# 9

# Custom Tags

## CERTIFICATION OBJECTIVES

It has been a long haul, but you are almost through the many objectives for JavaServer Pages and tag technology.

This chapter begins by sweeping up loose ends on tag handling code—showing you how tag handler code can access implicit variables. JSPs have implicit variables within their syntax. In the Java for tag handler code, you have to work a little harder.

Then we move on to two bigger topics, which are new to this version of the exam (and to JSP 2.0). The first topic is the simple tag life cycle. This is something of a misnomer, for tags are never that simple, but after fighting your way through the life cycle of Tag, IterationTag and BodyTag, you will probably appreciate the reduced frills of the new approach. The second topic is tag files. These give you a means of writing simple tags without writing a line of Java—everything is encapsulated in a JavaServer Page-like structure called a tag file.

The chapter finishes by looking at hierarchies of tags. Whether or not they follow the classic or simple model, some tags need to look up to their parents and grandparents. You learn about two techniques for tag handlers to get hold of other tag instances in the hierarchy.

## CERTIFICATION OBJECTIVE

# Tags and Implicit Variables (Exam Objective 10.2)

*Using the PageContext API, write tag handler code to access the JSP implicit variables and access web application attributes.*

Whereas JSPs have their own implicit variables (such as *request*, *session*, and *application*), your tag handler classes don't. What they have instead is a PageContext object. This part of the chapter explores how you can use methods on this object to get hold of the implicit variable equivalents. It's hard to write a meaningful tag handler without reference to the supplied PageContext, so we have inevitably covered some of this ground already. This short chapter section will amplify the knowledge you have already gained.

## The PageContext API

You have seen that in naked JSPs, you have access to a *pageContext* implicit object — of type javax.servlet.jsp.PageContext. Because this is an abstract class, you never meet an object of this type directly — your kindly JSP container provides an instance of a subclass that has implemented all the abstract methods it contains.

For the most part (in naked JSPs), you can avoid calling many of the methods on the *pageContext* implicit object — such as `getRequest()`, `getSession()`, and `getServletConfig()`. There's nothing stopping you from doing this, but you already have references to the objects returned by those methods. These take the shape of other implicit variables — such as *request*, *session*, and *application*.

In tag handler code, you don't have that luxury. The PageContext object is passed to you in the `setPageContext()` method. If your tag class inherits from TagSupport (or BodyTagSupport), `setPageContext()` saves the object to a protected instance variable so that you can access it directly in your code as *pageContext*—which conveniently matches the name of the *pageContext* implicit variable used directly inside JSP page source.

However, there is no equivalent mechanism for the remaining implicit variables. You could imagine the JSP container providing a call to a tag life cycle method called `setRequest(HttpServletRequest request)`, giving an opportunity to save the implicit request object, yet it doesn't. You have to do a little work yourself. But it's not hard. *pageContext* is a conduit to all the other implicit variables: You simply have to know the right method to call.

### Accessing JSP Implicit Variables

Table 9-1 lists the nine JSP implicit variables, together with the PageContext method needed to obtain the equivalent instance objects.

**exam Watch**

***Mostly, you can mentally convert the implicit variable name to the PageContext method name by prefixing "get" to the implicit variable. Just watch out for the two exceptions to this rule: config maps to getServletConfig, and application to getServletContext.***

**TABLE 9-1** JSP Implicit Variables and Equivalent PageContext Methods

| JSP Implicit Variable | PageContext Method Used to Obtain Equivalent Object |
|---|---|
| Request | `getRequest()` |
| Response | `getResponse()` |
| Out | `getOut()` (inherited from PageContext's parent class JspWriter) |
| Session | `getSession()` |
| Config | `getServletConfig()` |
| Application | `getServletContext()` |
| Page | `getPage()` |
| PageContext | (This *is* the PageContext object passed to your tag handler) |
| Exception | `getException()` |

The following example shows tag handler code using PageContext to exercise methods on objects that are otherwise implicit to the JSP:

```
public int doStartTag() throws JspException {
  ServletRequest request = pageContext.getRequest();
  String var1 = request.getServerName() + ":"
    + request.getServerPort();
  ServletResponse response = pageContext.getResponse();
  String var2 = response.getCharacterEncoding() + ";"
    + response.getContentType();
  ServletConfig config = pageContext.getServletConfig();
  String var3 = config.getServletName();
  try {
    Writer out = pageContext.getOut();
    out.write(var1 + ";" + var2 + ";" + var3);
  } catch (IOException e) {
    e.printStackTrace();
  }
  return Tag.EVAL_BODY_INCLUDE;
}
```

You can deploy this code from the WAR file in the CD at /sourcecode/ch09/examp0901.war. Here is the URL to run it:

```
http://localhost:8080/examp0901/asTag.jsp
```

When I run this code, the output I see is this:

```
localhost:8080;ISO-8859-1;text/html;jsp
```

The output of localhost:8080 originates from the calls to `getServerName()` and `getServerPort()` on the ServletRequest object returned by `pageContext.getRequest()`. Likewise, ISO-8859-1 and text/html are derived from `getCharacterEncoding()` and `getContentType()` calls on the ServletResponse object returned by `pageContext.getResponse()`. The string jsp is the servlet name retrieved from the ServletConfig object.

All of these pieces of information are held in String variables and are concatenated together. `PageContext.getOut()` is used to return the JSP's associated Writer and send the output to the page.

If the above code were written within a JSP using implicit variables, it might look like this:

```
<p><%= request.getServerName() %>:<%= request.getServerPort() %>;
<%= response.getCharacterEncoding() %>;<%= response.getContentType() %>;
<%= config.getServletName() %></p>
```

This JSP is available in the same WAR file as you deployed above (examp0901.war), and the URL to run it is

```
http://localhost:8080/examp0901/asJSP.jsp
```

**exam Watch**

***The only implicit variable that isn't necessarily always available is* exception. *Recall that the* exception *implicit variable is available only in a JSP error page. The JSP container should be used to load your error page only indirectly—when another page specifies your error page and when that other page goes wrong. The exception produced by the other page is then available to your error page, as the implicit variable* exception *in the JSP source. If that JSP source has a custom tag within it, that custom tag can invoke* `pageContext.getException()` *and expect an Exception object back. Under other circumstances, the call will return* null.

### Accessing Attributes with PageContext

You already know how to access attributes with the PageContext object—take a look again at Chapter 6 if you need a reminder. All four scopes are available to you: page, request, session, and application.

All you haven't yet done is use PageContext inside of tag handler code to set or get attribute values. We'll consider a small example where a tag handler performs a simple mathematical function using attributes to handle the input and outputs to the JSP page. Here's the JSP page first of all:

```
01 <%@ taglib prefix="mytags" uri="http://www.osborne.com/taglibs/mytags" %>
02 <html><head><title>Squaring Function</title></head>
03 <body><p>The square of ${param.input}
04 <% pageContext.setAttribute("input", request.getParameter("input")); %>
05 <mytags:square />
06 is ${output}
07 </p></body></html>
```

This example is available from the same WAR file you deployed previously, examp0901.war. You can call this JSP with a URL such as

```
http://localhost:8080/examp0901/square.jsp?input=2
```

Then the output is

```
The square of 2 is 4
```

The tag handler code in `doStartTag()` that performs the calculation looks like this:

```
01 String input = (String) pageContext.getAttribute("input");
02 int i = Integer.parseInt(input);
03 i = i * i;
04 pageContext.setAttribute("output", new Integer(i));
05 return Tag.SKIP_BODY;
```

Let's follow through the logic of the request to the JSP:

- In the URL an input parameter called *input* is passed with a value of 2.
- The JSP displays the value of this input parameter using EL on line 03: `${param.input}`.
- In line 04 the JSP uses the *pageContext* implicit variable to set up an attribute called *input*. The value of this attribute is taken from the request parameter we

just saw displayed. Because no scope is specified within the `setAttribute()` method, scope will default to `page`.

- At line 05 of the JSP, we find the bodiless `<mytags:square>` tag inserted. There is no output from this, but the tag handler code is executed.
- Within the tag handler code, at line 01 the value of the page context attribute called `input` is recovered into a local String variable.
- At line 02 the value of the String input is coerced to an **int** value. (This code is not production-ready as you can see—there's no defensive coding for bad input!)
- At line 03 we have the calculation: The input is squared (and still held in the **int** local variable *i*).
- At line 04 the result of the calculation is stored in an Integer object. This is used as the value of an attribute called *output*, set up in page scope.
- At line 05 we return from `doStartTag()`, skipping the body.
- Now—back at line 06 in the JSP—we are ready to display the result of the calculation. EL has direct access to attributes by name, hence the simple statement `${output}`.

From this small example, you see combined several techniques that you have learned so far. In particular, you see how attributes can be used to communicate between a JSP and a tag handler.

## EXERCISE 9-1


### Tags and Implicit Variables

In this exercise you will set up a tag handling class with instance variables whose names match the implicit variables in a standard JSP. You already have an example—in TagSupport—in the protected instance variable *pageContext*. You might even find this class useful as the base class for tag handlers of your own.

For this exercise, create a web application directory structure under a directory called ex0901, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch09/ex0901.war.

#### Create the Tag Handler Class

1. Create a Java source code file in /WEB-INF/classes or an appropriate package directory within it called ImplicitsTag.java.
2. In the class declaration, extend (javax.servlet.jsp.tagext.)TagSupport.

3. Declare instance variables called *request*, *response*, *session*, *application*, *config*, *out*, *page*, and *exception*. These should match their appropriate (implicit variable) type.
4. Override the `setPageContext()` method. However, invoke `super.setPageContext()` so that TagSupport's default action (to save the page context in a protected variable called pageContext) is honored. In the following lines of code, invoke the appropriate methods on pageContext to initialize the instance variables you declared in the previous request. For example,

   ```
   request = pageContext.getRequest();
   ```

   to initialize the ServletRequest instance variable.
5. In `doStartTag()`, exercise any methods you like on any of your instance variables. Use this as an opportunity to revise long-unused javax.servlet APIs that you last tried in the early chapters.
6. Whichever methods you do choose, make sure to obtain some information from each available implicit attribute, and record this in a request attribute. So, for example, you might use the `request.getRemoteHost()` method to return String information to record in an attribute called `requestInfo`.
7. Conclude the `doStartTag()` method by skipping the body.
8. Save and compile ImplicitsTag.java.

### Create a Tag File

9. Create a TLD file called mytags.tld in /WEB-INF/tags.
10. Include within it a definition for a tag named *implicits*. The tag class should tie in with the ImplicitsTag class you created previously. Remember that the tag has no body.

### Create a JSP File

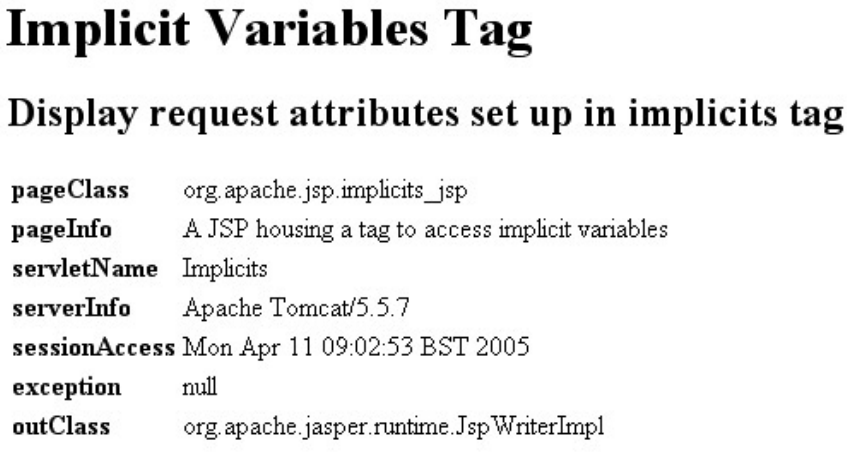
11. Create a JSP file called implicits.jsp directly in the context root directory ex0901.
12. Declare the tag file mytags.tld, with a prefix of mytags.
13. Include `<mytags:implicits />` somewhere in the JSP.
14. Somewhere after the tag, display the attributes you set up in the `doStartTag()` method of ImplicitsTag.java.

### Create, Deploy, and Test a WAR File

15. Create a WAR file from your ex0901 context directory, and deploy this to the Tomcat server.
16. Test your application by invoking the JSP with a URL such as

```
http://localhost:8080/ex0901/implicits.jsp
```

17. The output will, of course, vary depending on the choices you made about information to retrieve to the displayed attributes. The solution code produces the following output:




## CERTIFICATION OBJECTIVE

# The "Simple" Custom Tag Event Model (Exam Objective 10.4)

*Describe the semantics of the "Simple" custom tag event model when the event method (doTag) is executed; write a tag handler class; and explain the constraints on the JSP content within the tag.*

In Chapter 8, you learned about the "classic" tag event model. Although this is highly flexible, it is complex to learn. This explains part of the motivation for the J2EE designers to produce a "simple" model for tag production. Instead of three life cycle choices based on the interfaces Tag, IterationTag, and BodyTag, you have only one. This is based on the javax.servlet.jsp.tagext.SimpleTag interface, and in this section, we'll see how to write a tag based on this interface and learn how the J2EE container makes use of it. You'll find that you can do almost all the things you are able to do in the classic model, but you "do it yourself" instead of relying on different API calls and method return codes.

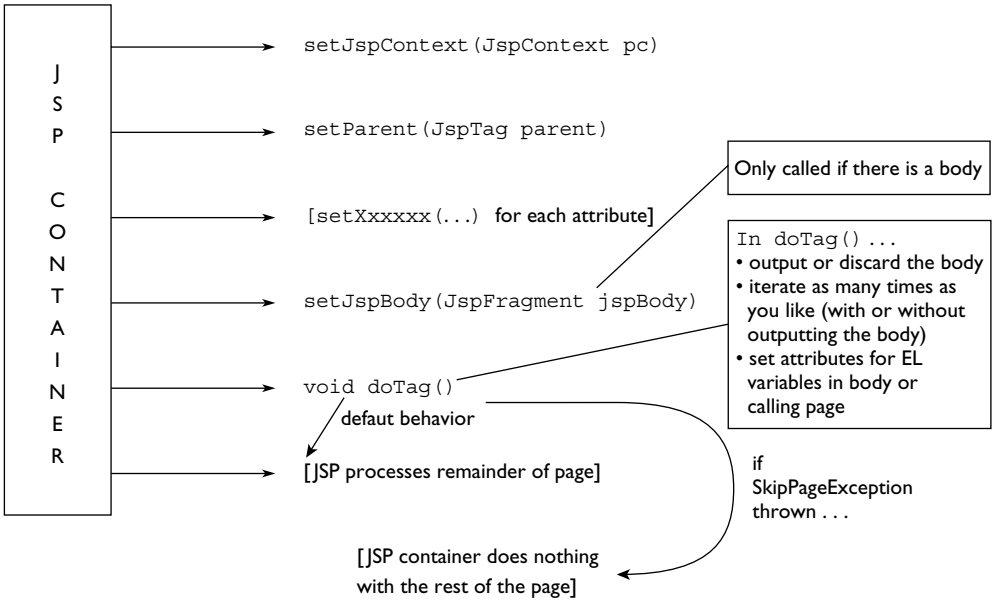
Since SimpleTag is a new innovation with JSP 2.0, you can expect plenty of exam questions relating to it. Don't make the mistake of thinking that simple tags are simple—perhaps relative to classic tags they are, but they still demand study and practice!

## The Simple Tag Model Life Cycle

The simple model for tags is shown in Figure 9-1. At first sight, it looks like the classic model. There is a method that provides some kind of content object. There is a method to set the parent tag, followed by methods to set attributes. Following this,

**FIGURE 9-1** JSP container processing one occurrence of a tag implementing the SimpleTag Interface in one JSP page:

The Simple Model Tag Life Cycle



another method provides the body content of the tag to the tag handler. So far, so classic—though you might have already noticed some unfamiliar parameter types (such as JspContext, JspTag, and JspFragment—where you might have expected PageContext, Tag, and BodyContent).

Next comes the `doTag()` method, and this is where the big difference lies. This method replaces all of `doStartTag()`, `doAfterBody()`, and `doEndTag()`. All the processing that would have occurred in those methods moves to `doTag()`. Furthermore, `doTag()` returns nothing at all. With simple tags, you don't use return codes (such as Tag.SKIP_BODY or BodyTag.EVAL_BODY_BUFFERED) to influence the life cycle. The life cycle as it directly affects the tag output is controlled instead by your code inside `doTag()`.

## Life Cycle Details

Any tag handler class you write needs to implement the javax.servlet.jsp.tagext .SimpleTag interface. Nearly all its methods are designed to have implementations that are called by the JSP container, as shown in Figure 9-1. Let's explore the complete life cycle in more detail. It begins within the thread running the servlet generated from your JSP page that uses an occurrence of the simple tag, and ends once that occurrence has been processed. The following table lists simple tag events in order.

| | |
|---|---|
| Construction | When the JSP container meets an occurrence of a simple tag, it makes a new instance of the tag handler class. The container calls the zero-argument constructor, so the tag handler class must have one, either explicitly defined or implicitly put there by the Java compiler. Because (unlike classic model tags) you get a new instance for every use, you can safely initialize variables in your constructor or instance member declarations. There is no "pool" for simple tags. |
| `setJspContext (JspContext pc)` | Saves the JspContext object for later access to attributes in all scopes and the JspWriter currently associated with the page. |
| `setParent (JspTag tag)` | Saves the JspTag for later access to this action's immediate parent. This method is called only if the custom action has a custom action as a parent. |
| `set<AttributeName> (<Type> attributeValue)` | The JSP container calls any "set" methods for attribute names defined in the tag library descriptor (just as for classic tag handlers). |
| `setJspBody(JspFragment jspBody)` | The JSP container calls this method only if the custom action has a body. If so, save the JspFragment object for later use—you'll need it to process the body later (the JSP container doesn't do that for you automatically). |

| | |
|---|---|
| `doTag()` | Within this method, you can do whatever you like. You are most likely to want to do one or both of the following: 1. Process the body (more than once if required) 2. Write directly to page output |
| Variable Synchronization | This is an advanced topic that should not come up on the exam. In brief, you can specify the equivalent of output parameters from your simple tag handler. The variable synchronization process moves these into attributes accessible in your page after the end of your tag. Of course, you can achieve this effect quite easily manually by setting up your own attributes in the tag handler code—as happens with the tag examples in the book. |
| Garbage Collection | There is no `release()` method for simple tags. Once a tag instance has been used, it is thrown into touch. So any cleanup must happen in `doTag()` (or a method called from `doTag()`), for this is the last method called by the container before garbage collection. You could include a `finalize()` method, although there is never an absolute guarantee that `finalize()` will be called. |

### SimpleTagSupport

Just as TagSupport and BodyTagSupport exist to provide default implementations of the classic tag handler interfaces, so SimpleTag has one of its own. The name won't come as any surprise: SimpleTagSupport. Consequently, it is usually easiest to extend SimpleTagSupport for your own simple tag handler classes, instead of implementing the SimpleTag interface directly.

SimpleTagSupport provides some predictable support for each of SimpleTag's methods, plus some extra useful methods as explained below:

- `setJspContext(JspContext pc)`, `setParent(JspTag parent)`, and `setJspBody(JspFragment jspBody)` store the objects passed in by the container for later access in your code by corresponding get methods (`getJspContext()`, `getParent()`, `getJspBody()`).

**exam Watch**

***A pedantic point, but one that might underpin a more picky examination question: All three set methods in the SimpleTagSupport class are implementations of SimpleTag interface methods. However, only one of the get methods—`getParent()`—derives from the interface. `getJspBody()` and `getJspContext()` are provided out of the kindness of SimpleTagSupport's designer's heart (or more likely because you need them to access otherwise private instance variables).***

- `doTag()` is a do-nothing implementation.
- `findAncestorWithClass(JspTag from, Class klass)` is an interesting static method that we look at in the final section of this chapter, on exploring tag hierarchies.

## An Example: A Unicode Converter

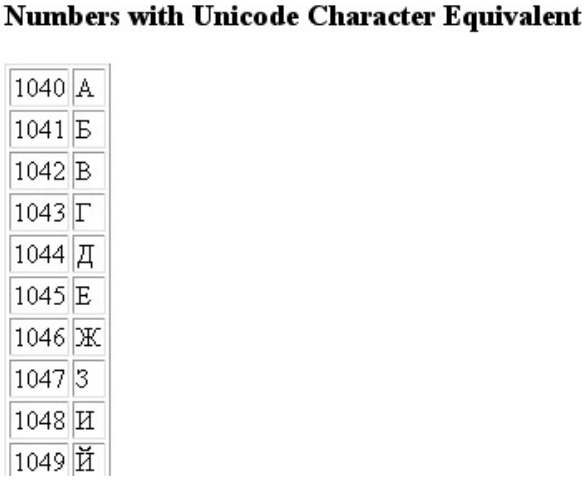
Enough theory on the APIs—let's look at a working example. This lives on the CD in /sourcecode/examp0902.war—deploy this as you will. This example uses a simple tag handler to list a sequence of numbers in an HTML table, together with their corresponding Unicode character equivalents. If you use a range of numbers from 0 to 255, you will see the extended ASCII character set. You may find it more exciting to explore some of the upper ranges of Unicode. The supplied JSP that uses the simple tag sets a range of 1040 to 1100, which causes the web page to display Cyrillic characters as shown in Figure 9-2. Call it with a URL such as

```
http://localhost:8080/examp0902/unicodeDisplay.jspx
```

If you don't see Cyrillic characters (just boring boxes instead), the chances are that you don't have a Cyrillic font installed.

**FIGURE 9-2**

Unicode Character Converter—First Few Characters Shown



Here is a JSP document that uses the simple tag (called unicodeConverter). You can see nothing to differentiate it from a classic tag invocation—the mechanism is the same:

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:mytags="http://www.osborne.com/taglibs/mytags">
  <jsp:output omit-xml-declaration="true" />
  <jsp:directive.page contentType="text/html" />
  <head><title>Unicode Converter</title></head>
  <body>
    <h3>Numbers with Unicode Character Equivalent</h3>
    <table border="1">
      <mytags:unicodeConverter begin="1040" end="1116">
        <tr><td>${number}</td><td>${character}</td></tr>
      </mytags:unicodeConverter>
    </table>
  </body>
</html>
```

You can see in the body of the `unicodeConverter` action a table row with two table cells, their data supplied from two EL variables: `${number}` and `${character}`. You can infer from Figure 9-2 and the JSP document source above that the tag handler is preoccupied with two tasks: setting up these EL variables and adding as many rows as requested. The number of rows—and range of characters displayed—is determined by the *begin* and *end* attribute values for the action.

Before we see the tag handler code, we will peek at the TLD. There's not much to see here—again, there's nothing particular to indicate that this is a simple tag we are dealing with. The declaration looks just like a classic tag:

```
<tag>
  <description>Shows Unicode Character for Number</description>
  <name>unicodeConverter</name>
  <tag-class>webcert.ch09.examp0902.CharConvSimpleTag</tag-class>
  <body-content>scriptless</body-content>
  <attribute>
    <name>begin</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
```

```
    <name>end</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

It's true that the `<tag-class>` (CharConvSimpleTag) hints at the tag's simple origins, but that's merely a naming choice by this developer. The only constraint on simple tag file declaration within the tag library descriptor is that `<body-content>` is restricted to three (instead of four) allowed values: `empty`, `tagdependent`, and `scriptless`. The fourth value—`JSP`—may get through XML schema validation, but the JSP container will give you a translation error: Simple tags are not allowed the full range of JSP syntax. Java language syntax within scriptlets, declarations, or expressions is disallowed. After all, this is meant to be a *simple* tag.

So to the tag handler code. Here it is in its entirety:

```
package webcert.ch09.examp0902;
import java.io.IOException;
import javax.servlet.jsp.JspContext;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.JspFragment;
import javax.servlet.jsp.tagext.SimpleTagSupport;
  public class CharConvSimpleTag extends SimpleTagSupport {
  private int begin;
  private int end;
  public int getBegin() { return begin; }
  public void setBegin(int begin) { this.begin = begin; }
  public int getEnd() { return end; }
  public void setEnd(int end) { this.end = end; }
  public CharConvSimpleTag() { super(); }
  public void doTag() throws JspException, IOException {
    for (int thisChar = getBegin(); thisChar <= getEnd(); thisChar++) {
      JspContext ctx = getJspContext();
      ctx.setAttribute("number", new Integer(thisChar));
      ctx.setAttribute("character", new Character((char) thisChar));
      JspFragment fragment = getJspBody();
      fragment.invoke(null);
    }
  }
}
```

This class extends SimpleTagSupport, so most of the life cycle methods are inherited from there. Beyond that, over half of the class is taken up with import statements and attribute handling code (instance variables and getters and setters for the attributes *begin* and *end*). The interest is in the `doTag()` method, which does the following things:

- Defines a **for** loop, designed to start at the value specified in the begin attribute and finish at the end attribute value.
- Gets hold of the JspContext object, using the `getJspContext()` method inherited from SimpleTagSupport.
- Sets up an attribute on the JspContext object, called *number*. This reflects the numeric value of the loop counter held in variable *thisChar*.
- Sets up a second attribute on the JspContext object, called *character*. The value for the attribute is a Character wrapper object. This is constructed with a **char** value derived from downcasting the **int** counter variable *thisChar*. This supplies the Unicode character when *character* is later accessed in the JSP page as an EL variable (`${character}`).
- Next, the code gets the JspFragment defining the body of the custom action—in other words, the following line of JSP page source:

```
<tr><td>${number}</td><td>${character}</td></tr>
```

- Next, the code calls the `invoke()` method on the fragment. The effect of this is to process this piece of JSP page source. Template text (such as `<tr><td>`) is written directly to page output. Any EL expressions (such as `${number}`) are evaluated before being sent to page output. Which writer is used? The answer lies in the parameter passed into the invoke method. You can define your own Writer and divert the JspFragment output there. But by supplying **null** as the parameter value, you are writing to the JspWriter associated with the JspContext. In other words, `getJspBody().invoke(null)` is shorthand for `getJspBody().invoke(getJspContext().getOut())`.
- This marks the end of the loop—which goes around again and is repeated as many times as necessary before reaching the *end* attribute value. Each time, the body of the custom action is reevaluated afresh—with a new table row, number, and character.

## INSIDE THE EXAM

Get clear in your mind the differences between the following:

- PageContext and JspContext
- BodyContent and JspFragment

PageContext and BodyContent belong to the classic tag model. JspContext and JspFragment are their equivalents (roughly speaking) in the simple tag model. There are many similarities, but there are significant differences as well. First, some points about javax.servlet.jsp .JspContext and javax.servlet.jsp.PageContext:

- Both PageContext and JspContext are classes, not interfaces.
- As a page author, you're never supposed to make a new one of either of these classes: You let the JSP container supply the instances.
- JspContext is the parent of PageContext (PageContext extends JspContext).
- JspContext contains all the methods to do with
  - Attribute access (e.g., `getAttribute()`, `setAttribute()`)
  - Writer access (`getOut()`)
  - Programmatic access to the EL evaluator (`getExpressionEvaluator()`, `getVariableResolver()`)—not something you are likely to encounter in the exam or encounter early in your simple tag development career.
- PageContext adds methods to do with
  - Accessing implicit objects in a servlet environment (e.g., `getRequest()`, `getResponse()`, `getServletContext()`).
  - Redirection (`forward()`, `include()`).

The idea behind JspContext was to abstract away all the parts that are not specific to the HTTP servlet environment. Of course, you are likely to be using JspContext in a HTTP servlet environment most, if not all, of the time. And indeed, many JspContext methods are designed to accept constants defined in the PageContext class, as in the following:

```
myJspContext.getAttribute("mySessionId", PageContext.SESSION_SCOPE)
```

However, at least you can be aware of the differences so you're not fooled by exam questions that include simple tag handler code of the following kind:

## INSIDE THE EXAM (continued)

```
HttpServletRequest request = myJspContext.getRequest();
```

which, of course, won't compile.

So what about the differences between javax.servlet.jsp.tagext.BodyContent and javax.servlet.jsp.tagext.JspFragment? These are more pronounced:

- BodyContent inherits from javax.servlet.jsp.JspWriter, which is a java.io.Writer.
- JspFragment isn't a Writer of any sort; it inherits directly from java.lang.Object.
- BodyContent has some content in it already when your classic custom tag handler code gets hold of it. This is the result of the JSP container evaluating the body of the tag—processing any scriplets or EL contained therein.
- JspFragment is the opposite: Its content constitutes the body *before* any evaluation has taken place. In your simple tag handler code, you control when to do the evaluation—if at all—by calling the `invoke()` method on the JspFragment object.
- There is no concept of buffering with JspFragment, as there is with BodyContent. Nothing of the body is buffered because nothing has been output until you decide. You can *simulate* a buffer by all means: Have the `invoke()` method write to a StringWriter, and there you have evaluated content, very like a BodyContent object.

### Terminating Early

Back in Figure 9-1, the simple tag life cycle diagram, you can see two routes out of the `doTag()` method: one the normal route, which results in the JSP container processing the rest of the page, the other bypassing evaluation of the rest of the page entirely.

Because `doTag()` has no return code associated with it, the mechanism for bypassing the rest of the page is through an exception. The exception to throw is javax.servlet.jsp.SkipPageException, a subclass of javax.servlet.jsp.JspException (which is in the *throws* clause of the `SimpleTag.doTag()` method signature).

## EXERCISE 9-2


### The "Simple" Custom Tag Event Model

This exercise is going to use a combination of an HTML file, JSP, and simple tag handler to display the contents of a given directory. For this exercise, create a web application directory structure under a directory called ex0902, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch09/ex0902.war.

#### Create the HTML File

1. Create an HTML file called fileBrowser.html directly in the context directory ex0902.
2. Include a form in the file that has a text input field, named *initDir*. The user will type into this field the name of the directory to browse. Don't forget to include a submit button in the form.
3. Make the action of the form a JSP document called fileBrowser.jspx.
4. Save and exit the file.

#### Create the JSP Document

5. Create a JSP document called fileBrowser.jspx directly in ex0902.
6. Include the following heading information, which ensures proper HTML output and declares standard actions, the core tag library, and your own tag library:

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:mytags="http://www.osborne.com/taglibs/mytags">
  <jsp:output omit-xml-declaration="true" />
  <jsp:directive.page contentType="text/html" />
```

7. Include an HTML table in the JSP document with three headings: "Name," "Size," and "Directory?"
8. After the headings, and before the main table row, include a tag declaration like this:

```
<mytags:fileBrowser initdir="${param.initdir}" size="true">
```

You can see that this is the opening tag of a tag named fileBrowser, with two attributes: *initdir* and *size*. The value for the *size* attribute is set to a constant of true. The value for *initdir* is supplied from the *initDir* request parameter set up in the HTML page (rendered through EL).

9. In the following table row, place `${file}` in the first table cell. This EL picks up an attribute set by the tag handler code we have yet to write.
10. In the next table cell, place `${size}`.
11. Place `${isDir}` in the next table cell.
12. After the table row, place the closing fileBrowser tag: `</mytags:FileBrowser>`.

### Create the Tag Library Descriptor File

13. Create a TLD file called mytags.tld in /WEB-INF/tags.
14. Include within it a definition for a tag named fileBrowser. The tag class should be webcert.ch09.ch0902.FileBrowserSimpleTag (follow whatever package naming convention you are using). The body content should be scriptless.
15. Define two attributes for the tag: *initDir* and *size*. Both are required, and both should allow run-time expressions.
16. Save and exit the TLD file.

### Enter a Tag Library Mapping in the Deployment Descriptor

17. In web.xml, ensure you have a tag library mapping. The URI is `http://www.osborne.com/taglibs/mytags`, and the tag library descriptor location is `/WEB-INF/tags/mytag.tld`.

### Write the Simple Tag Handler Code

18. Create a Java source code file in /WEB-INF/classes or an appropriate package directory within it. Call the file FileBrowserSimpleTag.java.
19. In the class declaration, extend (javax.servlet.jsp.tagext.)SimpleTagSupport.
20. Provide instance variables and getters and setters for the two attributes *initDir* and *size*.
21. Override the `doTag()` method. Within the method, create a java.io.File object from the value given for *initDir*. Use the `listFiles()` method on the File object to obtain an array of Files.

22. Still in the `doTag()` method, obtain the JspContext as a local variable.
23. Start a **for** loop based on the contents of the File array. For each file found, use `setAttribute()` on the JspContext local variable to set three attributes. The first is called *file*, and the value is the name of the current file in the loop. The second is called *size*, and the value is the size of the current file in the loop (`length()` method on File). The third is *isDir*, and the value is a Boolean object created from the output of the `isDirectory()` method on the current file in the loop.
24. Still within the loop, get the JspFragment associated with the tag handler (`getJspBody()`). Call the `invoke()` method on the fragment, passing in a **null** value. This has the effect of processing the body of the tag in the JSP. The body contains references to the three attributes whose values you have set.

### Create, Deploy, and Test a WAR File

25. Create a WAR file from your ex0902 context directory, and deploy this to the Tomcat server.
26. Test your application by invoking the JSP with a URL such as

```
http://localhost:8080/ex0902/
fileBrowser.html
```

27. Enter a valid directory (such as C:>), and click the submit button.
28. The output for the solution code (which includes one or two improvements not included in the exercise instructions) looks like the illustration to the left.

**Show Files in Requested Directory**

**Directory: C:\Program Files**

| Name | Size | Directory? |
|---|---|---|
| Adobe | (Dir) | Yes |
| Ahead | (Dir) | Yes |
| Apache Software Foundation | (Dir) | Yes |
| C-Media 3D Audio | (Dir) | Yes |
| CLOX | (Dir) | Yes |
| Common Files | (Dir) | Yes |
| ComPlus Applications | (Dir) | Yes |
| CyberLink | (Dir) | Yes |
| Google | (Dir) | Yes |
| HighMAT CD Writing Wizard | (Dir) | Yes |
| IBM | (Dir) | Yes |
| IDM Computer Solutions | (Dir) | Yes |

CERTIFICATION OBJECTIVE

# The Tag File Model (Exam Objective 10.5)

*Describe the semantics of the Tag File model; describe the web application structure for tag files; write a tag file; and explain the constraints on the JSP content in the body of the tag.*

Now we come to a variant of the simple tags we have met and written so far: tag files. In appearance and behavior, tag files are like JavaServer Pages. They contain a mixture of template text and elements. They are translated into Java source code and then compiled. The result is a simple tag class—yet with no necessity to write a simple tag handler.

Tag files are also convenient to deploy. You don't write tag entries in a tag library descriptor. Tag files are self-contained, with their own deployment directives.

We'll rewrite the simple tags we created in the last section as tag files and make comparisons between the two approaches. And along the way, we'll tease apart the exam objectives for tag files.

## Rewriting Simple Tags as Tag File

In the last section, you saw an example of a simple tag used to display a range of Unicode characters on a web page. We'll now look at a version of the tag when it is rewritten as a tag file—with some slight improvements. You may want to run the code, which is in the CD at /sourcecode/ch09/examp0903.war. Simply deploy the WAR file as you would any other solution code file. To run the code, point your browser to the following URL (adapt this to suit if you are not using standard Tomcat settings):

```
http://localhost:8080/examp0903/unicodeDisplay.jspx
```

This looks no different from the original version of the code (refer to Figure 9-2 to see how this looks in the browser). However, the underlying mechanism is different. In the remainder of this section, we'll revisit this example to explain how tag files work.

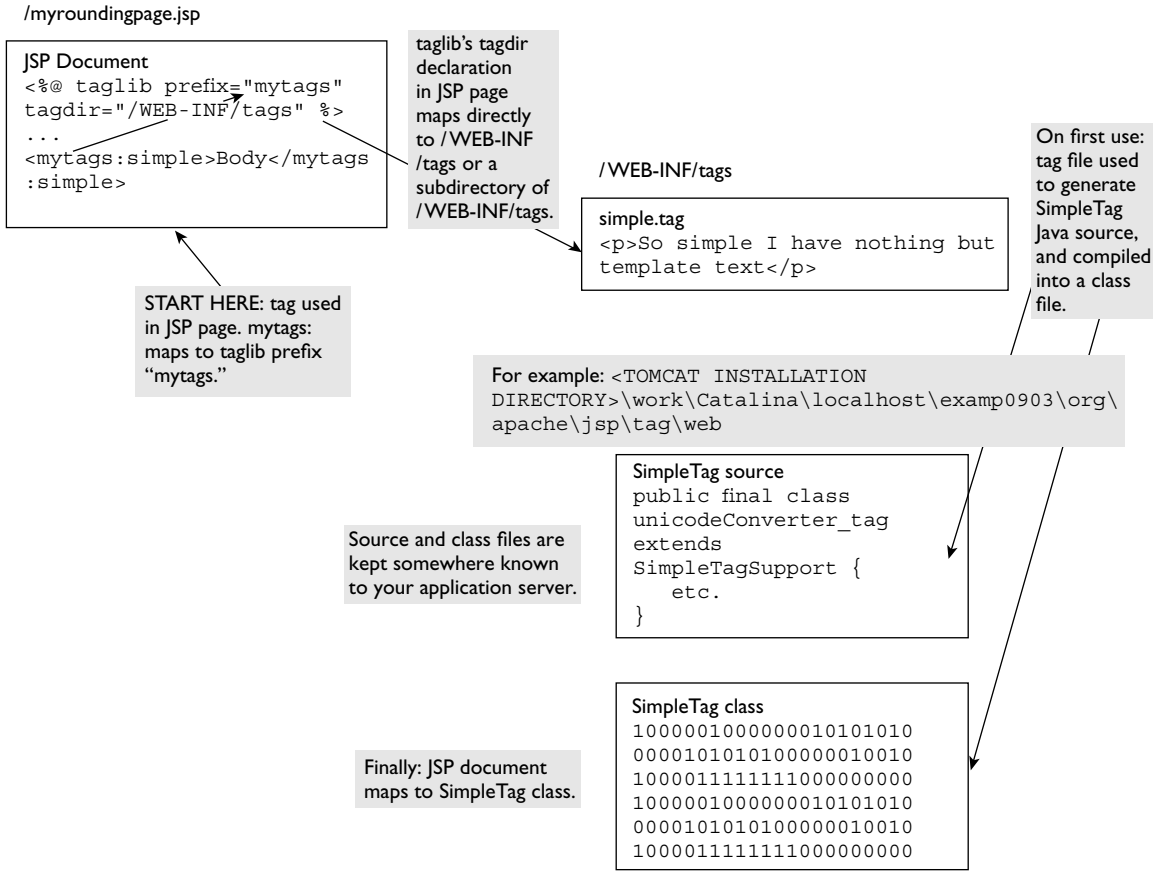
### Where Does the JSP Container Find the Tag File?

To ensure that your web container will find your tag file, you have to ensure that it has an extension of .tag and that it is located in one of these places within your web application:

- /WEB-INF/tags
- A subdirectory of /WEB-INF/tags
- In a JAR file kept in /WEB-INF/lib. The directory of the tag file within the JAR file must be /META-INF/tags — or a subdirectory of /META-INF/tags.

Let us first consider the case of a tag file in /WEB-INF/tags. What happens at run time, when the tag file is first accessed? This is shown in Figure 9-3. Tag files bear the same relationship to tag handler classes as JSP pages do to servlets. Both are used as sources from which a class file is generated. For tag files, a SimpleTag class is created

**FIGURE 9-3** Tag File Deployment and Run Time



when the tag file is first used—in much the same way as the use of a JSP triggers generation and compilation of a servlet.

So your JSP page source invokes the tag, which maps to a tag file in a tag directory. If this is its first use, the container generates a Java source code file that implements SimpleTag (usually by extending SimpleTagSupport—JSP container writers like to keep their work to a minimum too!). This source is kept and compiled in a location known to the server—Figure 9-3 illustrates the current situation for Tomcat. Thereafter, the server keeps an internal mapping between invocations of the tag file and the actual compiled class.

Beyond that, everything works just as if this were a bona fide simple tag declared in a tag handler, and with a handcrafted tag handler. Life cycle methods and rules are exactly the same.

When a tag file sits inside a JAR file, there is one difference in the packaging: The tag file must have a declaration inside a tag library descriptor. Here is a very short TLD, containing a single tag file declaration:

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <short-name>webcert</short-name>
  <tag-file>
      <name>mytag</name>
      <path>/META-INF/tags/mytag.tag</path>
  </tag-file>
</taglib>
```

You can mix tag file declarations with regular tag declarations and EL functions, all in the same TLD. Crucially, though, the `<name>` component must be unique across all the different types (you can't have a tag file called *mytag* existing in the same TLD as a regular tag called *mytag*).

For tag files kept directly in /WEB-INF/tags (or a subdirectory), there is no need for a tag library descriptor entry, although it is permitted: You could do this if (for some reason) you wanted a tag name that is different from the tag file name.

## Tag File Source

Let's return to the Unicode characters example. What has changed from the last section, where this was a simple tag developed in Java source? The JSP document using the tag—unicodeDisplay.jspx—has hardly changed at all. However, there is a subtle difference in the namespace declaration. There's no need to have a namespace referencing a tag library. Instead, you must reference a tag *directory*—a directory that acts as a repository for tag files and must be declared in using JSPs. Here is the original declaration in unicodeDisplay.jspx:

```
<html xmlns:mytags="http://www.osborne.com/taglibs/mytags" ... >
```

Here it is again, the change highlighted:

```
<html xmlns:mytags="urn:jsptagdir:/WEB-INF/tags/" >
```

In the first case, the URL String "http://www.osborne.com/taglibs/mytags" doesn't point to anything—it just has to match the corresponding `<taglib-uri>` setting in the web deployment descriptor. In the second case, the URN (of type JSP tag directory) designates a real location within the web application: the directory /WEB-INF/tags.

**exam Watch**

***The truth is that you will still probably encounter more traditional JSPs than JSP documents as questions in the exam. So you need to know the traditional variant of the tag directive. This is the <% taglib %> directive—but with a change to one of its attributes. It retains prefix, but tagdir is substituted for uri. So the full declaration equivalent to xmlns:mytags="urn:jsptagdir:/WEB-INF/tags/" in a JSP document is <% taglib prefix="mytags" tagdir="/WEB-INF/tags" %>. You will often find questions that test whether tagdir and uri can coexist in the same <% taglib %> directive. They can't. prefix is always present, and you can have either tagdir or uri—not both.***

Within the JSP document, the use of the tag hasn't changed very much, except that it no longer has a body with EL variables. All elements of the HTML table declaration have disappeared as well. Here's how it looks:

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:mytags="urn:jsptagdir:/WEB-INF/tags/">
  <jsp:output omit-xml-declaration="true" />
  <jsp:directive.page contentType="text/html" />
  <head><title>Unicode Converter—from Tag File</title></head>
  <body>
    <h3>Numbers with Unicode Character Equivalent—from Tag
File</h3>
    <mytags:unicodeConverter begin="1040" end="1116" />
  </body>
</html>
```

All the removed elements for presentation and data have migrated to the tag file, whose source we will see in a moment.

But before we consider that, how does the JSP find the tag? We've already said that no tag library descriptor is involved. So how is the name of the tag (`unicode Converted`) derived? The mechanism is very simple. The tag directory (/WEB-INF/tags) referenced in the XML name space declaration contains the tag file. The tag file itself has the name `unicodeConverter.tag`. So the name before the file extension corresponds to the name as used in the JSP document. The extension must always be .tag.

So now we know how the JSP document finds it, let's look at the complete tag file—shown below, with line numbers:

```
01 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
02 <%@ taglib prefix="mytags" uri="http://www.osborne.com/taglibs/mytags" %>
03 <%@ attribute name="begin" %>
04 <%@ attribute name="end" %>
05 <table border="1">
06 <c:forEach begin="${begin}" end="${end}" step="1" varStatus="counter">
07   <tr>
08     <td>${counter.index}</td>
09     <td>${mytags:unicodeConverter(counter.index)}</td>
10   </tr>
11 </c:forEach>
12 </table>
```

Because there is no TLD entry to refer to, the tag file has to assume its own responsibility for providing some of the information. This explains lines 03 and 04 of the file, which declare attributes using the `<%@ attribute %>` directive. This may sound incestuous, but the `<%@ attribute %>` directive has attributes of its own. The only mandatory attribute is *name*, as used in the example here. This is entirely equivalent to a tag library descriptor file containing these elements:

```
<attribute>
  <name>begin</name>
</attribute>
```

`<% attribute %>` has other attributes, and, like *name*, all of them (*required*, *rtexprvalue*, *description*, *type*, and *fragment*) are equivalent to the TLD subelements of `<attribute>` we have already met.

As well as playing the declarative role formerly taken by the TLD, the tag file needs to find its own way of substituting the Java logic otherwise placed in the `doTag()` method of a genuine simple tag handler class. Actual Java logic is

disallowed: A tag file chokes as soon as you try to introduce the smallest of Java language scriptlets. Instead, you use custom actions, and you are especially likely to choose actions from the JSTL. Hence, line 01 declares a reference to the JSTL core tag library.

Line 05 onward is a mixture of template text (HTML table elements), core library syntax, and EL. Line 06 introduces a `<c:forEach>` loop. The *begin* and *end* attributes in `<c:forEach>` have values supplied from the *begin* and *end* attributes defined in the tag—and represented in EL. The *step* attribute tells us we will be incrementing by 1 each time, and the *varStatus* attribute creates an EL variable called *counter* that we use in the table. This occurs at line 08: The *index* property of *counter* is displayed, which gives the current loop value. Remember that we're starting this loop at a number set in the JSP document: 1040. So the numbers we want to display are 1040, 1041, 1042, . . . (not 1, 2, 3, . . .).

What if there is logic that is beyond the bounds of what JSTL and EL can accomplish? Well, you can always access "real Java" by means of a (real) custom tag (not a tag file) used within the tag file. Or, as this example shows, through an EL function. At line 09, we want to display the Unicode character corresponding to the number displayed. Converting an integer to a character value is beyond simple EL; we need some Java to get the job done. Consequently, line 09 invokes a preprepared EL function called `unicodeConverter`, which accepts *counter.index* and returns an equivalent java.lang.Character wrapper object for display. (The code to achieve this is in CharConvFunction.java—not reproduced here, for it's not central to the explanation of tag files but present in the example code.) This, incidentally, explains why there is a tag library declaration at line 02: This references the tag library descriptor, which declares the EL function `unicodeConverter`.

## Tag File Directives

There are two directives in the tag file example: `<%@ taglib %>`, which you have met many times before in JSPs, and `<%@ attribute %>`, which you are meeting for the first time and which only works in tag files. Let's consider—first of all—those attributes found in JSPs and how they are used in tag files:

- `<%@ page %>`—*only* suitable for JSP pages. Tag files have their own equivalent—`<%@ tag %>`.
- `<%@ taglib %>`, `<%@ include %>`—both work identically in tag files and in JSP pages.

There are three directives that are only for tag files:

- `<%@ tag %>`, which shares several attributes in common with `<%@ page %>` for JSPs (*language*, *import*, *pageEncoding*, *isELIgnored*)
- `<%@ attribute %>`, like the `<attribute>` subelement of `<tag>` in a TLD
- `<%@ variable %>`, like the `<variable>` subelement of `<tag>` in a TLD

`<%@ tag %>` sounds as though it should be crucial, but all its attributes are optional. For a full list (and for absolutely thorough exam preparation), check Table JSP.8-2 in the JSP 2.0 specification. Only one attribute is crucial to understand in terms of the exam objective that headed this chapter section, and that is *body-content*. The default value for the *body-content* attribute is `scriptless`. The only other allowed values are `empty` and `tagdependent`. Their meaning is identical to the equivalent values in the `<body-content>` element of a custom or simple tag declaration in a TLD. Because a tag file becomes a simple tag handler class (after generation and compilation), `JSP` is a strictly disallowed value.


***So far, tag files have led us back to a more traditional JSP syntax. Although Java language is absent, the `<% ... %>` syntax predominates for all kinds of necessary declaration. What if you want your tag files to be written in pure XML, as JSP documents? That's fine; you can do that. There are custom actions equivalents, such as `<jsp:directive.tag ...>`. See the JSP 2.0 specification, especially sections 8.7 and 10.1.***

## EXERCISE 9-3


### The Tag File Model

In this exercise you will rewrite the file browser from Exercise 9-2 as a slightly enhanced version of the original. You will also use a tag file to provide the core functionality instead of a tag handler class. The tag file makes use of a servlet, which houses some of the code that would otherwise be difficult or impossible to render as a combination of JSTL and EL within the tag file.

For this exercise, create a web application directory structure under a directory called ex0903, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch09/ex0903.war.

### Copy the HTML File from Exercise 9-2

1. Copy the HTML file fileBrowser.html from context directory ex0902 to the current context directory for this exercise: ex0903.
2. Change the action in the form to target JSP page fileBrowser.jsp (instead of JSP document fileBrowser.jspx).

### Create the JSP Page

3. Create a JSP page called fileBrowser.jsp directly in ex0903.
4. Include a `taglib` directive with a prefix of `mytags` and with the tag directory attribute set to "/WEB-INF/tags."
5. Most of the template text in the original JSP document from Exercise 9-2 is going to migrate to the tag file. Consequently, all you need to set up in the JSP page is (1) some template elements (`<html>`, `<body>`, etc.) to establish the page as proper HTML and (2) a call to the tag just as before:

```
<mytags:fileBrowser initDir="${param.initDir}" size="true" />
```

### Set Up the Servlet and the Helper Class

6. The tag file we have yet to write is going to use a servlet, which in turn uses a helper class. You aren't going to write them yourself—they can be copied from the solution code WAR file.
7. Unzip the WAR file.
8. Copy /ex0903/WEB-INF/classes/webcert/ch09/ex0903/FileBrowserServlet.class to the equivalent directory located in your own directory structure for this exercise.
9. Do the same for /ex0903/WEB-INF/webcert/ch09/ex0903/FileFacade.class.
10. Also copy the deployment descriptor /ex0903/WEB-INF/web.xml. This sets up the servlet with a URL mapping of /FileBrowser.
11. Look at the source for FileBrowserServlet (in /ex0903/WEB-INF/src/webcert/ch09/ex0903/FileBrowserServlet.java). Note how this obtains a value for the parameter *initDir*, representing the directory to start with. See how this is used to create an array of Files in that directory and how each File in turn is transferred to a collection (ArrayList) of FileFacade objects. Note how the collection is set up as a request attribute called *fileList*.

12. Now look at the source for FileFacade.java (same solution directory as the servlet source code). Note how this is a bean that "wrappers" a genuine java.io.File object, with the express purpose of exposing three properties: *size* (of file), *directory* (is the file in fact a directory?), and *nameAndPath* (the full name of the file). This will make it easier to write EL in the tag file in the next stage of the exercise.

### Write the Tag File

13. Create a file called fileBrowser.tag directly in directory /WEB-INF/tags (create the directory as necessary).
14. Include a `taglib` directive to reference the JSTL core tag library.
15. Define two attributes for the tag (in attribute directives) with the names *initDir* and *size*. *initDir* represents the directory that the tag file operates on. *size* indicates whether or not the "size" column should appear in the displayed table.
16. Now use the `<c:import>` core library action to invoke the URL /FileBrowser, which will cause the FileBrowserServlet to process the request. Within `<c:import>`, pass a parameter (using `<c:param>`) with a value of `${initDir}`—the directory the servlet should operate on.
17. Include a table with two columns, with the headings "Name" and "Size." Use `<c:if>` and the `${size}` attribute to include only the Size column if it is requested.
18. Use `<c:forEach>` to iterate over the request attribute *fileList* set up by the servlet. Pull each object in the collection (a FileFacade object) back into a variable called *file*.
19. Within the `<c:forEach>` loop, display the file's full name (with the *nameAndPath* property on file) and (if requested) the file's size.
20. Optionally, detect whether the file is a directory or not. When it is a directory, set the displayed name up as a hyperlink back to the JSP fileBrowser.jsp, forwarding the link directory name as the `initDir` parameter.

### Create, Deploy, and Test a WAR File

21. Zip up your context directory in a WAR file, and deploy this to your server.
22. Test with a URL such as

```
http://localhost:8080/ex0903/fileBrowser.html
```

23. Supply an initial directory name (e.g., C:\Documents and Settings All Users), and click the submit button.
24. The solution page is shown in the following illustration. If you have included hyperlinks to other directories, check to see that they work.

**Show Files in Requested Directory (using Tag File)**

**Directory: C:\Documents and Settings\David**

| Name | Size |
|---|---|
| C:\Documents and Settings\David\.WASRegistry | 59 |
| C:\Documents and Settings\David\Application Data | N/A |
| C:\Documents and Settings\David\Cookies | N/A |
| C:\Documents and Settings\David\Desktop | N/A |
| C:\Documents and Settings\David\Favorites | N/A |
| C:\Documents and Settings\David\IBM | N/A |
| C:\Documents and Settings\David\Local Settings | N/A |
| C:\Documents and Settings\David\My Documents | N/A |
| C:\Documents and Settings\David\NetHood | N/A |
| C:\Documents and Settings\David\NTUSER.DAT | 1835008 |
| C:\Documents and Settings\David\NTUSER.DAT.LOG | 1024 |

## CERTIFICATION OBJECTIVE

# Tag Hierarchies (Exam Objective 10.3)

*Given a scenario, write tag handler code to access the parent tag and an arbitrary tag ancestor.*

The good thing about this exam objective is that you have already worked on it. In the last section of Chapter 8, you learned about iteration tags, following a card dealing example. Within that example, you saw how an inner tag could make use

**exam Watch**

***For the exam, you need to know how hierarchies work for both classic and simple tags.***

of methods in an outer tag. This chapter will round out your knowledge on this topic. In the Chapter 8, you considered how you could access parent tags (and parents of parents) in the classic tag model. This section shows you how the simple tag model can also access members of a simple tag hierarchy, and how you can even mix simple and classic tags together in the same hierarchy.

## Tag Hierarchies

Instead of having a practical example (such as card dealing), we're going to strip hierarchies back to their bare essentials in this section. You'll meet both a classic tag and a simple tag that have only a single purpose: to display the name of their parent. You'll see that *name* is an attribute of the tags. Then you'll see the tags embedded in all combinations:

- Simple within simple
- Classic within classic
- Simple within classic
- Classic within simple

You'll see how this works in three out of the four cases, but how one case (in which a classic tag is embedded inside a simple tag) proves to be problematic. However, the JSP container provides a solution, which we will explore.

### The Big Picture

The entirety of this example is available in the CD in /sourcecode/ch09/examp0904.war. You may want to deploy and run the code first of all. All the source code is available in the WAR file, so not all of this is reproduced in the pages that follow. Once deployed, run the code using a URL such as

```
http://localhost:8080/examp0904/nestingTags.jsp
```

This should produce a page in your browser similar to that shown in Figure 9-4.

We'll explore each of the five cases illustrated one by one.

**FIGURE 9-4** **Experiment with nesting tags**

Nesting Tags Example Output

| | |
|---|---|
| Simple within Simple | My name is: outerSimple; my parent's name is: No parent<br>My name is: innerSimple; my parent's name is: outerSimple |
| Classic within Classic | My name is: outerClassic; my parent's name is: No parent<br>My name is: innerClassic; my parent's name is: outerClassic |
| Simple within Classic | My name is: outerClassic; my parent's name is: No parent<br>My name is: innerSimple; my parent's name is: outerClassic |
| Classic within Simple (Broken) | My name is: outerSimple; my parent's name is: No parent<br>My name is: innerClassic; my parent's name is: No parent |
| Classic within Simple (Fixed) | My name is: outerSimple; my parent's name is: No parent<br>My name is: innerClassic2; my parent's name is: outerSimple |

## Simple within Simple

The JSP code that produces the "simple within simple" output in Figure 9-4 is as follows:

```
<mytags:nestingSimple name="outerSimple">
  <mytags:nestingSimple name="innerSimple" />
</mytags:nestingSimple>
```

You can see that there is an outer occurrence of a tag called nestingSimple, with a name attribute of `outer`. There is an inner occurrence of the same tag—this time without a body—called `inner`. The outer occurrence has no parent, so it displays the following:

```
My name is: outerSimple; my parent's name is: No parent
```

The inner occurrence does have its own name ("innerSimple"), and we would expect its parent to be "outerSimple." Again, this performs as expected. The next line of output in the web page is

```
My name is: innerSimple; my parent's name is: outerSimple
```

So far, so good.

The crucial code to accomplish this output in the `doTag()` method of the tag handler (which extends SimpleTagSupport) goes as follows:

```
String bodyMessage;
JspTag parent = getParent();
if (parent instanceof NestingSimpleTag) {
  NestingSimpleTag nst = (NestingSimpleTag) parent;
  bodyMessage = nst.getName();
} else {
  bodyMessage = "No parent";
}
bodyMessage = "My name is: " + getName() + "; my parent's name is: " +
bodyMessage + "<br />";
```

Just as for classic tags (as implemented in TagSupport), so any SimpleTag (as implemented in SimpleTagSupport) has a `getParent()` method. Note that this returns not a Tag, but a JspTag—useless by itself, for it has no methods. However, we suspect that the parent might be an instance of NestingSimpleTag—if it is, we can cast the parent reference to a NestingSimpleTag and get hold of its name. The rest is finagling with the String output (the code to send the String to the associated page output Writer has been omitted).

### Classic within Classic

The code for a classic tag to retrieve its classic parent is hardly any different. Here's the highly predictable JSP code first of all:

```
<mytags:nestingClassic name="outerClassic">
  <mytags:nestingClassic name="innerClassic" />
</mytags:nestingClassic>
```

The relevant code in the `doStartTag()` method of the tag handler (which extends TagSupport) looks like this:

```
String bodyMessage;
Tag parent = getParent();
if (parent instanceof NestingClassicTag) {
  NestingClassicTag nct = (NestingClassicTag) parent;
  bodyMessage = nct.getName();
} else {
  bodyMessage = "No parent";
}
```

Since this is a Tag (not a SimpleTag), the `getParent()` method retrieves a Tag rather than a JspTag—a subtle difference, but one that proves important later. And this time, the code tests for an instance of a NestingClassicTag rather than a NestingSimpleTag—naturally enough. The output is just as expected for the outer and inner invocations:

```
My name is: outerClassic; my parent's name is: No parent
My name is: innerClassic; my parent's name is: outerClassic
```

### Simple within Classic

What happens, though, when we embed our nestingSimple tag in our nestingClassic tag, as in the following JSP code?

```
<mytags:nestingClassic name="outerClassic">
  <mytags:nestingSimple name="innerSimple" />
</mytags:nestingClassic>
```

The desired output is clear enough:

```
My name is: outerClassic; my parent's name is: No parent
My name is: innerSimple; my parent's name is: outerClassic
```

Our tag handler code won't quite work as it stood before. We need some additional code to make the right cast to a classic tag:

```
String bodyMessage;
JspTag parent = getParent();
if (parent instanceof NestingSimpleTag) {
  NestingSimpleTag nst = (NestingSimpleTag) parent;
  bodyMessage = nst.getName();
} else if (parent instanceof NestingClassicTag) {
  NestingClassicTag nct = (NestingClassicTag) parent;
  bodyMessage = nct.getName();
} else {
  bodyMessage = "No parent";
}
```

This allows the code to call the `getName()` method on the right sort of tag—nothing more than this. The important part is that the `getParent()` call doesn't change. Even though we are in simple tag handler code, `getParent()` still retrieves

the classic tag handler instance parent: as a JspTag. That's no problem because classic tag handler instances are all Tags—and in JSP 2.0, Tag inherits from JspTag. JspTag, as we have seen before, is the common ancestor for both classic and simple tags. So this works.

### Classic within Simple (Broken)

Only one combination to go—a classic tag trying to find a simple parent, as in this JSP code:

```
<mytags:nestingSimple name="outerSimple">
  <mytags:nestingClassic name="innerClassic" />
</mytags:nestingSimple>
```

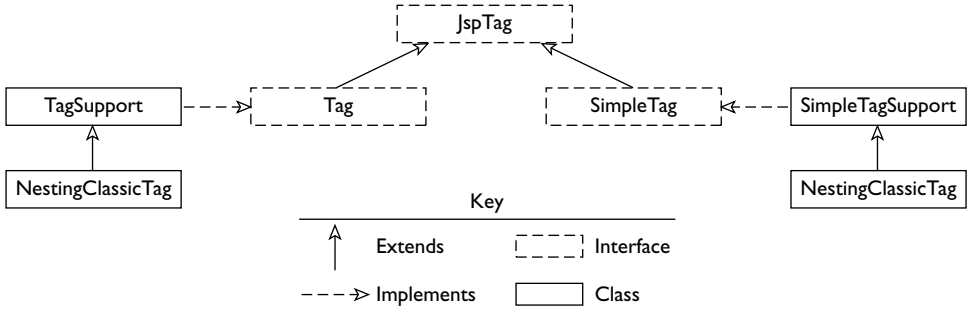
The naïve approach to making our nestingClassic tag find a nestingSimple parent is to write a mirror image of the adaptations we made to our simple tag handler code, which looks like this:

```
String bodyMessage;
Tag parent = getParent();
if (parent instanceof NestingClassicTag) {
  NestingClassicTag nct = (NestingClassicTag) parent;
  bodyMessage = nct.getName();
} else if (parent instanceof NestingSimpleTag) {
  NestingSimpleTag nst = (NestingSimpleTag) parent;
  bodyMessage = nst.getName();
} else {
  bodyMessage = "No parent";
}
```

However, this doesn't work. It doesn't break at run time; instead, we're told that the innerClassic tag has no parent, which is simply untrue:

```
My name is: outerSimple; my parent's name is: No parent
My name is: innerClassic; my parent's name is: No parent
```

Why doesn't the test `(parent instanceof NestingSimpleTag)` prove true? Consider that we are in an implementation of Tag and so the `getParent()` method has a return type of Tag, not JspTag. NestingSimpleTag can *never* be an instance of Tag—it's in the wrong hierarchy, as the following illustration shows.




So what does `getParent()` return? Maybe **null** because classic tags perhaps can't detect simple parents? On the contrary—the JSP container has an ace up its sleeve. One step ahead of our tag handler, it detects the mixed hierarchy and shrouds the NestedSimpleTag object in a TagAdapter instance. The process is as follows:

- The JSP container calls `doTag()` on `mytags:nestingSimple` (the outer—simple—tag).
- The `doTag()` method chooses to process the body of the simple tag (by calling the `JspFragment.invoke()` method).
- The body of the simple tag contains `mytags:nestingClassic` (the classic tag)—so the JSP container exercises the classic model life cycle on this tag.
- The classic model life cycle includes a call to `setParent()` for the classic tag. The JSP container knows the parent is a simple tag. However, `setParent()` will only accept a Tag as a parameter—and simple tag is a JspTag, not a Tag.
- The JSP container gets out of the dilemma by creating an instance of TagAdapter, which is of type Tag. The JSP Container passes the simple tag to the TagAdapter's one-argument (and only) constructor.
- Now the JSP container calls `setParent()` on the classic tag, passing in the TagAdapter instance.
- When—later—the code in the classic tag calls `getParent()`, the method returns this TagAdapter instance.
- In a moment, we'll see what methods the classic tag can use from TagAdapter to get at the real parent: the simple tag.

The TagAdapter is a class that you don't need to instantiate in your own code—it's there for the JSP container. The TagAdapter is a "wrapper" around the simple

tag. You encountered this kind of behavior before in the SCJP exam, in the java.io library. Recall that if you have a byte stream, but need to deal with characters, you can pass your byte stream to the constructor of an InputStreamReader. Thereafter, you deal with the character-oriented InputStreamReader—though under the covers, this is manipulating the original stream of bytes. So with the TagAdapter—it allows the JSP container code to deal with simple tags as classic tags in situations where there is no alternative.

### Classic within Simple (Fixed)

Armed with this information, we can set about making NestedClassicTag work properly (which is done in the nestingClassic2 tag, with tag handler class NestedClassicTag2). TagAdapter has two useful methods:

- `getAdaptee()` returns the wrappered-up JspTag (i.e., your simple tag instance).
- `getParent()` returns the adaptee's parent (by running `getAdaptee().getParent()`). If *that* parent isn't an instance of a Tag, then it in turn is wrappered up as a TagAdapter, then returned from this method.

**exam Watch**

***The TagAdapter class also contains a number of deliberately useless methods. Because TagAdapter implements the Tag interface, it includes all the classic life cycle methods: `setPageContext()`, `setParent()`, `doStartTag()`, `doEndTag()`, and `release()`. These mustn't be called directly—by you or the JSP container. To do so results in an UnsupportedOperationException in all cases.***

So back in our classic tag handler: `getParent()` returns—in these circumstances—an instance of TagAdapter. Given that, we can use the `getAdaptee()` method to fix our code, as follows:

```
String bodyMessage;
Tag parent = getParent();
if (parent instanceof NestingClassicTag2) {
  NestingClassicTag2 nct = (NestingClassicTag2) parent;
  bodyMessage = nct.getName();
} else if (parent instanceof TagAdapter) {
  JspTag simpleParent = ((TagAdapter) parent).getAdaptee();
  if (simpleParent instanceof NestingSimpleTag) {
```

```
      bodyMessage = ((NestingSimpleTag) simpleParent).getName();
    } else {
      bodyMessage = "No parent";
    }
  } else {
    bodyMessage = "No parent";
  }
```

If the parent is a classic tag, nothing has changed—we test for an instance of the classic tag class (true, there has been a name change to NestingClassicTag2—to differentiate it from the broken original). Next, the code—suspecting a simple tag as parent—tests for `getParent()` returning an instance of TagAdapter. If it has, the simple tag is extracted from the TagAdapter by using the `getAdaptee()` method. The output from this is a JspTag, and finally this instance can be tested to see if it is the tag we're hoping for, the NestingSimpleTag. When this code is run, the output is fixed, as shown:

```
My name is: outerSimple; my parent's name is: No parent
My name is: innerClassic2; my parent's name is: outerSimple
```

**exam Watch**

***Just as TagSupport has a `findAncestorWithClass()` method, so SimpleTagSupport has a method of the same name. It has some important differences, though. For one thing, it returns a JspTag rather than a Tag: It's intended to hunt down simple classes. For another (and in keeping with this), any TagAdapter found in the hierarchy is not taken at face value—instead, the simple tag it contains (obtained with `getAdaptee()`) is compared with the class passed in as the second parameter to the `findAncestorWithClass()` method. Check out the API documentation for chapter and verse.***

## EXERCISE 9-4


### Tag Hierarchies

In this exercise you will build a version of the card dealing example that you first met in Chapter 8. This will give you a good feel for classic tag hierarchies (to complete the picture, in the lab at the end of the chapter, you'll convert this example to use a mixed hierarchy with classic and simple tags).

For this exercise, create a web application directory structure under a directory called ex0904, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch09/ex0904.war.

### Copy the CardDealingTag Code from the Solution

1. The main custom action (code in the CardDealingTag class) contains mostly the "business" method for the card dealing solution. Rather than attempt to recreate this yourself, copy it from the solution code WAR file to a directory called /ex0904/WEB-INF/classes/webcert/ch09/ex0904.
2. Copy the source file CardDealingTag.java to /ex0904/WEB-INF/src/webcert/ch09/ex0904.
3. Open the source file. Note the following methods:

   `doStartTag()`—this shuffles the card every time it's invoked, and returns EVAL_BODY_INCLUDE to ensure that the body is processed.

   `doAfterBody()`—this continues to loop until all cards are "dealt" (as indicated by the currentCard value).

   `dealCard()`—this is the method that must be called to "deal" cards from the pack and will be called by child tag handlers you will write.

### Write Child Tag Handler Classes

4. Create two Java source files in your context directory, under a suitable package directory in /WEB-INF/classes, called CardTag.java and CardTag2.java.
5. Have CardTag extend TagSupport. Override the `doEndTag()` method. This should use `getParent()` to get hold of the parent tag, and cast this to a CardDealingTag reference. Call the `dealCard()` method on the `CardDealingTag()` instance so that one card is dealt. `dealCard()` returns a String holding the card name — output this to the Writer for the page context object.
6. Have CardTag2 extend TagSupport as well. Copy the code from `CardTag.doEndTag()` into CardTag2. Change the code so that instead of using `getParent()`, it uses `TagSupport.findAncestorWithClass()` to return a reference to the parent CardDealerTag.

### Write the Tag Library Descriptor File

7. Create a TLD file called mytags.tld in /WEB-INF/tags (create the directory as necessary).
8. Declare three tags called cardDealer, card, and card2. These reference (respectively) the CardDealerTag, CardTag, and CardTag2 classes. The body content of cardDealer should be set to JSP, and for card and card2 should be set to empty.

### Reference the TLD in the Deployment Descriptor

9. Place a `<jsp-config>` element in web.xml, which defines a mapping from taglib URI http://www.osborne.com/taglibs/mytags to a location of /WEB-INF/tags/mytags.tld.

### Define a JSP Document to Use the Tags

10. Create a JSP document called cardGame.jspx directly in the ex0904 context directory.
11. Include the following "boilerplate" namespace declarations for your own tag library, JSP standard actions, and the JSTL core library:

```
<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">
```

12. Include the following standard actions to ensure HTML rather than XML output to your browser:

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

13. Create an HTML table with four columns. Include table headings labeled Player 1 through to Player 4.
14. Surround the main table row with opening and closing cardDealer tags.
15. In each of the four cells for the main table row, include a self-closing card tag.

### Create, Deploy, and Test a WAR File

16. Zip up your context directory in a WAR file, and deploy this to your server.
17. Test with a URL such as

```
http://localhost:8080/ex0904/cardGame.jspx
```

18. The solution output is shown in the following illustration. Each refresh of the browser should result in a newly shuffled deal.

# Bridge Hand

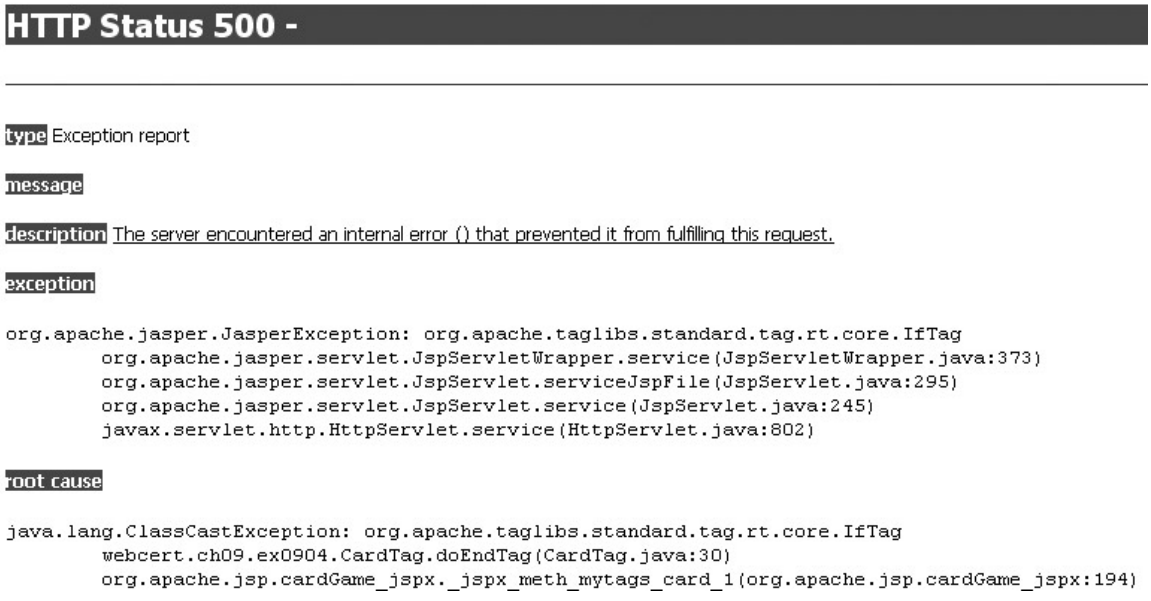
| Player 1 | Player 2 | Player 3 | Player 4 |
|---|---|---|---|
| Three of spades | Eight of spades | Nine of hearts | Ten of hearts |
| Six of clubs | Four of spades | Queen of hearts | Four of clubs |
| Four of hearts | Queen of spades | Ten of clubs | Nine of spades |
| Two of spades | Queen of diamonds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

19. Now amend cardGame.jspx (you can, if you wish, amend the deployed copy under Tomcat; otherwise, change the document in your context directory, remake the WAR, and redeploy it).
20. Change one of the card tags to be surrounded by a `<c:if>` action like this:

```
<c:if test="${true}"><mytags:card2 /></c:if>
```

The test will always be true, so the `<c:if>` is spurious. However, it separates the <mytags:card> action from its proper parent (`<mytags:cardDealer>`) in the hierarchy.

21. Retest the document. You should get a ClassCastException, like this:



**HTTP Status 500 -**

---

**type** Exception report

**message**

**description** The server encountered an internal error () that prevented it from fulfilling this request.

**exception**

```
org.apache.jasper.JasperException: org.apache.taglibs.standard.tag.rt.core.IfTag
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:373)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:295)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:245)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
```

**root cause**

```
java.lang.ClassCastException: org.apache.taglibs.standard.tag.rt.core.IfTag
        webcert.ch09.ex0904.CardTag.doEndTag(CardTag.java:30)
        org.apache.jsp.cardGame_jspx._jspx_meth_mytags_card_1(org.apache.jsp.cardGame_jspx:194)
```

22. Now change the tag enclosed in the `<c:if>` to `<mytags:card2>`. When you retest this time, the page should work. This is because `<mytags:card2>` uses the `findAncestorByClass()` method and is not sensitive to intervening actions in the hierarchy.

## CERTIFICATION SUMMARY

In this chapter you started by learning about accessing implicit variables in tag handler code. You saw that this involved gaining a thorough knowledge of methods on the PageContext class that provide equivalent instances to the JSP's implicit variables, such as `getRequest()` for *request*.

You also revised the topic of attributes and PageContext: Any attribute in any scope can be manipulated through the PageContext object.

You went on from there to explore the simple tag model. You saw that simple tags implement the SimpleTag interface or, more usually, extend the SimpleTagSupport class, which provides useful default implementations of methods. You learned all about the simple tag life cycle and maybe agreed that although it is not that simple, it removes a lot of the complexity that goes with classic tags. You learned that a new simple tag gets created for every invocation (no pooling as there is—potentially—for classic tags), so instance variables are safe to use. You saw that `setJspContext()` is the first life cycle method to be called, and this provides the tag handler with a JspContext object—very like a PageContext (which is JspContext's subclass). You saw that a call to `setParent()` followed—passing in a JspTag reference (as opposed to a Tag for the equivalent classic tag method). You saw that attributes are set after this (much as classic tags), followed by a call to setJspBody, providing something called a JspFragment to your simple tag. You then learned that `doTag()` is called and that this can write directly to page output and cause the body to be executed using the `JspFragment.invoke()` method. You finally learned that even though `doTag()` has no return value, you can skip the rest of the page by throwing a SkipPageException.

Next you saw that simple tags can be written without any Java source—in the form of tag files. You saw that tag files are written very much like JSP pages (though without any Java syntax—only template text, actions, and EL allowed). You saw that tag files must have a .tag extension and are normally placed in /WEB-INF/tags or a subdirectory thereof. You learned that the JSP container generates simple tag source code from a tag file and compiles this into a class that then functions just as a handcrafted simple tag.

You learned that tag file source shares some directives in common with JSPs (`<%@ taglib %>`, `<%@ include %>`), has one that it doesn't share (`<%@ page %>`), and has three directives all of its own (`<%@ tag %>`, `<%@ attribute %>`, `<%@ variable %>`). You learned that the attribute and variable directives have attributes very similar to subelements of the `<attribute>` and `<variable>` elements in a tag library descriptor. You also saw that the tag directive has several attributes in common with the JSP page directive, but also has some attributes (such as *body-content*) that reflect subelements of the `<tag>` element in a TLD.

In the final section of the chapter, you revisited the subject of tag file hierarchies. You saw that both TagSupport and SimpleTagSupport have identically named methods for retrieving instances of tag handlers: `getParent()`, for retrieving an immediate parent in the hierarchy, and `findAncestorWithClass()`, for retrieving

a tag handler somewhere farther up the hierarchy of the class identified in the parameter.

However, you learned that there are subtle differences—Tags and SimpleTags do not belong to the same class hierarchy (although they do both have JspTag as their ultimate parent). Tag hierarchy methods return Tags. SimpleTags return JspTags, and a SimpleTag can never be a Tag. You saw that to stop a fatal collision where a classic tag has a simple parent, simple tags may be cloaked in a TagAdapter, which presents the SimpleTag as a classic Tag.

# TWO-MINUTE DRILL

## Tags and Implicit Variables

- ❏ In tag handler code, implicit variables are available through methods on the PageContext object.
- ❏ The PageContext object is handed to the tag handler code in the `setPageContext()` method.
- ❏ For ServletRequest *request*, use `PageContext.getRequest()`.
- ❏ For ServletResponse *response*, use `PageContext.getResponse()`.
- ❏ For Writer *out*, use `JspWriter.getOut()`. (JspWriter is the superclass for PageContext.)
- ❏ For HttpSession *session*, use `PageContext.getSession()`.
- ❏ For ServletConfig *config*, use `PageContext.getServletConfig()`.
- ❏ For ServletContext *application*, use `PageContext.getServletContext()`.
- ❏ For Object *page*, use `PageContext.getPage()` (it's usually a Servlet).
- ❏ For PageContext *pageContext*—that is the PageContext object passed to the tag handler through the setPageContext() method.
- ❏ For Exception exception, use `PageContext.getException()`.
- ❏ `PageContext.getException()` will return **null**, unless the enclosing JSP page is marked as an error page (`isErrorPage="true"`) and the JSP container has redirected to this page as the result of an error.
- ❏ PageContext can be used to access attributes in all scopes.

## The "Simple" Custom Tag Event Model

- ❏ Simple tags implement the SimpleTag interface or extend the SimpleTagSupport class.
- ❏ An instance of a simple tag is created for each run-time use.
- ❏ Simple tag life cycle methods are called in sequence by the JSP container. None of the methods produce return codes.
- ❏ Following construction, `setJspContext` is called to provide a context object (in much the same way that `setPageContext` is called for a classic tag).
- ❏ Next: `setParent()`—passing a reference to a JspTag.

- ❏ Next: `setXXX()` methods for attributes.
- ❏ Next: `setJspBody()`—passing in a JspFragment representing the JSP code within the body of the tag (called only if there is a body).
- ❏ Next: `doTag()`. All looping, body processing, and other page output occurs within this method.
- ❏ Normal exit from `doTag()` allows the JSP container to process the remainder of the page.
- ❏ A SkipPageException thrown from `doTag()` makes the JSP container skip the rest of the page.
- ❏ JspFragment is not like BodyContent. BodyContent represents evaluated content, buffered. JspFragment is content that hasn't been evaluated yet (so no need for a buffer).

## The Tag File Model

- ❏ Tag files are simple tags whose source is in JSP-syntax form.
- ❏ Tag files must have a .tag extension (or .tagx if they are XML documents).
- ❏ Tag files are kept in /WEB-INF/tags or a subdirectory of /WEB-INF/tags.
- ❏ A JSP page using a tag file has access to it through a tag library directive: The tagdir attribute is used instead of uri—for example,

```
<%@ taglib prefix="mytags" tagdir="/WEB-INF/tags" %>
```

- ❏ Tag files may also be kept in a JAR file in /WEB-INF/lib.
- ❏ Tag files within a JAR must reside in the /META-INF/tags directory (or a subdirectory of this).
- ❏ Tag files within a JAR are only accessible if declared in a `<tag-file>` element in a tag library descriptor (TLD).
- ❏ The `<tag-file>` element in the TLD has two subelements: `<name>` and `<path>`.
- ❏ `<name>` must contain a name unique across all tag files, tags, and EL functions contained in the TLD.
- ❏ `<path>` must begin with /META-INF/tags.
- ❏ Tag files have limitations on body content: `empty`, `tagdependent`, or `scriptless`.

- The fourth value for body content—JSP—is disallowed in tag files (as for all simple tags).
- `scriptless` is the default.
- To set another value for body-content, use the `tag` directive:

```
<% tag body-content="tagdependent" %>
```

- Tag files have three directives that may only occur in tag files: `tag`, `attribute`, and `variable`.
- The `tag` directive is like the page directive in the JSP (and has four attributes in common with it—`language`, `import`, `pageEncoding`, and `isELIgnored`).
- The `attribute` directive is like the `<attribute>` element within a TLD.
- The `variable` directive is like the `<variable>` element within a TLD.
- At run time, Java source is generated for a SimpleTag from the tag file source and compiled into a class file.

## Tag Hierarchies

- Tag has a `getParent()` method, implemented in TagSupport.
- `getParent()` retrieves the immediate parent of a tag—if there is one.
- `getParent()` returns a Tag reference, which can be cast to something more appropriate if specific methods are needed.
- SimpleTag also has a `getParent()` method, implemented in SimpleTagSupport.
- `getParent()` in SimpleTag also returns its immediate parent, if there is one.
- However, `SimpleTag.getParent()` returns a JspTag, so it can accommodate both simple or classic tag parents.
- As `Tag.getParent()` can accommodate only Tags, any simple tag higher in the hierarchy is "cloaked" by the JSP container in an instance of TagAdapter.
- `TagAdapter.getAdaptee()` returns the simple tag wrappered by the TagAdapter instance (as a JspTag reference—which you can cast as needed).
- TagSupport has a **static** method called `findAncestorWithClass()`.
- To this method, you pass in the Tag instance to start from (usually *this*) and the Class whose type should match a tag handler instance in the hierarchy.

- ❏ `TagSupport.findAncestorByClass()` returns a Tag reference — this could be a TagAdapter.
- ❏ SimpleTagSupport also has a static `findAncestorByClass()` method, which works in identical fashion, except that it receives a JspTag to start from (not a Tag), and returns a JspTag reference.
- ❏ When using `SimpleTagSupport.findAncestorByClass()` methods, where TagAdapters are encountered, the class comparison is to `TagAdapter.getAdaptee()` (not directly to the class of the TagAdapter instance).

# 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. Choose all the correct answers for each question.

## Tags and Implicit Variables

1. Given the following JSP and tag handler code, what is the result of accessing the JSP? (Choose one.)

```
JSP Page Source
<%@ taglib prefix="mytags" uri="http://www.osborne.com/taglibs/mytags" %>
<html><head><title>Questions</title></head>
<body><p><% session.setAttribute("first", "first"); %>
<mytags:question01 />
${second}
</p></body></html>
Tag Handler Code for <mytags:question01 />
(imports missing, but assume they are correct)
public class Question01 extends TagSupport {
  public int doStartTag() throws JspException {
    Writer out = pageContext.getOut();
    try {
      out.write("" + pageContext.getAttribute("first"));
    } catch (IOException e) {
      e.printStackTrace();
    }
    pageContext.setAttribute("second", "second", PageContext.SESSION_SCOPE);
    return super.doStartTag();
  }
}
```

   A. first first
   B. second second
   C. first second
   D. second first
   E. first null
   F. null second
   G. null null

2. Which of the following snippets of code, if inserted into the `doStartTag()` method of a tag handler class extending TagSupport, would compile? (Choose three.)

   A.
   ```
   pageContext.getSession().getId();
   ```

   B.
   ```
   pageContext.getRequest().getAttributeName();
   ```

   C.
   ```
   pageContext.getHttpResponse().getBufferSize();
   ```

   D.
   ```
   pageContext.getPage().getServletName();
   ```

   E.
   ```
   pageContext.getException().getStackTrace();
   ```

   F.
   ```
   pageContext.getExpressionEvaluator();
   ```

3. Identify true statements about the *exception* implicit object in tag handler code. (Choose two.)

   A. It can be obtained through the page context's `getError()` method.
   B. The JSP page housing the tag must have page directive isErrorPage set to true for exception to be non-null in the tag handler code.
   C. The JSP page housing the tag must have page directive errorPage set to true for exception to be non-null in the tag handler code.
   D. The page context's `getError()` method return type is java.lang.Throwable.
   E. The page context's `getException()` method return type is java.lang.Throwable.
   F. The exception implicit object has a `getLocalizedMessage()` method.

4. The following is an extract from a tag handler class that implements the Tag interface. Given the code in the `doStartTag()` method, what else is likely to be true for the tag handler to compile and run successfully? (Choose three.)

```
// ...all necessary imports supplied...
public class Question04 implements Tag {
  public int doStartTag() throws JspException {
    HttpServletRequest request = pageContext.getRequest();
    request.setAttribute("myattr", "myvalue");
    return Tag.SKIP_BODY;
  }
  // ...Other methods and instance variables defined...
}
```

A. pageContext should be defined as an instance variable of type PageContext.
B. The method `pageContext.getRequest()` should be replaced with `pageContext.getServletRequest()`.
C. The `setPageContext()` method must initialize a value for pageContext.
D. A cast needs to be inserted in the `doStartTag()` code.
E. The `release()` method should set pageContext to **null**.
F. A scope parameter should be provided to the `request.setAttribute()` method call.

5. Which of the following methods are available in the java.servlet.jspPageContext class? (Choose three.)

A. getPageScope
B. getServletConfig
C. include
D. getErrorData
E. getError
F. getApplication

## The "Simple" Custom Tag Event Model

6. For a tag implementing the SimpleTag interface, which of the following method sequences might be called by the JSP container? (Choose two.)

A. setParent, setPageContext, setJspBody
B. setParent, setJspContext, doInitBody
C. setJspContext, setAnAttribute, doTag
D. setPageContext, setParent, doTag

E. setJspBody, doAfterBody, doTag
F. setJspContext, setParent, doTag

7. Which of the following techniques causes the JSP container to skip the rest of the page after processing a custom action implementing the SimpleTag interface? (Choose one.)
   A. Returning Tag.SKIP_PAGE from the `doEndTag()` method
   B. Returning −1 from the `doPage()` method
   C. Returning −1 from the `doTag()` method
   D. Returning Tag.SKIP_PAGE from the `doTag()` method
   E. Throwing a SkipPageException within the `doTag()` method
   F. Throwing a JspException from the `doPage()` method

8. Consider the following JSP page code and SimpleTag code. What is the output from tag `<mytags:question08>` to the requesting JSP page? You can assume that all necessary deployment descriptor and tag library descriptor elements are set up correctly. (Choose one.)

**JSP Page Code:**

```
<%@ taglib prefix="mytags" uri="http://www.osborne.com/taglibs/mytags"
%>
<html><head><title>Question08</title></head>
<body><p><mytags:question08>a</mytags:question08></body>
</html>
```

**SimpleTag Tag Handler Code for <mytags:question08>:**

```
import java.io.IOException;
import java.io.StringWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.JspFragment;
import javax.servlet.jsp.tagext.SimpleTagSupport;
public class Question08 extends SimpleTagSupport {
  public void doTag() throws JspException, IOException {
    JspFragment fragment = getJspBody();
    StringWriter sw = new StringWriter();
    for (int i = 0; i < 3; i++) {
      fragment.invoke(sw);
      String s = "b" + sw;
      sw.write(s);
      fragment.invoke(null);
    }
  }
}
```

A. aaa
B. bababa
C. babaabaaa
D. baabaaabaaaa
E. No output

9. Identify true statements about tag declarations in tag library descriptors from the following list. (Choose three.)
   A. The element `<simpletag>` is used to differentiate actions following the simple tag model from the classic tag model.
   B. A tag whose `<body-content>` is declared as JSP must follow the classic tag model.
   C. Separate tag library descriptor files must be used to separate classic and simple tag declarations.
   D. A simple tag has a default `<body-content>` of scriptless.
   E. Simple tags are commonly declared with a `<body-content>` of scriptless.
   F. If a simple tag is declared with a `<body-content>` of "empty," the JSP container makes one less method call on the simple tag handler class.

10. Consider the following JSP page code and SimpleTag code. What is the output from tag `<mytags:question08>` to the requesting JSP page? You can assume that all necessary deployment descriptor and tag library descriptor elements are set up correctly. (Choose one.)

**JSP Page Code:**

```
<%@ taglib prefix="mytags" uri="http://www.osborne.com/taglibs/mytags"
%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><head><title>Question 10</title></head>
<c:set var="counter">1</c:set>
<body><p><mytags:question10>
<c:forEach begin="${counter}" end="3">${counter}</c:forEach>
</mytags:question10></p></body>
</html>
```

**SimpleTag Tag Handler Code for <mytags:question08>:**

```
package webcert.ch09.questions09;
import java.io.IOException;
import javax.servlet.jsp.JspContext;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;
public class Question10 extends SimpleTagSupport {
```

```
  public void doTag() throws JspException, IOException {
    JspContext context = getJspContext();
    int i = Integer.parseInt("" + context.getAttribute("counter"));
    for (; i < 4; i++) {
      context.setAttribute("counter", new Integer(i));
      getJspBody().invoke(null);
    }
  }
}
```

A. No output
B. 111 222 333
C. 1 2 3
D. 123 123 123
E. 111 22 3
F. 22 3

## The Tag File Model

11. From the list, identify correct techniques to make tag files available in a JSP page or JSP document. (Choose two.)

A.
```
<%@ taglib prefix="mytags" uri="/WEB-INF/tags/mytags.tld" %>
```

B.
```
<%@ taglib prefix="mytags" taglocation="/WEB-INF/tags" %>
```

C.
```
<html xmlns:mytags="http://www.tags.com/tags">
```

D.
```
<html xmlns:mytags="urn:jsptagdir:/WEB-INF/tags/">
```

E.
```
<%@ taglib prefix="mytags" tagdir="/WEB-INF/tags" %>
```

F.
```
<%@ taglib prefix="mytags" tagdir="/WEB-INF/tags/mytags.tld" %>
```

12. What is the result of accessing the following tag file? (Line numbers are for reference only and should not be considered part of the tag file source.) (Choose one.)

```
01 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
02 <c:set var="character">65</c:set>
03 <c:forEach begin="1" end="10" varStatus="loopCount" >
04 <% char c = (char)
Integer.parseInt(pageContext.getAttribute("character").toString());
05 pageContext.setAttribute("displayCharacter", new Character(c));
07 %>
08 ${displayCharacter}
09 <c:set var="character">${character + 1}</c:set>
10 </c:forEach>
```

A. Translation error at line 1

B. Translation error at line 4

C. Translation error at line 9

D. Run-time error at line 9

E. Output from tag file of A B C D E F G H I J

13. Which of the following are directives you might find in a tag file? (Choose three.)

A. page

B. tag

C. variable

D. import

E. attribute

F. jspcontext

G. scope

14. (drag-and-drop question) In the following illustration, match the numbered options on the right with the concealed lettered portions of the JSP page and tag file, such that the output from accessing the JSP page is as follows:

```
Body: 1 Tag File: 2 Body: 1 Tag File: 3 Body: 1 Tag File: 4
```

JSP Source

```
<%@ taglib prefix="mytags" tagdir="/WEB-INF/tags" %>
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core" %>
<html><head><title>Question 14</title></head>
<[ A ] var="counter" value="1" />
<body><mytags:question14 counter="${counter}">
  Body: [ B ]
</mytags:question14></body></html>
```

Tag File Source for /WEB-INF/tags/question14.tag

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core" %>
<%@ [ C ] name="counter" %>
<c:forEach begin="[D]"end="[E]"
varStatus="loopCounter">
  [ F ]
   Tag File: ${counter + [ G ]}
</c:forEach>
```

| | |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | tag |
| 6 | attribute |
| 7 | variable |
| 8 | c:set |
| 9 | loopCounter |
| 10 | loopCounter.count |
| 11 | loopCounter.index |
| 12 | <jsp:attribute /> |
| 13 | <jsp:doBody /> |
| 14 | <jsp:invoker /> |
| 15 | ${counter} |
| 16 | ${counter + 1} |
| 17 | ${counter + loopCounter} |

15. Which of the following is a valid location for a tag file? (Choose one.)
    - A. Directly in the context directory
    - B. Directly in the /WEB-INF directory
    - C. In any subdirectory of /WEB-INF
    - D. In /WEB-INF/tags or a subdirectory beneath /WEB-INF/tags
    - E. In a JAR file in /WEB-INF/lib, under the /META-INF/tags directory
    - F. C and D
    - G. D and E

## Tag Hierarchies

16. Consider the following hierarchy of actions. `<c:if>` is from the JSTL core library, and `<a:tagA>` and `<a:tagB>` are classic custom actions.

```
<a:tagA>
  <c:if test="${true}">
    <a:tagB />
  </c:if>
</a:tagA>
```

What options does tagB have for obtaining the enclosing instance of tagA? (Choose one.)

A. Use `TagSupport.findAncestorWithClass()`
B. Invoke `getParent().getParent()`.
C. Use `SimpleTagSupport.findAncestorWithClass()`
D. A, B, and C
E. A and B only
F. B and C only

17. Which of the following could not be returned by either of the `getParent()` methods in the JSP class libraries? (Choose two.)

A. An instance of a JSTL `core` library tag handler
B. An instance of a JSTL `xml` library tag handler
C. An instance of an HTML tag
D. An instance of an XML template tag in a JSP document
E. An instance of BodyTagSupport
F. An instance of SimpleTagSupport

18. What methods can you execute on the reference myAncestor at ??? in the code snippet below? (Choose two.)

```
JspTag myAncestor = SimpleTagSupport.findAncestorWithClass(MyTagClass);
myAncestor.???
```

A. `notifyAll()`
B. `setParent()`
C. `doStartTag()`

D. `doEndTag()`

E. `doAfterBody()`

F. `clone()`

G. `hashCode()`

19. (drag-and-drop question) Consider the `doStartTag()` method shown in the following illustration. The tag for this tag handler will always have a parent, which could obey the simple or classic model. Complete the code by matching numbered options from the right with the concealed letter options, such that the code will successfully identify and print out whether the parent is simple or classic (you may need to use some numbered options more than once).

```
public int doStartTag() {
  [ A ] enclosing = getParent();
  if (enclosing instanceof [   B   ]) {
    [     C     ] enclosingSimple =
      ([   D   ]) enclosing;
    JspTag simpleTag =
      enclosingSimple.[     E     ];
    System.out.println("Simple parent: " +
      simpleTag.getClass().getName());
  } else {
    System.out.println("Classic parent: " +
     [     F     ].getClass().getName());
  }
  return Tag.[       G       ];
}
```

| | |
|---|---|
| 1 | Tag |
| 2 | TagSupport |
| 3 | SimpleTag |
| 4 | SimpleTagSupport |
| 5 | TagAdapter |
| 6 | TagAdaptee |
| 7 | getAdapter() |
| 8 | getAdaptee() |
| 9 | enclosing |
| 10 | enclosingSimple |
| 11 | simpleTag |
| 12 | EVAL_BODY |
| 13 | EVAL_BUFFERED_BODY |
| 14 | EVAL_BODY_INCLUDE |

20. What strategies might a parent tag use to get hold of a child tag handler instance? (Choose two.)

A. Classic model: Child gets hold of parent and provides the parent with its own reference to the child. Parent uses the reference in `doAfterBody()` method.

B. Classic model: Child gets hold of parent and provides the parent with its own reference to the child. Parent uses the reference in `doStartTag()` method.

C. Simple model: Child gets hold of parent and provides the parent with its own reference to the child. Parent uses the reference after a call to `JspFragment.invoke()`.
D. Simple model: Child gets hold of parent and provides the parent with its own reference to the child. Parent uses the reference after `doTag()` method completes.
E. Parent tag uses `TagSupport.findDescendantWithClass()`.
F. Parent tag uses `this.getChild()`.

# LAB QUESTION

Before you finish shuffling cards altogether, attempt the following moderately difficult undertaking. Take the solution code from Exercise 9-4, and convert the CardDealingTag to a simple tag. Leave Card and Card2 as classic tag handlers. You'll need to make some adjustments, though, to cope with having a simple tag handler as a parent.

# SELF TEST ANSWERS

## Tags and Implicit Variables

1. ☑ **F** is the correct answer: The output is null second. In the JSP a session attribute called *first* is set to a value of "first." Although the tag handler code attempts to retrieve an attribute called "first," it uses the `pageContext.getAttribute()` method without supplying a scope parameter—hence, the code looks in page scope and retrieves a null reference that is output to the page within the code. Still in the tag handler code, another attribute called *second* is set up with a value of "second"—again in session scope. Back in the JSP, EL is used to output the attribute. Because EL uses `findAttribute()` under the covers, the JSP searches through all scopes until finding the `second` attribute.
   ☒ **A**, **B**, **C**, **D**, **E**, and **G** are incorrect according to the reasoning in the correct answer.

2. ☑ **A**, **E**, and **F** are the correct answers. **A** returns the ID for a session. **E** prints the stack trace for an exception object. Of course, the exception object will be **null** unless this tag handler is called from a JSP error page. So you might get a run-time error (NullPointer Exception), but the question is about compilation. **F** is correct, for there is a method called `getExpressionEvaluator()` on the pageContext object (not that you are likely to need it very often).
   ☒ **B** is incorrect because there is a ServletRequest method called `getAttributeNames()`, but not one called `getAttributeName()`. **C** is incorrect; although ServletResponse has a getBufferSize method, to get the ServletResponse object from PageContext, you should use (just) `getResponse()` (there is no `getHttpResponse()` method). **D** is incorrect; although the page implicit variable is likely to be a servlet object (with a `getServletName()` method), the `getPage()` method returns java.lang.Object, so some casting is necessary before using servlet methods.

3. ☑ **B** and **F** are the correct answers. **B** is correct because only error pages (designated by `isErrorPage` set to true) have an exception implicit variable. This is then available to the tags used in the page as a non-null object available through the pageContext object. **F** is correct because all exception objects have a `getLocalizedMessage()` method, for they must implement java.lang.Throwable. This has little to do with web coding, but don't forget that the SCWCD exam assumes you can remember things from the SCJP exam!
   ☒ **A** is incorrect—the PageContext object doesn't have a `getError()` method. This also rules out answer **D**. **C** is incorrect because although there is a page directive errorPage, it's for pages to specify a *separate* page as their chosen error page to divert to when things go wrong. The actual page with the errorPage directive won't have an exception implicit object available.

**E** is incorrect: Although PageContext has a `getException()` method (the one you do use to get hold of the exception implicit object), it returns java.lang.Exception—not java.lang.Throwable.

4. ☑ **A**, **C**, and **D** are the correct answers. **A** is correct because—as this is a direct implementation of the Tag interface (not an extension of TagSupport)—you need to do the work to set up a pageContext variable. **C** is also correct because having set up the pageContext variable, `setPageContext()` is the appropriate method for supplying a value. **D** is correct because `pageContext.getRequest()` returns a ServletRequest, not an HttpServletRequest (but a cast to HttpServletRequest is likely to work in an HTTP environment).
   ☒ **B** is incorrect: The method name (`getRequest()`) is already correct, so should not be changed. **E** is incorrect—there's nothing wrong with setting pageContext to **null**, but it doesn't contribute to making the code compile and run correctly. **F** is incorrect: `request.setAttribute()` sets an attribute in request scope—you can't adjust the scope to something else.
5. ☑ **B**, **C**, and **D** are the correct answers. All these methods (`getServletConfig`, `include`, and `getErrorData`) are available on PageContext.
   ☒ **A** is incorrect: PageContext has a `getPage()` method and a static **int** value defined of PageContext.PAGE_SCOPE, but the method `getPageScope()` is made up and meaningless. **E** is incorrect: There is a `getException()` method that you can legitimately call when a tag is located within a JSP error page, but there is no `getError()`. **F** is incorrect: Although there is an implicit variable called *application* in JSPs, the correct way to access this through the PageContext object is through the getServletContext method.

## The "Simple" Custom Tag Event Model

6. ☑ **C** and **F** are the correct answers. **C** is correct because `setJspContext()` is always the first method to be called for a SimpleTag. Next is `setParent()`, missing in this sequence, and it will be bypassed if this tag doesn't have a custom action for a parent. Next comes attribute setting—so `setAnAttribute()` is a plausible method call. Next is `setJspBody()`, again missing in this sequence, but bypassed by the JSP container if the SimpleTag doesn't have a body. Finally, the JSP container does call `doTag()`. **F** is correct because `setJspContext()`, `setParent()`, and `doTag()` is a plausible sequence for a tag with no attributes and no body.
   ☒ **A** is incorrect because SimpleTags do not have a `setPageContext` method, and even if they did, it wouldn't come after `setParent()` (setting of contexts is the first thing to happen). **B** is incorrect because `setJspContext()` should come before `setParent()`, and SimpleTags do not have a `doInitBody()` method. **D** is incorrect because—although the order

is plausible—SimpleTags have a `setJspContext()` method, not `setPageContext()` (as for classic tags). Finally, **F** is incorrect because SimpleTags do not have a `doAfterBody()` method.

7. ☑ **E** is the correct answer. To make the JSP container skip the rest of the JSP page, throw a SkipPageException (subclass of JspException) from the `doTag()` method of a SimpleTag. You can do this anyway and should rethrow the exception if it originates from any enclosed custom actions.
   ☒ **A** is incorrect for SimpleTags (this is the correct technique for classic tags, but of course, simple tags do not have a `doEndTag()` method). **B** and **F** are incorrect because there is no such method (in the simple or classic tag model) as `doPage()`. **C** and **D** are incorrect because the `doTag()` method returns nothing (**void**), so any attempt to return an integer literal or constant will result in a compilation error.

8. ☑ **A** is the correct answer. All the work that the code does using the java.io.StringWriter goes nowhere. When `JspFragment.invoke(null)` is executed at the end of the **for** loop, however, the contents of the body (the letter *a*) are output to the JspWriter associated with the page. This happens three times within the loop—hence, `aaa` is output to the page from the tag.
   ☒ **B**, **C**, **D**, and **E** are incorrect according to the reasoning in the correct answer.

9. ☑ **B**, **E**, and **F** are the correct answers. **B** is correct because simple tags are not allowed JSP content: `scriptless`, `tagdependent`, and `empty` are the only valid values for a simple tag's `<body-content>`. **E** is correct because the most common use of simple tag handler code is to manipulate attributes that are accessed by EL variables in the body of the action—and for the EL variables to be interpreted, the only valid setting is scriptless. **F** is correct: If `<body-content>` is declared as empty, the JSP container will not call the `setJspBody()` method on the simple tag handler class.
   ☒ **A** is incorrect as there is no such element as `<simpletag>`; the `<tag>` element is used to declare both classic and simple tags. **C** is also incorrect, for simple and classic tags can be mixed freely in the same tag library descriptor file—as long, of course, as their names are unique. **D** is incorrect, though almost right—there is no default for simple tags (though as discussed in the correct answer **E**, `scriptless` is a common choice).

10. ☑ **E** is the correct answer. First a page attribute is set up in the JSP called `counter`, with a value of 1. This is retrieved in the tag handler code into the **int** value *i*. On entry to the **for** loop, the page attribute *counter* is set to the value of *i*, so it remains at 1. Then the body of the action is invoked. This executes a `<c:forEach>` loop beginning at the current value of *counter*, and ending at 3. So the `<c:forEach>` loop executes three times, printing *counter* (with a value of 1) three times, through the EL syntax `${counter}`. Back now to the **for** loop in the tag handler code: second iteration. *i* is incremented to 2, so the attribute *counter* is incremented to

2. The body is invoked again, and the `<c:forEach>` loop outputs "2" twice. Repeat the process, and "3" is output once. So the entire output is: 111 22 3.

☒ **A**, **B**, **C**, **D**, and **F** are incorrect following the reasoning in the correct answer.

## The Tag File Model

11. ☑ **D** and **E** are the correct answers. **D** looks strange but is the namespace convention for declaring a tag file directory in a JSP document (the namespace doesn't have to be declared in an `<html>` opening tag but is usually declared in the root element for the document). **E** is the right way to reference a tag file directory in a conventional `taglib` directive.

☒ **A** is incorrect—it depicts the correct way to declare a tag library descriptor, not a directory for tag files. **B** is incorrect because `taglocation` is a made-up attribute for the `taglib` directive. **C** is incorrect because the value for the namespace is not a URN for a tag directory. Finally, **F** is incorrect because the *tagdir* attribute must reference a directory, not a TLD file.

12. ☑ **B** is the correct answer. A translation error occurs as soon as the JSP scriptlet begins at line 4. You are not allowed to embed scriptlets or scriptlet expressions (or any other sort of direct Java language syntax) in a tag file.

☒ **A** and **C** are incorrect because all remaining syntax in the tag file is correct apart from the presence of the scriptlet. **D** is incorrect because the tag file will never be run. **E** is incorrect because you will never get output—although if you transfer this code to a normal JSP file, it will produce the output shown.

13. ☑ **B**, **C**, and **E** are the correct answers: tag, attribute, and variable are the three possible directives present in a tag file.

☒ **A** is incorrect: The `page` directive is valid in JSPs, not tag files—where the tag file replaces it (and shares many of the same attributes). **D** is incorrect: `import` is not a directive, but an attribute of the tag directive. **F** is incorrect—although a tag file has an associated JspContext, jspcontext is not a directive. Finally, **G** is incorrect—*scope* is an attribute of the variable directive.

14. ☑ **A** maps to **8** (`c:set`), **B** maps to **15** (`${counter}`), **C** to **6** (`attribute`), **D** to **1** (the figure 1), **E** to **3** (the figure 3), **F** to **13** (`<jsp:doBody>`), and **G** to **11** (`loopCounter .index`). The key things to note about the code: `${counter}` in the JSP file is independent of `${counter}` in the tag file. However, as the body is re-executed three times (through placing `<jsp:doBody>` in the `<c:forEach>` loop in the tag file), the original value of `${counter}` (a value of 1) is passed through as the initial value for the attribute named *counter* on each occasion.

☒ All other combinations are incorrect.

15. ☑ **G** is the correct answer. Tag files can live directly in the /WEB-INF/tags directory (or any subdirectory of it) or can be packaged with a JAR file. The directory in the JAR file must be /META-INF/tags (or a subdirectory of this), and the JAR file must be located in /WEB-INF/lib.
☒ **A** is incorrect: Tag files in the context directory will just be treated as web content. **B** and **C** are incorrect: /WEB-INF and any arbitrary subdirectory of /WEB-INF won't do according to the specification. **F** is wrong, for it includes one incorrect location. **D** and **E** are correct locations—but because both are correct, one choice on its own is not a correct answer to the question.

## Tag Hierarchies

16. ☑ **D** is the correct answer. All the techniques listed in answers **A**, **B**, and **C** are viable. `TagSupport.findAncestorWithClass()` is a natural approach: You would pass as parameter the class for `<a:tagA>`. `getParent().getParent()` is viable in this circumstance—this technique will return the grandparent tag `<a:tagA>`. However, unless you can be very sure of the tag hierarchy, it's not a great approach. What may come as a surprise is that `SimpleTagSupport.findAncestorWithClass()` works in the context of classic tags. In fact, it's more flexible than `TagSupport.findAncestorWithClass()`—it returns a JspTag rather than a Tag. This means that the method can be used to find both simple and classic tags in the hierarchy (whereas `TagSupport.findAncestorWithClass()` is restricted to classic tags only).
☒ **A**, **B**, **C**, **E**, and **F** are incorrect according to the reasoning in the correct answer. (**A**, **B**, and **C** are correct approaches, but because all three are correct, you have to select answer **D**.)

17. ☑ **C** and **D** are the correct answers. Tags that qualify as template text—whether XML or HTML—are not part of the JSP custom tag hierarchy accessible through methods such as `getParent()`.
☒ **A**, **B**, **E**, and **F** are incorrect answers. All are instances of objects that implement JspTag and mostly Tag as well—so all are possible to recover with one of the two `getParent()` methods (provided in the Tag and SimpleTag interfaces).

18. ☑ **A** and **G** are the correct answers. Because the JspTag interface has no methods of its own (it is only a marker interface), then the only methods available in a JspTag reference are those on java.lang.Object. Hence, `notifyAll()` and `hashCode()` would be valid method calls.
☒ **B**, **C**, and **D** are incorrect, for all are methods in the Tag interface, which is a subinterface of JspTag. **E** is incorrect because `doAfterBody()` is a method in the IterationTag interface, yet farther down the hierarchy. **F** is incorrect, for although `clone()` is a method on Object, it is protected—only **public** methods are available to call on a local reference.

19. ☑ **A** maps to **1** (Tag); **B**, **C**, and **D** map to **5** (TagAdapter); **E** maps to **8** (`getAdaptee()`); **F** maps to **9** (the variable *enclosing*); and **G** maps to **14** (the return value EVAL_BODY _INCLUDE—beware of fake constants in the answers!).
☒ All other combinations are incorrect.

20. ☑ **A** and **C** are the correct answers. **A** is correct, for `doAfterBody()` will be executed after the evaluation of the body that contains the child tag. **C** is correct because by calling the `invoke()` method, the parent tag forces execution of the body containing the child tag, which sets the reference to the child tag that it needs.
☒ **B** is incorrect because the body (containing the child tag) hasn't yet been evaluated, so the child tag cannot have set a reference for the parent tag to use. **D** is incorrect because there is no life cycle method invoked on a simple tag after the `doTag()` method completes. **E** and **F** are incorrect because the methods are made up: There are no parent-to-child methods. This is a pity, but obvious when you consider that a JSP is processed in strict sequence, so parent tags always come before child tags.

# LAB ANSWER

Deploy the WAR file from the CD called lab09.war, in the /sourcecode/chapter09 directory. This contains a sample solution. You can call the initial document with a URL such as:

```
http://localhost:8080/lab09/cardGame.jspx
```

The output should be no different from before—see Exercise 9-4 for an illustration.