Chapter 19

JSP: Servlets Turned Inside Out

In our last chapter, the BudgetPro servlet example spent a lot of code generating the HTML output for the servlet to send back to the browser. If you want to change the HTML for any page (for example, add a background color), you would have to modify the Java code (obviously)—but you're not really wanting to modify the logic of the servlet, you only want to tweak its output. The HTML that a servlet generates can be scattered among output statements, string concatenations, classes, and method calls. Servlets, we might say, bury the HTML deep inside the code. We're now going to take a look at JavaServer Pages (JSP) which do the opposite—they expose the HTML and hide the code down inside.

This technique has been given the fancy description, *document-centric server-side programs*. They are "document-centric" because the HTML code is so visible—JSP content looks like (and is) HTML with some additions. They are "server-side" because all the work is done on the server and all the additions and special features of JSP are boiled down to a simple stream of HTML by the time it gets to the browser.

19.1 WHAT YOU WILL LEARN

- Theory of operation: how JSP can be thought of as servlets "inside out."
- Three simple JSP directives: `scriptlet`, `declaration`, `expression`.
- Servlet variables made available: `request`, `response`, `out`, `session`.
- Server-side includes.
- A tiny bit about tags.
- `jsp:useBean`.
- A look at our BudgetPro using JSP.
- The correct spelling of JavaServer Pages.

19.2 SERVLETS TURNED INSIDE OUT: JSP

Take a look at the `AccountView.java` class in the BudgetPro servlet example. It consists almost entirely of

```
sb.append("</a></td>");
```

method calls which build up a string of HTML. Instead, this could have been calls to do the output right then and there:

```
out.println("</a></td>");
```

Either way, if we want to modify the HTML, we have to modify the Java code. While that's not difficult, it can be error-prone. It would be nice to *not* have the Java syntax in the way when we want to modify the HTML. (That's especially true when you want to put quotation marks in your HTML:

```
out.println("<input name=\"name\" size=\"20\">");
```

It's not that it can't be done; the `\"` just gets hard to read and hard to get right the first time.)

One way to externalize all the HTML is to put it into a file. Then our Java application could read the file at runtime and send its contents to the browser. Not bad, but what about the dynamic parts? Remember how we generated the table from the `for` loop in `AccountView.java`:

```
for (Iterator actit = acct.getAllSubs(); actit.hasNext(); ) {
    Account suba = (Account) actit.next();
    sb.append("<tr>");
    sb.append("<td><a href=\"BudgetPro?name="+suba.getName());
    sb.append("&func=cd\">");
    sb.append(suba.getName());
    sb.append("</a></td>");
    sb.append("<td>albing</td>");
    sb.append("<td>");
    sb.append(suba.getTotal().toString());
    sb.append("</td>");
    sb.append("</tr>\n");
} // next acct
```

That would be hard to do with file-based HTML.

Another approach, the one used by JavaServer Pages, would be to use the HTML file as input to a converter program—one which would take each line of HTML, for example

```
<input name="name" size="20">
```

and produce a line of Java code:

```
out.println("<input name=\"name\" size=\"20\">");
```

Notice how the converter would be the one to handle the escape sequence for the quotation marks; we get to write straight HTML—it has to deal with the backslashes.

This is the basic idea behind JavaServer Pages. JSP files are nothing more than HTML (with some additions that we'll discuss shortly) which are compiled into Java programs—servlets, to be exact—that are then run to produce the Web page. The conversion happens no later than the first time the Web server tries to serve up that JSP. If it hasn't yet been converted, it will convert it into Java code and start the servlet. Thereafter, other requests to that page go directly to the servlet. If you modify the JSP file, then the Web server recognizes that the file has been modified and reconverts it.

But why go to all this trouble? It's not for the static HTML that we need JSP, but rather for the dynamic bits. Remember that `for` loop, above, used to make the HTML table of subaccounts? Let's look at part of a JSP that does the same thing:

```
<table border=1 width=50%>
<tr>
<th>Account</th>
<th>Owner</th>
<th>Value</th>
</tr>
<% // for each subaccount:
  for (Iterator actit = acct.getAllSubs(); actit.hasNext(); ) {
    Account suba = (Account) actit.next();
%>
    <tr>
    <td><a href="BPControl?name=<%= suba.getName() %>&func=cd">
    <%= suba.getName() %>
    </a></td>
    <td>albing</td>
    <td>
    <%= suba.getTotal().toString() %>
    </td>
    </tr>
<%
 } // next acct
%>
</table>
```

Notice how it starts off as simply the HTML building the table opening. Then we encounter some Java source code, enclosed in delimiters (`<% ... %>`), then back to plain HTML. There's even a line which intermixes HTML and Java:

```
<td><a href="BPControl?name=<%= suba.getName() %>&func=cd">
```

To understand what's going on here, let's take a look at four pieces of syntax that are the keys to JSP.

19.3 How to Write a JSP Application

Writing a JSP application consists, syntax-wise, of writing your desired output page in HTML and, where you need the dynamic bits, putting Java code and/or other special syntax inside special delimiters that begin with `<%`.

There are four special delimiters that we should describe if you're going to work with JSP. The bulk of your JSP will likely be HTML. But interspersed among the HTML will be Java source or JSP directives, inside of these four kinds of delimiters:

- `<% code %>`
- `<%= expression %>`
- `<%! code %>`
- `<%@ directive %>`

Let's look at them one at a time.

19.3.1 Scriptlet

The code that appears between the `<%` and `%>` delimiters is called a *scriptlet*. By the way, we really hate the term "scriptlet." It seems to imply (falsely) a completeness that isn't there. It is too parallel to the term "applet," which is a complete Java program that runs inside a browser. A scriptlet isn't necessarily a complete anything. It's a snippet of code that gets dropped inside the code of the servlet generated from the JSP source.

Recall that servlets may have a `doPost()` and a `doGet()` methods, which we collapsed in our example by having them both call the `doBoth()` method. Same sort of thing is happening here with the JSP, and the `doBoth()` ends up doing all the output of the HTML. Any snippets of Java code from within the `<%` and `%>` delimiters get dropped right in place between those output calls, becoming just a part of a method.

It can be useful to keep this in mind when writing JSP. It helps you answer the questions of scope—who has access to what, where are variables getting declared and how long will they be around? (Can you answer that last question? Since any variable declared inside the `<%` and `%>` will be in the JSP equivalent of our `doBoth()` method, then that variable will only be around for the duration of that one call to the `doBoth()`, which is the result of one `GET` (or `POST`) from the browser.)

The source code snippets can be just pieces of Java, so long as it makes a complete construct when all is converted. For example, we can write:

```
<% if (acct != null) { // acct.getParent() %>
    <a href="BudgetPro?func=back">
    <img src="/back.gif">
    </a>
<% } else { %>
    <img src="/back.gif">
<% } %>
```

Notice how the `if-else` construct is broken up into three separate scriptlets—that is, snippets of code. Between them, in the body of the `if` and the `else`, is plain HTML. Here is what that will get translated into after the JSP conversion:

```
if (acct != null) { // acct.getParent()
    out.println("<a href=\"BudgetPro?func=back\">");
    out.println("<img src=\"/back.gif\">");
    out.println("</a>");
} else {
    out.println("<img src=\"/back.gif\">");
}
```

Do you also see why we describe it as being "turned inside out"? What was delimited becomes undelimited; what was straight HTML becomes delimited strings in output statements.

As long as we're on the topic of conversion, let's consider comments. There are two ways to write comments in JSP, either in HTML or in Java. In HTML we would write:

```
<!-- HTML comment format -->
```

but since we can put Java inside of delimiters, we can use Java comments, too:

```
<% // Java comment format %>
```

or even:

```
<% /*
    * Larger comments, too.
    */
%>
```

If you've been following what we've been saying about translation of JSP into Java code, you may have figured out the difference. The Java comments, when compiled, will be removed, as all comments are, from the final executable.

The HTML-based comments, however, will be part of the final HTML output. This means that you'll see the HTML comments in the HTML that reaches the browser. (Use the **View Source** command in your browser to see them. As HTML comments, they aren't displayed on the page, but they are sent to the browser.) This is especially something to be aware of when writing a loop. Remember our loop for generating the table?

```
<% // for each subaccount:
  for (Iterator actit = acct.getAllSubs(); actit.hasNext(); ) {
    Account suba = (Account) actit.next();
 %>
    <!-- Next Row -->
    <tr>
    <td><a href="BPControl?name=<%= suba.getName() %>&func=cd">
...
<% } // next acct %>
```

We've put a comment just prior to the `<tr>` tag. What will happen is that the comment will be part of the generated HTML, and since this is a loop, the comment, just like the `<tr>` and other tags, will be repeated for each iteration of the loop. Now we're not saying this is undesirable—in fact it makes the resultant HTML more readable. But be aware that these comments will be visible to the end user. Be careful in what you say in them. The additional transmission time required for these few extra bytes is probably imperceptible, unless your comments are large and repeated many times.

19.3.2 Declaration

The other place that code can be placed is outside the `doGet()` and `doPost()`. It is still inside the class definition for the servlet class that gets generated from the JSP, but it is not inside any method. Such code is delimited like this:

```
<%! code %>
```

The exclamation mark makes all the difference. Since it's outside any method, such code typically includes things like variable declarations and complete method declarations. For example:

```
<%! public static MyType varbl;

public long
countEm()
{
    long retval = 0L;
    retval *= varbl.toLong();
    return retval;
}
%>
```

If you tried to do something like this inside of a scriptlet, you would get errors when the server tries to compile your JSP. Such syntax belongs at the outer lexical level. The use of the `<%! ... %>` syntax puts it there.

19.3.3 Expression

This delimiter is a shorthand for getting output from a very small bit of Java into the output stream. It's not a complete Java statement, only an expression that evaluates into a `String`. Here's an example:

```
<h4>As of <%= new java.util.Date() %></h4>
```

which will create a Java `Date` object (initialized, by default, with the current date/time) and then call the `toString()` method on that object. This yields a date/time stamp as part of an `<h4>` heading.

Any methods and variables defined inside the previously described delimiters are OK to use with this expression shorthand.

There are also a few predefined servlet variables.

We've described how the JSP is converted into a servlet—the HTML statements become `println()` calls. This all happens inside of an `HttpServlet`-like class, just like our `BudgetProServlet` extends `HttpServlet` in the previous chapter. In such a class, the method called when a request arrives from a browser looks very much like our `doBoth()` method:

```
doBoth(HttpServletRequest request, HttpServletResponse response)
```

> **TIP**
>
> If you want to see the source for the servlet that gets generated when a JSP is converted, and if you're using NetBeans, right-click on the filename (in the **Explorer** view) and, from this menu, choose **Compile**. Then do it again and you'll notice that the second choice on the menu is **View Servlet** (Figure 19.1).
>
> If you are using Apache Tomcat as your Web server, just look in the `work` subdirectory in the directory where Tomcat is installed. In the appropriate subdirectory you will find both the `.java` and `.class` files for your converted JSP with the `.jsp` suffix converted to `$jsp.java` and `$jsp.class` respectively. For example, `BPAcct.jsp` becomes `BPAcct$jsp.java` and is compiled into `BPAcct$jsp.class`.

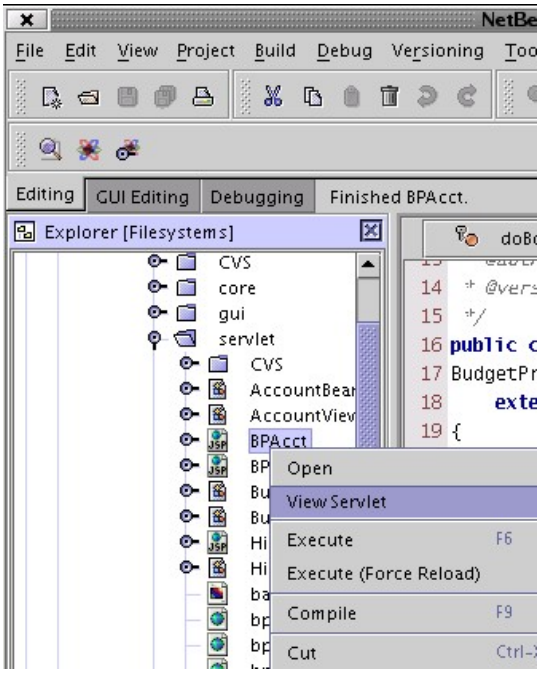

Figure 19.1 Viewing the converted JSP in NetBeans

The point here is that a request object and a response object are defined by the way the servlet is generated. They are called, oddly enough, `request` and `response`. In addition to these, a `session` is defined and initialized, just like we did in our servlet example. (What were we thinking?)

There are a few other variables that the converted servlet has created that we can use. We'll summarize them in Table 19.1. To read more about how to use them, look up the Javadoc page for their class definition.

Remember that these can be used not only by the `<%= %>` expressions, but also by the `<% %>` snippets of code.

19.3.4 Directive

The last of the special delimiters that we will discuss is the one that doesn't directly involve Java code. The `<%@ ... %>` delimiter encompasses a wide variety of directives to the JSP converter. We don't have the space or the patience to cover them all, so we'll cover the few that you are most likely to need early on

Table 19.1 JSP predefined variables

Type	Variable name
`PageContext`	`pageContext`
`HttpSession`	`session`
`ServletContext`	`application`
`ServletConfig`	`config`
`JspWriter`	`out`

in your use of JSP. We have some good JSP references at the end of this chapter for those who want all the gory details of this feature.

```
<%@page import="package.name.*" %>
```

is the way to provide Java `import` statements for your JSP. We bet you can guess what that happens in the generated servlet.

Here's another useful page directive:

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

You'll see this as the opening line of our JSP, to identify the output MIME type for our servlet.

JSP also has an `include` directive:

```
<%@include file="relative path" %>
```

This directive is, for some applications, worth the price of admission alone. That is, it is such a useful feature that even if they use nothing else, they could use JSP just for this feature. It will include the named file when converting the JSP—that is, at compile time.

It can be used for common header and footer files for a family of Web pages. (If you're a C programmer, think `#include`.) By defining one header file and then using this directive in each JSP, you could give all your JSP the same look—say, a corporate logo and title at the top of page and a standard copyright statement and hyperlink to your webmaster's e-mail address at the bottom.

Be aware that this inclusion happens at compile time and is a source-level inclusion. That is, you are inserting additional source into the JSP, so if your

included file contains snippets of Java code, they will be part of the resulting program. For example, you could define a variable in the included file and reference in the including file.

Also, since this inclusion happens at compile time, if you later change the included file, the change will not become visible until the JSP files that do the including are recompiled. (On Linux, this is simply a matter of **touch**ing all the JSP, as in:

```
$ touch *.jsp
```

assuming all your JSP files are in that directory. Touching them updates their time of last modification, so the Web server thinks they've been modified so the next access will cause them to be reconverted and their generated servlets reloaded. You get the idea.

There is another way to do an include in JSP—one that happens not at compile time, but at runtime. The syntax is different than the directives we've seen so far, but more on that in minute. First, an example of this kind of include:

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

In this format, the page specified by the URL (relative to this Web application's root) is visited and its output is included in place amongst the output of this JSP, the one doing the include.

A few quick notes:

- Be sure to include the ending "`/`" in the directive; it's part of the XML syntax which is a shorthand for ending the element in the same tag as you begin—that is, `<p />` instead of `<p> </p>`.
- When all is working, `flush` being `true` or `false` doesn't matter; when the included page has an error, then `flush="true"` causes the output to the browser to end at the point of the include; with `flush="false"`, the rest of the page will come out despite the error in the include.
- The page that is being included is turned into its own servlet. That is, it is its own JSP. You don't have to just include static HTML, you can include a JSP.
- Since this is a runtime include, all you are including is the output of that other page. You can't, with this mechanism, include Java snippets or declarations, but only HTML output.

Table 19.2 New XML syntax for JSP constructs

Standard format	New HTML format
`<% code %>`	`<jsp:scriptlet> code </jsp:scriptlet>`
`<%! code %>`	`<jsp:declaration> code </jsp:declaration>`
`<%= expr %>`	`<jsp:expression> expr </jsp:expression>`

19.3.5 New Syntax

But what about that new syntax? It's an XML-conformant syntax, and it's the syntax for all the newer features added to JSP. In fact, even the old JSP syntax, the statements that we've discussed, have an alternative new syntax (Table 19.2). Prior to JSP 2.0, that syntax was reserved for JSP that produce XML rather than HTML. (That's a whole other can of worms that we won't open now.) Now, as of JSP 2.0, both forms can be used, if your Web server is JSP 2.0 compliant.

You can see that the old syntax is more compact and less distracting than the large tags. We suspect that means the old syntax is likely to continue to be used for a long time yet.[1]

This new syntax is also used for the last two parts of JSP that we will cover, `useBean` and tag libraries.

19.3.6 JavaBeans in JSP

For those who really want to avoid doing any Java coding inside of a JSP, there is additional syntax that will provide for a lot of capability but without having to explicitly write any Java statements. Instead, you write a lot of arcane JSP directives, as we'll show you in just a bit. Is this any better? In some ways yes, but in other ways, no, it's just different syntax.

What we'll be able to do with this additional syntax is:

1. Instantiate a Java class and specify how long it should be kept around
2. Get values from this class
3. Set values in this class

1. The newer XML-style syntax would be useful if your JSP are generated by an XSLT stylesheet or are validated against a DTD, both topics being beyond the scope of our discussion.

The syntax looks like this:

```
<jsp:useBean id="myvar" class="net.multitool.servlet.AccountBean" />
```

which will create a variable called `myvar` as an instance of the `AccountBean` class found in the `net.multitool.servlet` package. Think of this as:

```
<%! import net.multitool.servlet.AccountBean; %>

<% AccountBean myval = new AccountBean(); %>
```

So can `AccountBean` be any class? Well, sort of. It can be any class that you want, as long as it is a bean. It doesn't have to end in "`Bean`", but it does have to be a class which has:

- A null constructor (you may have noticed there is no syntax to support arguments to the constructor on the `useBean` statement).
- No public instance variables.
- Getters and setters for instance variables.
- Getters and setters named according to a standard: `getTotal()` or `isTotal()` and `setTotal()` for a variable called `total` (`isTotal()` would be used if we had a boolean getter, that is, if the getter returned a `boolean`; otherwise it would expect `getTotal()` as the getter's name).

Otherwise, its a normal class. These restrictions mean that you can call the class a "JavaBean" or just "bean," and there is additional JSP syntax to manipulate the class. Specifically:

```
<jsp:getProperty name="myvar" property="total" />
```

will do, in effect, the following:

```
<%= myvar.getTotal() %>
```

or

```
<% out.print(myvar.getTotal()); %>
```

Similarly, we can set a value in the JSP with this syntax:

```
<jsp:setProperty name="myvar" property="total" value="1234" />
```

which will do, in effect, the following:

```
<% myvar.setTotal("1234"); %>
```

So this would hardly seem worth it, but there are other syntax constructs that make this much more powerful. Remember that we're working with Web-based stuff, with a JSP that will be invoked via a URL. That URL may have parameters on it, and we can map those parameters onto a bean's properties—that is, connect the parameters to setters for a given bean. We replace the `value` attribute with a `parameter` attribute, for example:

```
<jsp:setProperty name="myvar" property="total" parameter="newtot" />
```

which works the same as:

```
<% myvar.setTotal ( request.getParameter("newtot") ); %>
```

We can take that one step further and map all the parameters that arrive in the URL to setters in one step:

```
<jsp:setProperty name="myvar" parameter="*" />
```

So if you design your JSP and your HTML well, you can get a lot done automatically for you. One other thing going on behind the scenes that we've glossed over is the type of the argument to the setter. The parameters all come in as `String`s. However, if your setter's type is a Java primitive, it will automatically convert to that type for you, instead of just passing you `String`s.

One final twist on using beans is the duration of the bean and its values. If you don't specify otherwise (and we have yet to show you syntax to do otherwise) your bean will be around for the duration of the request, at which time it will be available to be garbage-collected. Any values in the bean will not be there on the next visit to that URL (i.e., the next call to that servlet).

Here is the syntax to make that bean last longer:

```
<jsp:useBean id="myvar" class="net.multitool.servlet.AccountBean"
             scope="session" />
```

which will make it stay for the duration of the session. You may remember (or you can flip back and look up) how we created and used session variables in the servlet. The same mechanism is at work here, but behind the scenes. You

only use the specific syntax in the `useBean` tag, and it does the rest (getting and storing) for you.

19.3.7 Tag Libraries

Well, we're almost done with JSP, but the one topic that we have yet to cover is huge. It's the trap door, or the way back out, through which JSP can get to lots of other code without the JSP author having to write it. Tag libraries are specially packaged libraries of Java code that can be invoked from within the JSP. Just like the `useBean`, they can do a lot of work behind the scenes and just return the results.

There are lots of available libraries, which is one reason for this topic to be so huge. We could spend chapters just describing all the various database access routines, HTML generating routines, and so on available to you. Perhaps the leading tag library is the JSP Standard Tag Library (JSTL).

Here are two of the most common directives used with tag libraries. First is the directive that declares a library to be used:

```
<%@ taglib prefix="my" uri="http://java.sun.com/jstl/core" %>
```

You then use the prefix as part of the tag name on subsequent tags that refer to this library. For example, if we had an `out` directive in our library, we could use `my` as the prefix, separated by a colon: `<my:out ...>`.

The second directive we will show is a `for` loop. The `for` loop mechanism provided by this library is in some ways simpler than using Java scriptlets. It comes in many forms, including one for explicit numeric values:

```
<my:forEach var="i" begin="0" end="10" step="2">
```

This example will loop six times with `i` taking on the values `0`, then `2`, then `4`, and so on. Another variation of the `forEach` loop can also make it easy to set up the looping values:

```
<my:forEach var="stripe" items="red,white,blue">
```

In this example it will parse the `items` string into three values: `red`, `white`, and `blue`, assigning each, in turn, to the variable `stripe`. In fact the `items` attribute can also store an array, or collection, or iterator from the Java code that you may have declared (or that is implicit from the underlying

servlet). The `forEach` will iterate over those values without you having to code the explicit `next()` calls or index your way through an array.

The bottom of the loop is delimited by the closing tag:

```
</my:forEach>
```

For more information on these and other tags, check out

- `http://java.sun.com/products/jsp/jstl`
- `http://jakarta.apache.org/products/jsp/jstl`
- The references at the end of this chapter

Beyond the standard library of tags, there are other third-party collections of tags; you can also create your own libraries, called *custom tag libraries*. While a useful and powerful thing to do if you have a large JSP-based application, such details would expand this book well beyond its scope. If you're interested in this topic, please follow up with some of the excellent references at the end of this chapter.

19.4 USING JSP WITH BUDGETPRO

We could have taken the BudgetPro example from the previous chapter and simply translated it all into JSP files. The reason we didn't is that it's not how you are likely to find JSP used "in the wild." Since JSP files become servlets, it is not uncommon to find JSP and servlets mixed together—not arbitrarily, but in a sensible way. Remember the Model/View/Controller (MVC) pattern from your readings on object-oriented programming and design patterns?[2] Well, JSP makes for a reasonable View, and a plain servlet can act as the Controller. The Model is typically the database behind all this. That's what we've done with the BudgetPro example.

We've taken the two main chunks of output code—that for the main account display and the form used for creating subaccounts—and turned those

2. If not, then a) you should do some more reading, and b) the MVC pattern is a "classic" way to divide the work of a GUI into three distinct parts: Model—the data behind what you are doing or displaying; View—a particular way to display that data; and Controller—an object that acts as the "traffic cop" to various inputs and events, sending messages to either the View, or Model, or both.

into JSP files. The main servlet class (`BudgetProServlet.java`) is thus "gutted" of its output, and the new version (`BudgetProController.java`) acts as the controller. Requests come to it via HTTP requests, but for output, it redirects the browser making that request over to the appropriate JSP.

This introduces a new bit of servlet syntax—redirecting a request to another URL. The action is taken by means of a method call on the HTTP response object:

```
response.sendRedirect("BPAcct");
```

Whereas in the previous, servlet version of BudgetPro, we would create an object that was the next page of output:

```
nextPage = new AccountView(current);
```

In this version, we instead redirect the response to a JSP that produces the output for that page.

So how does the JSP know for which account it should display information? That is shared between the JSP and the controller servlet via the session information. As with the previous, servlet-base BudgetPro, the session is used to store the current account. It can be retrieved from the session information, as seen in line 11 of `BPAcct.jsp`:

```
11:   <% Account acct = (Account) session.getAttribute("current");
```

That variable (`acct`) is then used throughout the JSP to get the appropriate data for display, as in:

```
21:   Account: <%= acct.getName() %>
```

We could also have used a session JavaBean. Such a mechanism requires more setup on both sides, the controller and the JSP, but has the advantage of removing more literal Java code from the JSP. ("We leave this as an exercise for the reader!")

19.5 Review

We've looked at server-side Java processing with JavaServer Pages which can be thought of as servlets turned inside out. From that simple concept we looked

at our servlet example and converted it to use JSP. We also looked briefly at the syntax for JSP tags and the JSTL, but encouraged you to do more reading on this topic.

19.6 WHAT YOU STILL DON'T KNOW

We didn't yet discuss the spelling of JavaServer Pages. If you've read through this chapter, you may have noticed that there is no space between Java and Server but there is a space between Server and Pages. If you've read this chapter, you may also have some idea of why it's spelled this way: It's the JavaServer that's doing the work—serving up the Pages. OK, it's not a huge deal, but it is worth knowing how to spell something kerectly, rite?

There are volumes that we could have written about tag libraries. Large scale projects, and any project with a database connection behind it, will find tag libraries invaluable at providing standard mechanisms for database access. Check out the resources, below, for more information on tag libraries.

19.7 RESOURCES

Some of the best material on JavaServer Pages comes from two of the books we mentioned in the previous chapter. You now understand how interrelated the two topics of servlets and JSP are, and these two books cover both topics very well:

- *Core Servlets and JavaServer Pages* by Marty Hall and Larry Brown, ISBN 0-13-009229-0, a Prentice Hall PTR book.
- Its sequel, *More Servlets and JavaServer Pages* by Marty Hall, ISBN 0-13-067614-1, also by Prentice Hall PTR.

As we said, the topic of tag libraries is huge, and just writing about JSTL could fill it's own volume. It has. We recommend:

- *Core JSTL: Mastering the JSP Standard Tag Library* by David Geary, ISBN 0-13-100153-1, Sun Microsystems Press.

To get it straight from the horse's mouth, there is the official Sun specifications for JSP, available at

- `http://java.sun.com/products/jsp/ download.html#specs`

19.8 EXERCISES

1. Convert the controller and the JSP to share their data via JavaBeans.
2. Add a control button to each page (`BPAcct.jsp`) to return not just one level upwards, but back to the top level account. (Hint: The controller can store a reference to the top level account in a session variable named `top`.)