CHAPTER 4

The Standard Tag Library

Chapter 3 explained how to get values from beans to pages with the `jsp:getProperty` tag, along with a number of limitations in this process. There was no good way to display the tracks on a CD, because the page has no way to know how many tracks a bean will be holding. The quiz was unable to determine whether the user's answer was correct, because the page has no way to compare two values in a bean.

Both of these problems can be solved by a new set of tags: the standard tag library. Although these tags are not technically a portion of the JSP specification, they are closely related and can be used in any application server that supports JSPs. This chapter looks at what these tags can do, after a few words on how tags in JavaServer Pages work in general.

4.1 Tag Libraries

We have already seen tags that deal with things ranging from including other JSPs to manipulating beans. These tags are all useful and perform their specific tasks well, but almost from the beginning, the authors of the JSP specification realized that no set of tags could possibly do everything that everyone would need from JSPs. To address that issue, those authors provided a mechanism for programmers to create new tags that could do anything possible and an easy way for pages to use these custom tags. The topic of the creation of new tags is covered in Chapter 13. Listing 4.1 illustrates how a page loads and uses a tag.

Listing 4.1 A JSP that uses a custom tag

```
<%@ taglib prefix="awl"
    uri="http://jspbook.awl.com/samples" %>
The time, in two different formats:<p>
<awl:date format="EEEE, MMMM dd yyyy 'at' hh:mm"/><br>
<awl:date format="hh:mm:ss MM/dd/yy"/><br>
```

The tag library is loaded with the first line. The URI (Uniform Resource Identifier) specifies the location of the tag library definition, and the prefix specifies the name that will be used to access the tags. Here, the prefix is `awl`, but it could be anything, as long as it is used consistently. One of the tags from this library, `time`, is used twice in the last two lines. The name of the tag is prepended by the prefix specified at the top.[1]

The `awl:time` tag itself simply sends the current time to the page, in a format specified by the `format` property. If this looks familiar, it is because this does essentially the same thing as Listing 3.2. That example used a bean with an input for the format and an output for the time. Using a custom tag, the input is specified as a named property, and the output is implicit in the way the tag works.

Technically, neither example was particularly good. Because they play the part of models in the model/view/controller paradigm, beans should not be concerned with how their data will be presented. Hence, the bean used in Listing 3.2 should not have had to deal with formatting issues. Similarly, tags are intrinsically part of the view portion and so should not deal directly with data, but the `awl:time` tag in Listing 4.1 holds data in the form of the current time. With some effort, the standard tag library can help make such separations of roles between tags and beans easier to manage, as will be seen later in this chapter.

4.2 Tags with Bodies

Custom tags can do more than output data controlled by parameters. A custom tag can have a body, which it can control in arbitrary ways. Recall a similar tag, `jsp:useBean`, which renders its body only when the bean it is accessing is created. Listing 4.2 shows such a custom tag that can be used to display its body, hide it, or even reverse it. The result is shown in Figure 4.1.

[1] Formally, the tag lives in an XML namespace specified by the prefix. Custom tags can be loaded with any namespace; formally, the portion before the colon is not part of the name. In the text however, this prefix will always be included to avoid possible confusion between tags, such as `c:param` and `sql:param`.

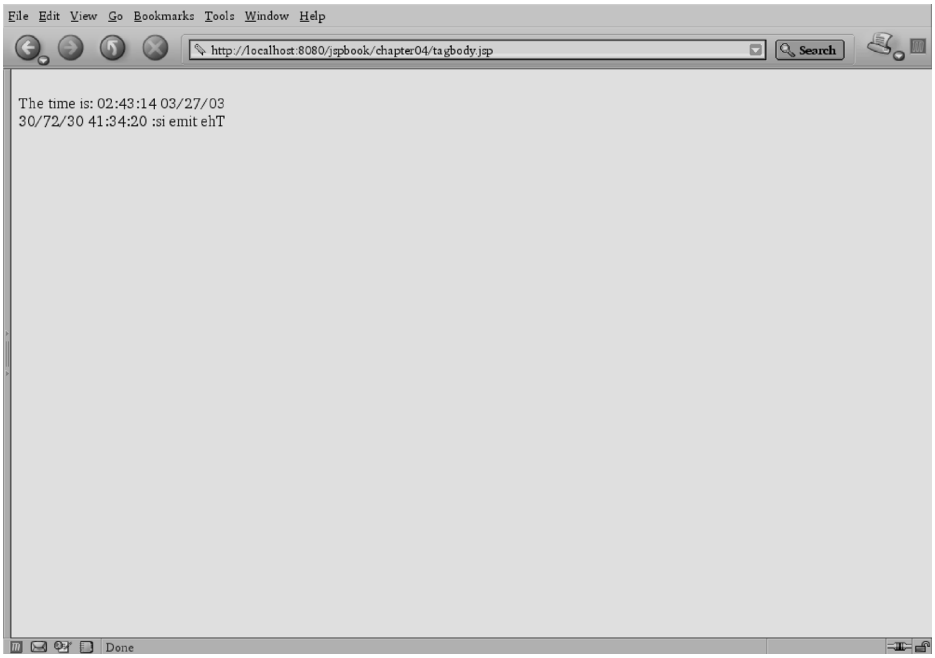

Figure 4.1. The result of a custom tag.

Listing 4.2 A custom tag with a body

```
<%@ taglib prefix="awl"
     uri="http://jspbook.awl.com/samples" %>
<awl:maybeShow show="no">
You can't see me!
</awl:maybeShow><br>

<awl:maybeShow show="yes">
  The time is:
  <awl:date format="hh:mm:ss MM/dd/yy"/>
</awl:maybeShow><br>

<awl:maybeShow show="reverse">
  The time is:
  <awl:date format="hh:mm:ss MM/dd/yy"/>
</awl:maybeShow><br>
```

This example loads the same tag library used in Listing 4.1 and again specifies that it will be using the `awl` prefix to access the tags. The tag used this time is called

`awl:maybeShow`, and it has a parameter, `show`, that controls what the tag should do with its body. This parameter may be set to `no`, in which case the body is hidden from the page; `yes`, in which case the body is displayed; or `reverse`, in which case the body is shown backward.

Note that the body of the `awl:maybeShow` tag may include anything, including other JSP tags. This was also true of the `jsp:useBean` tag and in fact is true of any custom tag that has been properly programmed. This property is described by saying that JSP tags can be *nested*. From here on, it will simply be assumed, unless otherwise noted, that the body of any tag can contain any other tag.

4.3 Dynamic Attributes in Tags

For the standard tag library to be able to do all the wonderful things it claims to do, the tags will need to take parameters that are more complicated than such simple instructions as "yes" and "no." In fact, the parameters to the standard tag library comprise a full language, although one that is significantly simpler than Java itself and much better suited for building pages.

This language is built into the very core of JSPs in the latest version of the JSP specification. This means that programmers creating new tags may use this language for their own purposes; this will also be illustrated in Chapter 13.

Expressions in this language are surrounded by braces and preceded by a dollar sign. The simplest kinds of expressions in the language are constants, such as strings or numbers:

```
${23}
${98.6}
${'hello'}
```

These expressions don't mean anything on their own, but when used as the value of a parameter, they are evaluated by the expression language before they are sent to the tag. Because numbers and strings evaluate to themselves, this means that the following two expressions mean the same thing:

```
<awl:maybeShow show="${'yes'}">

<awl:maybeShow show="yes">
```

Note that within an expressions, literals are surrounded by single quotes and that the whole expression is surrounded by double quotes.

Errors to Watch For

If an expression is written incorrectly, such as leaving off a closing quote or a brace, a JSP page error will report something like

```
An error occurred while parsing custom action
```

Now for the fun part: The scripting language can also refer to beans and properties of beans. Listing 3.1 used a bean to display some static properties, including the seventh prime number. Suppose that bean were loaded into a page with this tag:

```
<jsp:useBean id="bean1" class="com.awl.jspbook.ch03.Bean1"/>
```

In that case, then the scripting language would refer to the seventh prime number property as

```
${bean1.seventhPrimeNumber}
```

Note the pattern: first, the name of the bean as defined in the `jsp:useBean` tag, then a dot, then the name of the property. This is not exactly equivalent to the `jsp:getProperty` tag, as dropping this script fragment into a page will not display the value. In fact, it will not do anything at all. However, this would serve perfectly as a way to send the seventh prime number to a custom tag. Admittedly, there would probably never be any need to do such a thing, but often it will be necessary to send a value from a form to a tag. We now have the means to do this: Send the form inputs into a bean with the `jsp:setProperty` tag and then send the value from the bean to a tag with a scripted parameter.

Errors to Watch For

If an attempt is made to access a property that does not exist, a page error that looks like the following will be generated:

```
Unable to find a value for "property"
in object of class "beanClass"
```

Listing 4.3 shows a simple form that lets the user choose whether to show, hide, or reverse a block of text.

Listing 4.3 A form that will be used by a tag

```
<html>
<body>

<form action="show_result.jsp" method="post">
Shall I display the tag body?
<select name="shouldShow">
<option>yes
<option>no
<option>reverse
</select><br>

<input type="Submit" name="Go" value="Go">
</form>

</body>
</html>
```

The page that will use this form is shown in Listing 4.4. It combines many of the things that have been discussed so far: a bean, the `awl:maybeShow` tag, and a scripted parameter.

Listing 4.4 Using a bean and a tag together

```
<%@ taglib prefix="awl"
    uri="http://jspbook.awl.com/samples" %>
<jsp:useBean
  id="form"
  class="com.awl.jspbook.ch04.FormBean"/>
<jsp:setProperty name="form" property="*"/>

<awl:maybeShow show="${form.shouldShow}">
  The time is:
  <awl:date format="hh:mm:ss MM/dd/yy"/>
</awl:maybeShow>
```

The first portion of this example should be old hat by now: First, a tag library is loaded, and then a bean is obtained and fed the form values. The second part

uses the tag almost exactly as in Listing 4.2. The only difference is that the `show` parameter is not a fixed value but comes from the bean via a script. Using a bean, a custom tag, and the scripting language, we can now dynamically control a whole block of text!

4.4 Displaying Expressions

The ability to use a bean to control a tag is certainly powerful, but often such values must be shown to the user rather than used by a tag. A standard tag, `c:out`, renders values to the page, and the use of this tag is quite straightforward. Listing 4.5 revisits the example from Listing 3.1, which displayed various values from a bean. Listing 4.5 use the same bean but now displays values using the new tag.

Listing 4.5 The `out` tag

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<jsp:useBean
  id="bean1"
  class="com.awl.jspbook.ch03.Bean1"/>

<p>Here is some data that came from bean1:</p>

<ul>

<li>The name of this bean is:
<c:out value="${bean1.name}"/>

<li>The 7th prime number is:
<c:out value="${bean1.seventhPrimeNumber}"/>

<li>The current time is:
<c:out value="${bean1.currentTime}"/>

</ul>
```

Because this does exactly the same thing as Listing 3.1, it may not be immediately clear why anyone would use the `c:out` tag instead of the `jsp:getProperty` tag. Although `c:out` is somewhat shorter, the real reason to use it is that it has many advantages, all of which are derived from the fact that what is being shown is the result of a script, not a simple property.

The expression language allows page developers to manipulate properties in many ways. For example, it is possible to write an expression that will add two numbers right in the page, without needing to rely on the bean to do it. Listing 4.6 shows another version of our calculator from Listing 3.6, only doing the addition in the page.

Listing 4.6 Addition in the expression language

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<jsp:useBean
  id="calc"
  class="com.awl.jspbook.ch04.CalcBean"/>
<jsp:setProperty name="calc" property="*"/>

The sum is:
<c:out value="${calc.value1 + calc.value2}"/>
```

It is now possible to extend this easily to do more complex calculations, such as finding the average of the two numbers or raising one to the power of the other, and so on.

Note that although this is very powerful, it also breaks the model/view/controller paradigm, as the model is now being manipulated directly from the view. Sometimes, this is worth doing, but as a general rule of thumb, it is better to leave such calculations in the bean.

Another advantage to the `c:out` tag is that it can display things other than beans. Every JSP has available a number of *implicit objects*, that is, objects that the system provides without the developer's needing to load or name them explicitly. One of these is the `pageContext` object, which contains a great deal of information about the action currently being performed, such as the name of the page being generated, the name of the computer from which the request came, and so on. Listing 4.7 uses the `pageContext` object to display some of the available information.

Listing 4.7 The `request` object

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

<li>Your computer is called
<c:out value="${pageContext.request.remoteHost}"/>
```

```
<li>This page came from server
<c:out value="${pageContext.request.serverName}"/>

<li>This page came from port
<c:out value="${pageContext.request.serverPort}"/>

</ul>
```

This example illustrates a new kind of syntax: expressions with multiple dots. This will make more sense following the discussion of compound data later in this chapter.

Another important implicit variable is `param`, and it holds all the values that have been sent to a page by a form. This variable acts like a special bean in that it does not have a predefined set of properties but instead has a property for every value in the form.[2] Suppose, for example, that a form has an input like this:

```
<input type="text" name="color">
```

The user's response could be displayed on a page using the following:

```
<c:out value="${param.color}"/>
```

This feature of the expression language provides a fix for a problem with the Java News Today site. Recall that the page title appears in `top.jsp`, which is shared by every page, but this title really should change to identify each page. This can be accomplished as follows:

```
<title>
  Java News Today: <c:out value="${param.title}"/>
</title>
```

Here, the parameter `title` will not come from a form but instead can be passed in through a variation of the `jsp:include` tag. For example, the index page will now include the top portion of the page with

```
<jsp:include page="top.jsp">
  <jsp:param name="title" value="Welcome!"/>
</jsp:include>
```

[2]For readers familiar with Java, `param` is in instance of a class that implements the java.util.Map interface. The expression language handles the dot operator following the name of a map by treating the identifier after the dot as a key.

The availability of `param` also means that the bean isn't needed in Listing 4.4 at all! The whole page can be reduced to

```
<awl:maybeShow show="${param.shouldShow}">
  The time is:
  <awl:date format="hh:mm:ss MM/dd/yy"/>
</awl:maybeShow>
```

Likewise, the calculator could do without its bean, reducing the page to

```
The sum is:
<c:out value="${param.value1 + param.value2}"/>
```

Errors to Watch For

Most of the comments about possible errors when using the `jsp:getProperty` tag also apply to `c:out` and other tags that use expressions. In particular, trying to reference a property that the bean does not possess will result in an error.

In addition, trying to reference a bean that does not exist, such as `c:out value="${someBean.someProperty}"` if `someBean` has not been loaded, will not result in an error but simply in nothing being displayed. This can result in problems that may be difficult to find and fix, for example, if the name of a bean is simply misspelled.

Because `c:out` acts like an enhanced version of `jsp:getProperty`, it is not surprising that an equivalent of the `jsp:setProperty` tag is in the standard library. This tag, `c:set`, looks like this:

```
<c:set
     target="bean"
     property="property name"
     value="property value"/>
```

This tag sets the property called *`property name`* in the bean identified as `bean name` to `value`. Unlike the `jsp:setProperty` tag, the `c:set` tag can not set all the properties in a bean at once by using the special property `*`. However, each of the parameters to `c:set` may be a script, which allows properties to be set with dynamic values.

Errors to Watch For

When using the `c:set` tag, it is very important to mind the distinction between something like `target="bean"` and `target="${bean}"`. The former is a name that has no properties; the latter is a bean obtained from the name by the expression language. This can be a natural source of confusion, as the `jsp:setProperty` tag does use the name. Even if the reason is not completely clear at this point, remember that the target should always take an expression, not simply a name.

4.5 Formatting Output

Consider once again the calculator from Listing 4.6. If the user enters large numbers—say, 1264528 and 9273912—the sum will be 10538440. This is certainly the right answer, but it is not in a particularly readable format. It would be much better if it could be displayed as 10,538,440. The issue of formatting comes up frequently when designing Web pages, as there are often particular rules about how numbers, currencies, and dates should be displayed.

Another portion of the standard library provides tags for displaying formatted values. This library can be imported by using

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

Once loaded, a number of new tags are available, some of which work similarly to the `c:out` tag but allow a format to be specified. For example, the `c:out` tag from Listing 4.6 could be replaced with

```
<fmt:formatNumber
  value="${calc.value1 + calc.value2}"
  pattern="###,###"/>
```

The `pattern` indicates that there should be a comma after every three digits. It would also be legal to provide a decimal point, as in `###,###.##`, which would indicate that a comma should be placed every three digits, with two digits following the decimal point.

A number of examples have contained custom mechanisms for formatting dates. Now a more general solution to this problem is available: the `fmt:formatDate` tag. This tag works very much like `fmt:formatNumber` tag but expects its value to be a date. If the bean from Listing 3.1 were loaded with `jsp:useBean`, the `date` property could be formatted with

```
<fmt:formatDate
  value="${bean1.date}"
  pattern="hh:mm:ss MM/dd/yy"/>
```

The valid expressions for `pattern` can be found in the documentation for the `java.text.SimpleDateFormat` class, but note for the moment that any of the expressions from Listing 3.2 would work.

The formatting tags can do a great deal more than has been shown here. Different countries have different standard ways to express numbers and dates, and the format tags can ensure that data is formatted in an appropriate way for each country, through a mechanism called *internationalization*. The format tags can also be used to parse values, which would allow the calculator to accept inputs with commas and decimal points. These topics are beyond the scope of this book, but now that the basic functionality of these tags is clear, interested readers can see the remaining details in section 9 of the JavaServer Pages Standard Tag Library specification, which is available from http://java.sun.com/products/jsp/.

4.6 Compound Data in the Expression Language

Up until now, all the bean properties have have been simple types: strings of text or numbers. This feature of the examples that have appeared is not a fundamental restriction on beans themselves. Beans can contain *compound values* as well.

Compound values, as the name implies, contain multiple pieces of data. If this sounds familiar, it should; beans themselves hold multiple pieces of data. Indeed, beans can contain other beans, which can contain yet other beans, and so on, indefinitely.

As an example of how this might be useful, consider how a bean would be used to model a home entertainment system, which may contain many individual components, such as an amplifier, CD player, cassette player, and radio tuner. The system as a whole may have certain properties, such as which component is currently playing and the overall color and size of the system. In addition, each individual component has its own set of properties. The CD player has a property representing the name of the disc it currently contains, the tuner has a property indicating the station it is currently tuned to, and so on.

It would in principle be possible to give all the properties of the components to the system as a whole, but this is bad design. It is much better to *encapsulate* logical units as separate beans. This design allows more complex beans to be constructed incrementally by using the individual building blocks, in the same way that using the `jsp:include` tag allows complex pages to be built up from smaller ones.

Given the home entertainment system bean, there is no way in which the `jsp:getProperty` tag could be used to determine the name of the CD in the CD player. The whole CD player bean could be obtained with

```
<jsp:getProperty
  id="homeEntertainment"
  property="cdPlayer"/>
```

However, this will display the whole CD player bean. Beans as a whole have no standard representation; this might display as something cryptic, such as `com.awl.ch04.jspbook.CdBean10b053`, or it might display as a list of all the properties of the bean or anything else that the bean programmer has chosen. In any case, it is unlikely to display only the name of the current disc. What is needed is a way to traverse a set of compound data. Fortunately, the expression language provides a mechanism to do this.

As discussed previously, within the expression language, a single dot between two names indicates that the name on the left should be a bean and the name on the right a property. This extends in a natural way; if a property is itself a bean, it is legal to add another dot followed by the name of a property within that bean, and so on. Getting the name of CD from a CD player within a home entertainment system would therefore look something like

```
${homeEntertainment.cdPlayer.currentDisk}
```

The meaning of the multiple dots in Listing 4.7 should make more sense now. An object called `pageContext` holds information about the page currently being generated. Within this object is another object, called `request`, which holds information pertaining to the request being processed. Finally, the `request` object has such data as the name of the local computer, the remote computer, and so on.

4.6.1 Repeating a Section of a Page

Another important kind of compound data is a collection of an arbitrary number of values. A CD has a number of tracks, but as this number is different for different CDs, a CD bean cannot simply have a different property for each track. Similarly, a

shopping cart bean will contain a number of items, but this number will change as the bean is used.

Java has many ways to manage collections of varying size, but the simplest is called an *array*. Arrays are lists of objects of the same type, such as arrays of strings, arrays of numbers, and arrays of CDs. Within these arrays, items are referenced by a number called the *index*, starting with 0.

The expression language makes it possible to pull a particular element out of such an array by placing its index within brackets. Obtaining the first track of a CD could be done with an expression like this:

```
${cd.tracks[0]}
```

Again, note how this logically follows from the way properties work: `${cd.tracks}` would return the entire array; following this with `[0]` pulls out a particular element from that array.

Errors to Watch For

If a request is made for an index beyond the number of elements in the array, the result will be empty.

It is unusual to need to access a particular element in an array; it is more common to need to repeat some action for every element, regardless of how many there are. This process is known as *iteration*, and it should come as no surprise that a tag in the standard library handles it: `jsp:forEach`. Recall that Listing 3.13 obtained information about a CD from a serialized bean. At that point, however, there was no way to list the tracks, because the page could not know in advance how many there would be. Listing 4.8 uses the `c:forEach` tag to solve this problem, and the resulting page is shown in Figure 4.2.

Listing 4.8 The `forEach` tag

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<jsp:useBean
 id="album"
 beanName="tinderbox4"
 type="com.awl.jspbook.ch04.AlbumInfo"/>
```

Figure 4.2. Iteration used to display every element in an array.

```
<h1><c:out value="${album.name}"/></h1>

Artist: <jsp:getProperty name="album"
            property="artist"/><p>
Year: <c:out value="${album.year}"/></p>

Here are the tracks:
<ul>
<c:forEach items="${album.tracks}" var="track">
  <li><c:out value="${track}"/>
</c:forEach>
</ul>
```

The `c:forEach` tag takes a number of parameters. The first is the items to iterate over, which is specified by a script. The second is a name to use as a variable; within the body of `c:forEach`, this variable will be set to each element in the array in turn. This variable can be accessed by the expression language as a bean, which means, among other things, that the `c:out` tag can be used to display it.

Errors to Watch For

If something other than an array is used as the `items` parameter, the `c:forEach` tag will treat it as if it were an array with one element.

4.6.2 Optionally Including Sections of a Page

Iteration allows a page to do one thing many times. The other major type of control a page may need is determining whether to do something at all. The custom `awl:maybeShow` tag introduced at the beginning of this chapter handled a limited version of that problem, but the standard tag library provides a number of much more general mechanisms, called collectively the *conditional* tags. The most basic of these tags is called `c:if`.

In its most common form, the `c:if` tag takes a single parameter, `test`, whose value will be a script. This script should perform a logical check, such as comparing two values, and facilities are provided to determine whether two values are equal, the first is less than the second, the first is greater than the second, and a number of other possibilities. Listing 4.9 shows how the `c:if` tag can work with a bean to determine whether to show a block of text.

Listing 4.9 The `if` tag

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="awl"
    uri="http://jspbook.awl.com/samples" %>
<jsp:useBean
  id="form"
  class="com.awl.jspbook.ch04.FormBean"/>
<jsp:setProperty name="form" property="*"/>

<c:if test="${form.shouldShow == 'yes'}">
 The time is:
 <awl:date format="hh:mm:ss MM/dd/yy"/>
</c:if>
```

Note the expression in the script for the `test` parameter. Two equal signs, `==`, are used to check two values for equality. Here, the first value comes from a property

and is obtained with the normal dotted notation. The second value, `yes`, is a constant, or *literal*, which is reflected by the single quotes around it in the script. If these quotes were not present, the expression language would look for a bean called `"yes"`; as no such bean exists, the result would be an error.

Listing 4.9 is similar to Listing 4.3; the major difference is that Listing 4.9 uses the standard tag instead of the custom `awl:maybeShow`. The downside is that the `c:if` tag cannot reverse a block of text; all it can do is decide whether to include its body content in the final page.

This may seem like a shortcoming but in fact reflects a good design pattern. Note that `awl:maybeShow` does two completely unrelated things: checks whether a value is `yes`, `no`, or `reverse` and reverses a block of text. Rather than making one tag do two things, it is better to have two different tags. According to the so-called UNIX philosophy of software, each piece of code should do only one thing and do it well, and there should be easy ways to knit these small pieces together. For tags, this means that each tag can be used independently or combined with other tags. In this case, if an `awl:reverse` tag did nothing but reverse its body content, it could be combined with the `c:if` tag to do the same thing as Listing 4.3. This is shown in Listing 4.10.

Listing 4.10 Splitting tags

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="awl"
    uri="http://jspbook.awl.com/samples" %>
<jsp:useBean
   id="form"
   class="com.awl.jspbook.ch04.FormBean"/>
<jsp:setProperty name="form" property="*"/>

<c:if test="${form.shouldShow == 'yes'}">
 The time is:
 <awl:date format="hh:mm:ss MM/dd/yy"/>
</c:if>

<c:if test="${form.shouldShow == 'reverse'}">
 <awl:reverse>
   The time is:
```

```
    <awl:date format="hh:mm:ss MM/dd/yy"/>
  </awl:reverse>
</c:if>
```

Note that two `c:if` tags are used here: one to check whether the value is `yes` and another to check whether it is `reverse`. The body content of both of these tags is the same, which is rather wasteful. It means that if the body ever needs to change, it will need to be modified in two places in order to keep everything consistent. It would be better in this case to put the body in a separate file and then have both of the `if` tags include that file with a `jsp:include` tag. Now that the functionality of `awl:maybeShow` has been divided into two pieces, the `c:if` tag can be used for many other things, and the `awl:reverse` tag can be used to reverse unconditionally a block of text, should such a thing ever be useful.

Listing 4.10 imports two tag libraries: the standard one, which is installed as `c` and provides the `c:if` tag, and the custom one installed as `awl`, which provides the `awl:reverse` tag. This is perfectly valid; often a page will need many different tags from different libraries, and it will then need to import all of them. The only catch is that each tag library must be given a different prefix.

4.7 Browser Detection

Web programmers face many difficult decisions, not the least of which is how to deal with the fairly horrible state of modern browsers. A popular Web site is likely to receive requests from versions of Internet Explorer 3 through 6, Mozilla, Netscape 4.7, Opera, various AOL browsers, and numerous custom browsers now available in consumer devices, such as phones and PDAs (personal digital assistants). Each of these is likely to render HTML slightly differently, support different media types, and handle JavaScript differently, if at all.

One way of dealing with this variability is to use the "lowest common denominator," that is, only those features that are supported and work the same in every browser. This makes things easier for the Web developer but means that the user will be getting a site that looks like something from the early 1990s, which may disappoint many users. Alternatively, Web developers may design a site for one browser—perhaps Mozilla 1.0—and put up a note encouraging other users to switch to this browser. This is likely to infuriate many users who either don't want to or can't change browsers simply to get to one site.

Finally, developers can create parallel versions of all the browser-specific HTML and JavaScript and so on and send out the appropriate version, based on which browser is being used. The browser makes this possible by identifying itself with

every request, and JSPs make this possible through the conditional tags. A skeleton of code that accomplishes this is shown in Listing 4.11.

Listing 4.11 Browser detection

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<jsp:useBean
  id="browser"
  class="com.awl.jspbook.ch04.BrowserBean"/>

<c:set
  target="${browser}"
  property="request"
  value="${pageContext.request}"/>

You are using a browser that identifies itself as
<c:out value="${browser.fullName}"/><p>

<c:if test="${browser.type == 'Gecko'}">
... include Mozilla code here ...
</c:if>

<c:if test="${browser.type == 'MSIE'}">
... include IE code here ...
</c:if>
```

This example uses `BrowserBean`, a utility bean that extracts browser information from the request. In order to obtain this information, `BrowserBean` must have access to the `request` object. This object is obtained from the `pageContext`, as was done in Listing 4.7, and passed with a `c:set` tag to the bean.

A bean such as `BrowserBean` is needed for two reasons: first, because the browser name is not available as a simple property, such as the ones shown in Listing 4.7; second, because the full name of the browser is likely to be something unwieldy, such as Mozilla/5.0 (X11; U; FreeBSD i386; en-US; rv:1.0rc3) Gecko/20020607, which contains information about the specific revision and operating system on which the browser is running. This is generally more information than needed to select the appropriate browser-specific code for a page. This second problem is solved by having the bean recognize major browser types, and it is this type that is used by the `c:if` tags.

4.8 Combining Tags

As mentioned previously, the bodies of JSP tags can contain anything, including other JSP tags. An example is the `c:out` tag within the `c:forEach` tag in Listing 4.8. To demonstrate this further, the `c:if` and `c:forEach` tags work together in the following example.

If given an empty array, a `c:forEach` tag will not render its body content at all. This is fine but can lead to some odd-looking pages. In Listing 4.8, if the CD is empty, the page will display "Here are the tracks" and then stop. This is technically correct but to the user may look as though the page stopped generating halfway through. It would be better to inform the user that the CD is empty rather than to display a list with no elements. This can be accomplished by putting the `c:forEach` tag inside a `c:if` tag, as shown in Listing 4.12.

Listing 4.12 Tags working together

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<jsp:useBean id="album" beanName="tinderbox4"
 type="com.awl.jspbook.ch04.AlbumInfo"/>

<h1><jsp:getProperty name="album" property="name"/></h1>

Artist: <jsp:getProperty name="album"
            property="artist"/><p>
Year: <jsp:getProperty name="album" property="year"/></p>

<c:if test="${empty album.tracks}">
There are no tracks!  What a boring CD.
</c:if>

<c:if test="${!(empty album.tracks)}">
  Here are the tracks:
  <ul>
  <c:forEach items="${album.tracks}" var="track">
    <li><c:out value="${track}"/>
  </c:forEach>
  </ul>
</c:if>
```

Conceptually, the only new thing about this example is the check that is done in the `c:if` tag. The `empty` in the test checks whether the named property exists,[3] and if it does exist and is an array, whether it has any elements. The exclamation point in the test should be read as "not." It means that if the following test would be true, it returns `false`, and vice versa.

4.9 Selecting among Multiple Choices

Once again, the preceding example had to use two `c:if` tags, although the bodies are different in this case. However, this is still somewhat clumsy, as the same check is being performed twice: once to see whether it is true and once to see whether the reverse is true. This double check is needed because the `c:if` tag is capable of deciding only between two alternatives: to include its body or not to include it. Another set of tags allows *multiway branching*, or choosing from among several mutually exclusive possibilities.

Unlike the other tags seen so far, three tags work together to obtain the desired result. The outermost tag, `c:choose`, has no parameters; it merely serves as a container for a collection of two other tags: `c:when` and `c:otherwise`. Each individual `c:when` tag acts a lot like a `c:if` tag. Both tags take a parameter called `test`, which should be a script, and render their body content if the condition in the script is `true`. The difference is that multiple `c:if` tags will each be checked in turn, whereas a `c:choose` tag will stop after finding the first `c:when` tag with a `test` that is `true`.

In other words, consider a set of possible values for a bean property, such as the colors red, green, and blue. The following snippet of code would check each of these possibilities regardless of the value:

```
<c:if test="${bean.color == 'red'}">...</c:if>
<c:if test="${bean.color == 'green'}">...</c:if>
<c:if test="${bean.color == 'blue'}">...</c:if>
```

The following snippet will check whether the color is red; if so, it will stop and will not then have to check whether it is green and then blue:

```
<c:choose>
  <c:when test="${bean.color == 'red'}">...</c:when>
  <c:when test="${bean.color == 'green'}">...</c:when>
  <c:when test="${bean.color == 'blue'}">...</c:when>
</c:choose>
```

[3] Technically, it tests whether the value equals `null`, as will be discussed in Chapter 9.

Clearly, the second option is more efficient. In addition, using the `c:choose` tag groups related code in one place and so makes JSPs easier to read and understand.

The `c:choose` tag works with another tag: `c:otherwise`. This tag also has no parameters; its body will be evaluated if none of the `c:when` tags has a `true` condition.

It is now clear how it would be possible to avoid doing the check twice in Listing 4.11—by using one `c:when` and a `c:otherwise`—rather than by using two `c:if` tags. This is shown in Listing 4.13.

Listing 4.13 The `choose` tag

```
<c:choose>
  <c:when test="${empty album.tracks}">
    There are no tracks!  What a boring CD.
  </c:when>

  <c:otherwise>
    Here are the tracks:
    <ul>
    <c:forEach items="${album.tracks}" var="track">
      <li><c:out value="${track}"/>
    </c:forEach>
    </ul>
  </c:otherwise>
</c:choose>
```

This code is a little more verbose than Listing 4.12 but has the advantage of avoiding one redundant `test`. Using the `c:choose` tag also makes it clear that the conditions are mutually exclusive, and hence only one of the bodies will ever be rendered.

Chapter 2 briefly mentions the `jsp:forward` tag, which sends the user from one page to another. This tag can be combined with the `c:choose` tag to provide a type of control called *dispatching*, whereby one page determines where the appropriate content lives and sends the user to that page. This is illustrated in Listing 4.14.

Listing 4.14 Using the `choose` tag as a dispatcher

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

```
<c:choose>
  <c:when test="${param.whichPage == 'red'}">
    <jsp:forward page="red.jsp"/>
  </c:when>

  <c:when test="${param.whichPage == 'green'}">
    <jsp:forward page="blue.jsp"/>
  </c:when>

  <c:when test="${param.whichPage == 'blue'}">
    <jsp:forward page="blue.jsp"/>
  </c:when>

  <c:otherwise>
    <jsp:forward page="select_page.jsp"/>
  </c:otherwise>
</c:choose>
```

This page looks for a form parameter, `whichPage`, which should be `red`, `green`, or `blue`, and, based on this value, sends the user to one of three pages. If no value has been provided, the `otherwise` tag forces the user to `"select_page.jsp"`, which contains the form to be filled out.

4.10 Summary and Conclusions

Now we're cooking! This chapter introduced the concept of custom tags and the standard tag library now part of the JSP specification. These tags give page authors full control over what the user ends up seeing, by providing the means to show arbitrary values, repeat a section of a page as many times as needed, conditionally remove a section of page, or choose from many possible sections. In the next chapter, we'll see how these tags, together with beans, allow Java News Today to build its site.

4.11 Tags Learned in this Chapter

`c:forEach` Repeats a section of the page for every item in an array
Parameters:

`items`: An expression specifying the array to use, most likely a bean property

`var`: The name of the variable with which each element in the array will be referred

Body: Arbitrary JSP code

`c:out` Displays a value
Parameters:

`value`: An expression to be evaluated and displayed

Body: Arbitrary JSP code; the body content will be displayed if `value` is `null`

`c:if` Conditionally include a portion of the page
Parameters:

`test`: An expression that should be a logical test of a property

`var`: If present, names a variable where the result of the expression will be stored

Body: Arbitrary JSP code

`c:choose` Includes one of several portions of a page
Parameters: None
Body: Arbitrary number of `c:when` tags and, optionally, one `c:otherwise` tag

`c:when` One possibility for a `c:choose` tag
Parameters:

`test`: An expression that should be a logical test of a property

Body: Arbitrary JSP code

`c:otherwise` The catch-all possibility for a `c:choose` tag. If none of the expressions in the `c:when` tags evaluates to `true`, the body of the `c:otherwise` will be included.
Parameters: None
Body: Arbitrary JSP code

`c:set` Set a property in a bean
Parameters:

`target`: The name of a bean

`property`: The property within the bean to set

`value`: The value to assign; may be a script

Body: None

`fmt:formatNumber` Format a number for output
Parameters:

`value:` The value to be formatted; may be a script

`pattern:` A pattern specifying how the number should be formatted
Body: None

`fmt:formatDate` Format a date and/or time for output
Parameters:

`value:` The value to be formatted; may be a script

`pattern:` A pattern specifying how the date should be formatted
Body: None