses the `msajsp: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 3–27 shows a typical result.

Listing 3.56 *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="msajsp-taglib.tld" prefix="msajsp" %>
<msajsp:if>
  <msajsp:condition>true</msajsp:condition>
  <msajsp:then>Condition is true</msajsp:then>
  <msajsp:else>Condition is false</msajsp:else>
</msajsp:if>
<P>
<msajsp:if>
  <msajsp:condition><%= request.isSecure() %></msajsp:condition>
  <msajsp:then>Request is using SSL (https)</msajsp:then>
  <msajsp:else>Request is not using SSL</msajsp:else>
</msajsp:if>
<P>
Some coin tosses:<BR>
<msajsp:repeat reps="10">
  <msajsp:if>
    <msajsp:condition><%= Math.random() < 0.5 %></msajsp:condition>
    <msajsp:then><B>Heads</B><BR></msajsp:then>
    <msajsp:else><B>Tails</B><BR></msajsp:else>
  </msajsp:if>
</msajsp:repeat>
</BODY>
</HTML>
```

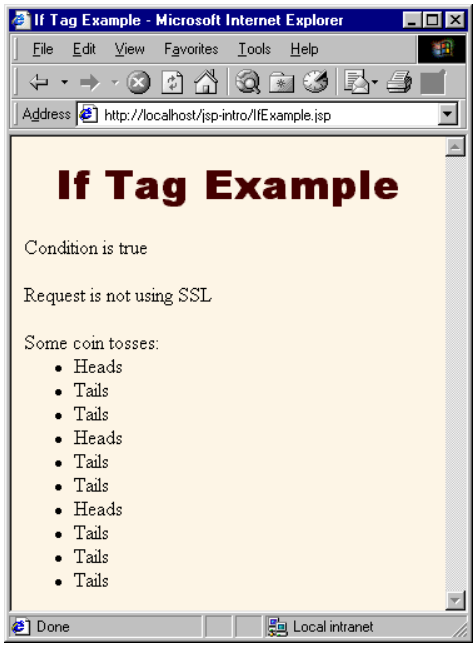

Figure 3–27 Result of *IfExample.jsp*.

3.8 Integrating Servlets and JSP: The MVC Architecture

Servlets are great when your application requires a lot of real programming to accomplish its task. 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 (see Figure 3–28), 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 (MVC)* or *Model 2* architecture. For code that supports a formalization of this approach, see the Apache Struts Framework at *http://jakarta.apache.org/struts/*.

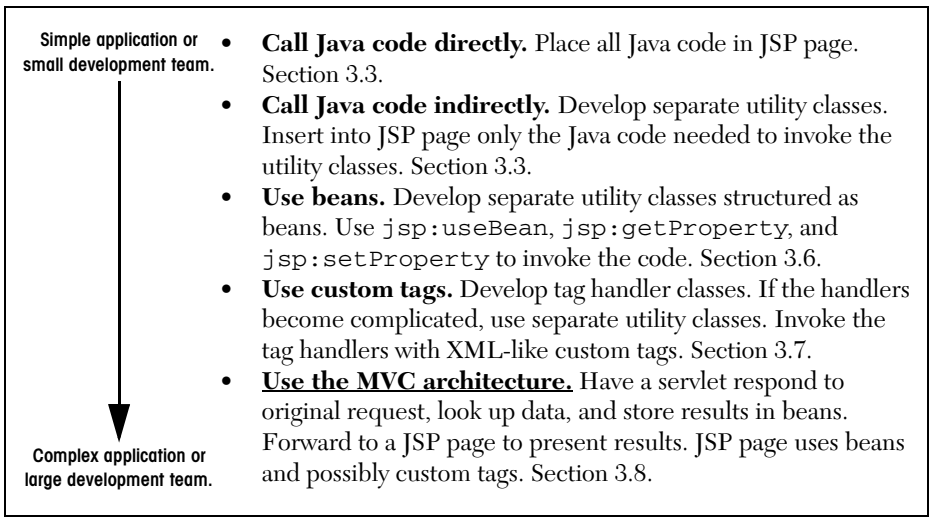

Figure 3–28 Strategies for invoking dynamic code from JSP.

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 `HttpServletResponse` as arguments. Both methods throw `ServletException` and `IOException`. For example, Listing 3.57 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, I 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 3.57 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

A servlet can store data for JSP pages in three main places: in the `HttpServletRequest`, in the `HttpSession`, and in the `ServletContext`. These storage locations correspond to the three nondefault values of the `scope` attribute of `jsp:useBean`: that is, `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 an arbitrary location 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. Section 4.5 (Handling Relative URLs in Web Applications) discusses several approaches to this problem. One simple solution, however, 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=...>`.

Using Alternative Means to Get a RequestDispatcher

Servers that support version 2.2 or 2.3 of the servlet specification have 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 (see Section 5.3, "Assigning Names and Custom URLs"), 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 Online Travel Agent

Consider the case of an online travel agent that has a quick-search page, as shown in Figure 3–29 and Listing 3.58. Users need to enter their email 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 in accordance with 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 3–30). 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.

To accomplish the desired behavior, the front end (Listing 3.58) submits the request to the top-level travel servlet shown in Listing 3.59. This servlet looks up the

customer information (see *http://www.moreservlets.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 `moreservlets.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 3.60 and the result in Figure 3–30) looks up the customer information by means of

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

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

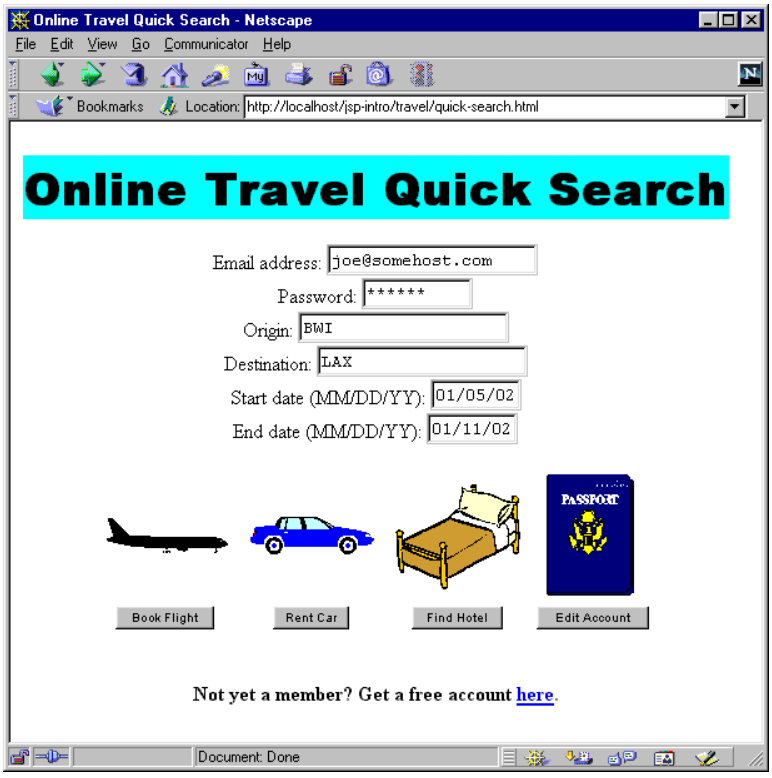

Figure 3–29 Front end to travel servlet (see Listing 3.58).

Listing 3.58 *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/moreservlets.Travel" METHOD="POST">
<CENTER>
Email address: <INPUT TYPE="TEXT" NAME="emailAddress"><BR>
Password: <INPUT TYPE="PASSWORD" NAME="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" NAME="flights" VALUE="Book Flight">
      </SMALL>
  ...
</TABLE>
</CENTER>
</FORM>
...
</BODY>
</HTML>
```

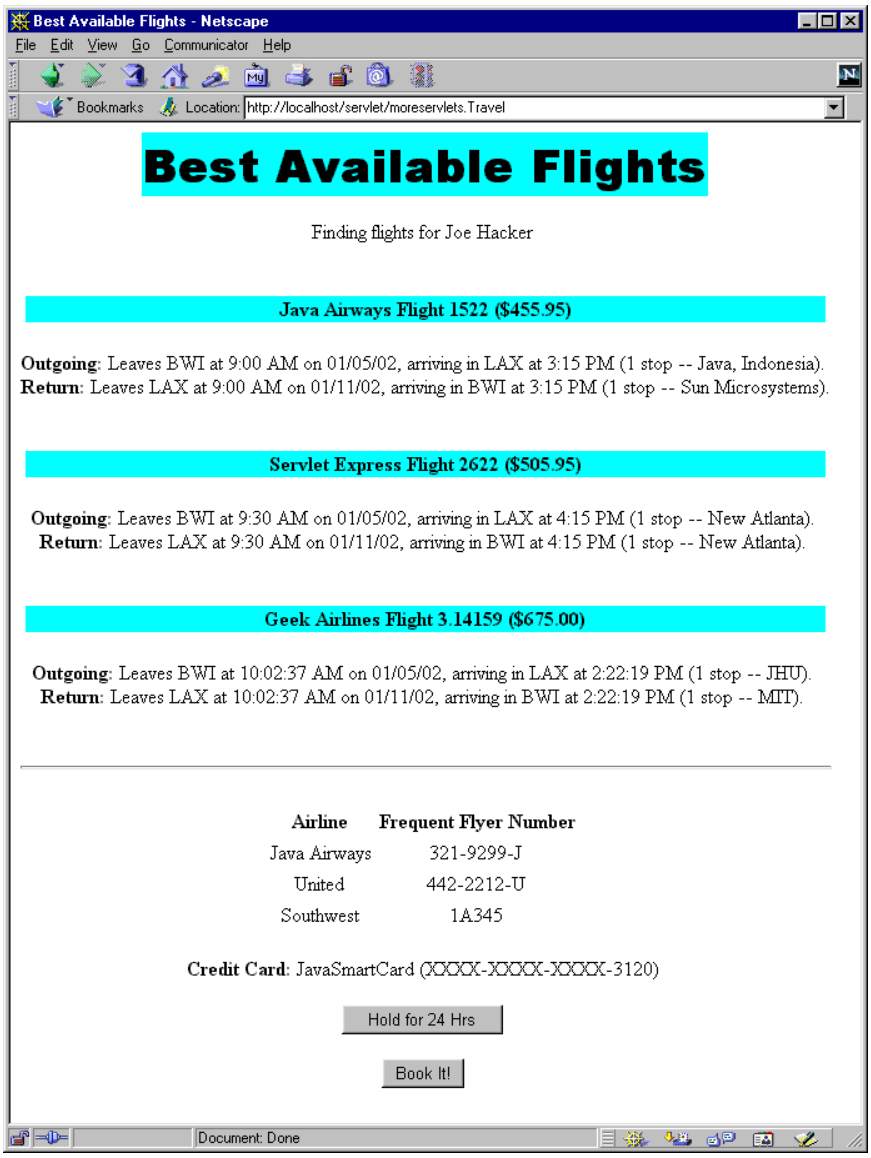

Figure 3–30 Result of travel servlet (Listing 3.59) dispatching request to *BookFlights.jsp* (Listing 3.60).

Listing 3.59 *Travel.java*

```
package moreservlets;

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. However,
   *  the use of POST means that you cannot forward
   *  the request to a static HTML page, since the forwarded
   *  request uses the same request method as the original
   *  one, and static pages cannot handle POST. Solution:
   *  have the "static" page be a JSP file that contains
   *  HTML only. That's what accounts.jsp is. The other
   *  JSP files really need to be dynamically generated,
   *  since they make use of the customer data.
   */

  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("/jsp-intro/travel/accounts.jsp",
               request, response);
    }
```

Listing 3.59 *Travel.java (continued)*

```
    // 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("/jsp-intro/travel/BookFlights.jsp",
               request, response);
    } else if (request.getParameter("cars") != null) {
      gotoPage("/jsp-intro/travel/RentCars.jsp",
               request, response);
    } else if (request.getParameter("hotels") != null) {
      gotoPage("/jsp-intro/travel/FindHotels.jsp",
               request, response);
    } else if (request.getParameter("account") != null) {
      gotoPage("/jsp-intro/travel/EditAccounts.jsp",
               request, response);
    } else {
      gotoPage("/jsp-intro/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 3.60 *BookFlights.jsp*

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

You should pay careful attention to the `TravelCustomer` class (shown partially in Listing 3.61, with the complete code available at *http://www.moreservlets.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. Other supporting classes (*FrequentFlyerInfo.java*, *TravelData.java*, etc.), JSP pages (*RentCars.jsp*, *FindHotels.jsp*, etc.), and the *travel-styles.css* style sheet can be found at *http://www.moreservlets.com*.

Listing 3.61 *TravelCustomer.java*

```
package moreservlets;

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);
```

Listing 3.61 *TravelCustomer.java (continued)*

```
    setStartDate(startDate);
    setEndDate(endDate);
    setFrequentFlyerData(frequentFlyerData);
    setRentalCarData(rentalCarData);
    setHotelData(hotelData);
  }

  public String getEmailAddress() {
    return(emailAddress);
  }

  public void setEmailAddress(String emailAddress) {
    this.emailAddress = emailAddress;
  }

  // See http://www.moreservlets.com for missing code.
  public String getFrequentFlyerTable() {
    FrequentFlyerInfo[] frequentFlyerData =
      getFrequentFlyerData();
    if (frequentFlyerData.length == 0) {
      return("<I>No frequent flyer data recorded.</I>");
    } else {
      String table =
        "<TABLE>\n" +
        "  <TR><TH>Airline<TH>Frequent Flyer Number\n";
      for(int i=0; i<frequentFlyerData.length; i++) {
        FrequentFlyerInfo info = frequentFlyerData[i];
        table = table +
                "<TR ALIGN=\"CENTER\">" +
                "<TD>" + info.getAirlineName() +
                "<TD>" + info.getFrequentFlyerNumber() + "\n";
      }
      table = table + "</TABLE>\n";
      return(table);
    }
  }

  // This would be replaced by a database lookup
  // in a real application.

  public String getFlights() {
    String flightOrigin =
      replaceIfMissing(getOrigin(), "Nowhere");
    String flightDestination =
      replaceIfMissing(getDestination(), "Nowhere");
    Date today = new Date();
```

Listing 3.61 *TravelCustomer.java (continued)*

```
    DateFormat formatter =
      DateFormat.getDateInstance(DateFormat.MEDIUM);
    String dateString = formatter.format(today);
    String flightStartDate =
      replaceIfMissing(getStartDate(), dateString);
    String flightEndDate =
      replaceIfMissing(getEndDate(), dateString);
    String [][] flights =
      { { "Java Airways", "1522", "455.95", "Java, Indonesia",
          "Sun Microsystems", "9:00", "3:15" },
        { "Servlet Express", "2622", "505.95", "New Atlanta",
          "New Atlanta", "9:30", "4:15" },
        { "Geek Airlines", "3.14159", "675.00", "JHU",
          "MIT", "10:02:37", "2:22:19" } };
    String flightString = "";
    for(int i=0; i<flights.length; i++) {
      String[] flightInfo = flights[i];
      flightString =
        flightString + getFlightDescription(flightInfo[0],
                                            flightInfo[1],
                                            flightInfo[2],
                                            flightInfo[3],
                                            flightInfo[4],
                                            flightInfo[5],
                                            flightInfo[6],
                                            flightOrigin,
                                            flightDestination,
                                            flightStartDate,
                                            flightEndDate);
    }
    return(flightString);
  }

  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) {
```

Listing 3.61 *TravelCustomer.java (continued)*

```
    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);
  }

  private String replaceIfMissing(String value,
                                  String defaultValue) {
    if ((value != null) && (value.length() > 0)) {
      return(value);
    } else {
      return(defaultValue);
    }
  }

  public static TravelCustomer findCustomer
                                (String emailAddress,
                                 TravelCustomer[] customers) {
    if (emailAddress == null) {
      return(null);
    }
    for(int i=0; i<customers.length; i++) {
      String custEmail = customers[i].getEmailAddress();
      if (emailAddress.equalsIgnoreCase(custEmail)) {
        return(customers[i]);
      }
    }
    return(null);
  }
}
```

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 code 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 %>" />
```

The `jsp:forward` action, like `jsp:include`, can make use of `jsp:param` elements to supply extra request parameters to the destination page. For details, see the discussion of `jsp:include` in Section 3.5.