Invoking Java Code with JSP Scripting Elements

Topics in This Chapter

- Static vs. dynamic text
- Dynamic code and good JSP design
- The importance of packages for JSP helper/utility classes
- JSP expressions
- JSP scriptlets
- JSP declarations
- Servlet code resulting from JSP scripting elements
- Scriptlets and conditional text
- Predefined variables
- Servlets vs. JSP pages for similar tasks

Chapter 11

This chapter discusses the "classic" approach to invoking Java code from within JSP pages. This approach works in both JSP 1 (i.e., JSP 1.2 and earlier) and JSP 2. Chapter 16 discusses the JSP expression language, which provides a concise mechanism to indirectly invoke Java code, but only in JSP 2.0 and later.

11.1 Creating Template Text

In most cases, a large percentage of your JSP document consists of static text (usually HTML), known as *template text*. In almost all respects, this HTML looks just like normal HTML, follows all the same syntax rules, and is simply "passed through" to the client by the servlet created to handle the page. Not only does the HTML look normal, it can be created by whatever tools you already are using for building Web pages. For example, we used Macromedia Dreamweaver for many 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 `<%` or `%>` in the output, you need to put `<\%` or `%\>` 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 client normally.

11.2 Invoking Java Code from JSP

There are a number of different ways to generate dynamic content from JSP, as illustrated in Figure 11–1. 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, hard to reuse, and hard to divide among different members of the development team. See Section 11.3 (Limiting the Amount of Java Code in JSP Pages) for details. Nevertheless, many pages are quite simple, and the first two approaches of Figure 11–1 (placing explicit Java code directly in the page) work quite well. This chapter discusses those approaches.

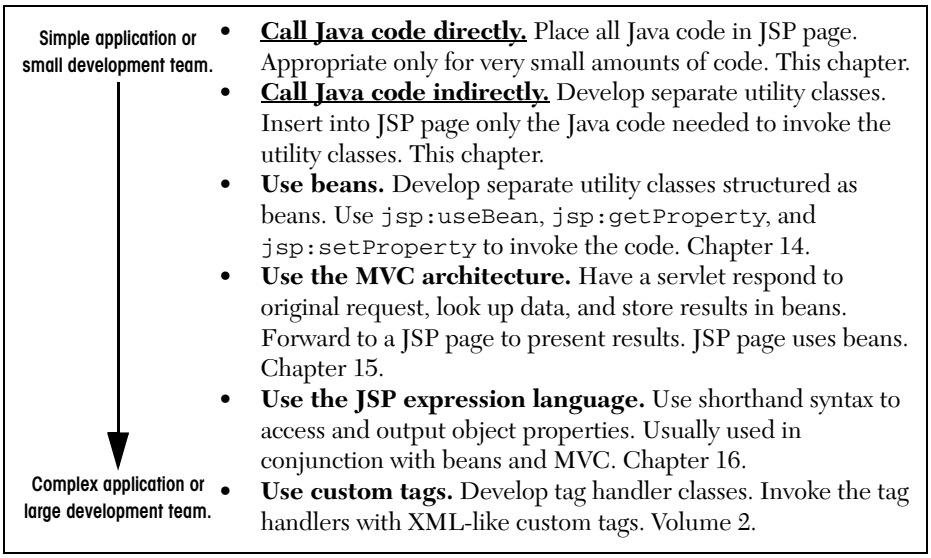

Figure 11–1 Strategies for invoking dynamic code from JSP.

Types of JSP Scripting Elements

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

1. **Expressions** of the form `<%=` *Java Expression* `%>`, which are evaluated and inserted into the servlet's output.
2. **Scriptlets** of the form `<%` *Java Code* `%>`, which are inserted into the servlet's `_jspService` method (called by `service`).
3. **Declarations** of the form `<%!` *Field/Method Declaration* `%>`, which are inserted into the body of the servlet class, outside any existing methods.

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

11.3 Limiting the Amount of Java Code in JSP Pages

You have 25 lines of Java code that you need to invoke. You have two options: (1) put all 25 lines directly in the JSP page, or (2) put the 25 lines of code in a separate Java class, put the Java class in `WEB-INF/classes/`*directoryMatchingPackageName*, and use one or two lines of JSP-based Java code to invoke it. Which is better? The second. The second. The second! And all the more so if you have 50, 100, 500, or 1000 lines of code. Here's why:

- **Development.** You generally write regular classes in a Java-oriented environment (e.g., an IDE like JBuilder or Eclipse or a code editor like UltraEdit or emacs). You generally write JSP in an HTML-oriented environment like Dreamweaver. The Java-oriented environment is typically better at balancing parentheses, providing tooltips, checking the syntax, colorizing the code, and so forth.
- **Compilation.** To compile a regular Java class, you press the Build button in your IDE or invoke `javac`. To compile a JSP page, you have to drop it in the right directory, start the server, open a browser, and enter the appropriate URL.
- **Debugging.** We know this never happens to you, but when *we* write Java classes or JSP pages, we occasionally make syntax errors. If there is a syntax error in a regular class definition, the compiler tells you right away and it also tells you what line of code contains the error. If

there is a syntax error in a JSP page, the server typically tells you what line *of the servlet* (i.e., the servlet into which the JSP page was translated) contains the error. For tracing output at runtime, with regular classes you can use simple `System.out.println` statements if your IDE provides nothing better. In JSP, you can sometimes use print statements, but where those print statements are displayed varies from server to server.

- **Division of labor.** Many large development teams are composed of some people who are experts in the Java language and others who are experts in HTML but know little or no Java. The more Java code that is directly in the page, the harder it is for the Web developers (the HTML experts) to manipulate it.
- **Testing.** Suppose you want to make a JSP page that outputs random integers between designated 1 and some bound (inclusive). You use `Math.random`, multiply by the range, cast the result to an `int`, and add 1. Hmm, that sounds right. But are you sure? If you do this directly in the JSP page, you have to invoke the page over and over to see if you get all the numbers in the designated range but no numbers outside the range. After hitting the Reload button a few dozen times, you will get tired of testing. But, if you do this in a static method in a regular Java class, you can write a test routine that invokes the method inside a loop (see Listing 11.13), and then you can run hundreds or thousands of test cases with no trouble. For more complicated methods, you can save the output, and, whenever you modify the method, compare the new output to the previously stored results.
- **Reuse.** You put some code in a JSP page. Later, you discover that you need to do the same thing in a different JSP page. What do you do? Cut and paste? Boo! Repeating code in this manner is a cardinal sin because if (when!) you change your approach, you have to change many different pieces of code. Solving the code reuse problem is what object-oriented programming is all about. Don't forget all your good OOP principles just because you are using JSP to simplify the generation of HTML.

"But wait!" you say, "I have an IDE that makes it easier to develop, debug, and and compile JSP pages." OK, good point. There is no hard and fast rule for exactly how much Java code is too much to go directly in the page. But no IDE solves the testing and reuse problems, and your general design strategy should be centered around putting the complex code in regular Java classes and keeping the JSP pages relatively simple.

Core Approach

Limit the amount of Java code that is in JSP pages. At the very least, use helper classes that are invoked from the JSP pages. Once you gain more experience, consider beans, MVC, and custom tags as well.

Almost all experienced developers have seen gross excesses: JSP pages that consist of many lines of Java code followed by tiny snippets of HTML. That is obviously bad: it is harder to develop, compile, debug, divvy up among team members, test, and reuse. A servlet would have been far better. However, some of these developers have overreacted by flatly stating that it is *always* wrong to have *any* Java code directly in the JSP page. Certainly, on some projects it is worth the effort to keep a strict separation between the content and the presentation and to enforce a style where there is no Java syntax in any of the JSP pages. But this is not always necessary (or even beneficial).

A few people go even further by saying that *all* pages in *all* applications should use the Model-View-Controller (MVC) architecture, preferably with the Apache Struts framework. This, in our opinion, is also an overreaction. Yes, MVC (Chapter 15) is a great idea, and we use it all the time on real projects. And, yes, Struts (Volume 2) is a nice framework; we are using it on a large project as the book is going to press. The approaches are great when the situation gets moderately (MVC in general) or highly (Struts) complicated.

But simple situations call for simple solutions. In our opinion, all the approaches of Figure 11–1 have a legitimate place; it depends mostly on the complexity of the application and the size of the development team. Still, be warned: beginners are much more likely to err by making hard-to-manage JSP pages chock-full of Java code than they are to err by using unnecessarily large and elaborate frameworks.

The Importance of Using Packages

Whenever you write Java classes, the class files are deployed in `WEB-INF/classes/`*`directoryMatchingPackageName`* (or inside a JAR file that is placed in `WEB-INF/lib`). This is true regardless of whether the class is a servlet, a regular helper class, a bean, a custom tag handler, or anything else. All code goes in the same place.

With regular servlets, however, it is sometimes reasonable to use the default package, since you can use separate Web applications (see Section 2.11) to avoid name conflicts with servlets from other projects. However, with code called from JSP, you should always use packages. And, since when you write a utility for use from a servlet, you do not know if you will later use it from a JSP page as well, this strategy means that you should *always* use packages for *all* classes used by either servlets or JSP pages.

Core Approach

Put all your classes in packages.

Why? To answer that question, consider the following code. The code may or may not contain a `package` declaration but does not contain `import` statements.

```
...
public class SomeClass {
  public String someMethod(...) {
    SomeHelperClass test = new SomeHelperClass(...);
    String someString = SomeUtilityClass.someStaticMethod(...);
    ...
  }
}
```

Now, the question is, what package will the system think that `SomeHelperClass` and `SomeUtilityClass` are in? The answer is, whatever package `SomeClass` is in. What package is that? Whatever is given in the `package` declaration. OK, fine. Elementary Java syntax. No problem. OK, then, consider the following JSP code:

```
...
<%
  SomeHelperClass test = new SomeHelperClass(...);
  String someString = SomeUtilityClass.someStaticMethod(...);
%>
```

Now, same question: what package will the system think that `SomeHelperClass` and `SomeUtilityClass` are in? Same answer: whatever package the current class (the servlet that the JSP page is translated into) is in. What package is that? Hmm, good question. Nobody knows! The package is not standardized by the JSP spec. So, packageless helper classes, when used in this manner, will only work if the system builds a packageless servlet. But they don't always do that, so JSP code like this example can fail. To make matters worse, servers sometimes *do* build packageless servlets out of JSP pages. For example, most Tomcat versions build packageless servlets for JSP pages that are in the top-level directory of the Web application. The problem is that there is absolutely no standard to guide when they do this and when they don't. It would be far better if the JSP code just shown always failed. Instead, it sometimes works and sometimes fails, depending on the server or even depending on what directory the JSP page is in. Boo!

Be safe, be portable, plan ahead. Always use packages!

11.4 Using JSP 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. This evaluation is performed at runtime (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` (the method that replaces `doGet` in servlets that result from JSP pages). These implicit objects are discussed in more detail in Section 11.12, but for the purpose of expressions, the most important ones are these:

- **`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 12.4).
- **`out`**, the `Writer` (a buffered version of type `JspWriter`) used to send output to the client.
- **`application`**, the `ServletContext`. This is a data structure shared by all servlets and JSP pages in the Web application and is good for storing shared data. We discuss it further in the chapters on beans (Chapter 14) and MVC (Chapter 15).

Here is an example:

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

JSP/Servlet Correspondence

Now, we just stated that a JSP expression is evaluated and inserted into the page output. Although this is true, it is sometimes helpful to understand what is going on behind the scenes.

It is actually quite 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 11.1 shows a small JSP sample that includes some static HTML and a JSP expression. Listing 11.2 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, we 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 11.1 Sample JSP Expression: Random Number

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

Listing 11.2 Representative Resulting Servlet Code: Random Number

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html");
  HttpSession session = request.getSession();
  JspWriter out = response.getWriter();
  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 Autogenerated Servlet Source Code

install_dir/work/Standalone/localhost/_
(The final directory is an underscore. More generally, in
install_dir/work/Standalone/localhost/*webAppName*.
The location varies slightly among various Tomcat versions.)

JRun Autogenerated Servlet Source Code

install_dir/servers/default/default-ear/default-war/WEB-INF/jsp
(More generally, in the WEB-INF/jsp directory of the Web application to which the JSP page belongs. However, note that JRun does not save the .java files unless you change the `keepGenerated` element from `false` to `true` in *install_dir*/servers/default/SERVER-INF/default-web.xml.)

Resin Autogenerated Servlet Source Code

install_dir/doc/WEB-INF/work
(More generally, in the WEB-INF/work directory of the Web application to which the JSP page belongs.)

XML Syntax for Expressions

XML authors can use the following alternative syntax for JSP expressions:

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

In JSP 1.2 and later, servers are required to support this syntax as long as authors don't mix the XML version and the standard JSP version (`<%= ... %>`) in the same page. This means that, to use the XML version, you must use XML syntax in the *entire* page. In JSP 1.2 (but not 2.0), this requirement means that you have to enclose the entire page in a `jsp:root` element. As a result, most developers stick with the classic syntax except when they are either generating XML documents (e.g., xhtml or SOAP) or when the JSP page is itself the output of some XML process (e.g., XSLT).

Note that XML elements, unlike HTML ones, are case sensitive. So, be sure to use `jsp:expression` in lower case.

11.5 Example: JSP Expressions

Listing 11.3 gives an example JSP page called Expressions.jsp. We placed the file in a directory called jsp-scripting, copied the entire directory from our development directory to the top level of the default Web application (in general, to the top-level directory of the Web application—one level up from WEB-INF), and used a URL of http://*host*/jsp-scripting/Expressions.jsp. Figures 11–2 and 11–3 show some typical results.

Notice that we include `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. This convenience is an important factor in the use of JSP. JSP pages are not more *powerful* than servlets (they *are* servlets behind the scenes), but they are sometimes more *convenient* than servlets.

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. Moreover, servers are expressly prohibited from making content in WEB-INF/classes (or anywhere in WEB-INF) directly accessible to clients. So, it is impossible to put style sheets in the same directory as servlet class files, even if you use the web.xml `servlet` and `servlet-mapping` elements (see Section 2.11, "Web Applications: A Preview") to customize servlet URLs. 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` (see Chapter 15, "Integrating Servlets and JSP: The Model View Controller (MVC) Architecture").

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.coreservlets.com.

Listing 11.3 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.">
```

Listing 11.3 Expressions.jsp *(continued)*

```
<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>
```

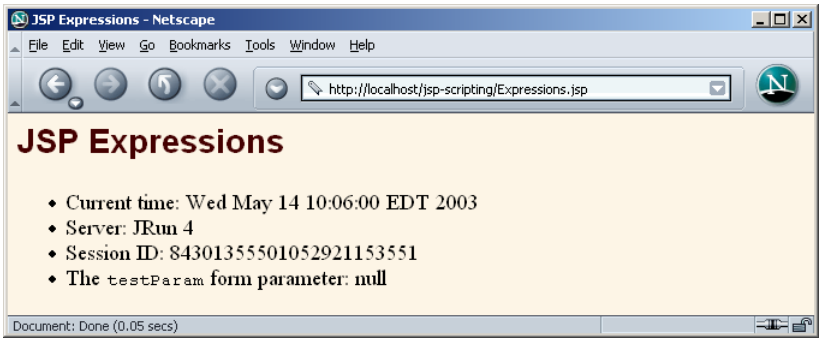

Figure 11–2 Result of Expressions.jsp using Macromedia JRun and omitting the `testParam` request parameter.

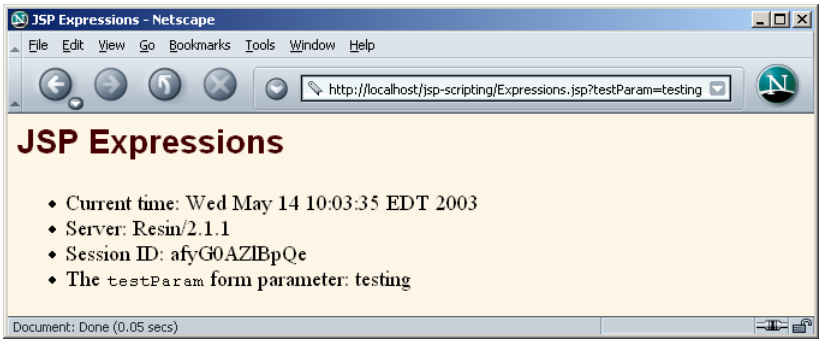

Figure 11–3 Result of Expressions.jsp using Caucho Resin and specifying `testing` as the value of the `testParam` request parameter.

11.6 Comparing Servlets to JSP Pages

In Section 4.3, we presented an example of a servlet that outputs the values of three designated form parameters. The code for that servlet is repeated here in Listing 11.4. Listing 11.5 (Figure 11–4) shows a version rewritten in JSP, using JSP expressions to access the form parameters. The JSP version is clearly superior: shorter, simpler, and easier to maintain.

Now, this is not to say that all servlets will convert to JSP so cleanly. JSP works best when the *structure* of the HTML page is fixed but the *values* at various places need to be computed dynamically. If the structure of the page is dynamic, JSP is less beneficial. Sometimes servlets are better in such a case. And, of course, if the page consists of binary data or has little static content, servlets are clearly superior. Furthermore, sometimes the answer is neither servlets nor JSP alone, but rather a combination of the two. For details, see Chapter 15 (Integrating Servlets and JSP: The Model View Controller (MVC) Architecture).

Listing 11.4 ThreeParams.java

```
package coreservlets;

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

public class ThreeParams extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Reading Three Request Parameters";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
                "<UL>\n" +
                "  <LI><B>param1</B>: "
                + request.getParameter("param1") + "\n" +
                "  <LI><B>param2</B>: "
```

Listing 11.4 ThreeParams.java *(continued)*

```
                + request.getParameter("param2") + "\n" +
                "  <LI><B>param3</B>: "
                + request.getParameter("param3") + "\n" +
                "</UL>\n" +
                "</BODY></HTML>");
  }
}
```

Listing 11.5 ThreeParams.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Reading Three Request Parameters</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Reading Three Request Parameters</H1>
<UL>
  <LI><B>param1</B>: <%= request.getParameter("param1") %>
  <LI><B>param2</B>: <%= request.getParameter("param2") %>
  <LI><B>param3</B>: <%= request.getParameter("param3") %>
</UL>
</BODY></HTML>
```

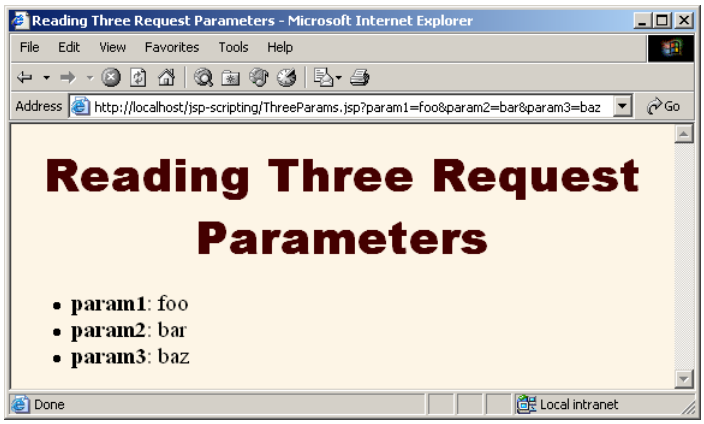

Figure 11–4 Result of ThreeParams.jsp.

11.7 Writing Scriptlets

If you want to do something more complex than output the value of 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 a combination of a scriptlet and a JSP expression, as below.

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

Or, you could have used a single JSP expression, as here.

```
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 Microsoft Word, not as HTML (which is the default). Since Microsoft Word can import HTML documents, this technique is actually quite useful in real applications.

```
<% response.setContentType("application/msword"); %>
```

It is important to note that you need not set response headers or status codes at the *very* top of 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. It is legal to set headers and status codes after a small amount of document content 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 Chapter 12 (Controlling the Structure of Generated Servlets: The JSP page Directive) 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 11.6 shows a small JSP sample that includes some static HTML, a JSP expression, and a JSP scriptlet. Listing 11.7 shows a `_jspService` method that might result. Note that the call to `bar` (the JSP expression) is not followed by a semicolon, but the call to `baz` (the JSP scriptlet) is. Remember that JSP expressions contain Java *values* (which do not end in semicolons), whereas most JSP scriptlets contain Java *statements* (which are terminated by semicolons). To make it even easier to remember when to use a semicolon, simply remember that expressions get placed inside `print` or `write` statements, and `out.print(blah;);` is clearly illegal.

Again, different vendors will produce this code in slightly different ways, and we 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 11.6 Sample JSP Expression/Scriptlet

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

Listing 11.7 Representative Resulting Servlet Code: Expression/Scriptlet

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

XML Syntax for Scriptlets

The XML equivalent of `<% Java Code %>` is

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

In JSP 1.2 and later, 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; if you use the XML version you must use XML syntax consistently for the entire page. Remember that XML elements are case sensitive; be sure to use `jsp:scriptlet` in lower case.

11.8 Scriptlet Example

As an example of code that is too complex for a JSP expression alone, Listing 11.8 presents a JSP page that uses the `bgColor` request parameter to set the background color of the page. Simply using

```
<BODY BGCOLOR="<%= request.getParameter("bgColor") %>">
```

would violate the cardinal rule of reading form data: always check for missing or malformed data. So, we use a scriptlet instead. `JSP-Styles.css` is omitted so that the style sheet does not override the background color. Figures 11–5, 11–6, and 11–7 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 by most browsers for historical reasons), respectively.

Listing 11.8 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.trim().equals(""))) {
  bgColor = "WHITE";
}
%>
<BODY BGCOLOR="<%= bgColor %>">
<H2 ALIGN="CENTER">Testing a Background of "<%= bgColor %>"</H2>
</BODY></HTML>
```

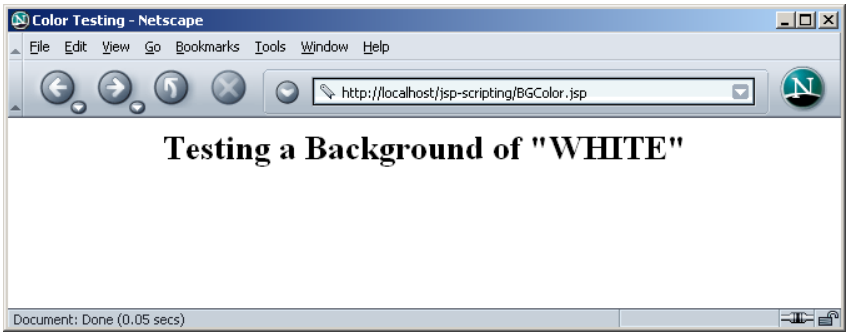

Figure 11–5 Default result of BGColor.jsp.

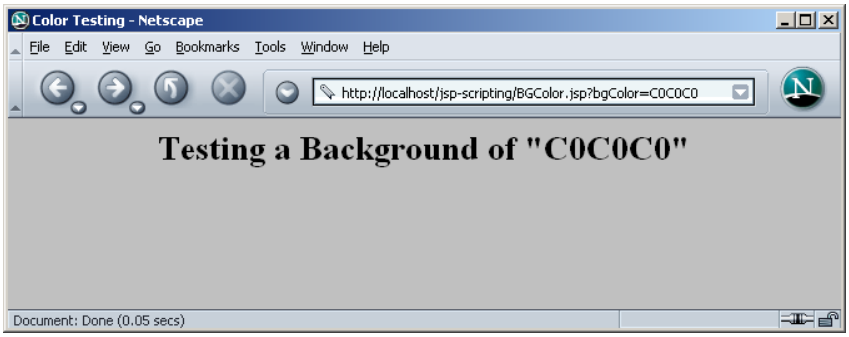

Figure 11–6 Result of BGColor.jsp when accessed with a `bgColor` parameter having the RGB value `C0C0C0`.

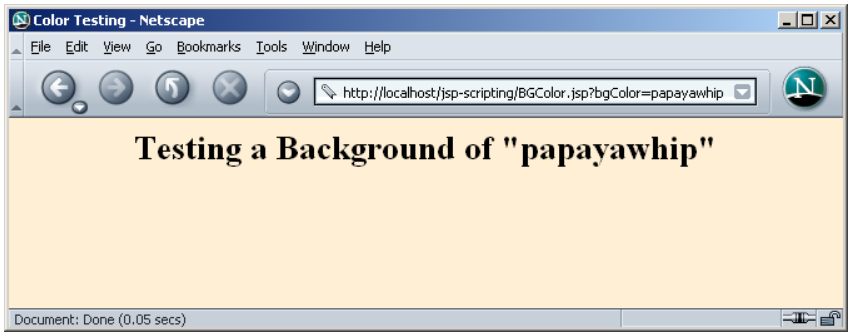

Figure 11–7 Result of BGColor.jsp when accessed with a `bgColor` parameter having the X11 color name `papayawhip`.

11.9 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 tag. Key to this approach are the facts that (a) code inside a scriptlet gets inserted into the resultant servlet's `_jspService` method (called by `service`) *exactly* as written and (b) that any static HTML (template text) before or after a scriptlet gets converted to `print` statements. This behavior means that scriptlets need not contain complete Java statements and that code blocks left open can affect the static HTML or JSP outside the scriptlets. For example, consider the JSP fragment of Listing 11.9 that contains mixed template text and scriptlets.

Listing 11.9 DayWish.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Wish for the Day</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<% if (Math.random() < 0.5) { %>
<H1>Have a <I>nice</I> day!</H1>
<% } else { %>
<H1>Have a <I>lousy</I> day!</H1>
<% } %>
</BODY></HTML>
```

You probably find the bold part a bit confusing. We certainly did the first few times we saw constructs of this nature. 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. See Figures 11–8 and 11–9.

Don't panic! Simply follow the rules for how JSP code gets converted to servlet code. Once 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("<H1>Have a <I>nice</I> day!</H1>");
} else {
  out.println("<H1>Have a <I>lousy</I> day!</H1>");
}
```

The key is that the first two scriptlets do not contain complete statements, but rather partial statements that have dangling braces. This serves to capture the subsequent HTML within the `if` or `else` clauses.

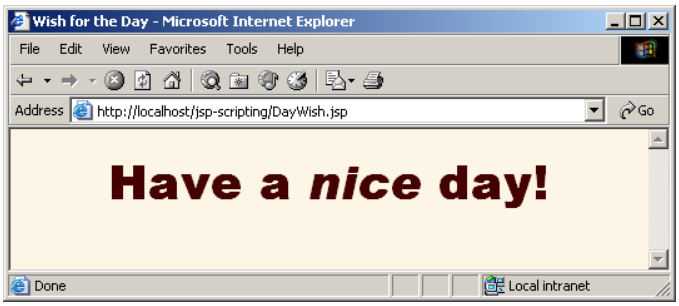

Figure 11–8 One possible result of DayWish.jsp.

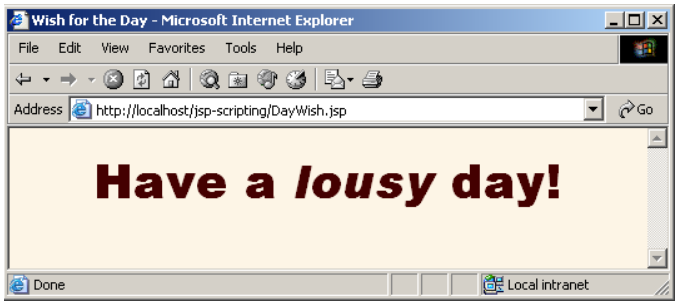

Figure 11–9 Another possible result of DayWish.jsp.

Overuse of this approach can lead to JSP code that is hard to understand and maintain. Avoid using it to conditionalize large sections of HTML, and try to keep your JSP pages as focused on presentation (HTML output) tasks as possible. Nevertheless, there are some situations in which the alternative approaches are also unappealing. The primary example is generation of lists or tables containing an indeterminate number of entries. This happens quite frequently when you are presenting data that is the result of a database query. See Chapter 17 (Accessing Databases with JDBC) for details on database access from Java code. Besides, even if *you* do not use this approach, you are bound to see examples of it in your projects, and you need to understand how and why it works as it does.

11.10 Using Declarations

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

```
<%! Field or Method Definition %>
```

Since declarations do not generate output, they are normally used in conjunction with JSP expressions or scriptlets. In principle, JSP declarations can contain field (instance variable) definitions, method definitions, inner class definitions, or even static initializer blocks: anything that is legal to put inside a class definition but outside any existing methods. In practice, however, declarations almost always contain field or method definitions.

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 in servlets that come from JSP.

Core Approach

For initialization and cleanup in JSP pages, use JSP declarations to override `jspInit` or `jspDestroy`, not `init` or `destroy`.

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 (compilation warnings give the right line numbers; 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 instance variables (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

Define most methods with separate Java classes, not 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 whether the code from declarations goes at the top or bottom of the servlet. For instance, Listing 11.10 shows a small JSP snippet that includes some static HTML, a JSP declaration, and a JSP expression. Listing 11.11 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 we oversimplified the `out` variable (which is a `JspWriter`, not the slightly simpler `PrintWriter` that results from a call to `getWriter`). Finally, the servlet will never implement `HttpJspPage` directly, but rather will extend some vendor-specific class that already implements `HttpJspPage`. So, don't expect the code your server generates to look *exactly* like this.

Listing 11.10 Sample JSP Declaration

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

Listing 11.11 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 {
```

Listing 11.11 Representative Resulting Servlet Code: Declaration *(continued)*

```
  response.setContentType("text/html");
  HttpSession session = request.getSession();
  JspWriter out = response.getWriter();
  out.println("<H1>Some Heading</H1>");
  out.println(randomHeading());
  ...
 }

  ...
}
```

XML Syntax for Declarations

The XML equivalent of `<%! Field or Method Definition %>` is

```
<jsp:declaration>Field or Method Definition</jsp:declaration>
```

In JSP 1.2 and later, 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. The entire page must follow XML syntax if you are going to use the XML form, so most developers stick with the classic syntax except when they are using XML anyhow. Remember that XML elements are case sensitive; be sure to use `jsp:declaration` in lower case.

11.11 Declaration Example

In this example, the following JSP snippet 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 two lines 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

the now-deprecated `SingleThreadModel` interface (see Section 3.7). 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, in principle, 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 that is similar to the previous snippet but that uses `synchronized` blocks to guarantee thread safety, see the discussion of the `isThreadSafe` attribute of the `page` directive in Chapter 12.

Listing 11.12 shows the full JSP page; Figure 11–10 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.

Second, even if you use `jspDestroy`, 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. (Note that session objects are automatically shared across distributed applications as long as the values are `Serializable`. But session values are specific to each user, whereas we need client-independent data in this case.)

Listing 11.12 AccessCounts.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>JSP Declarations</TITLE>
<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>
```

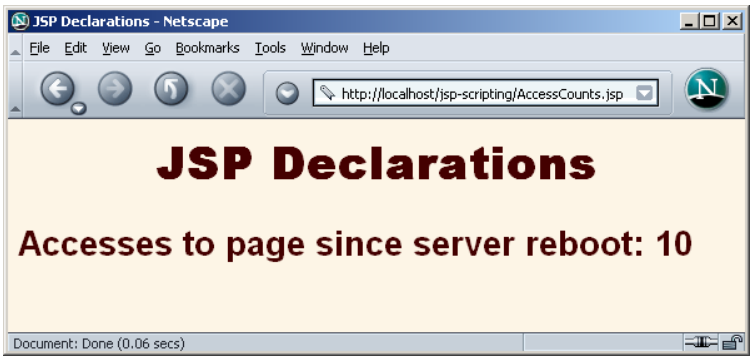

Figure 11–10 Visiting AccessCounts.jsp after it has been requested nine previous times by the same or different clients.

11.12 Using Predefined Variables

When you wrote a `doGet` method for a servlet, you probably wrote something like this:

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html");
  HttpSession session = request.getSession();
  PrintWriter out = response.getWriter();
  out.println(...);
  ...
}
```

The servlet API told you the types of the arguments to `doGet`, the methods to call to get the session and writer objects, and their types. JSP changes the method name from `doGet` to `_jspService` and uses a `JspWriter` instead of a `PrintWriter`. But the idea is the same. The question is, who told you what variable names to use? The answer is, nobody! You chose whatever names you wanted.

For JSP expressions and scriptlets to be useful, you need to know what variable names the autogenerated servlet uses. So, the specification tells you. You are supplied with eight automatically defined local variables in `_jspService`, sometimes called "implicit objects." Nothing is special about these; they are merely the names of the local variables. *Local* variables. Not constants. Not JSP reserved words. Nothing magic. So, if you are writing code that is not part of the `_jspService` method, these variables are not available. In particular, since JSP declarations result in code

that appears outside the `_jspService` method, these variables are not accessible in declarations. Similarly, they are not available in utility classes that are invoked by JSP pages. If you need a separate method to have access to one of these variables, do what you always do in Java code: pass the variable along.

The available variables are `request`, `response`, `out`, `session`, `application`, `config`, `pageContext`, and `page`. Details for each are given below. An additional variable called `exception` is available, but only in error pages. This variable is discussed in Chapter 12 (Controlling the Structure of Generated Servlets: The JSP page Directive) in the sections on the `errorPage` and `isErrorPage` attributes.

- **`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 Chapter 12), 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 (see Chapter 12). 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 the use of the `session` attribute of the `page` directive (Chapter 12) to disable automatic session tracking. In that case, attempts to reference the `session` variable cause errors at the time the JSP page is translated into a servlet. See Chapter 9 for general information on session tracking and the `HttpSession` class.

- **`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 and JSP pages 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. In principle, you can use it to read initialization parameters, but, in practice, initialization parameters are read from `jspInit`, not from `_jspService`.
- **`pageContext`**
  JSP introduced a class called `PageContext` to give a single point of access to many of the page attributes. The `PageContext` class has methods `getRequest`, `getResponse`, `getOut`, `getSession`, and so forth. 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 `request`, `response`, `out`, 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.

11.13 Comparing JSP Expressions, Scriptlets, and Declarations

This section contains several similar examples, each of which generates random integers between 1 and 10. They illustrate the difference in how the three JSP scripting elements are typically used. All the pages use the `randomInt` method defined in Listing 11.13.

Listing 11.13 RanUtilities.java

```
package coreservlets; // Always use packages!!

/** Simple utility to generate random integers. */

public class RanUtilities {

  /** A random int from 1 to range (inclusive). */

  public static int randomInt(int range) {
    return(1 + ((int)(Math.random() * range)));
  }

  /** Test routine. Invoke from the command line with
   *  the desired range. Will print 100 values.
   *  Verify that you see values from 1 to range (inclusive)
   *  and no values outside that interval.
   */

  public static void main(String[] args) {
    int range = 10;
    try {
      range = Integer.parseInt(args[0]);
    } catch(Exception e) { // Array index or number format
      // Do nothing: range already has default value.
    }
    for(int i=0; i<100; i++) {
      System.out.println(randomInt(range));
    }
  }
}
```

Example 1: JSP Expressions

In the first example, the goal is to output a bulleted list of five random integers from 1 to 10. Since the structure of this page is fixed and we use a separate helper class for the `randomInt` method, JSP expressions are all that is needed. Listing 11.14 shows the code; Figure 11–11 shows a typical result.

Listing 11.14 RandomNums.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Random Numbers</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Random Numbers</H1>
<UL>
  <LI><%= coreservlets.RanUtilities.randomInt(10) %>
  <LI><%= coreservlets.RanUtilities.randomInt(10) %>
  <LI><%= coreservlets.RanUtilities.randomInt(10) %>
  <LI><%= coreservlets.RanUtilities.randomInt(10) %>
  <LI><%= coreservlets.RanUtilities.randomInt(10) %>
</UL>
</BODY></HTML>
```

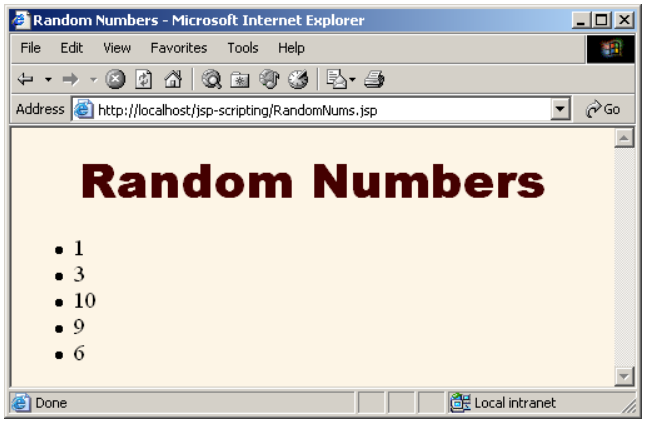

Figure 11–11 Result of RandomNums.jsp. Different values are displayed whenever the page is reloaded.

Example 2: JSP Scriptlets

In the second example, the goal is to generate a list of between 1 and 10 entries (selected at random), each of which is a number between 1 and 10. Because the number of entries in the list is dynamic, a JSP scriptlet is needed. But, should there be a single scriptlet containing a loop that outputs the numbers, or should we use the dangling-brace approach described in Section 11.9 (Using Scriptlets to Make Parts of the JSP Page Conditional)? The choice is not clear here: the first approach yields a more concise result, but the second approach exposes the `<LI>` element to the Web developer, who might want to modify the type of bullet or insert additional formatting elements. So, we present both approaches. Listing 11.15 shows the first approach (a single loop that uses the predefined `out` variable); Listing 11.16 shows the second approach (capturing the "static" HTML into the loop). Figures 11–12 and 11–13 show some typical results.

Listing 11.15 RandomList1.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Random List (Version 1)</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Random List (Version 1)</H1>
<UL>
<%
int numEntries = coreservlets.RanUtilities.randomInt(10);
for(int i=0; i<numEntries; i++) {
  out.println("<LI>" + coreservlets.RanUtilities.randomInt(10));
}
%>
</UL>
</BODY></HTML>
```

Figure 11–12 Result of RandomList1.jsp. Different values (and a different number of list items) are displayed whenever the page is reloaded.

Listing 11.16 RandomList2.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Random List (Version 2)</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Random List (Version 2)</H1>
<UL>
<%
int numEntries = coreservlets.RanUtilities.randomInt(10);
for(int i=0; i<numEntries; i++) {
%>
<LI><%= coreservlets.RanUtilities.randomInt(10) %>
<% } %>
</UL>
</BODY></HTML>
```

Figure 11–13 Result of RandomList2.jsp. Different values (and a different number of list items) are displayed whenever the page is reloaded.

Example 3: JSP Declarations

In this third example, the requirement is to generate a random number on the first request, then show the *same* number to all users until the server is restarted. Instance variables (fields) are the natural way to accomplish this persistence. The reason is that instance variables are initialized only when the object is built and servlets are built once and remain in memory between requests: a new instance is not allocated for each request. JSP expressions and scriptlets deal only with code inside the `_jspService` method, so they are not appropriate here. A JSP declaration is needed instead. Listing 11.17 shows the code; Figure 11–14 shows a typical result.

Listing 11.17 SemiRandomNumber.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Semi-Random Number</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
```

Listing 11.17 SemiRandomNumber.jsp *(continued)*

```
<BODY>
<%!
private int randomNum = coreservlets.RanUtilities.randomInt(10);
%>
<H1>Semi-Random Number:<BR><%= randomNum %></H1>
</BODY>
</HTML>
```

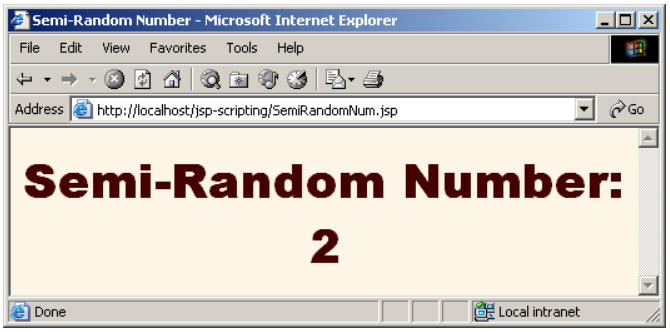

Figure 11–14 Result of SemiRandomNumber.jsp. Until the server is restarted, all clients see the same result.