8

JSP Tag Libraries

CERTIFICATION OBJECTIVES

- Tag Libraries
- JSTL
- EL Functions
- The "Classic" Custom Tag Event Model

✓ Two-Minute Drill

Q&A Self Test

In the previous two JSP chapters, you got to use the low-level facilities in the technology. In this and the next chapter, you'll meet some higher-level tools and techniques, and finally get to reduce Java language syntax in your JSP page source to little or none.

First, we'll look at how to use tag libraries. You'll learn how to use tags that are not delivered with the JSP container but that you either create yourself or obtain from elsewhere. You'll revisit the deployment descriptor to see the role it plays in supporting your own and third-party tag libraries.

In this chapter you'll also learn all about a "custom" tag library now provided as "standard"—if that's not too much of a contradiction: This is the Java Standard Tag Language core library. This supplies the missing logic elements you need in order to use EL without Java language syntax.

After that, we move back to Expression Language. You'll see how EL itself is underpinned by tag technology, and you'll learn how to write your own functions that can be accessed with EL syntax on a JSP page. You'll also see how these EL functions use definitions in tag libraries.

Finally in the chapter, you'll get to write your own custom tags—a step beyond making use of an existing tag library. This is one of the most challenging of the exam objectives but also one of the most rewarding: You'll see how to harness the full power of Java while keeping your JSP pages syntactically simple and elegant.

CERTIFICATION OBJECTIVE

Tag Libraries (Exam Objectives 9.1, 9.2, and 6.6)

For a custom tag library or a library of Tag Files, create the "taglib" directive for a JSP page.

Given a design goal, create the custom tag structure in a JSP page to support that goal.

Configure the deployment descriptor to declare one or more tag libraries, deactivate the evaluation language, and deactivate the scripting language.

The three exam objectives here take us some little way into the world of custom tags. In this section of the chapter, you'll get to use some existing custom tags within JSP page source. Later exam objectives and sections address how to write a custom tag from scratch.

You'll also begin to appreciate the benefits of tags. The JSP pages we look at here are scriptless and so are much more maintainable. Java code maintenance is pushed back to where it belongs—into standard Java classes.

The Custom Tag Development Process

There are four essential steps to writing a custom tag for use in your JavaServer Pages:

1. Writing a Java class called a tag handler
2. Defining the tag within a tag library definition (TLD) file
3. Providing details of where to find the TLD file in the deployment descriptor, web.xml
4. Referencing the TLD file in your JSP page source and using the tags from it

You can see from this process that there are four "artifacts" involved with a custom tag: a Java class file, a TLD file, web.xml, and the JSP page itself. The relationship among these four is shown in Figure 8-1.

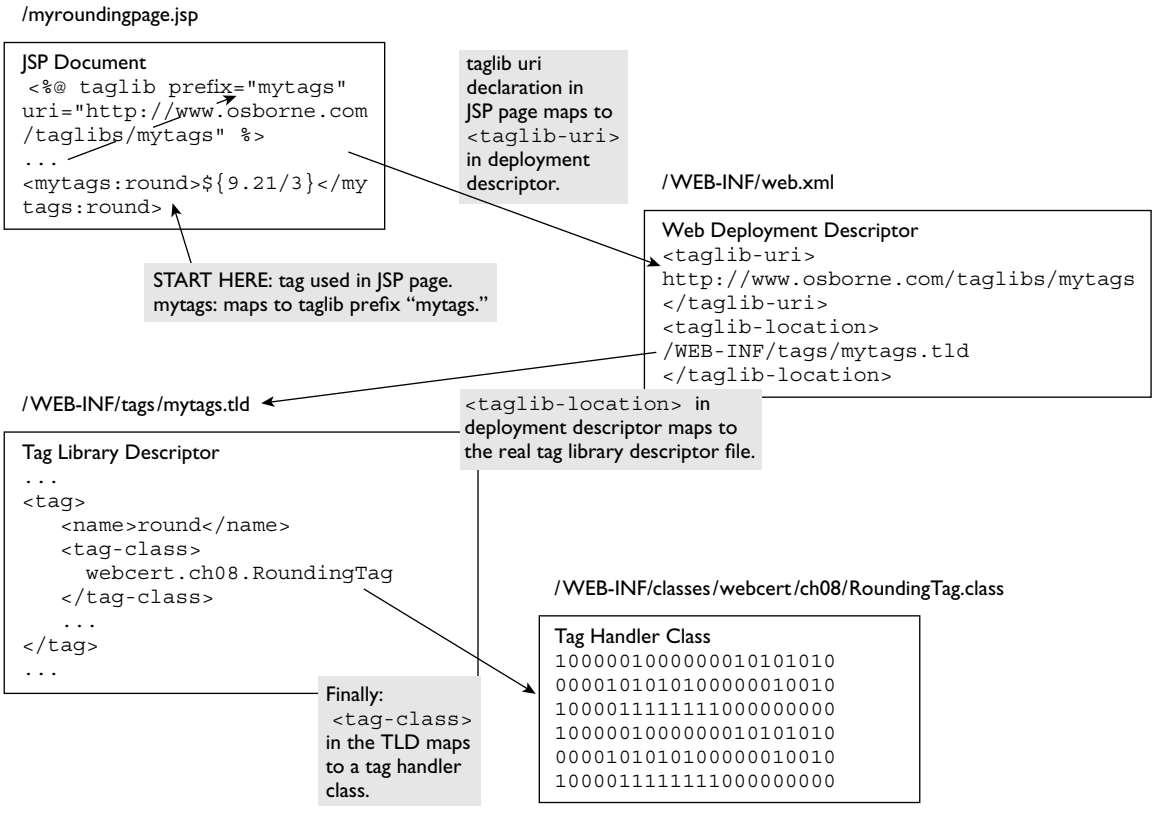
We're actually going to concentrate on steps 2, 3, and 4 in this section of the chapter—not the writing of a custom tag, but merely using one. Writing the tag handler code (step 1) comes in the last part of this chapter. As an example, we'll take a simple tag that solves (or at least masks) one of the annoyances of Expression Language arithmetic with double values.

Hunting the Tag

To illustrate the process of how the JSP container finds the tag, we'll use a custom tag for rounding a figure to an arbitrary number of decimal places. The process is shown in overview in Figure 8-1.

You'll recall from Chapter 7 that double arithmetic on most computers, contrary to most people's expectations, is not an exact science. Small errors creep in because of the nature of binary storage of double primitives. So the expression ${9.21 / 3} displays 3.0700000000000003 instead of the exact result you might expect—3.07. The solution—at least for display purposes—is to round the resulting figure to a convenient number of decimal places. However, EL doesn't have any kind of syntax to handle formatting. Instead, we'll pass the responsibility for this on to a custom tag.

FIGURE 8-1 Tag Building Blocks



Here's a complete JSP page in traditional syntax that uses a rounding tag:

```
01 <html>
02   <%@ taglib prefix="mytags" uri="http://www.osborne.com/taglibs/mytags" %>
03   <head><title>Rounding Example: JSP syntax</title></head><body>
04   <h1>Round EL Calculation To 2 Decimal Places</h1>
05   <p>EL calculation without rounding:${9.21 / 3}</p>
06   <p>EL calculation with rounding:
07      <mytags:round decimalPlaces="2">${9.21 / 3}</mytags:round></p>
08 </body></html>
```

You can see the tag on line 7. It looks just like a JSP standard action. The element is called `mytags:round`, and it consists of an opening and closing tag, with a body.

The body contains the EL expression we've just been discussing—`${9.21 / 3}`.The opening tag contains an attribute named `decimalPlaces`, set to a value of 2. The functionality is what you might expect—the result of the expression is rounded to two decimal places. This is displayed on the page instead of the "raw" result of the EL calculation.

What are the actual steps involved at run time to produce this result? Behind the scenes, the JSP container interacts with the real Java class associated with the tag, which happens to be called RoundingTag. Here's a high-level sequence of events:

1. The JSP container calculates the EL expression.
2. The JSP container locates an instance of RoundingTag.
3. The JSP container passes information to Rounding Tag—the value of the attribute (2 for decimalPlaces) and the result of the EL calculation.
4. RoundingTag does the necessary math to convert the result to two decimal places.
5. RoundingTag clears out the original EL calculation result from the JSP's output buffer and substitutes the rounded result instead.
6. The JSP container carries on outputting the rest of the page.

At this stage, we're only interested in how the JSP container locates RoundingTag. The process is reasonably straightforward. First of all, as used on the page, the tag has a name and a prefix:

```
07 <mytags:round ...
```

exam Watch

You have complete freedom of choice as to what prefix to use, within legal XML naming conventions. However, all the prefixes used in all the `taglib` directives on a given JSP page (and any of its statically included files) must be unique.

The prefix (`mytags`) must match the value of a prefix attribute in a `taglib` directive in the JSP page source, which it does:

```
02 <%@ taglib prefix="mytags"
...
```

The `taglib` directive has another attribute, `uri`:

```
02 <%@ taglib prefix="mytags"
uri="http://www.osborne.com/taglibs/mytags" %>
```

The URI doesn't point anywhere! Well, it might—but that's purely incidental to the tag library resolution going on here. The URI should match an entry in the deployment descriptor web.xml, which looks like this:

```
<web-app>
  <jsp-config>
    <taglib>
      <taglib-uri>http://www.osborne.com/taglibs/mytags</taglib-uri>
      <taglib-location>/WEB-INF/tags/mytags.tld</taglib-location>
    </taglib>
  </jsp-config>
</web-app>
```

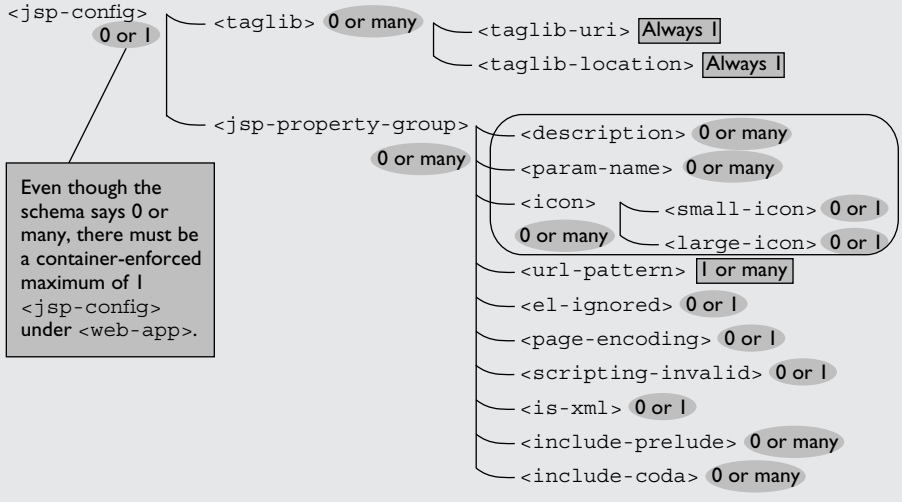
You've met `<jsp-config>` already—as you can see, it sits under the root element `<web-app>`. One of its subelements is `<taglib>`. `<taglib>` has two subelements of its own. The first—`<taglib-uri>`—has a body value that exactly matches the `uri` quoted in the `taglib` directive. The second subelement, `<taglib-location>`, gives the actual location in the web application where the taglib library descriptor (TLD file) is located. The usual rules apply on the path cited in `<taglib-location>`:

- If the path begins with a slash, it's an path beginning at the context root.
- If the path doesn't begin with a slash, it's a relative path—relative to this file, web.xml—so in other words, always relative to where web.xml is located, which is in the /WEB-INF directory.

INSIDE THE EXAM

One thing that has changed quite a bit since the Servlets 2.3/JSPs 1.2 (and so the last version of the exam) is the section of the deployment descriptor devoted to controlling matters that are tag-library related. It used to be very simple: There was an element under `<web-app>` called `<taglib>`, which had `<taglib-uri>` and `<taglib-location>` subelements. All these elements are retained at level 2.4 of the servlet specification, but they are now housed in a new element called `<jsp-config>`, which comes under the root element `<web-app>`. `<jsp-config>` isn't just about tag libraries, though—it controls many aspects of JSP behavior, some of which we have seen already. The following diagram shows the complete layout of the `<jsp-config>` element.

INSIDE THE EXAM (continued)



There are several elements whose meaning you don't have to know for the exam — these are grayed out in the illustration. Of the remainder,

- `<taglib>` and its subelements are described elsewhere in this chapter.
- `<is-xml>` is described in the JSP document section of Chapter 7.

These leaves two elements (`<el-ignored>` and `<scripting-invalid>`) that control expression language and Java as a scripting language. Here's an example `<jsp-config>` setting that does both:

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>/*</url-pattern>
    <el-ignored>true</el-ignored>
    <scripting-invalid>true</scripting-invalid>
  </jsp-property-group>
</jsp-config>
```

The `<url-pattern>` element works exactly as the same-named element for servlet declarations. So `/*` here indicates that all resources in the web application are affected by the settings shown. For other possible URL patterns, see Chapter 2, in

INSIDE THE EXAM (continued)

the "Deployment Descriptor Elements" section.

- `<el-ignored>` when set to true causes expression language to be unevaluated—so treated as template text. The default (if this element is omitted from `<jsp-property-group>`) is false.
- `<scripting-invalid>` when set to true causes a translation time error if JSP scriptlets, expressions, or declarations are used. The default (if this element is omitted) is false.

The only piece of the puzzle left is the TLD file itself, which looks like this:

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-
jsptaglibrary_2_0.xsd"
  version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>My Tag Library</short-name>
  <tag>
    <name>round</name>
    <tag-class>webcert.ch08.examp0801.RoundingTag</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
      <name>decimalPlaces</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

A few things to point out about this TLD file:

- The root element is `<taglib>`.
- The first mandatory element under the root is `<tlib-version>`. This denotes the tag library version number. That's a version number you impose for your own versions of the tag library (it doesn't represent—for example—the JSP version

you are using). So you're not tied to 1.0—when you revise the tag definitions, you might reset this to 1.1, or whatever.

- The next mandatory element is `<short-name>`—every tag library must have one.
- Anything else is optional—though you would expect something to be defined in the file! In this case, there is a single tag defined, which has its own `<tag>` element containing various subelements:
  - `<name>`—this is the name for the tag as used on the JSP page (on line 07—`<mytags:round decimalPlaces="2">`).
  - `<tag-class>`—the fully qualified name of the class implementing the tag functionality—in this case, webcert.ch08.exam0801.RoundingTag. For the tag to run, the class file must be available in one of the usual locations—directly in its package directory under WEB-INF/classes or inside a .jar file in WEB-INF/lib.
  - `<body-content>` dictates what can go in the body of the tag. There are four valid values:
    - `empty`—the body of the tag must be empty (so `<pfx:mytag />` or `<pfx:mytag></pfx:mytag>`).
    - `tagdependent`—the body of the tag contains something that isn't regular JSP source. The tag handler code works out what to do with it. A typical use is to put an SQL statement in the body so that the tag handler takes the statement, runs it, and (perhaps) returns the resulting data to the body of the tag in place of the original SQL statement.
    - `scriptless`—the body of the tag does contain regular JSP source, but nothing involving scripting language. So Java language syntax is forbidden—whether in scriptlets or expressions. EL, though, is absolutely fine in a body specified as scriptless—and will be evaluated.
    - `JSP`—the body of the tag contains any kind of regular JSP source. This is the most permissive value (and, as we'll learn later, not allowed for some kinds of tag handlers, the so-called "simple" kind). Java language syntax is just fine.
  - A tag may have any number of attributes, from zero to many. These are represented by `<attribute>` elements nested within the `<tag>` element. There's one attribute in our example, to represent the number of decimal places. The `<attribute>` element has a number of subelements:

- <name>—for the name of the attribute (decimalPlaces in the example).
- <required>— true if the attribute is mandatory, false otherwise.
- <rtexprvalue> (short for "run-time expression value")— true if you can use EL or an expression (<%= ... %> or <jsp:expression>) to provide the attribute's value at run time, false if the attribute's value must be a literal.

So finally, within the <tag-class> element of the <tag> element in the <taglib> root element of the tag library descriptor file, we find the actual Java that does the work—RoundingTag.class. We'll explore what goes on in that class in the last section of this chapter, and meanwhile just be grateful to have followed the trail from a tag reference in the JSP page to a real piece of code. Look back to Figure 8-1 for a "route map" that summarizes the entire trail.

on the job

Whatever happened to the <jsp-version> element, a mandatory element in JSP version 1.2 TLD files, and used to specify JSP version 1.2? The answer is that it has been replaced within the root element, taglib, which must now always contain the attribute/value pair version="2.0" to reference JSP specification level 2.0.

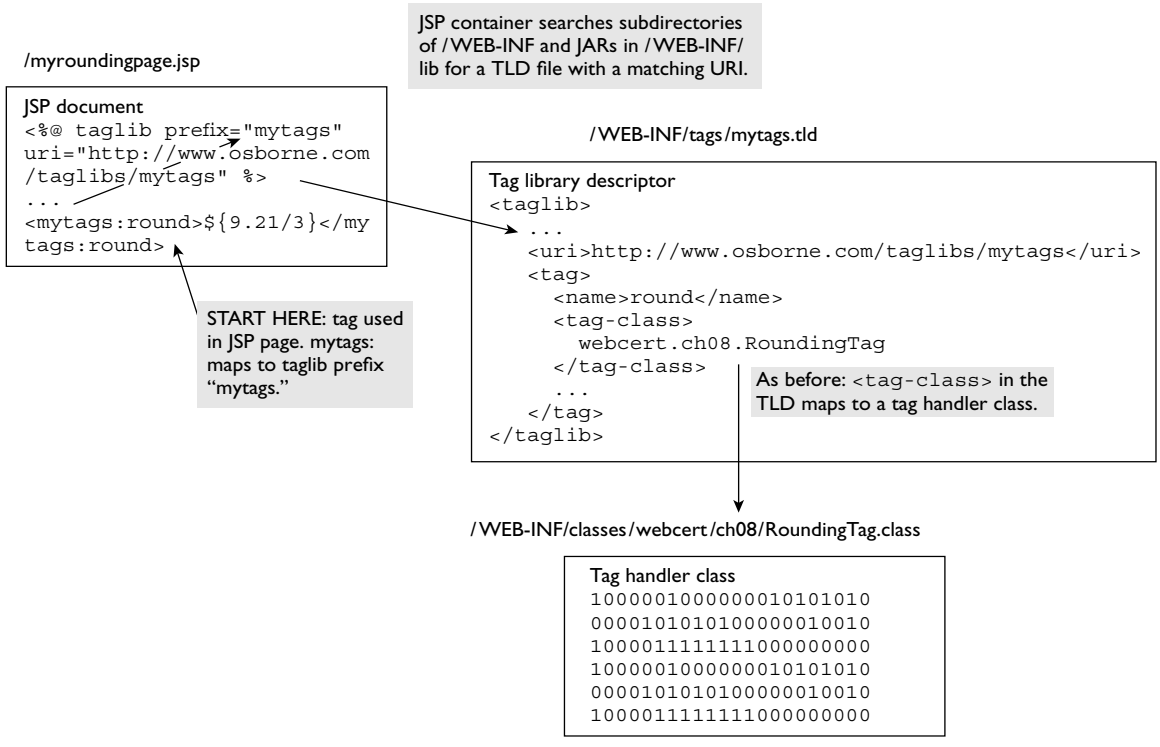
Other Ways to the TLD

The best way to set up mappings from your JSP pages to the TLDs lodged in your web application is through the web.xml entries we have described. This makes the tag library mapping explicit and obvious to anyone reading the deployment descriptor. However, there are alternatives where nothing is needed in the deployment descriptor—so-called "implicit" mapping entries for tag libraries.

First, you can place files with a .tld extension directly in the /WEB-INF directory or one of its subdirectories. Alternatively, you can package .tld files in a JAR file that is placed in /WEB-INF/lib. The .tld files within the JAR file must have a path that begins /META-INF (such that if you were to unpack the JAR file, the .tld would be located directly in the /META-INF directory or one of its subdirectories).

Apart from this restriction on location, the .tld files must contain the optional <uri> element, which must match the uri of the taglib directive or (if using JSP documents) the namespace value. Figure 8-1 showed the (superior) explicit technique. Figure 8-2 summarizes both the implicit mapping techniques for finding tag library descriptors.

FIGURE 8-2 **Routes to Tag Library Descriptors**



Tag Library Descriptors in JSP Documents

You might recall, though, that the tag directive doesn't have a direct equivalent in JSP document syntax—you have to use namespaces instead. Here's the same JSP page source for the rounding example converted to XML syntax in a JSP document:

```
01 <html xmlns:mytags="http://www.osborne.com/taglibs/mytags"
02       xmlns:jsp="http://java.sun.com/JSP/Page">
03   <jsp:output omit-xml-declaration="true" />
04   <jsp:directive.page contentType="text/html" />
05   <head><title>Rounding Example</title></head>
06   <body>
07     <h1>Round EL Calculation To 2 Decimal Places</h1>
```

```
08     <p>EL calculation without rounding:${9.21 / 3}</p>
09     <p>EL calculation with rounding:
10       <mytags:round decimalPlaces="2">${9.21 / 3}</mytags:round></p>
11  </body>
12 </html>
```

exam Watch

The tags you make look identical to JSP standard actions. This is no accident, for JSP standard actions derive from tag libraries. Given that, why do you not have to place a `taglib` directive when you use standard actions in a page? Well, it's a bit like the situation with writing standard Java source. When you use a class from java.lang, you don't have to include an import statement in your source; you get that for free. Such is the case with standard actions; the tag library is just understood to be there and available, with a `jsp:` prefix. Therefore, you can't use the `jsp:` prefix yourself for your `taglib` directives, even if you are not using any standard actions in your page. However, don't forget that when you use JSP document syntax, `taglib` directives disappear in favor of XML namespace definitions, as you've seen. In that case, even standard actions must be explicitly referenced with a namespace—in exactly the same way as your own custom tag libraries.

The essential part is on line 01: Instead of the tag library directive, you find a namespace (xmlns). The prefix `mytags` comes after `xmlns:`, and the URI `http://www.osborne.com/taglibs/mytags` is the value for the namespace. This approach is, in fact, exactly the same one taken for declaring a namespace for JSP custom actions on line 02. The actual way you use the custom tag—shown on line 10—hasn't changed at all from the regular JSP (nondocument) source that began this chapter.

exam Watch

There are certain "reserved prefixes" that you can't use for your own custom tag library declarations, whether as directives or namespaces: jsp, jspx, java, javax, servlet, sun, and sunw. They are case sensitive, so watch out for questions that feature prefixes that are not all lowercase, such as Sun or JAVA: These are in fact legal!

EXERCISE 8-1

Tag Libraries

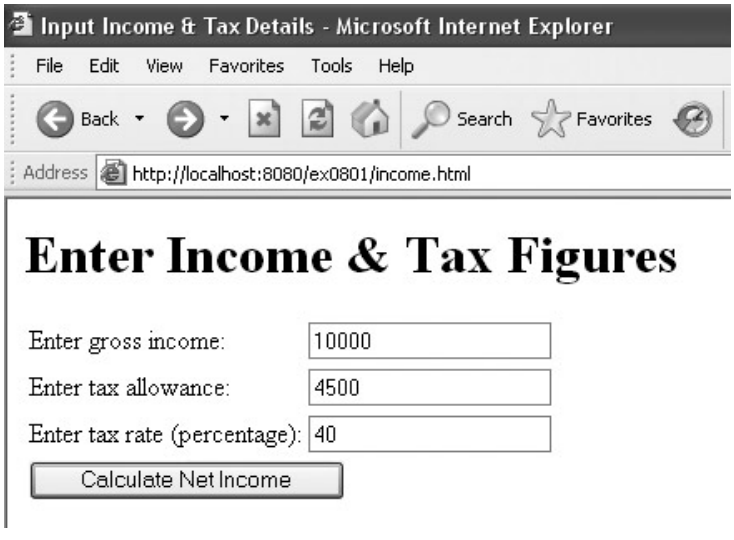
In this exercise you will finish off an incomplete JSP document and deployment descriptor by providing the details required to find and use a tag in an existing tag library—as per the main exam objective for this section. First, you'll install the solution code and make sure that the existing solution runs successfully. The theme is taxation—you enter a gross income figure and some other details required by the taxation office, and get back (through the custom tag) a net income figure. For those of you who hate filing tax returns (I guess I'm speaking to most readers here), grit your teeth and focus on the exam objective!

Install and Run the Solution Code

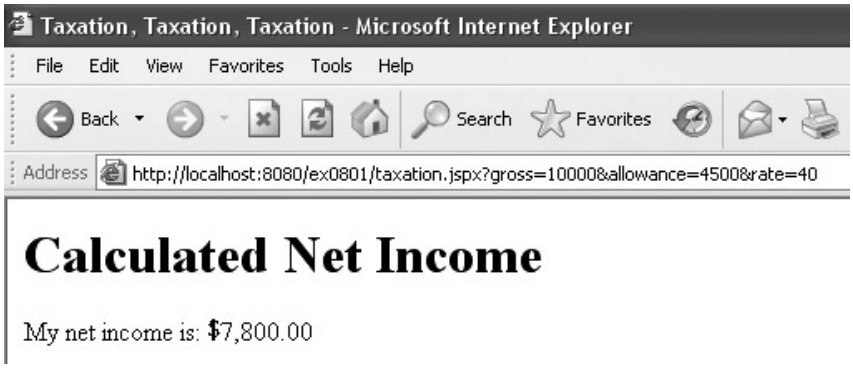
1. The solution file is in the CD as sourcecode/ch08/ex0801.war. Install this WAR file into Tomcat (or your web application server) in the normal way.
2. Run the code using a URL such as

```
http://localhost:8080/ex0801/income.html
```

3. You should see a page like the one illustrated here:



4. Enter a gross income figure (e.g., 10000), a tax allowance (the amount of the gross income that remains untaxed, e.g., 4500), and a percentage tax rate (e.g., 40). Then press the "Calculate Net Income" button.
5. You should see a page (produced by a JSP document) like the one shown here:



6. This page uses a custom tag to calculate the net income according to the formula: net income = tax allowance + ((gross income – tax allowance) * (100 – tax rate) / 100). If the figures supplied in the HTML form don't make sense or are missing, you should see a message on the page saying "Bad Input Figures." The logic is performed in a Java class called the tag handler—we explain tag handlers later in the chapter, but feel free to look at the source now in /WEB-INF/src/webcert/ch08/ex0801/TaxationTag.java.
7. Check the HTML code for the file /income.html. You're not going to change this in any way—just look to see how this sets up three parameters in a form and submits the result to the JSP document called /taxation.jspx.
8. Open the tag library descriptor file, /WEB-INF/tags/mytags.tld. You can see here that four attributes are defined for the tag.

Prepare Files for Your Own Solution

9. Once you're satisfied that the solution works, stop your server. You're going to edit some files in place where they are deployed on your server.
10. There are "unfinished" versions of the deployment descriptor and the JSP document. Perform the following renames so that the unfinished versions end up having the "real" names:
    - Rename /WEB-INF/web.xml to /WEB-INF/webSolution.xml.
    - Rename /taxation.jspx to /taxationSolution.jspx.

- Rename /WEB-INF/webUnfinished.xml to /WEB-INF/web.xml.
- Rename /taxationUnfinished.jspx to /taxation.jspx.

Complete the Deployment Descriptor

11. Open up web.xml, the deployment descriptor. Edit this to provide a reference to a tag library (if you need to, refer to earlier in the chapter to remind yourself how the XML elements are nested). The tag library location (as you saw in step 8) is /WEB-INF/tags/mytags.tld. Save and close the deployment descriptor.

Complete the JSP Document

12. Open up taxation.jspx. You'll see two clearly marked places where you need to intervene. Put in the namespace details for the tag library, together with an appropriate prefix. Then put in the call to the netincome tag. Three of the attributes—grossIncome, allowance, and taxRate—should be set with EL values from run-time parameters. Remember that the HTML form passes in these parameters, whose names are gross, allowance, and rate respectively. Hard-code the value for the fourth—currency—attribute. Any three-character ISO code for currency will be recognized (e.g., GBP, USD, EUR) provided that the relevant locale is recognized by your system. Save and close the JSP document.

Run and Test Your Solution

13. Restart your server, and access income.html as before (step 2). Of course, because of the renaming, when you click the submit button now, your solution page will be invoked.
14. If you don't get the desired result, compare your solution with the original solution (now held in files /WEB-INF/webSolution.xml and /taxation Solution.jspx).

CERTIFICATION OBJECTIVE

JSTL (Exam Objective 9.3)

Given a design goal, use an appropriate JSP Standard Tag Library (JSTL v1.1) tag from the "core" tag library.

We are finally ready to meet the JSP Standard Tag Library. This adds a host of facilities that are otherwise only available through Java language scripting. These facilities are entirely available through JSP tag technology — the same technology we have been exploring in this chapter. So what makes the JSTL special?

It's more the underlying philosophy. When custom tags became available, everybody started building tag libraries. Frameworks of tag libraries became available to simplify the application construction — mostly open source and free. You could guarantee that each of these frameworks would have some custom tag or other devoted to common tasks. Take the example of iterating through each item in a loop. One framework might have a tag called "iterate," while another might have one called "loop."

One such framework came out of the Jakarta Apache taglib project, which attempted to define a common standard and implementation for a universally useful set of tag libraries. These contain functionality that is common to practically every JSP development: from basic control flow (iteration, conditions) to XML parsing to database access. Such was the popularity of these libraries that their tag definitions have been officially adopted as part of the JavaServer Page specification by Sun as the JavaServer Page Standard Tag Library — or JSTL.

You'll need an implementation. JSP containers (such as Tomcat) don't necessarily come with one already supplied. Fortunately, it's easy and free to acquire an implementation, which is also easy to install into most containers.

JSTL

The JSTL comprises five tag libraries:

- **core**: custom actions that do the programming "grunt work" — such as conditions and loops — and also fundamental JSP tasks such as setting attributes, writing output, and redirecting to other pages and resources
- **xml**: custom actions that alleviate much of the work in reading and writing XML files
- **sql**: custom actions dedicated to database manipulation
- **fmt**: custom actions for formatting dates and numbers, and for internationalization of text
- **function**: a set of standardized EL functions.

For the exam, you are required to know only about the core library. This is fortunate, for there are sixteen or so custom actions in the core library alone, and these form a

mini tag language in their own right. Of course, you're likely to want to explore the other three libraries because they are liable to prove useful on your projects—but that's the last on them from this chapter. (Phil Hanna surveys all four libraries in his book *JSP 2.0* [McGraw-Hill Osborne].)

You can download the specification for JSTL from the Sun web site—a very useful resource that goes beyond its brief as a specification, and is almost a developer manual. What you don't get from Sun is an implementation of the tags themselves, and this is *essential for the rest of the work in this book*! Do take the following steps—either right now or just before you undertake the next exercise:

1. To get hold of the standard reference implementation, visit:

   ```
   http://jakarta.apache.org/taglibs/binarydist.html.
   ```

2. Ensure that you download JSTL 1.1, which goes with JSP version 2.0—this is the appropriate level for the exam.
3. Extract the downloaded JSTL zip or tar file anywhere you like.
4. Make the crucial files from the extracted distribution available to your server installation. For Tomcat, I take the following step: Copy standard.jar and jstl.jar to `<TOMCAT INSTALLATION>/common/lib`.
5. After a server restart, the JSTL features should be available to you.

There is a previous and still supported standard for JSTL, which is 1.0. This goes with JSP level 1.2. Quite a few of the details—including, for example, the URIs to access the standard tag libraries—are different in the earlier implementation. Note that this book talks only about the 1.1 version of JSTL, which goes with JSP level 2.0. The exam doesn't cover the earlier version at all.

To make use of the core tag library in your own JSP pages, you must include a `taglib` directive (or namespace reference) containing the right URI:

```
http://java.sun.com/jsp/jstl/core
```

You can choose whichever prefix you like to use with these tags, although popular convention suggests the use of "c." So a complete `taglib` directive to include the core JSTL library might look like this:

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

Groupings of Actions

In all, there are fourteen actions contained in the JSTL core library. The JSTL specification splits these up into four groups: General Purpose actions, Conditional actions, Iterator actions, and URL-Related actions. I keep these same headings in the explanations that follow. This table briefly summarizes the actions in each group.

Group	Actions	Purpose

General Purpose	`<c:out>`
	`<c:set>`
	`<c:remove>`
	`<c:catch>`
		Manipulating attributes, controlling output to the JSPWriter, catching exceptions

Conditional	`<c:if>`
	`<c:choose>`
	`<c:when>`
	`<c:otherwise>`
		Control flow (branching)

Iteration	`<c:forEach>` `<c:forTokens>`	Control flow (looping)

URL-Related	`<c:import>`
	`<c:url>`
	`<c:redirect>`
	`<c:param>`
		Use of other resources

General Purpose Actions

There are four "general purpose" actions in the core library. These are for setting (and removing) attribute values in any scope, writing output, and catching exceptions.

<c:out> This action is used for writing expressions and template text to page output. You may well ask why you might need such a tag—after all, can't you include expressions and template text directly in your JSP page source? The answer is that you can, and for most purposes this tag is "surplus." However, `<c:out>` has a couple of neat, specialized features.

The first is the provision of a default value if the main value expression evaluates to **null**. Here's how it looks:

```
<c:out value=${user.name} default="User name not recognized" />
```

If the name property of the user attribute has a value, that value is displayed. If the name property evaluates to **null**, then the text "User name not recognized" will be displayed instead. As an alternative, the default value can be placed in the body of `<c:out>`. The result is the same, and with rejigged syntax, here's the alternative form for the previous example:

```
<c:out value=${user.name}>User name not recognized</c:out>
```

The second feature is the ability to escape XML-unfriendly characters, such as < and >. Entities are substituted instead, such as `<` and `>`. Whatever characters go to output—whether in the value or the default value—they are escaped in this way. This feature is enabled by default but can be invoked explicitly:

```
<c:out value=${xmlUnfriendlyAttribute} escapeXml="true" />
```

Or switched off:

```
<c:out value=${xmlUnfriendlyAttribute} escapeXml="false" />
```

The following table shows the list of characters that are converted through the XML-escaping facility.

Character	Converted To
<	<
>	>
&	&
'	'
"	"

on the job

It can be limiting to have the value you want to display with `<c:out>` constrained to an attribute. It's fine for simple expressions (`<c:out value= ${user.name} />`) but no use if you want the value to contain, say, other tags. Under those circumstances, you can use the body of `<c:out>` to set up

a default value, using tags or anything else you want. Then force the default value to display by placing a constant of* null *for the value. Here's an example:

```
<c:out value="${null}" escapeXml="true">
  <jsp:include page="xmlUnfriendly.jsp" />
</c:out>
```

Since the value attribute contains the null literal expressed in EL, the default value (contained in the body) is processed. In the body is a* `<jsp:include>` *standard action to include a JSP file. Because* `escapeXml` *is set to "true," the page will be incorporated, and any XML-unfriendly characters implied in the included file's name will be turned into entities.

`<c:set>` This is used for setting attributes in any scope. This tag is a convenient and lightweight alternative to the standard action combination of `<jsp:useBean>` and `<jsp:getProperty>`. Here's an example:

```
01 <c:set value="9.21" var="numerator" />
02 <c:set value="3" var="denominator" />
03 <c:set value="${numerator/denominator}" var="calculationResult" />
04 ${calculationResult}
```

The `var` attribute is used to name a variable, and the `value` attribute sets a value for that variable. The variable becomes an attribute in page scope by default. So in the example, at line 01 a page attribute called `numerator` is set with a literal floating point decimal value of 9.21, and on line 02 another called `denominator` with a literal integer value of 3. On line 03, `<c:set>` is used to set another variable called `calculationResult`. This time, an EL expression is used to set the value instead of a literal:

```
value="${numerator/denominator}"
```

Finally, an expression is used to display the value of the `calculationResult` page attribute—just to show that something really happened. If you try out the code, you should find that the output is 3.0700000000000003.

As an alternative to using the `value` attribute, you can place the value in the body of the tag. Here's an alternative to line 03 in the example:

```
03 <c:set var="calculationResult">${numerator/denominator}</c:set>
```

If you want to set up attributes in different scopes, `<c:set>` has a `scope` attribute for the purpose—taking the expected values of page, request, session, and application. Here is how `calculationResult` can be put into session scope:

```
03 <c:set var="calculationResult"
scope="session">${numerator/denominator}</c:set>
```

You are not restricted to using attributes in some scope. Any available object can be the object of a `<c:set>` action, provided that the object can be construed as a Java Bean. An alternative syntax is used. As an example, consider that HttpSession has a maxInactiveInterval property—by virtue of having `set` and `getMaxInactiveInterval()` methods. You can set this property using the following `<c:set>` syntax:

```
<c:set value="28" target="${pageContext.session}"
      property="maxInactiveInterval" />
```

The `target` attribute specifies the bean under consideration—here, session is available in EL through the implicit variable `pageContext`. The `property` attribute specifies the name of the property, maxInactiveInterval. The `value` attribute (which we've seen before) specifies the value for the property—in this case, 28 (seconds). As before, the body of `<c:set>` can be used to specify the value, instead of using the `value` attribute:

```
<c:set target="${pageContext.session}"
      property="maxInactiveInterval">28</c:set>
```

exam Watch

It's very important to note which tag attributes are allowed to take expressions (and so be evaluated dynamically), and which are not. There are tables at the end of each of the four groupings of actions (general purpose, conditional, iterator, URL-related) that summarize all the attributes for all the core tags and show which ones allow expressions.

<c:remove> This is the inverse of `<c:set>`, and it can be used to remove a scoped attribute. The action has only two attributes—`var` to name the variable, and the optional `scope` to specify the scope. As always, `page` is the default when `scope` is not specified. So to remove a variable, a statement such as

```
<c:remove var="calculationResult" />
```

may be all that's required, or one such as

```
<c:remove var="calculationResult" scope="session" />
```

when it is necessary to specify the scope.

<c:catch> This action allows you to catch an error in your page, without it propagating to (for example) an error page defined in web.xml.

At first sight, it appears like a catch block in Java, but it's more like a try/catch rolled into one, where the catch does nothing: All errors are suppressed. If you look at the generated servlet code after inserting a `<c:catch>` action, you'll see that the exception caught is a java.lang. Throwable—right at the top of the hierarchy. So you should use `<c:catch>` only for minor elements of your JSP page whose successful execution really doesn't matter.

The `<c:catch>` action relates only to anything that goes on in its body. So it suppresses a Java language ArithmeticException in the following example:

```
<c:catch>
  <jsp:scriptlet>int zero = 0; out.write(3/zero);</jsp:scriptlet>
</c:catch>
```

You can specify a named, page-scope attribute to represent the exception, and access this after the `<c:catch>` block. You use the `var` attribute to do this (there is no scope attribute; the exception dies with the page). So, adapting the example above,

```
<c:catch var="numException">
  <jsp:scriptlet>int zero = 0; out.write(3/zero);</jsp:scriptlet>
</c:catch>
<p>If there was an exception, the message is: ${numException.message}</p>
```

Attribute Name	Run-time Expression Allowed?	Mandatory?	Default Value	Type
<c:out>				
value	Yes	Yes	None	Object
escapeXml	Yes	No	True	**boolean**
default	Yes	No	Empty String	Object
<c:set>				
value	Yes	No	None	Object
var	No	No	None	String
scope	No	No	Page	String
target	Yes	No	None	Object
property	Yes	No	None	String
<c:remove>				
var	No	Yes	None	String
scope	No	No	Page	String
<c:catch>				
var	No	No	None	String

Conditional Actions

JSTL is provided with conditional actions, which give you the branching aspects of a programming language. As always, the moral is to keep the logic simple! These actions are designed to ease complexity, not to let you replicate full-blown Java syntax in JSTL.

First, we'll consider the `<c:if>` action, which tests a condition and executes its body when this condition is true. Then we'll consider the trio of actions `<c:choose>`, `<c:when>`, and `<c:otherwise>`, which work in consort in a way not dissimilar from the Java language switch statement.

<c:if> This action evaluates its body content if a condition is true. The condition is expressed in EL as the value for the test attribute. The following example shows how the syntax works:

```
<c:if test="${user.loyaltyPoints gt 1000}">
  <p>Welcome to one of our best customers!</p>
</c:if>
```

The `<c:if>` action doesn't have to have a body, which at first may seem pointless. However, you can store the result of the test in a scoped variable, thanks to the `var` and `scope` attributes. There is some point to this if you want to use the result of a test later, without repeating the test. This may make even more sense in the context of EL functions that we examine later in this chapter but that we'll introduce very briefly here. An EL function is a Java method, called with EL syntax. Let's look at an example:

```
<c:if test="${mytags:checkRole(user, 'Manager')}"
var="userInManagerRole" scope="session" />
```

The test uses an EL function called checkRole, which receives two parameters—a user name and a role name, one an attribute called "user," the other a String literal "Manager." The checkRole function returns a **boolean** result. If the function returns **true**, the intent is that the user is allowed to see or do things within the application that are off-limits to users who are not within the "Manager" role. The act of checking the role might be quite expensive—perhaps involving a secure network call to another machine hosting a user-role registry. But the result is held in a session attribute called "userInManagerRole," whose value is a java.lang.Boolean. So anywhere else within this session, the application can conditionally check this variable instead of reperforming the function. The following test makes use of this variable, and returns sensitive information only if `userInManagerRole` evaluates to true.

```
<c:if test="${userInManagerRole}">Salary field: $1,234,567</c:if>
```

`<c:choose>` In combination with `<c:when>` and `<c:otherwise>`, this action works something like a Java **switch** statement. However, these three tags behave a little differently and more flexibly than the Java equivalent.

The structure is this: The `<c:choose>` action is just a container for two possible other actions—`<c:when>` and `<c:otherwise>`. Between the opening and closing tags for `<c:choose>`, you can include only white space (any amount) or these two actions. The rules on including `<c:when>` and `<c:otherwise>` are as follows:

- `<c:when>`: there *must* be at least one occurrence.
- `<c:otherwise>`: optional—but if included, there cannot be *more than* one occurrence.

As we'll see, <c:when> executes a test—like <c:if>. Within the body of the <c:choose> each <c:when> test is performed in strict order of appearance. If a test is **false**, the body of <c:when> is ignored—again, just like <c:if>. If a test is **true**, the body of that <c:when> action is executed. *Anything* that follows within the <c:choose> action—either <c:when> actions or the <c:otherwise> action—will be ignored once a true test has triggered (and this is where the similarity to the Java switch action ends in that you use **break** statements to prevent execution of the statements within other cases or the **default** block). If *none* of the <c:when> tests evaluate to true, then and only then will the body of <c:otherwise> be executed.

<c:when> This has only a single, mandatory attribute: `test`. The test expression must evaluate to a boolean **true** for the body to be executed. The test expression won't break if the result evaluates to non-boolean; instead, the expression will be treated as equivalent to a boolean **false**.

<c:otherwise> This has no attributes. The body will be executed only when the preceding <c:when> actions within the enclosing <c:choose> all evaluate to **false**. Note that <c:otherwise> must be the last action in the <c:choose> group, coming after the final <c:when> action.

Here's a complete example that shows the trio of actions together:

```
<c:choose>
  <c:when test="${userInDeveloperRole}">
    Welcome, fellow developer.
  </c:when>
  <c:when test="${userInManagerRole}">
    You are a manager! I'll take the next bit slowly for you.
  </c:when>
  <c:otherwise>
    Hmm—I'm not sure what you are. Should I be talking to you?
  </c:otherwise>
</c:choose>
```

The example assumes the preexistence of a couple of attributes, `userInDeveloperRole` and `userInManagerRole`. Actually, it doesn't matter if these attributes don't exist, but the test they appear in will be treated as evaluating to **false**. Should the first <c:when> test prove true (because the attribute `userInDeveloperRole` has the value true), then the "welcome" text will display. If not, the next <c:when> test will be performed. If `userInManagerRole` has a value of true, the associated text for that is displayed. If that test evaluates to false, then the catchall text within the body

of the <c:otherwise> action will be displayed. If a user is both a developer and a manager, note that only the text in the first <c:when> will be displayed: Processing resumes at the end of the <c:choose> block after a positive test.

Attribute Name	Run-time Expression Allowed?	Mandatory?	Default Value	Type
<c:if>				
test	Yes	Yes	None	boolean
var	No	No	None	String
scope	No	No	Page	String
<c:choose>				
No attributes. Body may contain only one or more <c:when> actions, and zero or one <c:otherwise> action at the end of the block.				
<c:when>				
test	Yes	Yes	None	boolean
<c:otherwise>				
No attributes				

Iterator Actions

There are only two iterator actions—and less is better when it comes to exam preparation! They are also very similar: The <c:forEach> action is a general-purpose construct for looping, and <c:forTokens> has nearly the same syntax, but is specialized for splitting up Strings in the same way as the Java StringTokenizer class. There is bad news, however: These actions have an impressive number of attributes and, consequently, a few variant syntaxes. A few examples should set us right on these.

<c:forEach> This has two main uses: to iterate over a collection of objects (like a Java language "while" loop working with an Iterator or Enumeration), or to iterate a fixed number of times (like a Java language standard "for" loop). EL helpfully provides some ready-made implicit variables that are collections of objects—for example, ${headerValues}, which represents the collection of values for request headers. Here is a <c:forEach> loop that displays these values in an HTML table:

```
<table border="1">
<c:forEach var="hdr" items="${headerValues}">
  <tr><td>${hdr.key}</td><td>${hdr.value[0]}</td></tr>
</c:forEach>
</table>
```

The value for the `items` attribute must contain the collection object to loop around—in this case, a java.util.Map object (`${headerValues}`—see Chapter 7 for a full explanation of this EL implicit object). On each circuit of the loop, an object from this collection is placed in the variable represented by the value for the var attribute: `hdr`. Each object in a Map is a Map.Entry object, with `getKey()` and `getValue()` methods, exposing the two bean-like properties key and value. So the EL syntax `${hdr.key}` has the effect of getting the request header's key value and writing this to the current JspWriter. Map.Entry's `getValue()` method returns an Object—but here some inside knowledge is necessary. We know (from the EL specification) that each Map.Entry value object in the `headerValues` Map is a String array. EL allows the use of Java-language such as array syntax (`${hdr.value[0]}`) to obtain—in this case—the first available value for the named request header.

The `<c:forEach>` action allows many different types for the `items` attribute. These are listed in the following table, together with the corresponding type for the `var` attribute exposed.

Type for `items` Attribute	Corresponding Type for `var` Attribute
java.lang.Array (of some type of Object)	The declared type for the Array
java.lang.Array (of primitives)	The wrapper type (Integer, Double, etc.) corresponding with the declared primitive type (**int**, **double**, etc.) for the array
java.util.Collection, java.util .Iterator, java.util.Enumeration	Whatever type of Object is returned from the underlying Collection, Iterator or Enumeration (and remember there is nothing that forces all the Objects in a collection to be of the same type, beyond all being Objects)
java.lang.String	java.lang.String (the String in items is split up into a separate Strings for each comma encountered—like the tokens in StringTokenizer)

You don't have to use `<c:forEach>` with a collection of items. Alternatively, it can be used to perform a set number of iterations—much like a Java **for** loop. This is what happens in the following example:

```
<table border="1">
  <c:set var="num" value="1" />
  <c:forEach begin="1" end="128" step="2">
    <c:set var="num" value="${num + num}" />
    <tr><td>${num}</td></tr>
  </c:forEach>
</table>
```

The JSTL tag `<c:set>` is used to set a variable called *num* with a value of 1. Then the `<c:forEach>` loop begins. The loop is set up to begin at 1 and end at 128, in steps of 2—as indicated by the `begin`, `end`, and `step` attribute values. So the loop counter will initially have a value of 1, then 3, then 5, and so on up to 127. On the next iteration, the counter will have a value of 129, which is greater than the end value, so the loop ends. Within the loop, another `<c:set>` action is used to set the *num* variable to double its present value, and display this within a table cell.

exam Watch

The possibilities of `<c:forEach>` go further than space in this book allows to give a complete account. You can combine use of a collection (expressed in the `items` attribute) with the `begin`, `end`, and `step` attributes to pick out particular objects within the collection. You can also define a `varStatus` attribute that exposes information about each iteration—most usefully, a number representing the current round. For complete exam readiness, check the documentation in the JSTL 1.1. specification document.

<c:forTokens> `<c:forTokens>` is a specialized version of the `<c:forEach>` action, designed to perform String tokenization much like the StringTokenizer class does in straight Java language syntax. In fact, `<c:forEach>` will also do String Tokenization when the items attribute is set to a String but it is restricted to observing commas as delimiters only. There's no way to make `<c:forEach>` break up a String on encountering tabs or white space. This is where `<c:forTokens>` comes in. It has mostly identical parameters but differs in two important respects:

- The `items` attribute will accept only a String as a value (not Maps, Collections, etc.).
- An additional attribute, `delims`, is used to specify the delimiter to recognize when breaking up the String into tokens.

The following example expands on the first `<c:forEach>` example we saw a few moments ago.

```
<table border="1">
<c:forEach var="hdr" items="${headerValues}">
  <tr><td>${hdr.key}</td><td>${hdr.value[0]}</td></tr>
  <c:if test="${hdr.key eq 'Accept'}">
    <c:set value="${hdr.value[0]}" var="acceptValues" />
  </c:if>
</c:forEach>
</table>
<table border="1"><tr><th><b>Accept values</b></th></tr>
<c:forTokens items="${acceptValues}" delims="," var="value">
  <tr><td>${value}</td></tr>
</c:forTokens>
</table>
```

The first part of the code is unchanged—a `<c:forEach>` action is used to iterate through request header values supplied by the EL implicit variable `${headerValues}`. Most request headers have single values, but this is rarely true of the header `Accept`, which lists a range of file (MIME) types that the requesting client will accept back in the HTTP response. The `Accept` header could be represented as multiple request headers, each with an individual value (the request header mechanism allows for this). But more usually, the file types are returned as a comma-delimited list. This is where the `<c:forTokens>` action comes in handy.

In a change from the original code, a `<c:if>` action is used to spot when the Accept header is under consideration, and save its values using `<c:set>` to an EL variable called *acceptValues*:

```
<c:if test="${hdr.key eq ' Accept'}">
  <c:set value="${hdr.value[0]}" var="acceptValues" />
</c:if>
```

This *acceptValues* variable becomes the input to the `<c:forTokens>` action, as the value for the `items` attribute. Each value within *acceptValues* is recognized as separate from the next because comma is set as the value for the `delims` attribute. The `var` attribute is used to produce the *value* EL variable to display in a table cell on each iteration of the loop:

```
<c:forTokens items="${acceptValues}" delims="," var="value">
  <tr><td>${value}</td></tr>
</c:forTokens>
```

When I run this code in my browser, the result looks like this:

Accept values
image/gif
image/x-xbitmap
image/jpeg
image/pjpeg
application/vnd.ms-excel
application/vnd.ms-powerpoint
application/msword
application/x-shockwave-flash
/

Attribute Name	Run-time Expression Allowed?	Mandatory?	Default Value	Type
<c:forEach>				
var	No	No	None	String
items	Yes	No	None	Object
varStatus	No	No	None	String
begin	Yes	No	None	int
end	Yes	No	None	int
step	Yes	No	None	int
<c:forTokens>				
Var	No	No	None	String
items	Yes	Yes	None	String
delims	Yes	Yes	None	String
varStatus	No	No	None	String
begin	Yes	No	None	int
End	Yes	No	None	int
Step	Yes	No	None	int

URL-Related Actions

The final group of actions we need to consider in the JSTL core library is URL related. You can use these actions to import resources from a given URL, re-encode URLs, redirect to URLs, as well as pass additional request parameters where necessary.

<c:import> This is a souped-up version of `<jsp:include>`. It performs request-time inclusion of other resources. But whereas `<jsp:include>` is restricted to resources within the same web application, `<c:import>` can be used for resources in other contexts on the same server, or even to other servers entirely. Furthermore, the result of the import doesn't have to be sent directly to page output. Instead, you can store the result for later use. Some examples follow. First is the simple case, importing a resource from the same application, and placing that in page output:

```
<c:import url="trailer.jsp" />
```

You can see that the only attribute you need to set up is *url*, which takes a relative or absolute URL value. When the URL is relative, it can have a forward slash or not. The usual rules apply: Minus a forward slash, the URL is relative to the file doing the importing. With a forward slash, the URL is interpreted as starting from the context root of the current web application.

If you want to go outside of the current context, the syntax is this:

```
<c:import url="/rounding.jspx" context="/examp0801" />
```

The *context* attribute specifies the context root for the resource named in the *url* attribute. The context must reside (or at least be known to) the same container that houses the context for the file doing the importing. The important thing to remember is that for this syntax, both the *url* and *context* values must begin with a forward slash.

Absolute URLs can be used too. Here's a slightly longer example that introduces two new concepts—importing from an external resource using an absolute URL and storing the result in a variable:

```
<c:import url="http://c2.com/index.html" var="importedPage" scope="page" />
<jsp:scriptlet>String fiftyChars = ((String)
  pageContext.getAttribute("importedPage")).substring(0,50);
  pageContext.setAttribute("fiftyChars", fiftyChars);
</jsp:scriptlet>
<pre>${fiftyChars}</pre>
```

To specify an external URL, you simply include the protocol and server name in the value for the *url* attribute. By using the *var* and *scope* attributes, the specified URL is imported into a page-scoped attribute called *importedPage*. Next, a scriptlet takes the first fifty characters of the *importedPage* attribute value, setting this in an attribute called *fiftyChars*. Beyond the scriptlet, an HTML `<pre>` tag surrounds some EL that displays the *fiftyChars* attribute.

An alternative to *var* is the *varReader* attribute. With this, you have the opportunity to import a page into a Reader object. In Exercise 8-2, we'll use this feature to build a mini-filter on the imported data. You can only make use of the reader within the body of the `<c:import>` tag—beyond that, it's unavailable. The code to set up a reader for use might look like this:

```
<c:import url="http://c2.com/index.html" varReader="myReader">
<!—Make use of myReader in the body of the import action—>
...scriptlet goes here...
</c:import>
```

<c:url> This works out a URL string for inclusion in page output. By default, the URL generated through this action is sent directly to page output—which is rarely useful. More usually, you make use of the *var* attribute to store the URL for later use—perhaps in an EL expression for the *href* parameter value for an HTML link, as shown here:

```
<c:url value="/rounding.jspx" context="/examp0801" var="myLink" />
<a href="${myLink}">Link to another page in another context</a>
```

The syntax has a lot in common with `<c:import>`. The *value* attribute replaces the *url* attribute. In the example, a different context (/examp0801) is specified with the *context* attribute, so the URL in the *value* attribute is interpreted beginning at the root of that context. The *var* attribute stores the generated URL in the named attribute (*myLink*) as a String. As usual, this attribute defaults to the page context—but by using the *scope* attribute (not shown), you can place the attribute holding the URL in any scope you like.

You may well think that it would be more straightforward to put the link in directly: Why bother with `<c:url>`? So the anchor link above would simply read

```
<a href=“/rounding.jspx”>Link to another page in another context</a>
```

For a start, this wouldn't get to the other context. The value for *href* would need to be an absolute URL (such as `http://myserver.com/examp0801/rounding`

`.jspx`) or a contorted relative URL (such as `../examp0801/rounding.jspx` – the .. to go "up one level" to escape the current context). Moreover, `<c:url>` is an improvement because it seamlessly rewrites URLs whenever necessary to include the session ID (recall the discussion in Chapter 4 about how you need to rewrite URLs when cookies are banned on the client to maintain session identity).

<c:redirect> This causes the client to redirect to a specified URL. This action uses the `HttpServletResponse.sendRedirect()` method under the covers (we talked about that in Chapter 2). There's one compulsory attribute (*url*) and one optional one (*context*). The URL can be a complete URL, including the protocol and host name and port number:

```
<c:redirect url="http://localhost:8080/examp0801/rounding.jspx" />
```

At the other extreme, *url* can have a value that doesn't begin with a forward slash. Then it is interpreted relative to the directory in which the page doing the redirection is located:

```
<c:redirect url="iteration.jspx" />
```

You can begin the *url* value with a forward slash, in which case it is interpreted as starting at the context root:

```
<c:redirect url="/iteration.jspx" />
```

There's also an option to supply another context on the same server, by using the context attribute. Suppose you point your browser to a JSP using the following URL:

```
http://localhost:8080/examp0802/redirecting.jsp
```

You can see from the URL that the context root is `examp0802`. Within the page, `redirecting.jsp`, `<c:redirect>` is used like this:

```
<c:redirect url="/rounding.jspx" context="/examp0801" />
```

In this case, the value for *url* must begin with a forward slash—because it begins from the context specified. In other words, `<c:redirect>` composes a URL like this to cause your browser to repoint to the specified page:

```
http://localhost:8080/examp0801/rounding.jspx
```

One final point: Like the `<c:url>` action, `<c:redirect>` has the capacity to rewrite URLs to include the session ID—when it needs to.

<c:param> Used to attach parameters to any of the previous three URL actions `<c:import>`, `<c:url>`, or `<c:redirect>`. The following example expands the original `<c:url>` example to add some parameters:

```
<c:url value="/rounding.jspx" context="/examp0801" var="myLink">
  <c:param name="firstName" value="David" />
  <c:param name="secondName">Bridgewater</c:param>
</c:url>
<a href="${myLink}">Link to another page in another context</a>
```

Attribute Name	Run-time Expression Allowed?	Mandatory?	Default Value	Type
<c:import>				
url	Yes	Yes	None	String
context	Yes	No	None	String
var	No	No	None	String
scope	No	No	Page	String
charEncoding	Yes	No	ISO-8859-1	String
varReader	No	No	None	String
<c:url>				
value	Yes	No	None	String
context	Yes	No	None	String
var	No	No	None	String
scope	No	No	Page	String

EXERCISE 8-2

JSTL

In this exercise you'll build a mini-search engine. It won't quite be on the scale of Google or Yahoo, but it will make use of a wide range of the JSTL actions you've learned about in this section of the chapter. You'll need to have Internet access to make this exercise work properly.

For this exercise, go back to creating the usual web application directory structure this time under a directory called ex0802, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch08/ex0802.war. This is a moderately difficult exercise, so don't feel any shame in deploying and looking at the solution code.

A big health warning is required at this point: *For this example to work, and for most of the subsequent examples in the book, you have to have installed an implementation of JSTL.* The steps for doing this are described at the beginning of the JSTL section in this chapter.

Create a File of URLs

1. Create a file directly in the web application root ex0802 called urls.txt. Type in a set of genuine URLs, each separated by a carriage return—for example,

```
http://c2.com/index.html
http://www.ibm.com
http://www.microsoft.com
http://www.osborne.com
http://java.sun.com
```

Create a JSP Document for Entry of a Search Word (search.jspx)

2. Create a file directly in the web application root ex0802 called search.jspx.
3. The page will be an HTML document. In the root `<html>` element, include namespaces for standard actions (`http://java.sun.com/JSP/Page`) and the JSTL core library (`http://java.sun.com/jsp/jstl/core`).
4. Include the following standard actions to ensure you get standard HTML output:

```
<jsp:output omit-xml-declaration="true" />
<jsp:directive.page contentType="text/html" />
```

5. In the body of the HTML page, include a heading saying "The following URLs will be searched:". Use the core JSTL `import` action to import urls.txt (the file you created in step 1) into a variable called *searchUrls*.
6. After this, use the core JSTL `forTokens` action to split up the imported content of the urls.txt file. The *items* attribute will be the *searchUrls* variable you created in step 5 (use EL to represent this). Each line in the original file is a separate URL, so the delimiter (*delims* attribute) should be the carriage return for your platform (on the Windows platform I'm using, that's carriage return and line feed, so `

`). Retrieve each line into a variable called *currentURL*. Display each URL found in a separate row of an HTML table. Don't forget to close the `forTokens` action with an appropriate end tag.
7. Under this table, introduce a conventional HTML form. The action of the form should be "searchResults.jspx." Have a text field in the form named "searchWord" and a submit button. This is where the user will type a search word for matching against text in the list of URLs. Optionally, express the action of the form as an EL variable—and preload this by using the core JSTL `url` action.

Create a JSP Document to Perform the Search and Display the Results

8. Create a file directly in the web application root ex0802 called search.jspx.
9. Repeat steps 3 and 4 for this file, to reference the right namespaces and ensure HTML output.
10. Use the JSTL actions `import` and `forTokens` exactly as you did in the search.jspx file, to iterate through the URLs in the urls.txt file. All the remaining steps for this document should be inserted within the `forTokens` loop.
11. Use the `import` action again, but this time to import the current URL (`${currentURL}`) into a variable called *currentFile*.
12. Insert the following scriptlet (code supplied here!), which scours the current file for the search word passed in as a parameter, and sets up an attribute called *isFound* to indicate whether the word has been found in the

file or not. To save typing, you can copy in the following lines of code from the electronic version of the book (or the solution code):

```
<jsp:scriptlet>
  // Get hold of the search word, and the file-as-String
  String searchWord = request.getParameter("searchWord").toLowerCase();
  String file =
    ((String) pageContext.getAttribute("currentFile")).toLowerCase();
  // Is the search word in the file?
  int foundAt = file.indexOf(searchWord);
  if (foundAt >= 0) {
    // Yes, it's found
    pageContext.setAttribute("isFound", new Boolean(true));
    // Find a position in the file a little bit before the search word
    int startAt = 0;
    if (foundAt > 50) {
      startAt = foundAt-50;
    } else {
      startAt = 0;
    }
    pageContext.setAttribute("startAt", new Integer(startAt));
  } else {
    // No, the search word isn't found in the file
    pageContext.setAttribute("isFound", new Boolean(false));
  }
</jsp:scriptlet>
```

13. Use the JSTL `if` action to test the *isFound* attribute set up by the previous scriptlet. All the remaining steps take place within the `if` action (the closing tag for `</c:if>` comes immediately before the closing tag for `</c:forTokens>`). This next part of the JSP document should be accessed only if *isFound* is true.
14. Use the JSTL `import` action to import the same file again, but this time into a Reader variable.
15. Insert the following scriptlet, which returns 200 characters of the file surrounding the search word. The idea is to present a small part of the content to the user of this application to place the search word in some kind of context—this is placed in an attribute called *firstBitOfFile* so that it can easily be accessed using EL later in the code. As an aside—note how

this scriptlet is surrounded in XML CDATA syntax. Why should this be? (Answer at the end of the exercise!)

```
<jsp:scriptlet><![CDATA[
  // Now you have the file as a Reader - skip to where you
  // want to start at, a little before the found search word.
  Reader r = (Reader) pageContext.getAttribute("currentReader");
  long skipAmount = ((Integer)
pageContext.getAttribute("startAt")).longValue();
  r.skip(skipAmount);
  int i;
  char c;
  int counter = 200;
  StringBuffer sb = new StringBuffer();
  try {
    // Just read a couple of hundred characters
      while ((i = r.read()) > -1 &&—counter > 0 ) {
      c = (char) i;
        if (c == '<') {
        sb.append("&lt;");
      } else if (c == '>') {
        sb.append("&gt;");
  } else {
    sb.append(c);
  }
    }
} catch (IOException e) {
    e.printStackTrace();
}
  pageContext.setAttribute("firstBitOfFile", sb.toString());
]]></jsp:scriptlet>
```

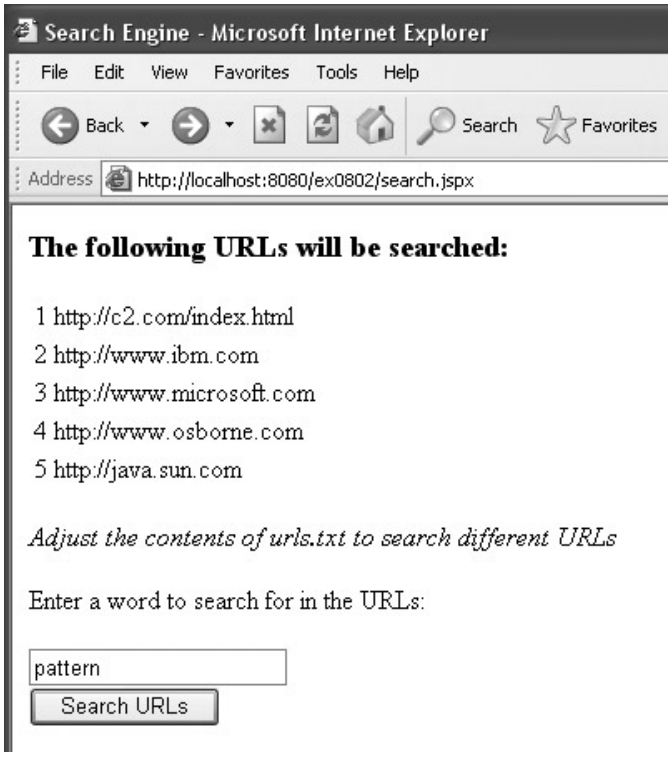
16. Using EL variables, display the current URL (preferably as a hyperlink) and the first bit of the file within table cells in an HTML table row.

Run and Test Your Code

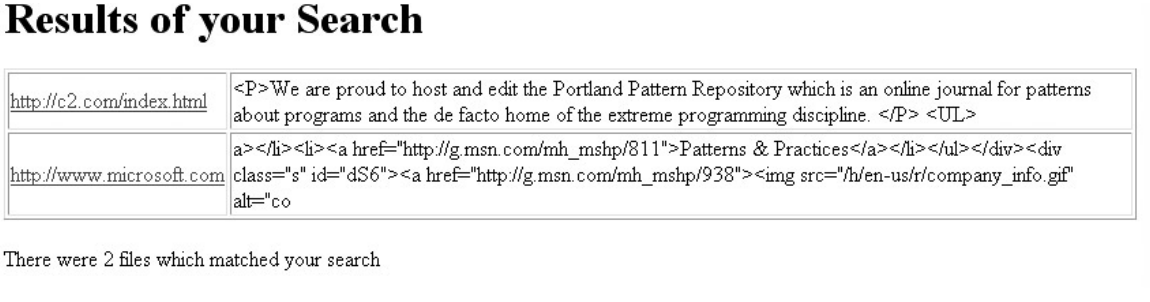
17. Create a WAR file that contains the contents of ex0802, and deploy this to your web server. Start the web server if it has not started already.
18. Use your browser to request search.jspx, with a URL such as

    ```
    http://localhost:8080/ex0802/search.jspx
    ```

19. Make sure that the URLs to be searched are displayed. Your page should look something like this:



20. Enter a search word, and click the submit button. The resulting page (searchResults.jspx) should look something like this:



Results of your Search

http://c2.com/index.html	<P>We are proud to host and edit the Portland Pattern Repository which is an online journal for patterns about programs and the de facto home of the extreme programming discipline. </P> <UL>
http://www.microsoft.com	a> Patterns & Practices <a href="http://g.msn.com/mh_mshp/938"><img src="/h/en-us/r/company_info.gif" alt="co

There were 2 files which matched your search

21. Remember the question earlier: Why is the scriptlet code surrounded in XML CDATA (character data) syntax? The answer is that you are writing XML documents, which have to contain legal XML. Because the code in the

second Java scriptlet contains characters that are XML-hostile (such as "<" in the `if` statement), it has to be demarcated as character data that XML parsers will effectively ignore.

CERTIFICATION OBJECTIVE

EL Functions (Exam Objective 7.4)

Given a scenario, write EL code that uses a function; write code for an EL function; and configure the EL function in a tag library descriptor.

EL functions can be used almost as an alternative to custom tags. Many things that you can accomplish with a custom tag can be achieved with an EL function instead. The payback is in simplicity—the development process is easier, and the use of the function within the JSP page is as easy as using a custom tag.

EL Functions

We'll take the rounding tag that we met at the beginning of this chapter and recast it as an EL function. The process of developing and using an EL function very closely parallels the same process for custom tags. There are four very similar stages:

1. Writing a Java class containing the method underpinning the EL function
2. Defining the function within a tag library definition (TLD) file
3. Providing details of where to find the TLD file in the deployment descriptor, web.xml
4. Referencing the TLD file in your JSP page source and using the EL functions from it

We'll look at these steps in turn in the headings that follow.

Writing Methods for EL Functions

Any class will do as a repository for EL functions. The only requirement of a *method* that acts as an EL function is that it should be declared (1) public and (2) static. This means that any existing public static method in any class—either of your own,

or as part of your Java environment—is already a candidate for EL function-hood. Most of the functions in java.lang.Math, for example, can be made available as EL functions without having to write a line of code—you simply follow steps 2 to 4 within the EL function-making process.

Here is the logic of the rounding tag we met in the first part of this chapter, rendered in a public static method for use as an EL function:

```
package webcert.ch08.examp0803;
public class Rounding {
  public static double round(double figure, int decimalPlaces) {
    /* Do the rounding */
    long factor = (long) Math.pow(10, decimalPlaces);
    // Shift decimal point to right...
    figure *= factor;
    // Do the rounding...
    long interimResult = Math.round(figure);
    /* Shift decimal point to left (cast of numerator to double
    * because you don't want
    * integer division to occur, and then promote to double) */
    double output = ((double) interimResult) / factor;
    return output;
  }
}
```

As you can see, there is nothing special about this class that makes it specific to the world of tags and JavaServer pages. You can use any prebuilt class—including those supplied with the Java environment.

exam Watch

You might have thought that returning nothing (void) from an EL function is illegal. However, there's nothing wrong with it—provided you state that void is returned from the function signature in the TLD:

```
<function-signature>void voidFunction(int)</function-signature>
```

Why might you want to do this? You might need to call a function that causes some effect in your application (sets up some attributes, perhaps) but shouldn't or mustn't return anything to the JSP page.

The class that implements an EL function has two usual locations—as a straight class file in WEB-INF/classes, or wrapped up in a JAR in WEB-INF/lib.

Defining the Function within the TLD File

Having written (or chosen) the class and method for your EL function, the next step is to define it—in a TLD file. You use the `function` element to do so. The definition for our rounding function might look like this:

```
<function>
  <description>Rounds figure to given number of decimal places</description>
  <name>round</name>
  <function-class>webcert.ch08.examp0803.Rounding</function-class>
  <function-signature>double round(double, int)</function-signature>
</function>
```

The subelements of `<function>` are as follows:

<description> This is an optional description for the function.

<name> This is the name of the function as it will be used within EL expressions. This doesn't have to match the Java method name—it can be any logical name.

<function-class> This is the fully qualified name of the Java class containing the implementation of the function.

<function-signature> This is the signature of the function, almost as it would appear in regular Java syntax. The similarities and differences are these:

- Qualifiers are omitted (after all, the method must be public and static, so these keywords would be redundant).
- A return type must be supplied (or **void**)—just like Java. If the function returns an object type (rather than a primitive), the fully qualified name of the class must be given. This applies even to classes in java.lang (java.lang.String, not just String).

- The method name follows, and this has to match the method name in the Java class. Just as in Java, a pair of parentheses follows the method name—to enclose the parameters.
- Parameters must be listed in order, following the order within the Java method. However, only the types are provided—the parameters don't have an accompanying name in the TLD signature. Primitives are listed as in Java, and—as for the return type—objects must have their fully qualified name.

Finding the TLD

Now we're at the point where we need to put the TLD file somewhere where pages in the web application can find it. This process is absolutely no different from locating a TLD for custom tags, as we explored at the beginning of the chapter. Refer back to that for an account of what to do.

exam Watch

You don't need a separate TLD for EL functions, different from a TLD that contains custom tags. You can mix EL functions and custom tags (and, as we'll see later, other elements as well) in the same TLD. The only rule is on names: Each must be unique across all functions, custom tags, and other top-level elements within the TLD.

Declaring the TLD and Using the EL Function

We're finally ready to use the EL function within a JSP page! This means two things:

1. Declaring the TLD with the page
2. Using the rounding function

Your options for declaring the TLD file have already been covered at the beginning of this chapter: You can either use the `taglib` directive (for JSP syntax) or reference the taglib in a namespace (JSP document XML syntax).

However, using the function is different. You use EL rather than tag syntax. Our call to the `round` function is shown here in its simplest form, plugging in constants for the parameters to the function:

```
${mytags:round(9.21 / 3, 2)}
```

As is customary for EL, the whole statement is surrounded by `${...}`. Within the curly braces, first comes the function name—declared just like a tag name: `mytags:round`. Immediately following the function name are parentheses to contain the parameters—just as in Java language method calls. With the parentheses, each parameter is separated by a comma from the next. So the first parameter—the figure to round—is a constant calculation (`9.21 / 3`), and the second parameter—the number of decimals to display—is a constant 2.

Here's another example of the use of the `round` function, in a more realistic setting. Parameters to functions can be run-time expressions. Take a look at the following example:

```
<c:set var="unrounded" value="${param.num/param.denom}" />
${mytags:round(unrounded, 2)}
```

First of all, the JSTL action `<c:set>` is used to load an attribute called *unrounded* with the value from an EL calculation. The EL calculation performs a division based on two request parameters: one called *num* and the other called *denom*. Then the *unrounded* variable is plugged in to the EL `round` function—as the first parameter.

EXERCISE 8-3

EL Functions

In this exercise we'll rewrite the Taxation Tag you met in Exercise 8-1 as an EL function. You'll reuse some of the pieces of that exercise but also add some new components. The functionality is identical—you enter some details about income, tax allowance, and tax rate on a web page; click on a submit button; and then see a JSP document that presents a calculated net income figure. Refer to Exercise 8-1 for screen shots of how the finished application should look.

Copy Files from Exercise 8-1

1. First, create a context directory called ex0803 with the usual web application subdirectories.

2. From Exercise 8-1, copy the files income.html and taxation.jspx from the context directory ex0801, and paste them directly into your new context directory ex0803. You'll leave income.html exactly as it is and make only one small change to taxation.jspx later in the exercise.
3. Also from Exercise 8-1, copy the deployment descriptor web.xml from ex0801/WEB-INF to ex0803/WEB-INF.

Create the EL Function Code

4. In /ex0803/WEB-INF/src, create a package directory called webcert/ch08/ex0803, and in that create a file called Taxation.java (belonging to package webcert.ch08.ex0803).
5. Include a method in this class called `calcNetIncome`, with the following signature:

```
public static String calcNetIncome(double gross,
double rate, double allow, String currency)
```

6. This method accepts four parameters, as shown, and should return a formatted net income figure (complete with correct currency symbol). You can write the code to do this yourself, though — of course — it has no direct bearing on the SCWCD exam. You may prefer to cheat and use the following code:

```
double taxToPay = ((gross–allow) * rate / 100);
double net = (gross–taxToPay);
NumberFormat nf = NumberFormat.getInstance();
Currency c = Currency.getInstance(currency);
nf.setCurrency(c);
nf.setMinimumFractionDigits(c.getDefaultFractionDigits());
nf.setMaximumFractionDigits(c.getDefaultFractionDigits());
return c.getSymbol() + nf.format(net);
```

7. Compile the code to directory /ex0803/WEB-INF/classes/webcert/ch08/ex0803.

Declare the Function in a TLD File

8. You have now to link the static method in the class you have just written to a function definition in a tag library descriptor file.

9. Create an empty file called mytags.tld in directory /ex0803/WEB-INF/tags.
10. Edit the file, and provide an enclosing `<taglib>` element. Make sure that you include any necessary attributes in the taglib opening tag, and also add mandatory subelements (for the tag library version and short name). Refer to Chapter 7 for details on "heading level" information on tag library descriptors. If you need to, use the tag library descriptor from Exercise 8-1 as a template for this one (it's also called mytags.tld, and is located in /ex0801/WEB-INF/tags).
11. Include a function element in the TLD, referring to the preceding material in this chapter to remind yourself how the syntax of the function element works.
    - The description can be anything you like.
    - The name should be `netincome`.
    - The function class should tie in to the Taxation class you wrote, webcert.ch08.ex0803.Taxation.
    - The function signature should match the `calcNetIncome` function. Remember that parameter types don't need names and that all nonprimitive parameters and return types (even those from java.lang) need to have a fully qualified name.
12. Save and close the TLD file.

Adjust the JSP Document

13. In the JSP document you copied from Exercise 8-1 (taxation.jspx), you already had a reference to the TLD document mytags.tld, with a prefix of mytags.
14. So you have only one thing to change. Currently, the document uses the tag declaration `<mytags:netincome>` to display the calculated net income. Remove this, prior to doing the following steps to replace it with an EL function call.
15. Place the function call to `mytags:netincome` in EL syntax.
16. Remembering that you navigate to this document from an HTML page containing a form, add four parameters as follows:

- The first should be from a request parameter called `gross`.
- The second should be from a request parameter called `rate`.
- The third should be from a request parameter called `allowance`.
- Hard-code the fourth (the currency code) as "GBP."

17. Hint: Remember that EL has an implicit variable for obtaining parameter values.

Run and Test the Application

18. Create a .war file from your ex0803 context directory, and deploy this to your server.
19. Test your application with a URL such as

```
http://localhost:8080/ex0803/income.html
```

20. If you need to, refer to Exercise 8-1 to see how the pages in the application should look.

CERTIFICATION OBJECTIVE

The "Classic" Custom Tag Event Model (Exam Objective 10.1)

Describe the semantics of the "Classic" custom tag event model when each event method (doStartTag, doAfterBody, and doEndTag) is executed, and explain what the return value for each event method means; and write a tag handler class.

We have looked at a number of JSP technologies now that rely on tag technology—standard actions, Expression Language, and JSTL actions. Formerly—without EL and JSTL—you would always have to write your own custom tags (or acquire some

from a third party) to avoid Java scripting. But although EL and JSTL are very flexible, there will come a time when the best solution to keeping your page Java-free is a custom tag. Hence, custom tags are still alive and well on the exam syllabus, and although they represent the most complex end of JSP technology, they are satisfying to write and powerful in scope.

In this section we'll tackle tags head-on by looking at the so-called "classic" custom tag event model. This is technology that has been around for a long time, featured in the older version of the SCWCD exam and still in the current one. In the next chapter, you'll meet newer technologies that make it possible to retain most or all of the power of classic tags, but with easier development approaches. But to appreciate these properly, you need to work through some classic tags first!

Tags for All Seasons

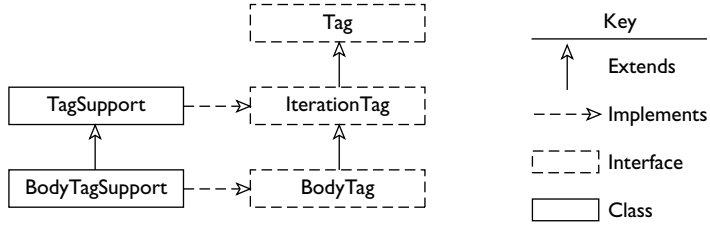
In actual fact, the classic custom tag event life cycle isn't a single life cycle. There are three possibilities you can choose from, based on three interfaces, all in the javax.servlet.jsp.tagext package: Tag, IterationTag, and BodyTag. Each of these extends the other, adding new possibilities into the basic life cycle:

- Tag is the simplest of the three.
- IterationTag extends Tag, adding (no surprises!) some looping capabilities.
- BodyTag extends IterationTag, adding further method calls to manipulate the body content of the tag (instead of just blindly appending more output to the page, as the other types do).

You can build your own custom tags by implementing any one of these three interfaces. More usually, you extend classes provided in javax.servlet.jsp.tagext that already implement these interfaces, and provide some useful base functionality. Figure 8-3 is a class diagram that shows the relationship among these classes and the interfaces already mentioned.

FIGURE 8.3

Tag Interfaces and Classes in javax.servlet.jsp.tagext



You can see from Figure 8-3 that there isn't a class that just implements the Tag interface. Your choices are TagSupport (implementing IterationTag as well as Tag) and BodyTagSupport (implementing everything possible).

Tag

If all you want to do is make the tag do something—you're not interested in the body at all—then the Tag interface is for you. Let's consider the case where you have so little interest in the body that you elect to leave it empty. The tag is there just to substitute some text or other into your page output. Perhaps you would like to insert the current date and time. You could use an expression such as this:

```
<%= new Date() %>
```

Here's how you might approach a tag replacement for this. First of all, you might write a class that implements the Tag interface. Here's the source for this:

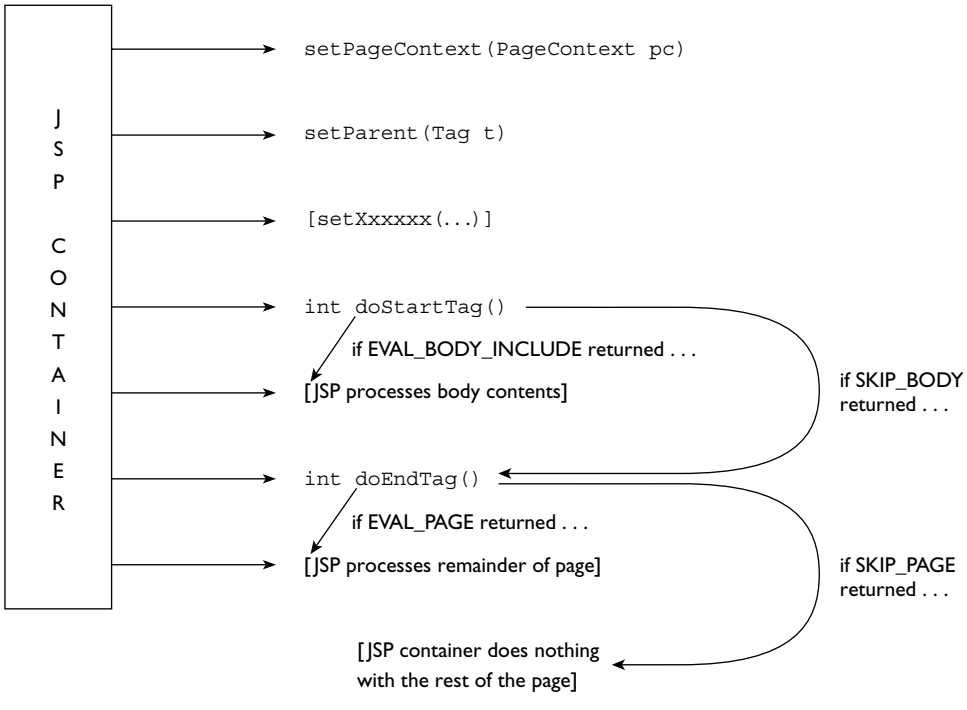
```
import java.io.IOException;
import java.util.Date;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.Tag;
public class DateStampTag1 implements Tag {
  private PageContext pageContext;
  private Tag parent;
  public void setPageContext(PageContext pageContext) {
    this.pageContext = pageContext;
  }
  public void setParent(Tag parent) {
    this.parent = parent;
  }
  public Tag getParent() {
    return parent;
  }
  public int doStartTag() throws JspException {
    dateStamp();
    return Tag.EVAL_BODY_INCLUDE;
  }
  public int doEndTag() throws JspException {
    return Tag.EVAL_PAGE;
  }
```

```
  protected void dateStamp() throws JspException {
    JspWriter out = pageContext.getOut();
    try {
      out.write("<i>" + new Date() + "</i>");
    } catch (IOException e) {
      throw new JspException(e);
    }
  }
  public void release() {
  }
}
```

This is more or less as simple as classic tag source gets. Let's examine the methods in turn. In the source, they appear roughly in "tag life cycle" order. The life cycle is illustrated in Figure 8-4.

FIGURE 8-4 JSP container processing one occurrence of a tag implementing the Tag Interface in one JSP page:

Custom Tag Life Cycle (1): The Tag Interface



- First of all, the JSP container makes an instance of the DateStampTag1 class. The class must have a no-argument constructor—either explicitly defined or (as here) provided by the compiler in the absence of other constructors.
- Next, the JSP container calls `setPageContext(PageContext pc)`. This gives the tag an opportunity to save a reference to the page context of the JSP page that contains the tag, and this is exactly what happens here. DateStampTag1 has been defined with a javax.servlet.jsp.PageContext instance variable. The page context so provided lets the remaining code in the tag handler class do all the things that might otherwise be done using the *pageContext* implicit variable in a scriptlet.
- After this, the JSP container calls `setParent(Tag parent)`. Again, the idea is to save a reference for later use—this time to the tag that immediately encloses this tag. Although DateStampTag1 makes no use of its parent tag, it has to provide the method to properly implement the Tag interface. So DateStampTag1 does the right thing, by defining an instance variable of type Tag, and saving the parameter passed into this method for later use.
- Again to satisfy the Tag interface, the `getParent()` method (returning the parent Tag—or **null**) must be defined. This isn't called by the container, but it can be used by a later method within the tag handler class.
- Now the JSP container calls any other set methods on DateStampTag1 that relate to attributes for the tag. As it happens, there aren't any; we'll introduce an attribute in the next version of the tag handler coming up soon (imaginatively called DateStampTag2).
- Next the JSP container calls `doStartTag()`. You can think of this as corresponding to the point where the JSP container processes the appearance of datestamp's opening tag within the JSP page source. In this case, `doStartTag()` calls its own internal method—`dateStamp()`—to do the work of writing the date to page output. After doing this, `doStartTag()` must return an **int** value—this is a return code to tell the JSP container what to do next. Valid values are defined as **public final static int** data members within the Tag interface. There are two possibilities:
  - Tag.EVAL_BODY_INCLUDE—any JSP page source between the opening and closing tags for this action should now be processed.
  - Tag.SKIP_BODY—the exact opposite: Any JSP page source between the opening and closing tags for this action should be ignored. The JSP container proceeds directly to the `doEndTag()` method for this action.

- Dealing with the `dateStamp()` method: This is the only method in the DateStampTag1 class that is not an implementation of a Tag interface method. It does the specific work associated with this tag: namely, getting hold of the JspWriter from the *pageContext* variable and writing a new date and time to it. This is equivalent to using the implicit variable *out* in a scriptlet, but with the benefit that this tag handler class is completely separate from JSP page source.
- Next, the JSP container calls `doEndTag()`. In this version of DateStampTag1, this does nothing functional. However, the method is still obliged to send back a return code to the container. Again, there are two options:
  - Tag.EVAL_PAGE tells the JSP container to process the rest of the page after the closing tag.
  - Tag.SKIP_PAGE effectively tells the JSP container to abort the rest of the page following the closing tag.
- Finally, the JSP container calls `release()`. However, this won't happen every time the tag appears in a JSP page. The instance of the tag is potentially reused over and over again. If the container decides to take a particular instance of a tag out of service, then `release()` is called, which you can use to clean up any expensive resources associated with the tag.

exam Watch

Instances of tag handler classes are reused—at least, potentially. So you don't necessarily get a new instance with every use of a particular tag on your JSP page, or even across JSP pages. That means that you can't rely on the constructor to do initialization that must take place for each specific use; it's far better to use `setPageContext()` for that, which will get executed for every tag appearance. Apart from the no-argument constructor and `release()`, all other Tag life cycle methods are executed for each appearance of the tag on the page, as shown in Figures 8-4, 8-5, and 8-6.

The difficult work—writing the tag handler class—is now done. However, there are additional things to do before we can use the datestamp tag within a page. For one thing, we must define the tag within a TLD file. We've seen how functions are defined already in a TLD file, and tags are—if anything—easier. Here's how the tag element looks (to save space, some mandatory taglib attributes are omitted, as are the top-level elements that go with tag libraries):

```
<taglib ...taglib attributes...>
<... elements omitted ...>
<tag>
  <name>dateStamp1</name>
  <tag-class>webcert.ch08.examp0804.DateStampTag1</tag-class>
  <body-content>empty</body-content>
</tag>
</taglib>
```

Each tag must have a `<tag>` element defined. The tag must have at least the three subelements shown:

- `<name>`: a unique name within the tag library. This name must remain unique not only within other tags but also within any EL functions or tag files defined in the tag library as well.
- `<tag-class>`: like `<function-class>` for EL function descriptors, or `<servlet-class>` for servlets—the fully qualified name of the tag handler class (but omitting the file extension ".class").
- `<body-content>`: the sort of content permitted in the body of the tag—between the opening and closing tags. There are four valid values: empty, scriptless, tagdependent, and JSP. In this case, dateStamp1 is defined as empty, so there can't be any body. The tag can appear in the page either as `<prefix:dateStamp1></prefix:dateStamp1>`, or—more usually—using the "self-closing" form `<prefix:dateStamp1 />`.

We covered the correct locations for the TLD file in the first section of this chapter.

So that leaves actually using the tag within the JSP page. The procedure is the same as for EL functions. Here is a complete JSP document that uses the dateStamp1 tag:

```
<html xmlns:mytags="http://www.osborne.com/taglibs/mytags"
      xmlns:jsp="http://java.sun.com/JSP/Page">
<jsp:output omit-xml-declaration="true" />
<jsp:directive.page contentType="text/html" />
<head><title>Date Stamp Example</title></head>
  <body>
    <h1>Date Stamp Example</h1>
    <p>Date Stamp 1: <mytags:dateStamp1 /></p>
  </body>
</html>
```

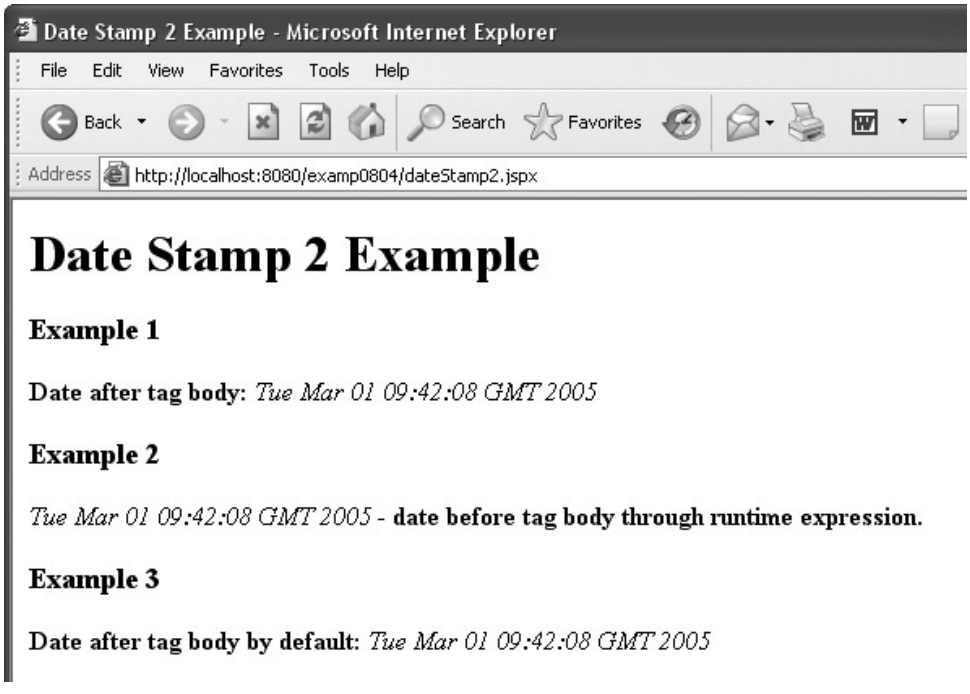
If you run this page, the output is as shown in the following illustration. As you can see, all you have to do to make the date stamp appear is to locate the tag within the template text at the right point. The JSP page container will run the code and substitute the output from the tag handler code into the page output at the right place.

Just because a tag that only implements the Tag interface doesn't have a way of manipulating the body is not to say that such a tag *can't* have a body. Let's now look at a modified version of the date stamp tag, which allows a body and demonstrates the use of an attribute associated with the tag. In this version, the tag logic has changed in one important respect. Dependent on the setting of an attribute called *beforeBody*, the location of the date stamp will change in respect of the body. We'll first look at how this works in action, then revisit the code in the tag handler. Here's the JSP page using the modified tag, dateStamp2:

```
<html xmlns:mytags="http://www.osborne.com/taglibs/mytags"
      xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c="http://java.sun.com/jsp/jstl/core" >
  <jsp:output omit-xml-declaration="true" />
  <jsp:directive.page contentType="text/html" />
  <head><title>Date Stamp 2 Example</title></head>
  <body>
    <h1>Date Stamp 2 Example</h1>
    <h3>Example 1</h3>
    <p><mytags:dateStamp2 beforeBody="false">
      <b>Date after tag body: </b>
```

```
    </mytags:dateStamp2></p>
    <h3>Example 2</h3>
    <c:set var="trueVariable" value="true" />
    <p><mytags:dateStamp2 beforeBody="${trueVariable}">
      <b> – date before tag body through runtime expression.</b>
    </mytags:dateStamp2></p>
    <h3>Example 3</h3>
    <p><mytags:dateStamp2>
      <b>Date after tag body by default: </b>
    </mytags:dateStamp2></p>
  </body>
</html>
```

This is what the output looks like when you access the document.



You can see that the dateStamp2 tag has one optional attribute, called *beforeBody*. When this is set to true, the date stamp appears before the body; and when set to false, after the body. Example 1 is the first of three uses of dateStamp2 in the JSP page source:

```
<h3>Example 1</h3>
<p><mytags:dateStamp2 beforeBody="false">
  <b>Date after tag body: </b>
</mytags:dateStamp2></p>
```

In this case, the attribute *beforeBody* is set from the constant value "false" in the opening tag. Unlike the previous tag examples, dateStamp2 has a body (the text "Date after tag body: " between the HTML bold face tags).

If the *beforeBody* attribute accepted only constant values, there would be very little point to having it. After all, you could just decide to place the bodiless version of the date stamp custom action (dateStamp1) either before or after some template text. However, there might be occasions when you want the placement decision made according to some logic in your JSP page. Consequently, the *beforeBody* attribute is set up to allow an expression for a value, and that's what you see in Example 2:

```
<h3>Example 2</h3>
<c:set var="trueVariable" value="true" />
<p><mytags:dateStamp2 beforeBody="${trueVariable}">
  <b>—date before tag body through runtime expression.</b>
</mytags:dateStamp2></p>
```

The `set` action from the JSTL core library is used to set a variable called *trueVariable* to a value of "true." Now in the dateStamp2 action's opening tag, the `beforeBody` attribute is set from an EL expression that pulls back the value from `trueVariable`. Because `beforeBody` is set to "true," the date stamp appears in front of the "—date before tag body. . ." template text. Though the example is trivial, it serves to make the point that run-time expressions can be fed into custom tag attribute values.

Finally, in Example 3, we see that the dateStamp2 custom action allows you to leave out the *beforeBody* attribute altogether:

```
<h3>Example 3</h3>
<p><mytags:dateStamp2>
  <b>Date after tag body by default: </b>
</mytags:dateStamp2></p>
```

In that case, the attribute still retains a default value, set by the tag handler code we'll see in a few moments. Because the default value of *beforeBody* is "false," the outcome is to place the date stamp after the "Date after tag body..." template text.

Here's the tag definition in the tag library (TLD file). The opening and closing tags used for tag library setup are omitted for brevity.

```
<...>
<tag>
  <name>dateStamp2</name>
  <tag-class>webcert.ch08.examp0804.DateStampTag2</tag-class>
  <body-content>JSP</body-content>
  <attribute>
    <name>beforeBody</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
<...>
```

Again we see the unique `<name>` for the tag—dateStamp2. The tag handler class is referenced in the `<tag-class>` element, as we've seen before. The `<body-content>` element is set to JSP, which means that anything permitted in JSP page source can be placed between the opening and closing dateStamp2 tags. This includes template text, scriptlets, expressions, standard actions, and custom actions—in fact, any element you want. While "empty" is the most restrictive, "JSP" is the most permissive setting for `<body-content>`.

Next we come to a set of elements not discussed previously, enclosed by the `<attribute>` element. You can have as many enclosing `<attribute>` elements as you want in a `<tag>`, from zero to many. `<attribute>`—like `<tag>`—has no data of its own, but contains three mandatory subelements:

- `<name>`—a unique name for the attribute. The unique rule applies only to all the attributes *within* a single tag (different tags can have attributes of the same name, as you've already seen with the JSTL core library. For example, *var* is an attribute name that appears in many tags—such as `c:set` and `c:import`).
- `<required>`—if set to "true," the attribute must appear in the opening tag. If "false," the attribute's presence is optional. This implies that the tag handler class supplies a default value or doesn't otherwise need any value for the attribute to evaluate the action.
- `<rtexprvalue>`—if set to "true," a run-time (EL) expression value is permitted for the attribute's value setting. You are not forced to use a run-time expression; a constant value is still perfectly valid. If set to "false," run-time expressions are disallowed for the attribute value—a translation error results if you try to do so.

Finally, here's the revised code for the tag handler class. The changed lines of code are shown in boldface.

```
import java.io.IOException;
import java.util.Date;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.Tag;
public class DateStampTag2 implements Tag {
  private PageContext pageContext;
  private Tag parent;
  private boolean beforeBody;
  public void setPageContext(PageContext pageContext) {
       this.pageContext = pageContext;
  }
  public void setParent(Tag parent) {
       this.parent = parent;
  }
  public Tag getParent() {
       return parent;
  }
  public boolean isBeforeBody() {
        return beforeBody;
  }
  public void setBeforeBody(boolean beforeBody) {
        this.beforeBody = beforeBody;
  }
  public int doStartTag() throws JspException {
        if (isBeforeBody()) {
             dateStamp();
        }
        return Tag.EVAL_BODY_INCLUDE;
  }
  public int doEndTag() throws JspException {
        if (!isBeforeBody()) {
              dateStamp();
        }
        return Tag.EVAL_PAGE;
  }
  protected void dateStamp() throws JspException {
        JspWriter out = pageContext.getOut();
        try {
              out.write("<i>" + new Date() + "</i>");
        } catch (IOException e) {
              throw new JspException(e);
        }
```

```
    }
    public void release() {
    }
}
```

In support of the attribute is a bean-style property, represented by the private instance variable *beforeBody*. The crucial aspect of attribute setup in the tag handler code is the presence of a setter method—in this case, `setBeforeBody()`—whose name matches the attribute name apart from the "set" portion, and the initial capital ("**B**eforeBody" instead of "**b**eforeBody"). The JSP container can infer from the attribute name what set method to call, passing in the value set within the page source.

As it happens, a "getter" is provided in this example, but that's not crucial to the tag life cycle (the JSP container will never call the "getter"). Because we're using a **boolean** for the attribute here, the getter is called "`isBeforeBody()`" (`"getBeforeBody()"` is still permissible—it's just that **booleans** more usually have "is"-style getter methods).

The `doStartTag()` logic has changed slightly. Remember this is executed when the JSP container encounters the opening dateStamp2 tag—before the body has been encountered. If the outcome of calling the `isBeforeBody()` method is true, then the date stamp is included at this point by calling the `dateStamp()` method (unchanged from before). Regardless of whether the date stamp is included at this point, the `doStartTag()` method returns the value Tag.EVAL_BODY_INCLUDE, indicating that the body should be evaluated in all circumstances.

The `doEndTag()` logic does a similar test. When the JSP container calls this method, the body of the tag has already been output. Now if the "`isBeforeBody()`" test evaluates to false, the `dateStamp()` method is called, placing the date stamp after the body. The return value from this method remains unchanged from before: Tag.EVAL_PAGE indicates that the JSP container should carry on and evaluate the rest of the page following on after this custom action.

IterationTag Interface and TagSupport Class

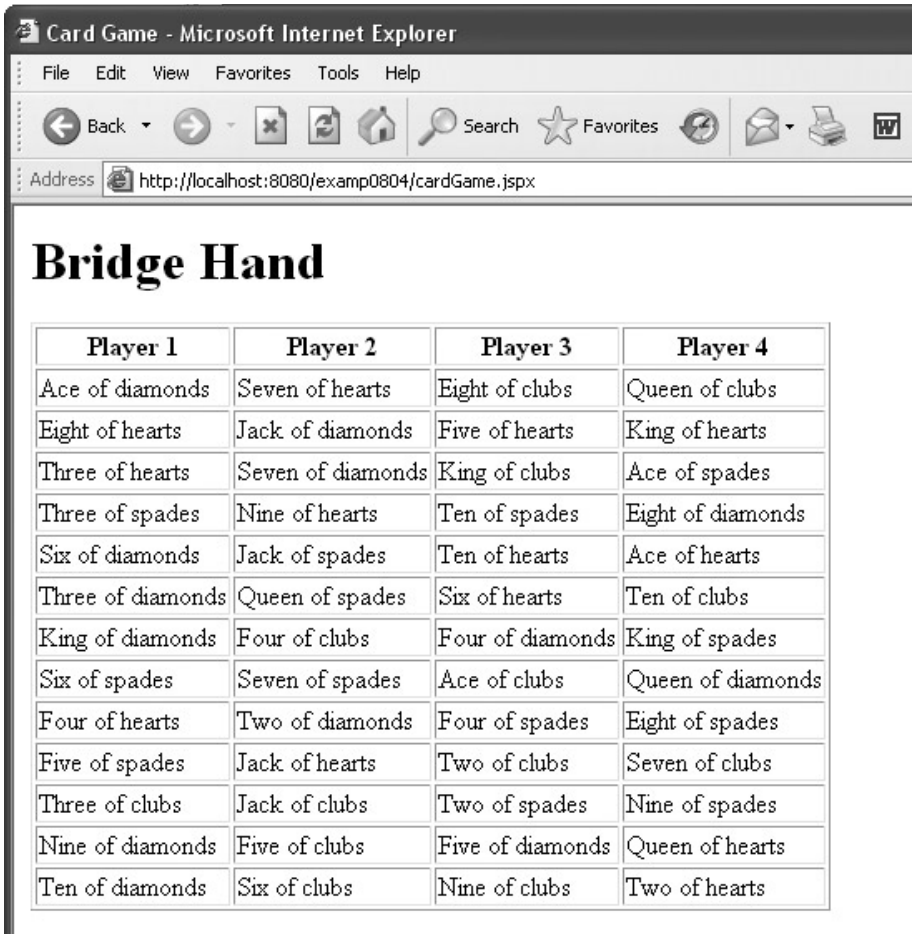
We've seen a simple case. Now we'll step up the complexity and consider a tag that repeats itself. This will also give an opportunity to look at how tags can interact when nested together. The example we'll take is the idea of a JSP page that shuffles and deals a pack of cards. We'll have a tag to manage the pack (called `cardDealer`)

and a tag within it that "receives" cards from the pile (called, simply, `card`). The tag is structured so that it deals out all fifty-two cards from the pack.

Let's see an example of the tags in action as a preliminary to considering the code. Here's the JSP page source:

```
<html xmlns:mytags="http://www.osborne.com/taglibs/mytags"
    xmlns:jsp="http://java.sun.com/JSP/Page">
  <jsp:output omit-xml-declaration="true" />
  <jsp:directive.page contentType="text/html" />
  <head><title>Card Game</title></head>
  <body>
    <h1>Bridge Hand</h1>
    <table border="1">
      <tr>
        <th>Player 1</th>
        <th>Player 2</th>
        <th>Player 3</th>
        <th>Player 4</th>
      </tr>
      <mytags:cardDealer>
      <tr>
        <td><mytags:card /></td>
        <td><mytags:card /></td>
        <td><mytags:card /></td>
        <td><mytags:card /></td>
      </tr>
      </mytags:cardDealer>
    </table>
  </body>
</html>
```

Most of the page is template text, setting up a table with headings for each of four players. The `cardDealer` custom action surrounds the data row of the table. Within each data row are four cells, one under each player heading. The `card` custom action has the effect of taking a card off the shuffled pack and displaying the value of the card. After each player "takes" a card, processing reaches the end tag for the `cardDealer` custom action. If there are more cards to deal, the JSP container processes the whole body of the tag (the table row) again, and so on until all the cards are used up. The following illustration below shows a sample hand.



Bridge Hand

Player 1	Player 2	Player 3	Player 4
Ace of diamonds	Seven of hearts	Eight of clubs	Queen of clubs
Eight of hearts	Jack of diamonds	Five of hearts	King of hearts
Three of hearts	Seven of diamonds	King of clubs	Ace of spades
Three of spades	Nine of hearts	Ten of spades	Eight of diamonds
Six of diamonds	Jack of spades	Ten of hearts	Ace of hearts
Three of diamonds	Queen of spades	Six of hearts	Ten of clubs
King of diamonds	Four of clubs	Four of diamonds	King of spades
Six of spades	Seven of spades	Ace of clubs	Queen of diamonds
Four of hearts	Two of diamonds	Four of spades	Eight of spades
Five of spades	Jack of hearts	Two of clubs	Seven of clubs
Three of clubs	Jack of clubs	Two of spades	Nine of spades
Nine of diamonds	Five of clubs	Five of diamonds	Queen of hearts
Ten of diamonds	Six of clubs	Nine of clubs	Two of hearts

The TLD descriptions for the two custom actions are pretty simple. Neither has any attributes. The notable difference is that `cardDealer` can contain any sort of body content, whereas `card` must be free of any body content.

```
<tag>
  <name>cardDealer</name>
  <tag-class>webcert.ch08.examp0804.CardDealingTag</tag-class>
  <body-content>JSP</body-content>
</tag>
```

```
<tag>
  <name>card</name>
  <tag-class>webcert.ch08.examp0804.CardTag</tag-class>
  <body-content>empty</body-content>
</tag>
```

Let's consider the tag handler code, first of the CardDealingTag class, which supplies the logic for the `cardDealer` custom action:

```
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.IterationTag;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.tagext.TagSupport;
public class CardDealingTag extends TagSupport {
  private static String[] suits =
    { "spades", "hearts", "clubs", "diamonds" };
  private static String[] values =
    { "Ace of", "Two of", "Three of", "Four of",
      "Five of", "Six of", "Seven of", "Eight of",
      "Nine of", "Ten of", "Jack of", "Queen of", "King of" };
private String[] pack = new String[52];
private int currentCard;
public int doStartTag() throws JspException {
  initializePack();
  shufflePack();
  currentCard = 0;
  return Tag.EVAL_BODY_INCLUDE;
}
public int doAfterBody() throws JspException {
  if (currentCard >= pack.length) {
    return Tag.SKIP_BODY;
  } else {
    return IterationTag.EVAL_BODY_AGAIN;
  }
}
public String dealCard() {
  String card = pack[currentCard];
  currentCard++;
  return card;
}
protected void initializePack() {
  for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 13; j++) {
      int cardIndex = (i * 13) + j;
```

```
    pack[cardIndex] = values[j] + " " + suits[i];
  }
    }
  }
  protected void shufflePack() {
    int packSize = pack.length;
      for (int i = 0; i < packSize; i++) {
        int random = (int) (Math.random() * packSize);
        // Swap two cards in the pack
        String card1 = pack[i];
        String card2 = pack[random];
        pack[i] = card2;
        pack[random] = card1;
      }
    }
  }
```

The first big difference about this tag handler class from the previous date stamp example is that it doesn't directly implement the Tag interface. Instead, it extends an existing class called javax.servlet.jsp.tagext.TagSupport. You can see from the class diagram (Figure 8-3) that TagSupport implements the IterationTag interface. Because IterationTag extends Tag, you get all the Tag methods as well, supplemented by those unique to IterationTag (there is only one!). However, there is a profound change to the life cycle with IterationTag, as you can see by looking at Figure 8-5 (compare this to the Tag life cycle in Figure 8-4).

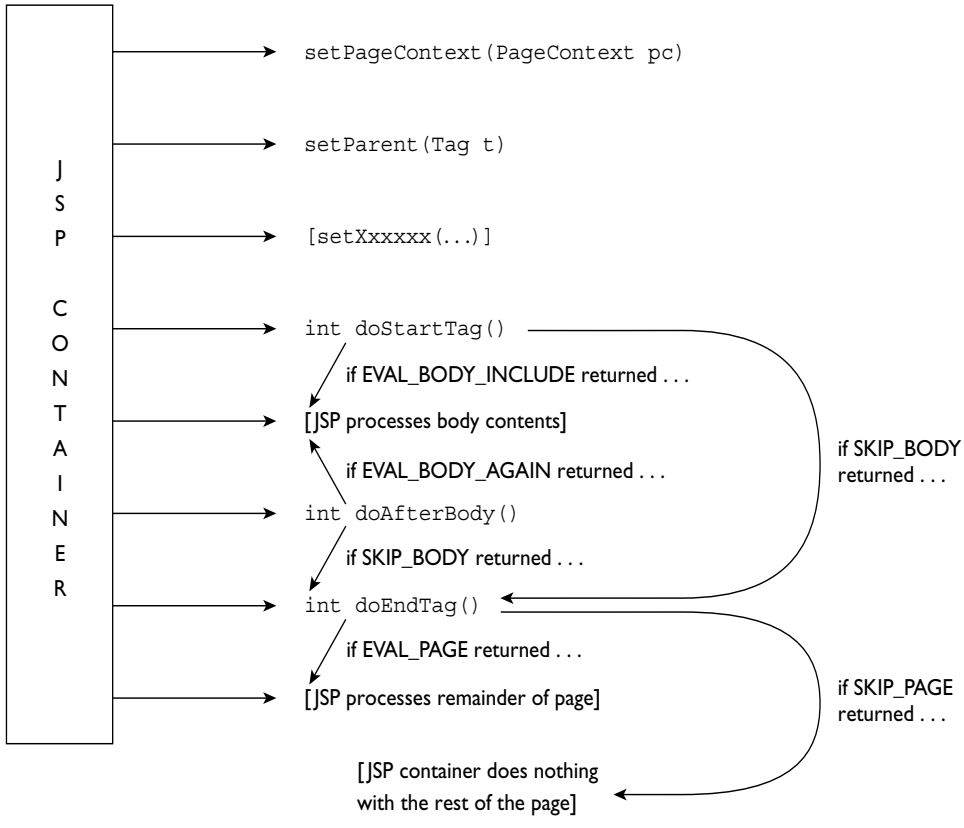
exam Watch

Although there are two instance variables declared in the CardDealerTag class, neither of these is an attribute of the tag (which has no attributes at all). They are merely there to support the internal workings of the tag. Even if these instance variables had getter and setter methods, they still wouldn't be attributes. For an attribute to be available on the tag, it must be declared in an <attribute> element in the tag library descriptor.

TagSupport—as it must—provides a default implementation of all the Tag and IterationTag methods, and supplements these with a few useful methods of its own that don't derive from any interface. So when writing your own tag that extends TagSupport, you are likely to have less work to do than if implementing IterationTag from scratch. You confine yourself to overriding only those methods where the default TagSupport behavior is insufficient. So, for example, `doEndTag()` is missing from our CardDealingTag code. The default TagSupport implementation—which is simply to return

FIGURE 8-5 JSP container processing one occurrence of a tag implementing the IterationTag Interface in one JSP page:

Custom Tag Life Cycle (2): The IterationTag Interface



the value Tag.EVAL_PAGE—is just what we want here. There's no special processing to be done when the closing tag is encountered in the page.

At the beginning of the class are two static String arrays. These provide the base data (suits and card values) for the next String array, called *pack*, which is a private instance variable in the class. After some initialization work, the pack holds 52 String values representing the 52 playing cards, in a random sequence. Next comes a private instance variable, of type **int**, called *currentCard*. This is used later as an index to the *pack* array, and it dictates the next "card" to "deal" from the pack.

Next we get the `doStartTag()` method, which is overridden from TagSupport. Remember that this will be invoked whenever the JSP container encounters the

opening cardDealer tag in any JSP page. The method calls the `initializePack()` and `shufflePack()` methods in the class, which serve to load up the pack String array with randomized card values. Their logic isn't explained in detail here, for it is straightforward and not central to the discussion of the tag life cycle (you can take the methods on trust!). The calls to these methods could have been placed in a no-argument constructor, but recall that tag instances get recycled—you don't get a dedicated instance for each access to the tag in your application. However, you can guarantee a `doStartTag()` call for every access to the tag, which guarantees a newly shuffled deck of cards every time. The `doStartTag()` method ends by returning Tag .EVAL_BODY_INCLUDE, which is necessary to ensure that the body is processed.

Following `doStartTag()` is a call to `doAfterBody()`. This is the one new method introduced by the IterationTag interface. The JSP container calls this method after evaluating the body but before `doEndTag()` (see Figure 8-5). The logic in the method demonstrates the two possible outcomes. If there are more cards to deal, then the method returns the **int** value Tag.SKIP_BODY (which, you'll remember, is also a valid return value from `doStartTag()`). This means that the JSP container won't process any further occurrences of the body, but proceed directly to the `doEndTag()` method. If, however, there are more cards available, then the method returns a new constant defined in the IterationTag interface: Iteration Tag.EVAL_BODY_AGAIN. As the name of this constant implies, the JSP container will evaluate the body of the tag again, and then once again invoke the `doAfterBody()` method. So you can see that there is no way out of this tag handler until Tag.SKIP_BODY is returned! You can also see that the body is evaluated at least once, for the logic tests present in `doAfterBody()` occur for the first time only after the body has been processed for the first time.

on the job

Should threading worry you? What if more than one person is trying to access the cardDealing.jspx file at the same time? Could there be other invocations of the same `cardDealer` tag instance that are trying to draw cards from the same pack? The answer is no. The JSP container must guarantee that any given instance of a tag handler class is dedicated to one thread at one given moment. How this is achieved is up to the JSP container designer—but you as a developer need not (in fact, should not) worry about synchronizing method calls within your tag handler classes.

So much for the life cycle methods of IterationTag. However, there is one "business" method left to explain, and that's `dealCard()`. It's an important method, for it retrieves a card from the pack—it's only by calling this method that the pack

is reduced and the loop implied by the logic in `doAfterBody()` can be broken. However, there is no code inside of the CardDealingTag class that calls this method, so where is it called from? The answer lies in the code of the tag handler class for the card tag — the CardTag class:

```
import java.io.IOException;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.Tag;
public class CardTag implements Tag {
  private PageContext pageContext;
  private Tag parent;
  private static int instanceNo;
  public CardTag() {
        instanceNo++;
  }
  public void setPageContext(PageContext pageContext) {
        this.pageContext = pageContext;
  }
  public void setParent(Tag parent) {
        this.parent = parent;
  }
  public Tag getParent() {
        return parent;
  }
  public int doStartTag() {
        return Tag.SKIP_BODY;
  }
  public int doEndTag() throws JspException {
        CardDealingTag dealer = (CardDealingTag) getParent();
        String card = dealer.dealCard();
        JspWriter out = pageContext.getOut();
        try {
              out.write(card);
              //out.write(card + "<br />" + instanceNo);
        } catch (IOException e) {
              throw new JspException(e);
        }
        return Tag.EVAL_PAGE;
  }
  public void release() {
      System.out.println("Releasing CardTag instance number: " + instanceNo);
  }
}
```

The CardTag tag handler is mostly simpler than the CardDealer tag handler. It implements the Tag interface rather than IterationTag, and has no body, as you can see from the tag library descriptor declaration:

```
<tag>
  <name>card</name>
  <tag-class>webcert.ch08.examp0804.CardTag</tag-class>
  <body-content>empty</body-content>
</tag>
```

This is reinforced by the `doStartTag()` method, which simply skips the body. All the action occurs in the `doEndTag()` method, whose purpose is to obtain the latest card from the pack and send the name of the card to page output. Its mechanism for doing this is simple enough.

`getParent()` works well when the tag in question is nested directly inside of its intended parent. But what if some other tag gets between the intended parent and child elements, by accident or design? Consider the following change to the JSP document that used the `cardDealer` and `card tags:`

```
<mytags:cardDealer>
<tr>
 <td><mytags:card /></td>
 <td><mytags:embolden><mytags:card /></mytags:embolden></td>
 <td><mytags:card /></td>
 <td><mytags:card /></td>
</tr>
</mytags:cardDealer>
```

You can see a new tag here, called "embolden," which surrounds the `card` tag for the second player (in the second column of the table). The tag's purpose is very simple: It introduces HTML boldface beginning and end tags to surround any body content (of course, it would be much easier just to write `<b><mytags:card /></b>;` bear with this as a teaching example!). For brevity, its tag handler code isn't shown here, but the class name is EmboldenTag. Now when the tag handler code for the card tag reaches the first line of code in its `doEndTag()` method:

```
CardDealingTag dealer = (CardDealingTag) getParent();
```

The actual tag handler instance returned by the `getParent()` method will be of type EmboldenTag—not the expected CardDealingTag instance as required. The page will fall down in a messy heap at run time with a ClassCastException.

There is a solution to this, provided by a static method on the TagSupport class, called `findAncestorByClass()`, which accepts two parameters: the first an instance of the tag whose ancestor is sought, and the second the class type of the ancestor tag. Here's the rewritten code for the `doEndTag()` method in `card`'s tag handler:

```
Class parentClass;
try {
  parentClass = Class.forName("webcert.ch08.examp0804.CardDealingTag");
} catch (ClassNotFoundException e1) {
  throw new JspException(e1);
}
CardDealingTag dealer = (CardDealingTag)
  TagSupport.findAncestorWithClass(this, parentClass);
// Code is unchanged from this point onwards...
String card = dealer.dealCard();
// etc.
```

First of all, the code makes an instance of Class type, using the static `Class.forName()` method. The instance required here is the CardDealingTag. Then in place of `getParent(), TagSupport.findAncestorWithClass()` is called, passing in "this" (the current instance of the CardTag tag handler), and the class instance we have just made. The method searches through the hierarchy of tags until it finds a parent, or grandparent, or great-great-grandparent (and so on) of the required class, CardDealingTag. The first (closest) relative is returned. The class match doesn't have to be exact: A subclass is acceptable (the API documentation states that the method will return "the nearest ancestor that implements the interface or is an instance of the class specified"). Under the covers, the code makes use of the `getParent()` method for each tag examined in the hierarchy, hence the importance of providing a functional `getParent()` method even if you never think your tag will need it.

This card-dealing example is not what I would regard as "production ready" code. If you forgot to include the `card` tag inside the `cardDealer` tag, you would produce an endless loop, which is not at all obvious just by looking at the tag declarations in the JavaServer Page source. There are mechanisms you can use—out of scope of the exam—to ensure that validation is performed at translation time. You might write a TagExtraInfo class to associate with a tag definition, and use it to perform complex validation on the tag's

attributes. However, to solve the validation problem posed here (checking the proper nesting of related tags), you would need to write a class extending javax.servlet.jsp.tagext.TagLibraryValidator. There are methods on this class called by the JSP container at translation time. In particular, the `validate()` method passes in an XML representation of the entire JSP page to be validated, giving you the opportunity to perform cross-tag checking and—if necessary—halting the translation process with an appropriate error message.

Other Methods on the TagSupport Class

TagSupport provides some other convenience methods that you should know about for the exam.

- `public void setId(String id)` and `public String getId()`—these methods assume that you will set up an attribute called `id` for your tag, intended to uniquely identify the tag on the page. It's up to you whether you do or not—but if you do, then the JSP container will invoke the `setId()` method in the normal way.
- `public void setValue(String name, Object o)`, `public Object getValue(String name)`, `public void removeValue(String name)`, and `public Enumeration getValues()`—these methods are underpinned by an instance variable Hashtable within the TagSupport class, just like the set of methods used to support attributes in different scopes (`request`, `session`, `etc.`). The idea is that you can associate named values with the tag instance and get them back according to their name (a unique String key). Note that the `getValues()` method returns not an Enumeration of values but of keys (you can use each key in turn with the `getValue()` method to return the underlying object).

exam Watch

Note that the `setValue(String name, Object o)` method in TagSupport is not a substitute for you providing individually crafted "setter" methods for each attribute you want to expose on your tag. The JSP container will not call the `setValue()` method; it's there for your own internal use in the tag handler code.

on the job

The IterationTag life cycle is, to a large extent, superseded by JSTL's iteration capabilities. You can probably imagine how the card-dealing example could be rewritten using <c:forEach>. However, I still like to use the IterationTag life cycle when I have complex business functionality to incorporate. Encapsulation is usually more readily achieved inside tag handler code than when using naked JSTL within JSP page source.

BodyTag Interface and BodyTagSupport Class

So to the third and last manifestation of the custom tag life cycle: the BodyTag interface (see Figure 8-6). Because IterationTag expands the base life cycle, BodyTag expands the iteration life cycle. The additional capability that a BodyTag has over its predecessors is the ability to manipulate the content of its body (the part that lies between the opening and closing tags).

This statement might seem initially confusing. Surely, the types of tag we have encountered have this capacity already? All tags, regardless of type, can dictate the type of body content allowed through the `<body-content>` element in the tag library descriptor—even, as we have seen, to the extent of banning body content altogether (when `<body-content>` is set to "empty"). And an IterationTag can cause its body to be evaluated as many times as it chooses.

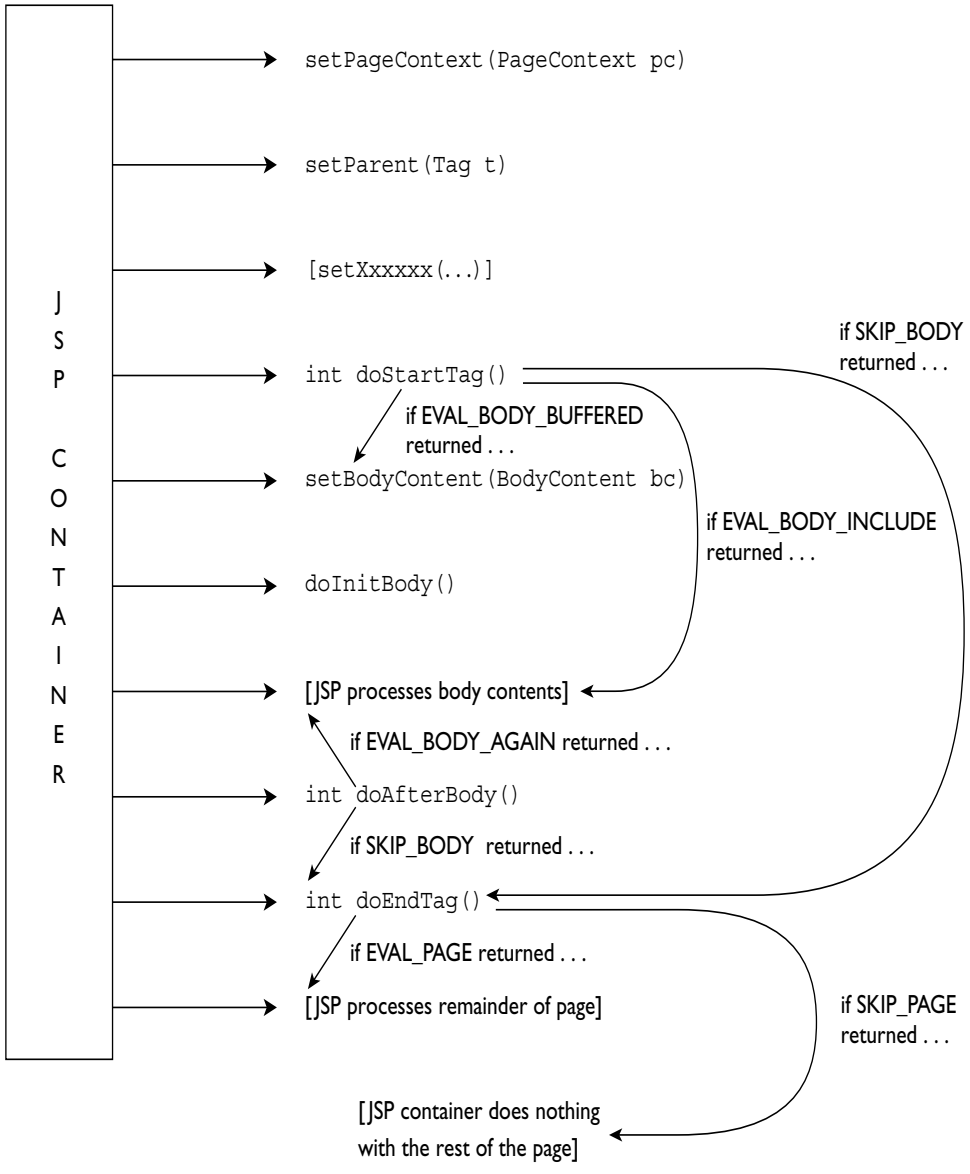
However, all we have really seen so far is the ability of tags to append (or prefix) to body content. None of the tags have actually taken what is in the body already and done something with it—perhaps changing the body out of all recognition or obliterating the body altogether. This is where the BodyTag interface comes in. Underpinning the BodyTag interface is a BodyContent object. This is a type of JspWriter, with an important distinction: The contents are buffered so that no part of the body has yet been committed to page output. This is what gives BodyTags their ability to take any liberties they wish with body content.

Let's first of all take a look at the BodyTag life cycle, shown in Figure 8-6. You can see two new life cycle methods over and above those provided by Tag and IterationTag.

- `setBodyContent()` gives you the opportunity to save the BodyContent object that will hold the buffered body contents. This method is called after `doStartTag()`, but before the next method, `doInitBody()`.
- `doInitBody()` is called before the JSP container enters and evaluates this tag's body for the first time. It is called only once per access to the tag, however many

FIGURE 8-6 Custom Tag Life Cycle (3): The BodyTag Interface

JSP container processing one occurrence of a tag implementing the BodyTag Interface in one JSP page:



times you choose to iterate through the body. Remember that at this point, the BodyContent object—though available—is empty: The body hasn't been evaluated at all yet.

One additional field is provided in the interface: BodyTag.EVAL_BODY_BUFFERED. This gives a new, third return code option from `doStartTag()`. Now, as well as being able to skip the body altogether (Tag.SKIP_BODY) or evaluating the body as normal (Tag.EVAL_BODY_INCLUDE), you can choose BodyTag.EVAL_BODY_BUFFERED, which has the effect of making the BodyContent object available. (Unless you set this return code, then the `setBodyContent()` method is not called.)

Beyond `doInitBody()`, the life cycle is exactly the same—with the important exception that `doAfterBody()` still has access to the BodyContent object. Understanding the methods on this object is the real key to effectively learning Body Tag-style tags.

The BodyContent Object

The BodyContent object effectively traps the output from any evaluations of the body that occur within the tag. As noted, the BodyContent is a JspWriter, so you can (in your tag handler code) write additional content to the BodyContent object before, between, and after evaluations of the tag body that arrive in the Body Content object as well.

Although the BodyContent object is a writer, there is no real underpinning stream; everything that accumulates there is held in a string buffer. One consequence of this is that you can write as much as you like to BodyContent, but nothing arrives in the page output, which can be baffling and confusing unless you know what is going on. What you are meant to do, usually when you are finished doing all you need to evaluate in the tag (most likely at the end of `doEndTag()`), is redirect the output to some other writer. BodyContent has a method exactly for this purpose: `writeOut(Writer w)`. The entire contents of the BodyContent object are appended to the Writer of your choice. But which Writer should you choose? Again, BodyContent comes to the rescue with the method `getEnclosingWriter()`, which passes back a JspWriter object. This is quite likely to be the JspWriter that is associated with the JSP page as a whole. However, if you consider that your tag might be nested within another tag implementing the BodyTag interface, the enclosing writer could be another BodyContent object itself. Really, though, this doesn't matter; almost invariably, the right thing to do is to pass the output to the enclosing writer, for whatever logic is controlling the enclosing writer should have a chance to

influence what happens to the output produced in your inner tag. So a typical invocation you will see at the end of your tag logic goes like this:

```
BodyContent bc = getBodyContent();
bc.writeOut(bc.getEnclosingWriter());
```

You might choose a different approach entirely, which is to use the capabilities of BodyContent to read what's been initially placed in the body of the tag, but never to write directly to BodyContent at all. You can always write whatever you like to the enclosing writer and silently drop the original content of the tag. This is especially useful when you set the content of your tag in the tag library to `<body-content> tagdependent</body-content>`. This means that the JSP container will make no effort at all to translate the body contents of your tag: It's up to the tag logic to do that. A very typical example of this use is when the body of the tag contains some completely non-Java dynamic content—such as an SQL statement to read a database. Suppose you have a JSP page with such a tag setup:

```
<mytags:sqlExecute>SELECT * FROM PRODUCT WHERE NAME LIKE
%1</mytags:sqlExecute>
```

Obviously, the user never wants to see "SELECT * FROM PRODUCT. . ."; he or she is far more likely to prefer to see a nicely formatted set of rows in an HTML table, showing details about a selection of products from the appropriate database table. Let's sketch out how you might approach writing a BodyTag to perform this translation:

- Ensure that you pass back EVAL_BODY_BUFFERED from the `doStartTag()` method (if you're extending BodyTagSupport, no need even to override the method—this is the default return code).
- Trap the BodyContent object in a private instance variable in your implementation of the `setBodyContent()` method (again—this work is done already if you're extending BodyTagSupport).
- Override `doAfterBody()`, calling the `getString()` method on the Body Content object—return this to a local String variable. This variable will contain the SQL string as an outcome of evaluating the body.
- Still in `doAfterBody()`, establish a data source connection to your database, and execute the SQL string. From the ResultSet returned, format the output as desired (in an HTML table, XML, or whatever). Write this to page output using the Jsp Writer returned by the BodyContent object's `getEnclosingWriter()` method.

- Still in `doAfterBody()`, release your data source connection, and return SKIP_BODY to prevent any further evaluations (no need for iteration here).

on the job

Before you develop database access tags along the lines of the pseudo-code drafted here, check out the capabilities of the JSTL SQL tag library.

EXERCISE 8-4

The "Classic" Custom Tag Event Model

In this exercise, you'll write a custom tag that will take a pair of numbers, and use this to generate a Fibonacci-like sequence of numbers. This works by taking the initial pair of numbers and adding them together, to produce a third. The second and third are then added together to produce a fourth, then the third and fourth to produce a fifth, and so on. If the pair of numbers you start with is 1 and 1, the output is the original Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, and so on. Here we see the output from the tag in the solution code, using 1 and 1 as the first and second numbers in the sequence:

Fibonacci Sequence

1
1
2
3
5
8
13
21
34
55
10 rows of Fibonacci-like sequence generated

For this exercise, create the usual web application directory structure under a directory called ex0804, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch08/ex0804.war.

Create an HTML Form (fibonacci.html)

1. Create a file directly in the web application root ex0804 called fibonacci.html.
2. Create an HTML form that allows you to type values into three input fields named seed1, seed2, and rowLimit. seed1 is for the first number in the sequence, seed2 is for the second number, and rowLimit determines how many numbers in the sequence will be generated and displayed.
3. For the action on the HTML form, specify Fibonacci.jspx.

Create a JSP Document (fibonacci.jspx)

4. Create a file directly in the web application root ex0804 called fibonacci.jspx.
5. Declare a URI for a tag library as a namespace attribute (xmlns) of the root `<html>` tag.
6. Include the namespace for standard actions (xmlns:jsp="http://java.sun.com/JSP/Page") and the following boilerplate code, which ensures HTML (rather than XML) output:

```
<jsp:output omit-xml-declaration="true" />
<jsp:directive.page contentType="text/html" />
```

7. In the body of the HTML, include a call to a tag called fibonacci. Provide an attribute called `rowLimit`, whose value is set using the EL variable representing the `rowLimit` request parameter passed from the fibonacci.html web page.
8. In the body of the tag (i.e., between the opening and closing tags), include two EL variables separated by a comma. The first should return the seed1 request parameter and the second the seed2 request parameter.

Write a Tag Handler Class (Fibonacci.java)

9. In an appropriate package directory under WEB-INF/src, create a Java source file called webcert.ch08.ex0804.FibonnacciTag.java. Make sure that the class declaration extends BodyTagSupport.

10. In the source file, define an instance variable called *rowLimit* of type **int**, together with `getRowLimit()` and `setRowLimit()` methods—this will handle the *rowLimit* attribute on the tag.
11. Define an instance variable called *currentRowNum* of type **int**—to keep count of how many iterations the tag has performed. Initialize the value of this to 1 in an overriding `doInitBody()` method.
12. Define instance variables (all of type **int**) to hold the current sequence number, the previous sequence number, and the previous to previous sequence number (the solution code calls these *currentValue*, *backOneValue*, and *backTwoValue*, respectively).
13. In an overriding `doAfterBody()` method, have logic that is performed only when the current row number (*currentRowNum*) is equal to 1. This should set up an HTML table heading, and print two table rows and cells containing the first two numbers in the sequence. These can be obtained by parsing the two comma-separated numbers in the tag body—available in the BodyContent object (available to you because this tag extends BodyTagSupport). Don't forget to store these two numbers in their appropriate instance variables (*backOneValue* and *backTwoValue*).
14. Next in `doAfterBody()`, have logic that detects when the number of rows requested in *rowLimit* has been exceeded. If this condition is true, return immediately from the method with a Tag.SKIP_BODY return code.
15. Next in *doAfterBody()*, calculate the current value of the sequence from the previous two values, and output this as a further table row. Shuffle the variable values (of *currentValue*, *backOneValue*, and *backTwoValue*) ready for the next iteration of the loop.
16. Finally in `doAfterBody()`, increment the current row number (*currentRowNum*), and return IterationTag.EVAL_BODY_AGAIN—you want to keep going filling up table rows until the limit has been reached.
17. In an overriding `doEndTag()` method, terminate the HTML table with an appropriate closing tag.

Set Up the TLD File

18. Create a TLD file called mytags.tld in a WEB-INF/tags directory.
19. Include a tag entry with a name of fibonacci, a tag class of webcert.ch08.ex0804.FibonacciTag, and a body content of "*scriptless*." (Note on *scriptless*:

This is the fourth and final legal value for body content that you have met in this chapter. This disallows Java language scriptlets, expressions, declarations, and the like, but still allows EL evaluations to go ahead.)

20. Include a definition with the tag entry for an attribute named *rowLimit*. This is a required attribute and should allow run-time expressions.

Add a Tag Library Mapping in the Deployment Descriptor (web.xml)

21. In WEB-INF/web.xml, ensure there is an appropriate `<taglib>` element within a `<jsp-config>` element.
22. The `taglib` URI should match whatever you declared in fibonacci.jspx.
23. The `taglib` location should point to /WEB-INF/tags/mytags.tld.

Run and Test the Code

24. Create a WAR file from your development directory ex0804, and deploy this to your server. Run and test the code with an appropriate URL such as

```
http://localhost:8080/ex0804/Fibonacci.html
```

CERTIFICATION SUMMARY

In this chapter you started by learning how to use preexisting tags from a tag library—how to use them in a JSP page or document, how to reference the tag library in the JSP page or document, and how to use the deployment descriptor to let the JSP container track the path from the JSP page or document to the tag library descriptor file.

You saw that tags within the JSP page are used in exactly the same manner as JSP standard actions, using XML-like syntax, declared in the form

```
<prefix:tagname attribute="value">any body content goes
here</prefix:tagname>
```

You further learned that you can use a `taglib` directive to declare the tag library that holds the tag you are using, and that this takes the form

```
<%@ taglib prefix="prefix" uri="anyStringYouLike" %>
```

You also touched on the alternative form found in JSP documents, which use namespace declarations instead of taglib directives, in this form:

```
<anyElement xmlns:prefix="anyStringYouLikeToRepresentTheURI">
```

You went on to discover how to reference a tag library in the web deployment descriptor, web.xml. You saw that this can be done within the `<taglib>` subelement of `<jsp-config>`. You saw that `<taglib>` has two subelements, one called `<taglib-uri>`—whose value must match the URI you place in the `taglib` directive—and another called `<taglib-location>`—whose value holds an path to the tag library descriptor file (.tld) holding tag definitions.

You learned more about the `<jsp-config>` element and, in particular, saw how this can be used to switch off EL for a group of JSP files defined in a `<jsp-property-group>`, by setting the `<el-ignored>` element to "true." You also saw how Java language scripting can be switched off for a group of JSP files by setting the `<scripting-invalid>` element to "true."

You then examined a tag library descriptor file in detail, seeing how to set up certain mandatory heading elements, such as `<tlib-version>` and `<short-name>`. You then learned about the elements used to define a tag, such as `<name>` (the name used in the JSP file after the prefix to target this tag), `<tag-class>` (to point to the actual Java tag handler class that does the work), and `<body-content>` (to define the permitted content between the opening and closing tags, with valid values ranging from empty through tagdependent, scriptless, and JSP). You also learned that a `<tag>` element might contain `<attribute>` elements, each with a trio of attributes: `<name>` (to specify the attribute name), `<required>` (true or false: whether mandatory or not), and `<rtexprvalue>` (true or false: whether a run-time expression, such as a Java language expression or piece of EL, can be used to set the attribute value).

In the next part of the chapter, you learned about the Java Standard Tag Libraries—in particular, the core library. You met fourteen custom tag actions you can use in your own JSP pages or documents, split across four groups: general purpose, conditional, iteration, and URL-related.

In the general purpose group, you saw that `<c:out>` can be used to output a value to the current JSPWriter, including the ability to escape XML-unfriendly characters. You used `<c:set>` to set values of attributes in the scope of your choice and `<c:remove>` to remove them again. Finally in this group, you learned how `<c:catch>` can be used to suppress minor errors within your JSP page.

In the conditional group, you saw how `<c:if>` can be used to perform a test, and so conditionally include pieces of JSP page source. You also saw how the result of the test can be saved to an attribute accessible through EL at a later time. You also learned how the `<c:choose>` action can be used to choose one true condition from many, defined in `<c:when>` subactions, with the possibility of including a single default action in a `<c:otherwise>` subaction.

In the iteration group, you saw that `<c:forEach>` can be used to work through all the objects in any kind of Collection or Map. You also saw that `<c:forEach>` has an alternative syntax, which allows iteration through a numeric sequence by steps—much like a Java "for" loop. You then met `<c:forTokens>` and found that most of its syntax is identical to `<c:forEach>` but that it specializes in breaking up Strings into separate Strings (much like Java's StringTokenizer class).

In the last JSTL core library group, URL-related actions, you learned about the `<c:import>` action and how this can be used in lieu of `<jsp:include>`. You saw how this action can be used to import local resources from the same web application, or resources in different web applications on the same server, or even external resources from other servers. You learned that the target of the import can be the JSPWriter directly or an interim reader (for more efficient processing of large imports). You also learned about `<c:url>`, for manufacturing URL strings (with session information automatically encoded when necessary), and `<c:redirect>`, to instruct clients to redirect to other resources. You finally met `<c:param>`, which can be used to provide additional request parameters to accompany any of the previous three URL-related actions.

Then you examined EL functions. You learned that any public static method in any class can be exposed as an EL function. You saw that making this happen requires a `<function>` element entry in the tag library descriptor—including the name of the function in the `<name>` element, the fully qualified name of the Java class containing the supporting method in the `<function-class>` element, and a stylized version of the method signature in the `<function-signature>` element. You learned the differences between `<function-signature>` and method signature: Keywords are omitted, parameters are expressed without parameter names (only types), and any Java objects returned or passed as parameters are always written with a fully qualified name. You finally learned that EL function-calling syntax looks like a mixture between tags and Java method calls: `${prefix:myfunction(param1, param2)}`.

In the last part of the chapter, you learned about custom tags. You saw that the custom tag life cycle grows in complexity, dependent on whether you implement the Tag, IterationTag or BodyTag interface. You learned that the package containing

these interfaces (javax.servlet.jsp.tagext) also contains two classes that save you work by providing default implementations: TagSupport (implementing IterationTag and Tag) and BodyTagSupport (implementing BodyTag).

You saw that in all cases, methods are called by the JSP container to provide a tag handling class with a PageContext object and (if relevant) the object representing the enclosing tag, and that this is followed by JSP container calls to set attribute values. You saw that—still, in all cases—`doStartTag()` is the next method called by the JSP container. You saw that a return code (Tag.SKIP_BODY) from `doStartTag()` might cause the JSP container to skip the body altogether and proceed directly to calling the `doEndTag()` method. You saw that `doEndTag()` can abort the rest of the page (by providing a return code of Tag.SKIP_PAGE) or allow it to be processed (Tag.EVAL_PAGE).

You then learned about the divergent parts of the life cycle. In IterationTag, you saw how an additional method—`doAfterBody()`—can cause the body to be processed again (with a return code of IterationTag.EVAL_BODY_AGAIN) or proceed to the `doEndTag()` method (Tag.SKIP_BODY). You saw that BodyTag adds to this by allowing `doStartTag()` an additional return code—BodyTag.EVAL_BODY_BUFFERED. You learned that when this is set, your tag is provided with a BodyContent object—representing the buffered contents of the body—and that by using this, you can manipulate the body contents before they are written out to the enclosing JSPWriter.

TWO-MINUTE DRILL

Tag Libraries

- Tags are used in a JSP page using the syntax `<prefix:tagname attribute="value">any body content goes here</prefix:tagname>`.
- The prefix used must match the value of the prefix attribute in the `taglib` directive.
- A `taglib` directive has the following syntax: `<%@ taglib prefix="prefix" uri="anyStringYouLike" %>`.
- For JSP documents, `taglib` directives are disallowed.
- JSP documents use XML namespace syntax as an alternative to the `taglib` directive.
- Namespace syntax can be attached to any element (but usually the root element), like this: `<anyElement xmlns:prefix="anyStringYouLikeTo RepresentTheURI">`.
- The URI declared in the `taglib` directive (or namespace value) should match the value of a `<taglib-uri>` element in the deployment descriptor.
- The `<taglib-uri>` element is paired with a `<taglib-location>` element, which points to the location within the web application where the tag library descriptor file is located.
- `<taglib-uri>` and `<taglib-location>` are both subelements of `<taglib>`, which is a subelement of `<jsp-config>`, which is a subelement of the root element `<web-app>`.
- Apart from `<taglib>`, `<jsp-config>` can have one other type of subelement: `<jsp-property-group>`.
- Within `<jsp-property-group>`, you can define a group of JSP files with the `<url-pattern>` element.
- Also within `<jsp-property-group>`, EL can be turned off for the defined group of JSP files by setting the `<el-ignored>` element to a value of "true."
- Again within `<jsp-property-group>`, scripting can be turned off for the defined group of JSP files by setting the `<scripting-invalid>` element to a value of "true."
- A tag library descriptor file (TLD) has a root element of `<taglib>`.

- A TLD must have elements `<tlib-version>` and `<short-name>` defined.
- A TLD can have any number of `<tag>` elements.
- The `<name>` subelement of `<tag>` defines the (unique) name of the tag, as it is called in the JSP page.
- The `<tag-class>` subelement of `<tag>` defines the fully qualified name of the Java tag handler class.
- The `<body-content>` subelement of `<tag>` defines what is allowed to appear between the opening and closing tags:
  - empty: The tag mustn't have a body.
  - tagdependent: The tag can have a body, but the contents (scriptlets, EL, etc.) are completely ignored and treated like template text.
  - scriptless: The tag body can contain EL or template text, but no Java language scripting constructs (expressions, scriptlets, declarations)—if these are present, a translation error results.
  - JSP: The tag body can contain anything: Java language scripting, EL, or template text.
- Another subelement of `<tag>` is `<attribute>`: This can appear zero, one, or many times.
- The `<attribute>` element is used to define attributes on a tag and has three subelements:
  - `<name>`—a name for the attribute (unique within the tag)
  - `<required>`—true or false—whether the attribute must be present or is optional.
  - `<rtexprvalue>`—true or false—whether the attribute value can be provided by an expression (Java language or EL).

JSTL

- There are five JavaServer Page Standard Tag Libraries (JSTL): core, relational database access (SQL), Formatting with Internationalization, XML Processing, and EL standard functions. The exam focuses only on the core library.
- The actions (tags) within the libraries represent a standard way of performing frequently required functionality within web applications.

- ❏ The following tag directive is used for access to core library actions in your pages:

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

- ❏ The "c" value for `prefix` is usual, but optional.
- ❏ The value for `uri` must be just as shown above.
- ❏ There are fourteen core library actions, divided into four groups: general purpose, conditional, iteration, and URL-related.
- ❏ There are four actions in the general purpose group: `<c:out>`, `<c:set>`, `<c:remove>`, and `<c:catch>`.
- ❏ `<c:out>` is for directing output to the JSPWriter. Its attributes are *value* (the output), *default* (the output if *value* is **null**), and *escapeXml* (for converting XML-unfriendly characters).
- ❏ `<c:set>` is for setting attributes in any scope. Its main attributes are*value* (contents of attribute), *var* (name of attribute), and *scope* (scope of attribute).
- ❏ `<c:set>` also has *target* and *property* attributes for setting properties on beans.
- ❏ `<c:remove>` is for removing attributes in any scope. It has only the attributes *var* (name of attribute to remove) and *scope* (scope of attribute).
- ❏ `<c:catch>` catches Throwable objects thrown from the statements it contains. It has only the one optional attribute, *var,* to store the Throwable object for later use.
- ❏ The conditional group in the JSTL library has four actions: `<c:if>`, `<c:choose>`, `<c:when>`, and `<c:otherwise>`.
- ❏ `<c:if>` is used to conditionally execute some JSP statements if a test proves true. Its attributes are *test* (expression for the test), *var* (optional attribute variable to hold the result of the test), and *scope* (scope of the optional attribute).
- ❏ `<c:choose>` is used to contain mutually exclusive tests, held in `<c:when>` actions.
- ❏ `<c:choose>` can only contain `<c:when>` and `<c:otherwise>` actions (and white space).
- ❏ `<c:when>` has only one attribute: *test*.
- ❏ Only the statements bounded by the first `<c:when>...</c:when>` action whose test is true will be executed.

- One `<c:otherwise>` can be included after any `<c:when>` actions.
- The statements within `<c:otherwise>` are executed only if all the preceding `<c:when>` tests prove false.
- There are two actions in the iterator group of the JSTL core library: `<c:forEach>` and `<c:forTokens>`.
- `<c:forEach>` is used to iterate through a series of items in a collection, or simply to loop for a set number of times.
- `items` can hold Arrays, Strings, and most collection types in java.util: Collections, Maps, Iterators, and Enumerations.
- When iterating through collections, `<c:forEach>` uses the attributes *items* (for the collection object) and *var* (to represent each item in the collection on each circuit of the loop).
- When looping a set number of times, `<c:forEach>` uses the attributes *begin* (the number to begin at), *end* (the number to end at), and *step* (the amount to step by when working through from the begin number to the end number).
- In either case, a special variable called *varStatus* can be used to obtain properties about the current iteration.
- All the above attributes can be combined in a hybrid syntax—for example, to step through every second item in a collection.
- `<c:forTokens>` works similarly to `<c:forEach>`—but is specialized for breaking up (tokenizing) Strings.
- It has the same six attributes as `<c:forEach>`, and an additional seventh of its own.
- The *items* attribute will accept only a String as input.
- The additional seventh parameter is *delims*, which holds the characters used to denote where to break up the String.
- There are four actions in the URL-related group of the JSTL core library: `<c:import>`, `<c:url>`, `<c:redirect>`, and `<c:param>`.
- `<c:import>` is used to include a URL resource within the current page at run time.
- `<c:import>` has six attributes. The main one is *url* (the expression representing the URL resource to import).
- The *context* attribute can be used to specify a different context housed in the same application server.

- ❏ The *var* and *scope* attributes can be used to place the contents of the URL resource in a scoped attribute (as a String).
- ❏ Alternatively, *varReader* can be used to keep the contents of the URL resource in a Reader object.
- ❏ `<c:url>` is used to compose URL strings (for use as links in documents, for example).
- ❏ `<c:url>` has four attributes: *value* (expression for the URL string), *context* (optional alternative context on the same web application server), *var* (optional String attribute to hold the result of the URL String expression), and *scope* (scope for *var*, if used).
- ❏ `<c:redirect>` is used to instruct the web client to point to an alternative resource.
- ❏ `<c:redirect>` has two attributes: *url* (the URL for the client to point to) and *context* (optional alternative context if the URL is in a different web application on the same server).
- ❏ `<c:param>` can be nested within `<c:url>`, `<c:import>`, or `<c:redirect>`.
- ❏ `<c:param>` is used to attach additional parameters to the requests made or implied by the other URL-related actions.
- ❏ `<c:param>` has two attributes: *name* (the name of the request parameter) and *value* (the value of the request parameter).

EL Functions

- ❏ Any **public static** method in any Java class can be exposed as an EL function.
- ❏ EL functions are defined in `<function>` elements in a tag library descriptor (TLD).
- ❏ Within the `<function>` element, an EL function must have three subelements defined: `<name>`, `<function-class>`, and `<function-signature>`.
- ❏ `<name>` is a unique name for the function.
- ❏ `<name>` must be unique not only within functions in the TLD but also within other elements that might be defined in the TLD, such as custom tags and tag files.
- ❏ `<function-class>` gives the fully qualified name of the Java class containing the method backing the EL function.

- `<function-signature>` reflects the signature of the method backing the EL function.
- `<function-signature>` must always use fully qualified names for Java classes returned or passed in to the function (even String must be expressed as java.lang.String).
- Parameter names are omitted from the function signature (only types are defined).
- The method name in the function signature must match the method name in the Java class.
- Qualifiers (such as **public** and **static**) are omitted in the function signature. So the Java method with signature . . .

```
public static String getDefinition(String word, int timeForSearch)
```

- . . . would yield the following `<function-signature>`:

```
<function-signature>java.lang.String getDefinition(java.lang.String,
int)</function-signature>
```

- This function might be called in the JSP page with the following EL syntax:

```
${myfunctions:getDefinition(wordHeldInAttribute, 30)}
```

The "Classic" Custom Tag Event Model

- Custom tags are supported by Java classes called tag handler classes.
- A tag handler class must implement one of the Tag, IterationTag, or BodyTag interfaces in the javax.servlet.jsp.tagext package.
- Each of these interfaces extends the next, so IterationTag augments the life cycle of Tag, and BodyTag augments the life cycle of IterationTag.
- Two classes implement default functionality for tags: TagSupport and BodyTagSupport.
- TagSupport implements IterationTag (and therefore Tag as well).
- BodyTagSupport implements BodyTag.
- When the JSP container meets an occurrence of a tag in the page, it makes calls to known methods on an instance of the tag handler class.
- The method calls and other events that can occur whatever the type of tag are shown here in order:

  - ❑ `setPageContext()`—to give the tag access to the PageContext object
  - ❑ `setParent()`—to give the tag instance access to the enclosing tag instance
  - ❑ `setXXX()`—to set any attribute values on the tag
  - ❑ `doStartTag()`
  - ❑ The processing of the tag body
  - ❑ `doEndTag()`
  - ❑ The processing of the remainder of the page
- ❑ `If doStartTag()` returns Tag.SKIP_BODY, the JSP container ignores anything between the opening and closing tags.
- ❑ `If doEndTag()` returns Tag.SKIP_PAGE, the JSP container aborts evaluation of the rest of the JSP page after the closing tag.
- ❑ `doStartTag()` can return Tag.EVAL_BODY, and `doEndTag()` can return Tag.EVAL_PAGE, which tell the JSP container to process the tag body and rest of page, respectively.
- ❑ IterationTag introduces a `doAfterBody()` method, called after the body is processed for the first time.
- ❑ If `doAfterBody()` returns IterationTag.EVAL_BODY_AGAIN, the JSP container processes the body again and then calls `doAfteBody()` again.
- ❑ `doAfterBody()` must return Tag.SKIP_BODY to break this loop.
- ❑ BodyTag introduces the concept of buffering the tag body (before it is committed to the JSPWriter) into a BodyContent object.
- ❑ For a BodyTag, the JSP container calls the methods `setBodyContent()` and `doInitBody()` immediately after the call to `doStartTag()`.
- ❑ However, for these calls to happen, `doStartTag()` must return BodyTag .EVAL_BODY_BUFFERED.
- ❑ The BodyContent object is a type of JSPWriter and can be used to manipulate the contents of the body before it is sent to page output.
- ❑ Alternatively, the body content can be discarded entirely.
- ❑ Important methods of BodyContent include `getEnclosingWriter()` and `writeOut(Writer out)`.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there might be more than one correct answer. The number of correct choices to make is stated in the question, as in the real SCWCD exam.

Tag Libraries

1. Given a tag declared as shown in the following tag library descriptor extract, what are valid uses of the tag in a JSP page? You can assume that the tag library is correctly declared in the JSP page with a prefix of "mytags." (Choose three.)

```
<tag>
<name>book</name>
<tag-class>webcert.ch08.examp0801.BookTag</tag-class>
<body-content>tagdependent</body-content>
<attribute>
  <name>isbn</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

A.

```
<mytags:book />
```

B.

```
<mytags:book isbn="<%= isbn %>">
```

C.

```
<mytags:book isbn="${isbn}" />
```

D.

```
<mytags:book isbn="1861979258" />
```

E.

```
<mytags:book isbn="${isbn}">Some default text if book not found</mytags:
book>
```

2. Which of the following XML fragments, if placed below the root element in the deployment descriptor, will deactivate the scripting language for all files in the web application with a .jsp extension? (Choose one.)

A.

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <scripting-invalid>true</scripting-invalid>
  </jsp-property-group>
</jsp-config>
```

B.

```
<jsp-config>
  <url-pattern>*.jsp</url-pattern>
  <scriptless>true</scriptless>
</jsp-config>
```

C.

```
<jsp-config>
  <url-pattern>/*.jsp</url-pattern>
  <el-ignored>true</el-ignored>
</jsp-config>
```

D.

```
<jsp-config>
  <jsp-property-group>
    <uri-pattern>*.jsp</uri-pattern>
    <script-invalid>true</script-invalid>
  </jsp-property-group>
</jsp-config>
```

E.

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <el-ignored>true</el-ignored>
  </jsp-property-group>
</jsp-config>
```

3. (drag-and-drop question) In the following illustration, match the correct numbered tag library descriptor element names to the letters masking the element names in the tag library descriptor source.

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-
jsptaglibrary_2_0.xsd"
  version="2.0">
  <[ A ]>1.0</[ A ]>
  <[ B ]>MyTagLib</[ B ]>
  <[ C ]>http://www.osborne.com/mytags.tld</[ C ]>
  <tag>
    <[ D ]>sometag</[ D ]>
    <[ E ]>a.b.MyClass</[ E ]>
    <[ F ]>empty</[ F ]>
    <attribute>
      <[ G ]>grossIncome</[ G ]>
      <[ H ]>true</[ H ]>
      <[ I ]>true</[ I ]>
    </attribute>
  </tag>
 </taglib>
```

1	version
2	name
3	expression
4	runtime-expression
5	tlib-version
6	tag-class
7	class
8	name
9	short-name
10	uri
11	url-pattern
12	body
13	body-content
14	required
15	mandatory
16	rtexprvalue

4. Which of the following deployment descriptors will successfully and legally deactivate Expression Language for an entire web application? (Choose two)

A.

```
<?xml version="1.0" ?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
</web-app>
```

B.

```
<?xml version="1.0" ?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
</web-app>
```

C.

```
<?xml version="1.0" ?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <jsp-config>
    <jsp-property-group>
      <url-pattern>*.*</url-pattern>
      <el-ignored>true</el-ignored>
    </jsp-property-group>
  </jsp-config>
</web-app>
```

D.

```
<?xml version="1.0" ?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <jsp-config>
    <jsp-property-group>
      <url-pattern>/*</url-pattern>
      <el-ignored>true</el-ignored>
    </jsp-property-group>
  </jsp-config>
</web-app>
```

E.

```
<?xml version="1.0" ?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
```

```
<jsp-config>
    <jsp-property-group>
      <url-pattern>*.*</url-pattern>
      <el-invalid>true</el-invalid>
    </jsp-property-group>
  </jsp-config>
</web-app>
```

5. From the following use of the tag `<mytags:convert>`, what statements must be true about its setup and use? You can assume that the tag translates and executes correctly. (Choose three.)

```
<mytags:convert currency="${param.cur}"><%= amount %></mytags:convert>
```

A. The `taglib` declaration has a prefix of "mytags."
B. In the TLD, the tag's body content element has a value of JSP.
C. In the TLD, the tag's name element has the value of currency.
D. In the TLD, the tag's currency attribute has the required element set to true.
E. In the TLD, the tag's currency attribute has the rtexprvalue element set to true.
F. In the TLD, the tag's mandatory element is set to true.

JSTL

6. Which of the following characters are not converted by the `<c:out>` action when the attribute *escapeXml* is set to false? (Choose one.)

A. {
B. <
C. ;
D. @
E. All of the above
F. None of the above

7. Which of the following are invalid uses of the `<c:set>` action? (Choose three.)

A.

```
<c:set scope="page">value</c:set>
```

B.

```
<c:set value="value" var="${myVar}" />
```

C.

```
<c:set var="myVar" scope="${scope}">value</c:set>
```

D.

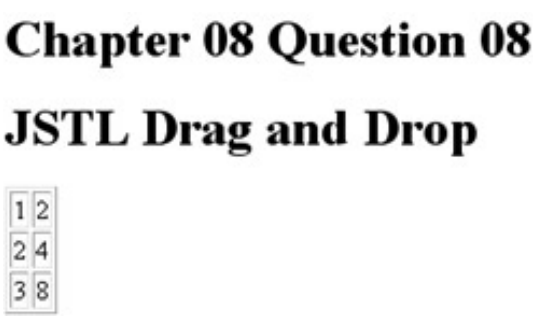
```
<c:set target="${myTarget}" property="myProp">propValue</c:set>
```

E.

```
<c:set value="${myVal}" target="myTarget" property="${myProp}" />
```

8. (drag-and-drop question) In the following illustration, match the numbered JSTL tag names, attribute names, and attribute values with the corresponding lettered points in the JSP document source shown. Your choices should lead to the output illustrated beneath the JSP document source.

```
<html xmlns:mytags="http://www.osborne.com/taglibs/mytags"
      xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c="http://java.sun.com/jsp/jstl/core" >
  <jsp:output omit-xml-declaration="true" />
  <jsp:directive.page contentType="text/html" />
  <head><title>JSTL Iterator Tags
Example</title></head>
  <body>
  <h1>Chapter 08 Question 08</h1>
  <h1>JSTL Drag and Drop</h1>
     <table border="1">
     <[ A ] [ B ]="num" [ C ]="1" />
     <[      D      ]begin="1" end="7" [ E ]="[F]"
[      G      ]="counter">
        <c:set var="num" [ H ]="${[      I      ]}" />
<tr><td>${counter.count}</td><td>${num}</td></tr>
     </c:forEach>
     </table>
  </[ J ]>
</html>
```



1	4
2	num * num
3	num + num
4	3
5	c:set
6	c:load
7	var
8	variable
9	val
10	value
11	varStatus
12	varCounter
13	counter
14	c:forEach
15	c:forEvery
16	step
17	skip
18	c:forToken

9. What is the result of attempting to access the following JSP page source? You can assume that the file countries.txt exists in the location specified. (Choose one.)

```
<html xmlns:mytags="http://www.osborne.com/taglibs/mytags"
      xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c="http://java.sun.com/jsp/jstl/core" >
  <jsp:output omit-xml-declaration="true" />
  <jsp:directive.page contentType="text/html" />
  <jsp:directive.page import="java.io.*" />
  <head><title>Question 9</title></head>
  <body>
    <c:import url="/countries.txt" varReader="myReader" />
    <jsp:scriptlet>
      Reader r = (Reader) pageContext.getAttribute("myReader");
      out.write(r.read());
    </jsp:scriptlet>
  </body>
</html>
```

A. The first character of the file countries.txt is sent to page output.
B. IOException occurs at run time.
C. NullPointerException occurs at run time.
D. Some other exception occurs at run time.
E. Translation error generating the source.
F. Translation error compiling the source.

10. What is the minimum number of attributes that must be specified in the `<c:forEach>` action? (Choose one.)

A. 1—*items*
B. 1—*collection*
C. 2—*var* and *items*
D. 2—*begin* and *end*
E. 3—*begin*, *end*, and *step*
F. 3—*var*, *collection*, and *varStatus*

EL Functions

11. Which of the following characteristics must a Java class have if it contains one or more EL functions? (Choose three.)

A. Instance variables matching the function attribute names
B. A no-argument constructor
C. A method that is **public**
D. A method that is **static**
E. A main method (signature: public static void main(String[] args))
F. A method that returns a nonvoid result

12. Which of the following represents a correct function declaration in the tag library descriptor? (Choose one.)

A.

```
<el-function>
  <description>Taxation Function</description>
  <name>netincome</name>
  <el-function-class>webcert.ch08.ex0803.Taxation</el-function-class>
  <el-function-signature>java.lang.String calcNetIncome(double, double,
      double, java.lang.String)</el-function-signature>
</el-function>
```

B.

```
<function>
  <description>Taxation Function</description>
  <name>netincome</name>
  <function-class>webcert.ch08.ex0803.Taxation.class</function-class>
  <function-signature>java.lang.String calcNetIncome(double, double,
      double, java.lang.String)</function-signature>
</function>
```

C.

```
<el-function>
  <description>Taxation Function</description>
  <name>netincome</name>
  <el-function-class>webcert.ch08.ex0803.Taxation</el-function-class>
  <el-function-signature>String calcNetIncome(double, double,
      double, String)</el-function-signature>
</el-function>
```

D.

```
<function>
  <description>Taxation Function</description>
```

```
    <name>netincome</name>
    <function-class>webcert.ch08.ex0803.Taxation</function-class>
    <function-signature>public static java.lang.String
      calcNetIncome(double grs, double allow, double rate,
      java.lang.String cur)
    </function-signature>
  </function>
```

E.

```
  <el-function>
    <description>Taxation Function</description>
    <name>netincome</name>
    <el-function-class>webcert.ch08.ex0803.Taxation</el-function-class>
    <el-function-method>public static String calcNetIncome(double, double,
        double, String)</el-function-method>
  </el-function>
```

13. What is the result of attempting to access the following JSP? You can assume that the EL functions are legally defined, that the EL function `mytags:divide` divides the first parameter by the second parameter, and that the EL function `mytags:round` rounds the result from the first parameter to the number of decimal places expressed by the second parameter. (Choose one.)

```
<html>
  <%@ taglib prefix="mytags" uri="http://www.osborne.com/taglibs/mytags" %>
  <head><title>Question 13</title></head>
  <body>
    <p>${mytags:round(${mytags:divide(arg1, arg2)}, 2)}</p>
  </body>
</html>
```

A. Translation error (in code generation).

B. Translation error (in code compilation).

C. Run-time error.

D. Zero, for arg1 and arg2 are not set to any value.

E. The expected result from the division, rounded to two decimal places.

14. Where in JSP page source can EL functions be used? (Choose two.)

A. In the body of a tag where body-content is set to scriptless

B. In the body of a tag where body-content is set to JSP

C. In the body of a tag where body-content is set to tagdependent
D. Within a JSP scriptlet
E. Within a JSP expression
F. Within a JSP declaration

15. Consider these pairings of Java method signatures and EL function method signatures for a TLD file. Which pairings go together and will work? (Choose two.)

A.

```
Java:
public String getNameForId(String id)
TLD:
public String getNameForId(String id)
```

B.

```
Java:
public static String getNameForId(int id)
TLD:
java.lang.String getNameForId(int)
```

C.

```
Java:
public static java.lang.String getNameForId(java.lang.String id)
TLD:
java.lang.String getNameForId(java.lang.String)
```

D.

```
Java:
static String getNameForId(String id)
TLD:
static java.lang.String getNameForId(java.lang.String id)
```

E.

```
Java:
public static String getNameForId(String id)
TLD:
public static java.lang.String getNameForId(java.lang.String)
```

The "Classic" Custom Tag Event Model

16. What is output from the following JSP document? (Choose one.)

JSP document:

```
<html xmlns:mytags="http://www.osborne.com/taglibs/mytags"
      xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c="http://java.sun.com/jsp/jstl/core" >
  <jsp:output omit-xml-declaration="true" />
  <jsp:directive.page contentType="text/html" />
  <head><title>Chapter 8 Question 16</title></head>
  <body>
    <c:set var="counter" value="0" />
  <p><mytags:question16>${counter}</mytags:question16></p>
  </body>
</html>
```

Tag Handler Code for question16 custom action:

```
import java.io.IOException;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.BodyTagSupport;
import javax.servlet.jsp.tagext.*;
public class Question16 extends BodyTagSupport {
  public int doAfterBody() throws JspException {
    int i = Integer.parseInt("" + pageContext.getAttribute("counter"));
    if (i > 5) {
      return Tag.SKIP_BODY;
    } else {
      pageContext.setAttribute("counter", "" + ++i);
  return IterationTag.EVAL_BODY_AGAIN;
    }
  }
}
```

A. 0123456
B. 01234
C. 12345
D. 23456
E. 123456
F. A blank page

17. Which of the following are valid statements relating to the `<body-content>` element in the tag library descriptor? (Choose two.)

    A. A closing tag may never be used for a custom action whose `<body-content>` element is set to "empty."

    B. A `<body-content>` setting of scripting-allowed permits JSP scriptlets in the body of the custom action.

    C. A `<body-content>` setting of jsp-document permits the use of JSP document syntax in the body of the custom action.

    D. To permit JSP expressions but not EL, `<body-content>` should be set to JSP.

    E. To permit EL but not JSP expressions, `<body-content>` should be set to scriptless.

    F. JSP expressions are legal in the body of a custom action whose `<body-content>` is set to tagdependent, but they will not be translated.

18. What is the result of accessing the following JSP document? (Choose one.)

```
JSP document:
<html xmlns:mytags="http://www.osborne.com/taglibs/mytags"
    xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core" >
  <jsp:output omit-xml-declaration="true" />
  <jsp:directive.page contentType="text/html" />
  <jsp:directive.page import="java.io.Writer" />
  <head><title>Chapter 8 Question 18</title></head>
  <body>
    <jsp:scriptlet>Writer myOut = pageContext.getOut();</jsp:scriptlet>
  <p><mytags:question18>
      <jsp:scriptlet>myOut.write("Body");</jsp:scriptlet>
    </mytags:question18></p>
  </body>
</html>
```

```
Tag handler code for question18 custom action:
import java.io.IOException;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.*;
public class Question18 extends BodyTagSupport {
  public int doAfterBody() throws JspException {
    try {
      bodyContent.write("Legs");
      bodyContent.writeOut(bodyContent.getEnclosingWriter());
```

```
    } catch (IOException e) {
      throw new JspException(e);
    }
    return Tag.EVAL_PAGE;
  }
  public void doInitBody() throws JspException {
    try {
      bodyContent.write("Head");
    } catch (IOException e) {
      throw new JspException(e);
    }
  }
}
```

A. Translation error (source generation)
B. Translation error (source compilation)
C. Run-time error
D. Output of HeadBodyLegs
E. Output of BodyHeadLegs
F. Output of BodyHeadBodyLegs

19. Which TLD tag declarations would best fit this tag handler code? (Choose two.)

Tag handler code:

```
package webcert.ch08.examp0804;
import java.io.IOException;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.tagext.TagSupport;
public class Question19 extends TagSupport {
  private int data;
  int getData() {
    return data;
  }
  void setData(int data) {
    this.data = data;
  }
  public int doEndTag() throws JspException {
    try {
      pageContext.getOut().write(id + ":" + data);
    } catch (IOException e) {
      throw new JspException(e);
```

```
    }
    return Tag.EVAL_PAGE;
  }
}
```

A.

```
<tag>
  <name>question19a</name>
  <tag-class>webcert.ch08.examp0804.Question19</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>id</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>data</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

B.

```
<tag>
  <name>question19b</name>
  <tag-class>webcert.ch08.examp0804.Question19</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>value</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

C.

```
<tag>
  <name>question19c</name>
  <tag-class>webcert.ch08.examp0804.Question19</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>data</name>
    <required>false</required>
```

```
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

D.

```
<tag>
  <name>question19d</name>
  <tag-class>webcert.ch08.examp0804.Question19</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>id</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

E.

```
<tag>
  <name>question19e</name>
  <tag-class>webcert.ch08.examp0804.Question19</tag-class>
  <body-content>empty</body-content>
</tag>
```

20. For the given interfaces, which of the following are valid sequences of method calls according to custom tag life cycle? (Choose three.)

A. Tag: setParent, setPageContext, doStartTag

B. Tag: setPageContext, setParent, doStartTag, doEndTag

C. IterationTag: doStartTag, doInitBody, doAfterBody, doEndTag

D. IterationTag: doStartTag, doAfterBody, doAfterBody, doAfterBody

E. BodyTag: setBodyContent, doInitBody, doStartTag, doAfterBody

F. BodyTag: doInitBody, setBodyContent, doStartTag, doEndTag

G. BodyTag: doStartTag, setBodyContent, doInitBody, doAfterBody

LAB QUESTION

In this lab you are invited to write a custom tag that extends BodyTagSupport but overrides every tag life cycle method. Give the tag some attributes that will dictate the return codes from each of the life cycle methods. So, for example, suppose the tag has an attribute called *rcStartTag,* which accepts a

value of 0, 1, or 2. Within the `doStartTag()` method for the tag handler, return Tag.SKIP_BODY if 0 is specified, Tag.EVAL_BODY_INCLUDE if 1 is specified, or BodyTag.EVAL_BODY_BUFFERED if 2 is specified for the *rcStartTag* attribute.

Write an HTML page that allows you to specify different values for each of the tag's attributes, and forward these to a JSP document that receives these values as parameters and uses EL to supply the values to an occurrence of your tag within the document. It doesn't really matter what the tag does, but try to get the page to reflect the tag life cycle methods called and the order in which they are called.

SELF TEST ANSWERS

Tag Libraries

1. ☑ **C**, **D**, and **E** are correct. **C** is correct because the attribute `isbn` can take a run-time expression and an EL expression is supplied using correct syntax. Also, the tag closes itself—although it's strange for a tagdependent tag to lack a body, it's not illegal. **D** is correct, this time supplying a constant for the required attribute `isbn`. **E** is correct—the opening tag has the same syntax as answer **C** (supplying an EL expression), but this time there is a body that is correctly rounded off by the end tag.
   ☒ **A** is incorrect because the attribute `isbn` is required, and it's missing in the syntax here. **B** is incorrect because the tag doesn't close itself, nor is there an end tag after a body. Though the tag may still work, the result could be catastrophic for any page source coming after, for this might be interpreted as the body of the tag.

2. ☑ **A** is the correct answer. **A** specifies the right sequence of elements—`<url-pattern>` and `<scripting-invalid>` nested inside `<jsp-property-group>` nested inside `<jsp-config>`. Furthermore, the URL pattern used (*.jsp) is correct—the leading slash is not required.
   ☒ **B** is incorrect: `<jsp-property-group>` is missing, and `<scriptless>` is an incorrect element name. **C** is incorrect because `<jsp-property-group>` is missing again, the value for `<url-pattern>` wrongly begins with a leading slash, and `<el-ignored>`—although a correct element name—controls expression language, not scripting. **D** is incorrect because `<uri-pattern>` and `<script-invalid>` are subtly wrong element names. **E** is incorrect—it's well formed with correct element names for ignoring expression language, but no good for suppressing scripting in all .jsp files.

3. ☑ **A** matches to **5** (tlib-version), **B** matches to **9** (short-name), **C** to **10** (uri), **D** to **8** (name), **E** to **6** (tag-class), **F** to **13** (body-content), **G** to **8** (name, again), **H** to **14** (required), and **I** to **16** (rtexprevalue).
   ☒ All other combinations are incorrect.

4. ☑ **A** and **D** are correct answers. **A** will work because web.xml is set to servlet version level 2.3, and, although empty of any detail, a JSP container will interpret that as meaning that EL should be ignored. **D** is correct, for although web.xml is at servlet version level 2.4, the necessary elements are included to turn off EL. Note the URL pattern to designate all files: /* (a single forward slash without the asterisk should also work, as this is the default mapping)

   ☒ **B** is incorrect, for the (empty) deployment descriptor is at servlet version level 2.4—at which EL is, by default, enabled. **C** is incorrect because the `<jsp-config>` element isn't

recognized in a 2.3 deployment descriptor. Your web application won't even start if XML validation is performed against the DTD. **E** is incorrect on two counts: A pattern of *.* is illegal, and the element name is `<el-ignored>`, not `<el-invalid>`.

5. ☑ **A**, **B**, and **E** are correct. **A** is correct because the prefix "mytags" must match the `taglib` directive. **B** is correct because the body of the tag contains a Java language expression, and JSP is the only value that will permit this within the body content. **E** is correct because to support an EL expression as the value for the *currency* attribute, the attribute must have its *rtexprvalue* set to true.
☒ **C** is incorrect because the tag's `name` element is `convert`—it's the `name` attribute of one of its attributes, which is set to `currency`. **D** is incorrect because we can't infer anything just by looking at the tag use about whether the currency attribute is required or not; it might be fine to leave it out altogether. **F** is incorrect because tags don't have a mandatory element—that's made up.

JSTL

6. ☑ **E** is the correct answer. If *escapeXml* is set to false, then no characters are converted, so all of the listed characters are not converted.
☒ **A**, **B**, **C**, and **D** are incorrect because no character conversion takes place. If *escapeXml* were set to true, then the less than sign (<) would be converted to the entity <. **F** is incorrect because it implies that all the listed characters would be converted.

7. ☑ **A**, **B**, and **C** are the correct answers. **A** is incorrect because the syntax including the *scope* attribute demands that there is a *var* attribute—the name of the variable to set to a value. **B** is incorrect because although the *var* attribute is present, it can't accept a run-time EL expression. **C** is incorrect because the optional *scope* attribute can't accept a run-time EL expression either.
☒ **D** is valid syntax, so an incorrect answer. The *target* and *property* attributes are set correctly (*target* can accept EL expressions), and the value is in the body of the tag. **E** is also valid syntax, and so an incorrect answer. This time, *value*, *target*, and *property* are set as attributes, with the values for *value* and *property* coming from EL expressions, which is legal.

8. ☑ **A** matches to **5** (c:set), **B** matches to **7** (var), **C** to **10** (value), **D** to **14** (c:forEach), **E** to **16** (step), **F** to **4** (3), **G** to **11** (varStatus), **H** to **10** (value, again), **I** to **3** (num + num), and **J** to **14** (c:forEach, again).
☒ All other combinations are incorrect.

9. ☑ **C** is the correct answer. A NullPointerException occurs. This is because the Reader returned by the `<c:import>` action is available only within the body of the action. Because `<c:import>` has no body in the example code, the reader drifts out of scope immediately. When the scriptlet code accesses myReader, **null** is returned, and when the container attempts to invoke the `read()` method, a NullPointerException occurs.
☒ **A** is incorrect, for the file is never read. This would, however, be the correct answer if the scriptlet were properly enclosed inside the body of the `<c:import>` action. **B** is incorrect; the NullPointerException preempts the possibility of an IOException. **D** is incorrect; no other sorts of exception occur, and **E** and **F** are incorrect because the page source generates and compiles successfully.

10. ☑ **A** is the correct answer. It is possible to specify the *items* attribute on its own, in which case the loop will work through each item in the given collection.
☒ **B** is incorrect because there isn't a *collection* attribute. **C** is incorrect because although it is very common to specify the *var* attribute as well as the *items* attribute, you don't have to have each object in the collection available to you within the loop. **D** and **E** are incorrect—although they specify a correct combination of attributes, it is possible and legal to use less as shown in the correct answer. Finally, **F** is incorrect both because the number of attributes is too high, and because *collection* is a made-up attribute.

EL Functions

11. ☑ **C**, **D**, and **E** are the correct answers. All a Java class needs to support an EL function is a method that is declared as both public and static (hence, both **C** and **D** have to be true for the same function—but there is nothing in the answers that says that this can't be so). **E** is also true: Bizarre as it may seem, a class's main method can be exposed as an EL function, just because it is public and static. The facts that main returns nothing (**void**) and receives an array as a parameter are not barriers to EL function status.
☒ **A** and **B** are incorrect, because anything at instance level within the class (such as instance variables and constructors) is irrelevant for EL functions: It doesn't matter if these are present in the class or not. **F** is incorrect because it is allowable for an EL function not to return anything to the JSP page that uses it (provided the TLD states that the function signature has a return type of "void").

12. ☑ **B** is the correct answer, for it has the correct syntax for an EL function declaration in the TLD file.
☒ **A** is incorrect because three of the element names are prefixed with `el-`, which they should not be. **C** is incorrect for the same reason as **A**, and also because the String return type and

String parameter must both be expressed in fully qualified form: java.lang.String. **D** is incorrect because the function signature must not contain the **public** or **static** keywords, and the parameters must not be named—only the type is present in the function signature (unlike Java). **E** is wrong because it contains a combination of the errors in the other wrong answers, already explained.

13. ☑ **A** is the correct answer—there is a translation error in code generation. You cannot embed one EL function inside another, as depicted. The JSP code generator will choke on the second { (before encountering the terminating }).
☒ **B** and **C** are incorrect because the code is never generated to compile and run. **D** and **E** would occur only if the code did run. The reasoning in **D** is also incorrect: Just because `arg1` and `arg2` don't have values in the JSP you can see, that is not to say that these could not be attributes set in some previous code in request, session, or application scope.

14. ☑ **A** and **B** are correct. **A** is correct—where body-content is designated as scriptless, that only means that Java language constructs are disallowed: scriptlets, declarations, and Java language expressions. EL functions (and other EL constructs), though, can be used in a scriptless body. **B** is correct—if body content is defined as JSP, then any kind of legal JSP syntax (EL functions included) goes.
☒ **C** is incorrect—although you can place EL function syntax in the body of a tagdependent action, it will be treated simply as template text; the container will not attempt to invoke the function. **D**, **E**, and **F** are incorrect—scriptlets, expressions, and declarations can contain only legal Java syntax. EL syntax is clearly not Java syntax, so it will cause compilation failure.

15. ☑ **B** and **C** are the correct answers. **B** is correct in all respects: The Java method is declared as public and static, and the matching TLD function signature uses fully qualified types where necessary. **C** is also correct: Although it isn't necessary to declare the String class with its full package name in the Java method signature (java.lang is implicitly available in every class), there is nothing wrong with doing that. Crucially, the TLD function signature does use fully qualified types for both the parameter and the return type.
☒ **A** is incorrect for several reasons: The Java method is not static; the TLD must not use the public keyword; the TLD must fully qualify the String return type and parameter (as java .lang.String); and the TLD must not name the parameter as the Java method signature does (so `(java.lang.String)` instead of `(String id)`). **D** is incorrect for several reasons as well: The Java method must be public; the TLD must not use the static keyword; the TLD must not name the *id* parameter (only state its type). **E** is incorrect only because the TLD uses the public and static keywords (which it must omit; in this respect, it doesn't match the Java method signature).

The "Classic" Custom Tag Event Model

16. ☑ **F** is the correct answer. Because the tag handler inherits from BodyTagSupport, and does not override the `doStartTag()` method, the default `doStartTag()` method is invoked, which returns a value of EVAL_BODY_BUFFERED. So all the body content—through all the iterations—is written to the BodyContent object. However, the contents of the BodyContent object are never sent to its enclosing writer—so they are completely lost.
☒ **A**, **B**, **C**, **D**, and **E** are incorrect because there is no output. If the problem outlined in the correct answer were fixed, then answer **A** would be correct—the body would be evaluated seven times before exit from the implicit loop in `doAfterBody()`—giving an output of 0123456.

17. ☑ **E** and **F** are the correct answers. **E** is correct because a value of `scriptless` for `<body-content>` does allow EL to run, but bans any language scripting—so JSP expressions, declarations, and scriptlets result in a translation error. **F** is correct as a value of `tagdependent`, for `<body-content>` turns off the JSP container's translation process (and validation) for the body of a tag. It's up to the tag handler logic to do something with the body content; the JSP container won't translate JSP expressions as it normally would.
☒ **A**, **B**, **C**, and **D** are incorrect. **A** is incorrect because although a tag whose `<body-content>` is set to "empty" must not have a body, a closing tag can still be present, like this: `<mytags:dosomething></mytags:dosomething>`. **B** is incorrect because there is no such allowed value as scripting-allowed for `<body-content>`; use a value of JSP when JSP scriptlets are permitted in the custom action's body. **C** is incorrect because there is no such setting as jsp-document either—and indeed, no setting of `<body-content>` has any effect on permitting or denying JSP document syntax. **D** is incorrect because the settings of `<body-content>` are cumulative. JSP is more permissive than scriptless. A setting of scriptless allows EL; a setting of JSP allows EL and language scripting as well. There is no way to allow scriptlets but disallow EL.

18. ☑ **E** is the correct answer. The key point to note is that the tag handler extends Body TagSupport. That means anything written to the BodyContent object is buffered. However, the scriptlet in the body of the tag uses the main JspWriter associated with the page. This is not buffered, so anything written to it is sent to page output straightaway, and *not* included in the body content for the tag. So the sequence of events is this: `doInitBody()` writes "Head" to BodyContent, which is buffered; the body is evaluated, so running the scriptlet that writes "Body" directly to page output; then `doAfterBody()` writes "Legs" to Body Content, which is buffered; then `doAfterBody()` writes the current accumulated content of

BodyContent ("HeadLegs") to the enclosing writer—which is in this case the main JspWriter for the page.

☒ **A**, **B**, and **C** are incorrect, for there are no translation or run-time problems. **D** and **F** are incorrect because of the reasoning given in the correct answer.

19. ☑ **D** and **E** are the correct answers. The only attribute exposed by the tag handler is *id* (`setId` is inherited from TagSupport). So answer **D**, which declares the custom action with the *id* attribute, is a good choice. So is answer **E**, which declares no attributes; there is no compulsion to use an available setter method in tag handler code as a declared attribute in the tag library descriptor.

☒ **A** is incorrect because it exposes *data* as an attribute (as well as *id*, which is fine). Although there is a `setData()` method within the tag handler code, the method access is at package level, meaning that JSP container code cannot call the method. So it doesn't count from the point of view of attribute definition. **B** is incorrect because although there is a `setValue(String key, Object value)` method inherited from TagSupport, this style of signature is wrong for exposing an attribute called *value* (this method is designed for a different purpose altogether). **C** is incorrect for the same reason as answer **A**.

20. ☑ **B**, **D**, and **G** are correct. **B** is correct because it correctly places the setting of page context before parent. **D** is correct because you can have several successive calls to `doAfterBody()` in an IterationTag. **G** is correct because the setting of body content comes before `doInitBody()`, and both these methods are positioned correctly between `doStartTag()` and `doAfterBody()`.

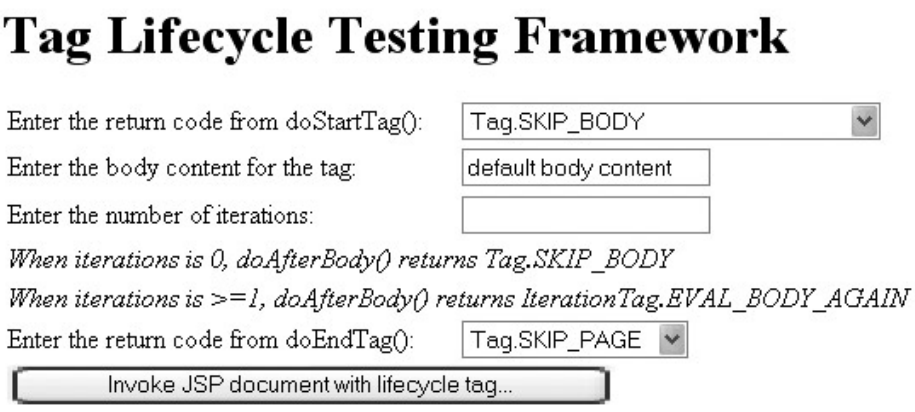
☒ **A** is incorrect because the setting of parent comes after the setting of page context—not before. **C** is incorrect because IterationTags don't have a `doInitBody()` method—only Body Tags do. **E** is incorrect because of the bad positioning of `doStartTag()`. **F** is incorrect because of the bad positioning of `doStartTag()`, and the reversing of `doInitBody()` and `setBody Content()`.

LAB ANSWER

Deploy the WAR file from the CD called lab08.war, in the /sourcecode/chapter08 directory. This contains a sample solution. You can call the initial HTML page with a URL such as

```
http://localhost:8080/lab08/lifecycle.html
```

The resulting page looks something like this:



Example output after pressing the submit button looks like the following illstration.

setPageContext()

setParent()

setRcEndTag - value: 5

setIterations() - value: 0

setRcStartTag() - value: 0

doStartTag(); returning Tag.SKIP_BODY

doEndTag(): returning Tag.SKIP_PAGE