3 First steps

This chapter covers

- Writing your first JSP page
- Simple dynamic content with JSP
- Basic session management
- Abstracting logic behind JSP pages

Now we're ready to see what JSP looks like. This chapter explores some of JSP's capabilities, giving you a quick tour of its basic functionality. The goal isn't to swamp you with technical details; we'll begin to consider those in the next two chapters, when we introduce the syntax and back-end implementation of JSP. The examples in this chapter should familiarize you with JSP pages' look and feel, which will be helpful when we discuss the syntax more formally.

About the examples

As indicated in chapter 1, a strength of JSP is that it lets you produce dynamic content using a familiar, HTML-like syntax. At the same time, however, the mixture of JSP elements and static text can make it difficult to look at a file and quickly find the JSP elements. To help remedy this problem for the examples in this book that mix JSP elements with static text, we have adopted the convention of marking JSP tags in such examples in **boldface**.

All of the examples presented in this chapter (except for one) are real, usable JSP, and you're encouraged to experiment with them yourself before moving forward. If you do not yet have a JSP environment in which to experiment, appendix B contains a quick installation guide for Tomcat, a free JSP container that provides the reference implementation for the JSP platform.

DEFINITION A *JSP container* is similar to a servlet container, except that it also provides support for JSP pages.

3.1 Simple text

No programming book would be complete without an example that prints "Hello, world!" This simple task serves as an excellent starting point for experimentation. Once you can use a language to print a text string of your choice, you're well on your way to becoming a programmer in that language.

For the web, it makes sense to print "Hello, world!" inside an HTML file. Here's a JSP page that does this:

```
<html>
<body>
<p>
Hello, world!
</p>
</body>
</html>
```

At this point, you're probably thinking, "Wait! That's nothing but a plain HTML file." And you're exactly right; this example is almost disappointingly simple. But it emphasizes an important point about JSP pages: they can contain unchanging text, just as normal HTML files do. Typically, a JSP page contains more than simple static content, but this static—or *template*—text is perfectly valid inside JSP pages.

If a JSP container were to use the JSP page in the example code to respond to an HTTP request, the simple HTML content would be included in the generated HTTP response unchanged. This would be a roundabout way of delivering simple, static HTML to a web browser, but it would certainly work.

Unfortunately, this example didn't show us much about what JSP really looks like. Here's a JSP page that prints the same "Hello, world!" string using slightly more of JSP's syntax:

```
<html>
<hody>
<p>
<%= "Hello, world!" %>
</p>
</body>
</html>
```

This example differs from the previous one because it includes a tag, or element, that has special meaning in JSP. In this case, the tag represents a *scripting element*. Scripting elements are marked off by `<%` and `%>`, and they let you include Java code on the same page as static text. While static text simply gets included in the JSP page's output, scripting elements let Java code decide what gets printed. In this case, the Java code is trivial: it's simply a literal string, and it never changes. Still, the processing of this page differs from that of the prior example: the JSP container notices the scripting element and ends up using our Java code when it responds to a request.

3.2 Dynamic content

If JSP pages could only print unchanging text, they wouldn't be very useful. JSP supports the full range of Java's functionality, however. For instance, although the scripting element in the last example contained only a simple Java string, it might have contained any valid Java expression, as in

```
<%= customer.getAddress() %>
```

or

```
<%= 17 * n %>
```

NOTE As we'll see in chapter 5, JSP is not strictly limited to Java code. The JSP standard provides for the possibility that code from other languages might be included between `<%` and `%>`. However, Java is by far the most common and important case, and we'll stick with it for now.

Let's take a closer look at some examples of JSP pages that produce content dynamically.

3.2.1 Conditional logic

One of the simplest tasks for a Java program, and thus for a JSP page, is to differentiate among potential courses of action. That is, a program can make a decision about what should happen next. You are probably familiar with basic conditional logic in Java, which might look like this:

```
if (Math.random() < 0.5)
    System.out.println("Your virtual coin has landed on heads.");
else
    System.out.println("Your virtual coin has landed on tails.");
```

This Java code, which simulates the flip of a coin, can transfer to a JSP page with only small modifications:

```
<html>
<body>
<p>Your virtual coin has landed on
<% if (Math.random() < 0.5) { %>
heads.
<% } else { %>
tails.
<% } %>
</p>
</body>
</html>
```

This example is similar to the Java code, except that JSP takes care of the output for us automatically. That is, we don't need to call `System.out.println()` manually. Instead, we simply include template text and let the JSP container print it under the right conditions.

So what, exactly, does this latest example do? Up until the first `<%`, the page is very similar to our first example: it contains just static text. This text will be printed for every response. However, the template text is interrupted by the Java code between the `<%` and `%>` markers. In this case, the Java code uses `Math.random()` to generate a pseudorandom number, which it uses to simulate the flip of a coin. If this number is less than 0.5, the block of JSP between the first two { and } braces gets evaluated. This block consists of static text (`heads.`), so this text simply gets included if the conditional check succeeds. Otherwise, the value `tails.` will be printed. Finally, the template text after the final `%>` marker gets included, unconditionally, into the output.

Therefore, this JSP page can result in two different potential outputs. Ignoring white space, one response looks like this:

```
<html>
<body>
<p>Your virtual coin has landed on
heads.
</p>
</body>
</html>
```

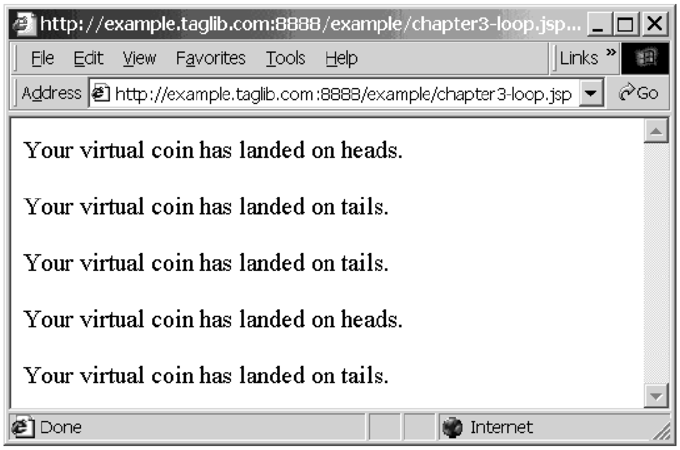
The other potential response is identical, except that the line containing the word "heads" is replaced with one containing "tails." In either case, the browser renders the resulting HTML. Recall from chapter 2 that browsers do not need to know how the HTML was generated; they simply receive a file and process it.

3.2.2 Iteration

Another fundamental task for programs is *iteration*—that is, looping. Like conditional logic, iterative code can be moved into JSP pages as well. Let's take the previous example and turn it into a page that flips five coins instead of one:

```
<html>
<body>
<% for (int i = 0; i < 5; i++) { %>
  <p>
  Your virtual coin has landed on
  <% if (Math.random() < 0.5) { %>
    heads.
  <% } else { %>
    tails.
  <% } %>
  </p>
<% } %>
</body>
</html>
```

How have we modified the example from the conditional logic section (other than by indenting it for clarity)? We've simply added a Java `for()` loop around part of it, embedded between the same `<%` and `%>` markers that we used earlier. As shown in figure 3.1, this loop causes its body to be executed five times.


http://example.taglib.com:8888/example/chapter3-loop.jsp...
File Edit View Favorites Tools Help
Links
Address http://example.taglib.com:8888/example/chapter3-loop.jsp
Go
Done
Internet

Your virtual coin has landed on heads.

Your virtual coin has landed on tails.

Your virtual coin has landed on tails.

Your virtual coin has landed on heads.

Your virtual coin has landed on tails.

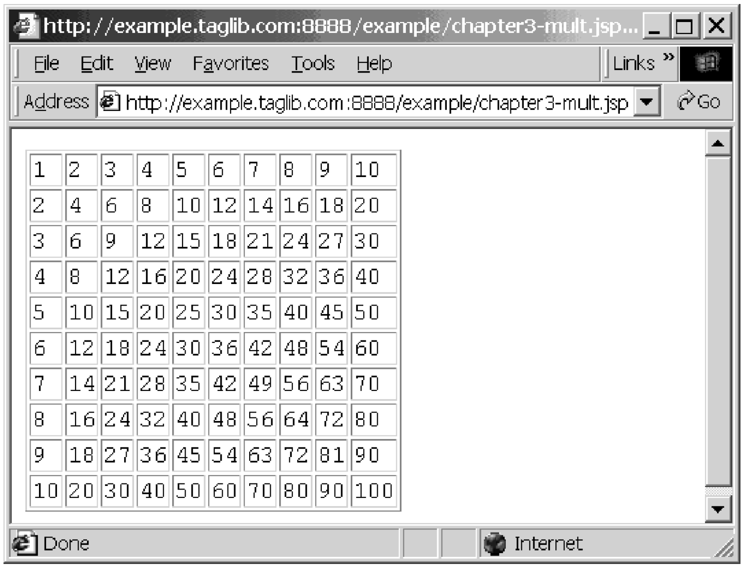
Figure 3.1 A sample run of our iteration example.

To get more of a feel for iteration, let's look at a simple JSP page that prints a traditional multiplication table in HTML (figure 3.2). This table would be tedious to type by hand, even for the most capable mathematicians. JSP turns the problem into a simple programming task that can be solved using two loops, one nested inside the other:

```
<table border="1">
<% for (int row = 1; row < 11; row++) { %>
   <tr>
   <% for (int column = 1; column < 11; column++) { %>
       <td><tt><%= row * column %></tt></td>
   <% } %>
   </tr>
<% } %>
</table>
```

How does this example work? First, we set up an HTML table with the `<table>` element. Then, for each number in the outer loop, we start a new row with the `<tr>` element. Within each row, we create columns for each number in the inner loop using the HTML `<td>` element. We close all elements appropriately and, finally, close the table.

Figure 3.3 shows the HTML source (from our browser's `View Source` command) for the HTTP response sent when the multiplication-table page runs. Note how, as we've emphasized before, the browser plays no part in the generation of this HTML. It does not multiply our numbers, for instance. It simply renders the HTML that the JSP engine generates.


http://example.taglib.com:8888/example/chapter3-mult.jsp...
File Edit View Favorites Tools Help
Links »
Address http://example.taglib.com:8888/example/chapter3-mult.jsp
Go
Done
Internet

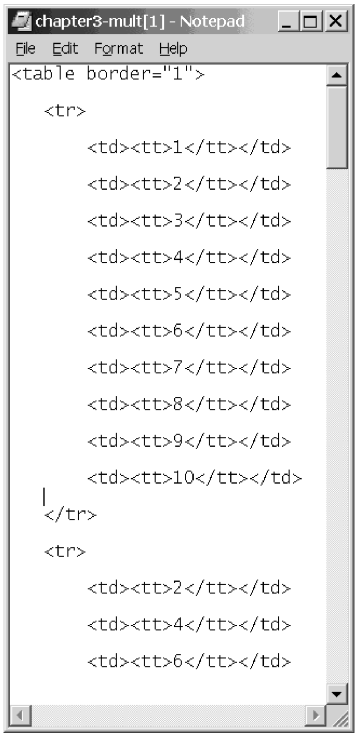
1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Figure 3.2 A multiplication table printed in a web browser

WARNING You might not have expected JSP processing to add some of the white space that appears in figure 3.3. JSP processing preserves the spaces in the source JSP file. For example, the body of the inner loop in our multiple-table example begins by starting a new line, for a line starts immediately after the inner `for()` loop's closing `%>` tag. In the majority of cases, you won't need to worry about the spacing of your output, but on rare occasions, you may need to eliminate extra white space in your source file.

This is the first example we've shown that mixes the simple `<%` marker with the `<%=` marker. As in the ASP environment, the `<%` marker introduces code that will simply be executed. By contrast, `<%=` introduces an expression whose result is converted to a string and printed. In the JSP code in the multiplication table example, the `for()` loops are structural and thus appear in blocks beginning with `<%`. When it comes time to print our `row * column` value, however, we include the Java code inside a block that starts with `<%=`.

NOTE We'll cover the details of these special markup tags—and describe more about iteration and conditional logic—in chapter 5.



```
<table border="1">

   <tr>

       <td><tt>1</tt></td>

       <td><tt>2</tt></td>

       <td><tt>3</tt></td>

       <td><tt>4</tt></td>

       <td><tt>5</tt></td>

       <td><tt>6</tt></td>

       <td><tt>7</tt></td>

       <td><tt>8</tt></td>

       <td><tt>9</tt></td>

       <td><tt>10</tt></td>

   </tr>

   <tr>

       <td><tt>2</tt></td>

       <td><tt>4</tt></td>

       <td><tt>6</tt></td>
```

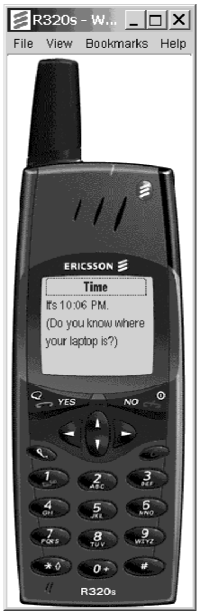
Figure 3.3
Output of the multiplication-table JSP page

3.2.3 *Non-HTML output*

JSP doesn't care about the form of static, template text. To demonstrate that JSP isn't tied to HTML exclusively, here's a simple JSP page that can be used as a time service for cell phones. It outputs WML, a form of XML that's used by some wireless devices:

```
<%@ page contentType="text/vnd.wap.wml;charset=UTF-8"
         import="java.text.*, java.util.*"
%><?xml version="1.0"?>
<%
  SimpleDateFormat df =
    new SimpleDateFormat("hh:mm a");
%>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
 "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
 <card id="time" title="Time">
   <p>It's <%= df.format(new Date()) %>.</p>
   <p>(Do you know where your laptop is?)</p>
 </card>
</wml>
```

Don't worry about the details of WML. JSP doesn't, and WML specifics are beyond the scope of this book. This example just demonstrates an application of JSP beyond the traditional, HTML-based web. Figure 3.4 shows sample output on an emulator for a particular wireless device, the Ericsson R320s.



Time
It's 10:06 PM.
(Do you know where your laptop is?)

Figure 3.4 Output of a WML emulator receiving input from a sample JSP page

NOTE For further details on the generation of non-HTML content, see chapter 15.

3.3 Processing requests and managing sessions

So far, our examples have performed simple tasks that aren't inherently web based. That is, you could write a command-line or Windows program analogous to each of the JSP examples presented so far. Let's move on to JSP pages that are web specific.

In JSP, several Java objects are exposed automatically to scripting code. When you write scripting code, you can refer to these objects without having to declare them by hand. Known as *implicit objects*, these variables—with names such as `request`, `session`, and `response`—give you a simple mechanism to access requests, manage sessions, and configure responses, among other tasks. The next few examples rely on features that the JSP container exposes through implicit objects. They make sense only in environments that are, like the Web, based on the request/response model described in chapter 2.

We'll go into further detail about implicit objects in chapter 6. For now, we introduce them just to demonstrate some more of JSP's functionality.

3.3.1 Accessing request parameters

In chapter 2, we saw how the Java Servlets API gives servlets access to information sent as part of the request. We also saw an example of a servlet that uses this information to greet the user by name. Compared to the servlet in listing 2.1, the JSP code to perform the same task is even simpler. Here's a JSP page that works just like the servlet in the last chapter:

```
<% String name = request.getParameter("name"); %>
<html>
<body>
<p>
<% if (name != null) { %>
```

```
  Hello, <%= name %>.
<% } else { %>
  Welcome, anonymous user.
<% } %>
You're accessing this servlet
from <%= request.getRemoteAddr() %>.
</p>
</body>
</html>
```

This example pulls out two pieces of information from the request: the value of the `name` parameter and the IP address of the machine that sent the request. (The calls work just as they did in listing 2.1.) Notice that we didn't need to declare the `request` variable; the environment has done so for us. The call to `request.getRemoteAddr()` means, "Get the IP address of the current request"; every time the JSP page runs, the value of the `request` object automatically represents the then-current request.

Accessing requests is very common in JSP, for access to requests lets JSP pages retrieve information from users. The `request` object is, by default, an instance of the same `HttpServletRequest` interface that we saw in chapter 2. All of the functionality of `HttpServletRequest` is thus available through the request object. For example, you can access the data entered into an HTML form by calling `request.getParameter()`, just as our example does.

3.3.2 Using sessions

Recall that HTTP is stateless, meaning that a web server starts with a blank slate as it processes each new request it receives. If you need to tie different requests—for example, all requests from the same user—into a *session*, you need either to program this yourself or to use a platform that handles the task for you.

Fortunately, JSP is one such platform. We'll see how JSP actually manages sessions later, in chapters 4 and beyond, but let's take a look now at how sessions might be used. As we mentioned, scripting elements in JSP pages have access to an implicit `session` object. You can store and retrieve session-related data by using methods this object provides.

As an example, imagine that during the processing of a request, you have built up an object called `userData` for a particular user. Suppose you wish to remember this object for subsequent requests that come from the same user. The session object lets you make this association. First, you would write a call like `session.setAttribute("login", userData)` to tie the `userData` object to the session. Then, for the rest of the session, even for different requests, you would be able to call `session.getAttribute("login")` to recover the same `userData` object. The session object keys data under particular names, much as a typical hash table, or an implementation of java.util.Map, does. In this case, the userData object is keyed under the name login.

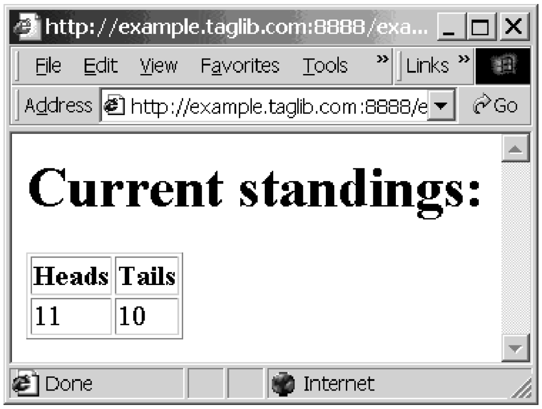
Let's see how sessions work in practice by converting our virtual coin-flip page from before into one that keeps track of how many times "heads" and "tails" have been chosen. Listing 3.1 shows the source code for such a page.

Listing 3.1 A small application that uses sessions

```
<%
   // determine the winner
   String winner;
   if (Math.random() < 0.50)
     winner = "heads";
   else
     winner = "tails";

   synchronized (session) {
     // initialize the session if appropriate
     if (session.isNew()) {
       session.setAttribute("heads", new Integer(0));
       session.setAttribute("tails", new Integer(0));
     }
     // increment the winner
     int oldValue =
       ((Integer) session.getAttribute(winner)).intValue();
     session.setAttribute(winner, new Integer(oldValue + 1));
   }
%>
<html>
<body>
<h1>Current standings:</h1>
<table border="1">
<tr>
  <th>Heads</th>
  <th>Tails</th>
</tr>
<tr>
  <td><%= session.getAttribute("heads") %></td>
  <td><%= session.getAttribute("tails") %></td>
</tr>
</table>
</body>
</html>
```

At its heart, this page is similar to the one from before that emulates a coin flip. However, the page contains extra logic to keep a tally of prior coin flips in the `session` object. Without getting too caught up in the details, the page initializes the session if it's new—that is, if `session.isNew()` returns `true`—and then it keeps track of the tally for "heads" and "tails," keying the data, imaginatively enough, under the names `heads` and `tails`. Every time you reload the page, it updates the tallies and displays them for you in an HTML table (figure 3.5). If you reload the page, the tallies change. If your friend, however, begins accessing the application from a different computer, the `session` object for your friend's requests would refer to a new session, not yours. When different users access the page, they will all receive their own, individual tallies of heads and tails. Behind the scenes, the JSP container makes sure to differentiate among the various users' sessions.


http://example.taglib.com:8888/exa...
File Edit View Favorites Tools Links
Address http://example.taglib.com:8888/e
Go
Done
Internet

Current standings:

Heads	Tails
11	10

Figure 3.5 A stateful tally of prior events, made possible by session management.

3.4 Separating logic from presentation

In the examples so far, Java code has been mixed right in with HTML and other static text. A single JSP file might contain some HTML, then some Java code, and finally some more HTML. While this mixture is a convenient way to generate dynamic content, it might be difficult for a large software-development team to maintain. For instance, programmers and HTML designers would need to manage the same combined JSP files. If problems are encountered, they might not be immediately clear whether they come from HTML problems or logic errors.

To help address these issues and provide for greater maintainability, JSP provides another mechanism for generating on-the-fly content. In addition to the simple scripting elements we've shown, JSP allows special, XML-based tags called *actions* to abstract Java code away from the JSP page itself.

Many actions look just like HTML tags, but they work like a signal to the JSP container to indicate that some processing needs to occur. When processing of a JSP page hits a block of static HTML text, like `<p>Hello!</p>`, such text is simply passed through to the JSP page's output. Processing for actions is different: when the JSP page hits an action, such as `<jsp:include>`, it runs extra code to figure out how processing should proceed. Unlike the Java between scripting elements `<%` and `%>` tags, however, the code for actions does not appear directly on the JSP page. Instead, it can either be built into the container or provided as a custom add-on by developers.

We'll cover actions in more depth in chapters 4 and 6. For now, let's take a look at how these tags might help you manage your JSP applications.

3.4.1 Reusing logic with JavaBeans

One common use of the special XML-based tags we've mentioned is to communicate with JavaBeans. In fact, JSP provides several standard action tags to help you communicate with these beans. As we discussed in chapter 1, JavaBeans are reusable Java components: they are Java classes that follow conventions, defined in the JavaBeans standard, that promote modularity and reusability. The details of this standard, as it relates to JSP pages, will be covered in chapter 8. For now, let's look at a simple JavaBean class so that we can present a JSP page that uses it:

```
package com.taglib.wdjsp.firststeps;
public class HelloBean implements java.io.Serializable {
  String name = "world";
  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

Indeed, this is a very simple Java class. It contains a single instance variable, `name`, which refers to a string. By default, this string has the value `world`, but it can be changed using the method `setName()`, which takes an instance of the Java `String` class as its parameter. Code outside the bean can retrieve the name by using `getName()`. These methods have names that the JavaBeans framework will look for, by default, when it needs to modify or retrieve the `name` variable, which in bean terms is called a *property*.

A JSP page may use this bean as follows:

```
<html>
<body>
<p>
<jsp:useBean id="hello"
  class="com.taglib.wdjsp.firststeps.HelloBean"/>
```

```
<jsp:setProperty name="hello" property="name"/>
Hello, <jsp:getProperty name="hello" property="name"/>!
</p>
</body>
</html>
```

The first action tag that appears is the `<jsp:useBean>` tag. As its name suggests, this tag lets the JSP page begin using a bean, specified by a particular class name and page-specific ID. In this case, we have indicated that we wish to use an instance of the `HelloBean` class and, for the purposes of the page, to call it `hello`. The appearance of the `<jsp:setProperty>` tag in the code causes the request parameter called `name`—if it exists and isn't an empty string—to be passed as the `String` parameter in a call to the bean's `setName()` method. We could have written

```
<% if (request.getParameter("name") != null
       && !request.getParameter("name").equals(""))
     hello.setName(request.getParameter("name"));
%>
```

and it would have had a similar effect, but `<jsp:setProperty>` is both easier to use and provides us with a level of abstraction. If we needed to set multiple properties in the `HelloBean`, `<jsp:setProperty>` would make our page substantially easier to read and less prone to errors.

The final action tag that appears in the example is `<jsp:getProperty>`, which retrieves the `name` property from the `HelloBean` and includes it in the JSP page's output. Therefore, the example prints a personalized greeting if it can retrieve the user's name from the request; if not, it simply prints `Hello, world!`, just like our first example.

The bean-centered approach gives our page several advantages in readability and maintainability. As we just mentioned, the tags beginning with `<jsp:` take care of various operations for us behind the scenes. This way, we don't have to write Java code that manually sets and retrieves information out of the bean.

Suppose that `HelloBean` were a little more complex. Instead of a bean that simply stores a name, imagine one that capitalizes names correctly or that uses the name as part of a database query that retrieves more information about the user. Even if `HelloBean` performed these extra tasks, its interface with the JSP page would be the same: `<jsp:getProperty>` and `<jsp:setProperty>` would still work just as they do in the example we just saw. If multiple pages in your application—or even multiple applications—need to use the logic contained inside the bean, they can all simply use different copies of the bean—or even the same copy—via the `<jsp:useBean>` tag. Beans therefore let you move more of your own Java code outside the JSP page itself, and they let you reuse this code among multiple pages. By