brief contents

Part 1  Getting started  1

1  Understanding Java servlets  3
2  Understanding JavaServer Pages  14
3  Web application and HTTP basics  21

Part 2  Servlets  29

4  The servlet model  31
5  Structure and deployment  67
6  The servlet container model  83
7  Using filters  97
8  Session management  119
9  Developing secure web applications  139

Part 3  JavaServer Pages and design patterns  163

10  The JSP technology model—the basics  165
11  The JSP technology model—advanced topics  188
12  Reusable web components  219
13  Creating JSPs with the Expression Language (EL)  236
14  Using JavaBeans  251
15  Using custom tags  285
16  Developing “Classic” custom tag libraries  309
17  Developing “Simple” custom tag libraries  352
18  Design patterns  376

Appendices

A  Installing Tomcat 5.0.25  403
B  A sample web.xml file  408
C  Review Q & A  412
D  Exam Quick Prep  475
CHAPTER 10

The JSP technology model—the basics

10.1 SP syntax elements 166
10.2 The JSP page life cycle 173
10.3 Understanding JSP page directive attributes 181
10.4 Summary 186
10.5 Review questions 186

EXAM OBJECTIVES

6.1 Identify, describe, or write the JSP code for the following expressions:
   • Template text;
   • Scripting elements (comments, directives, declarations, scriptlets, and expressions);
   • Standard and custom actions; and
   • Expression language elements.
   (Section 10.1)

6.2 Write JSP code that uses the directive:
   • ‘page’ (with attributes ‘import’, ‘session’, ‘contentType’, and ‘isELIgnored’)
   • ‘include’, and
   • ‘taglib’.
   (Section 10.3)

6.4 Describe the purpose and event sequence of the JSP life cycle:
   • JSP page translation
   • JSP page compilation
• Load class
• Create instance
• Call the jspinit method
• Call the _jspService method
• Call the _jspDestroy method

(Section 10.2)

**INTRODUCTION**

In the J2EE suite of specifications that includes servlets, JavaServer Pages (JSP), the Java Naming and Directory Interface (JNDI), Enterprise JavaBeans (EJB), and so forth, the JSP is a web-tier specification that supplements the Servlet specification and is useful in the development of web interfaces for enterprise applications. JSP is a technology that combines the HTML/XML markup languages and elements of the Java programming language to return dynamic content to a web client. For this reason, it is commonly used to handle the presentation logic of a web application, although the JSP pages may also contain business logic.

In this chapter, we will discuss the basic syntax of the JSP scripting language and the JSP page life cycle. This chapter gives you the basics you need to understand the JSP technology model and will help you grasp the more complex topics covered in the next chapter.

10.1 **JSP SYNTAX ELEMENTS**

Just like any other language, the JSP scripting language has a well-defined grammar and includes syntax elements for performing various tasks, such as declaring variables and methods, writing expressions, and calling other JSP pages. At the top level, these syntax elements, also called JSP tags, are classified into six categories. Table 10.1 summarizes the element categories and their basic use.

<table>
<thead>
<tr>
<th>JSP tag type</th>
<th>Brief description</th>
<th>Tag syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directive</td>
<td>Specifies translation time instructions to the JSP</td>
<td><code>&lt;%@ Directives %&gt;</code></td>
</tr>
<tr>
<td>Declaration</td>
<td>Declares and defines methods and variables</td>
<td><code>&lt;%! Java Declarations %&gt;</code></td>
</tr>
<tr>
<td>Scriptlet</td>
<td>Allows the developer to write free-form Java code in a</td>
<td><code>&lt;% Some Java code %&gt;</code></td>
</tr>
<tr>
<td></td>
<td>JSP page</td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>Used as a shortcut to print values in the output</td>
<td><code>&lt;%= An Expression %&gt;</code></td>
</tr>
<tr>
<td></td>
<td>HTML of a JSP page</td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>Provides request-time instructions to the JSP engine</td>
<td><code>&lt;jsp:actionName /&gt;</code></td>
</tr>
<tr>
<td>Comment</td>
<td>Used for documentation and for commenting out parts of</td>
<td><code>&lt;%-- Any Text --%&gt;</code></td>
</tr>
<tr>
<td></td>
<td>JSP code</td>
<td></td>
</tr>
</tbody>
</table>
The exam objectives covered in this chapter require you to know the syntax and purpose of the first four element types: directives, declarations, scriptlets, and expressions. We will briefly introduce actions in section 10.1.5, and explain them in detail in chapters 12 and 14. Although you don’t need to be familiar with comments to do well on the exam, they are very useful when writing JSP pages, and we will discuss them briefly in section 10.1.6.

Listing 10.1 is a simple JSP page that counts the number of times it is visited. It demonstrates the use of the different elements, which we will explain in the sections following the listing.

```
<html><body>
<%@ page language="java" %>
<%! int count = 0;       %>
<%  count++;             %>
Welcome! You are visitor number
<%= count                %>
</body></html>
```

When this file is accessed for the first time via the URL http://localhost:8080/chapter10/counter.jsp, it displays the following line in the browser window:

```
Welcome! You are visitor number 1
```

On subsequent requests, the counter is incremented by 1 before the message is printed.

### 10.1.1 Directives

Directives provide general information about the JSP page to the JSP engine. There are three types of directives: page, include, and taglib.

A `page` directive informs the engine about the overall properties of a JSP page. For example, the following `page` directive informs the JSP engine that we will be using Java as the scripting language in our JSP page:

```
<%@ page language="java" %>
```

An `include` directive tells the JSP engine to include the contents of another file (HTML, JSP, etc.) in the current page. Here is an example of an `include` directive:

```
<%@ include file="copyright.html" %>
```

A `taglib` directive is used for associating a prefix with a tag library. The following is an example of a `taglib` directive:

```
<%@ taglib prefix="test" uri="taglib.tld" %>
```
See section 10.3 for details on the page directive. In chapter 12, “Reusable web components,” we will take a close look at the include directive. Because the concept of a tag library is a vast topic in itself, the exam objectives devote two sections to it. We will learn about the taglib directive in detail in chapter 15, “Using custom tags,” and learn specific methods of tag development in chapters 16 and 17.

A directive always starts with `<%@` and ends with `%>`. The general syntax of the three directives is

```
<%@ page attribute-list %>
<%@ include attribute-list %>
<%@ taglib attribute-list %>
```

In the sample tags above, attribute-list represents one or more attribute-value pairs that are specific to the directive. Here are some important points to remember about the syntax of the directives:

- The tag names, their attributes, and their values are all case sensitive.
- The value must be enclosed within a pair of single or double quotes.
- A pair of single quotes is equivalent to a pair of double quotes.
- There must be no space between the equals sign (=) and the value.

## 10.1.2 Declarations

Declarations declare and define variables and methods that can be used in the JSP page. The following is an example of a JSP declaration:

```
<%! int count = 0; %>
```

This declares a variable named `count` and initializes it to 0. The variable is initialized only once when the page is first loaded by the JSP engine, and retains its value in subsequent client requests. That is why the `count` variable in listing 10.1 is not reset to 0 each time we access the page.

A declaration always starts with `<%!` and ends with `%>`. It can contain any number of valid Java declaration statements. For example, the following tag declares a variable and a method in a single tag:

```
<%! 
    String color[] = {"red", "green", "blue"}; 
    String getColor(int i) 
    { 
        return color[i]; 
    } 
%>
```

1 Theoretically, a JSP declaration can contain any valid Java declaration including inner classes and static code blocks. However, such declarations are rarely used.
We can also write the above two Java declaration statements in two JSP declaration tags:

```java
<%! String color[] = {"red", "green", "blue"}; %>
<%! 
    String getColor(int i)
    {
        return color[i];
    }
%>
```

Note that since the declarations contain Java declaration statements, each variable's declaration statement must be terminated with a semicolon.

### 10.1.3 Scriptlets

**Scriptlets** are Java code fragments that are embedded in the JSP page. For example, this line from the `counter.jsp` example (listing 10.1) is a JSP scriptlet:

```jsp
<% count++ %>
```

The scriptlet is executed each time the page is accessed, and the `count` variable is incremented with each request.

Since scriptlets can contain any Java code, they are typically used for embedding computing logic within a JSP page. However, we can use scriptlets for printing HTML statements, too. The following is equivalent to the code in listing 10.1:

```jsp
<%@ page language="java" %>
<%! int count = 0;  %>
<% 
    out.print("<html><body>");
    count++;
    out.print("Welcome! You are visitor number " + count);
    out.print("</body></html>");
%>
```

Instead of writing normal HTML code directly in the page, we are using a scriptlet to achieve the same effect. The variable `out` refers to an object of type `javax.servlet.jsp.JspWriter`. We will learn about `out` in chapter 11, “The JSP technology model—advanced topics.”

A scriptlet always starts with `<%` and ends with `%>`. Note, however, that unlike the other elements, the opening tag of a scriptlet does not have any special character following `<%`. The code within the scriptlet must be valid in the Java programming language. For example, this is an error because it does not terminate the print statement with a semicolon:

```jsp
<% out.print(count) %>
```
10.1.4 Expressions

Expressions act as placeholders for Java language expressions. This is an example of a JSP expression:

```<%= count %>```

The expression is evaluated each time the page is accessed, and its value is then embedded in the output HTML. For instance, in the previous `counter.jsp` example (listing 10.1), instead of incrementing the `count` variable in a scriptlet, we could have incremented it in the expression itself:

```html
<html><body>
<%@ page language="java" %>
<%! int count = 0;       %>

Welcome! You are visitor number <%= ++count %>

</body></html>
```

A JSP expression always starts with `<%=` and ends with `%>`. Unlike variable declarations, expressions must not be terminated with a semicolon. Thus, the following is not valid:

```
<%= count; %>
```

We can print the value of any object or any primitive data type (int, boolean, char, etc.) to the output stream using an expression. We can also print the value of any arithmetic or Boolean expression or a value returned by a method call. The exam may ask you to identify valid JSP expressions. Tables 10.2 and 10.3 contain some examples of valid and invalid JSP expressions based on the following declarations:

```%
  int anInt = 3;
  boolean aBool = true;
  Integer anIntObj = new Integer(3);
  Float aFloatObj = new Float(12.6);
  String str = "some string";
  StringBuffer sBuff = new StringBuffer();
  char getChar(){ return 'A'; }
%
```

Table 10.2 Valid JSP expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;%= 500 %&gt;</code></td>
<td>An integral literal</td>
</tr>
<tr>
<td><code>&lt;%= anInt*3.5/100-500 %&gt;</code></td>
<td>An arithmetic expression</td>
</tr>
<tr>
<td><code>&lt;%= aBool %&gt;</code></td>
<td>A Boolean variable</td>
</tr>
<tr>
<td><code>&lt;%= false %&gt;</code></td>
<td>A Boolean literal</td>
</tr>
</tbody>
</table>

continued on next page
10.1.5 Actions

Actions are commands given to the JSP engine. They direct the engine to perform certain tasks during the execution of a page. For example, the following line instructs the engine to include the output of another JSP page, copyright.jsp, in the output of the current JSP page:

```
<jsp:include page="copyright.jsp" />
```

There are six standard JSP actions:

- jsp:include
- jsp:forward
- jsp:useBean
- jsp:setProperty
- jsp:getProperty
- jsp:plugin

The first two, jsp:include and jsp:forward, enable a JSP page to reuse other web components. We will discuss these two actions in chapter 12, “Reusable web components.”

The next three, jsp:useBean, jsp:setProperty, and jsp:getProperty, are related to the use of JavaBeans in JSP pages. We will discuss these three actions in chapter 14, “Using JavaBeans.”

The last action, jsp:plugin, instructs the JSP engine to generate appropriate HTML code for embedding client-side components, such as applets. This action is not specified in the exam objectives, and its details are beyond the scope of this book.
In addition to the six standard actions, a JSP page can have user-defined actions. These are called custom tags. We will learn about custom tags in chapters 15 (“Using custom tags”), 16 (“Developing classic custom tag libraries”), and 17 (“Developing simple custom tag libraries”).

The general syntax of a JSP action is

```html
<jsp:actionName attribute-list />
```

In this tag, `actionName` is one of the six actions mentioned and `attribute-list` represents one or more attribute-value pairs that are specific to the action. As with directives, you should keep in mind these points:

- The action names, their attributes, and their values are case sensitive.
- The value must be enclosed within a pair of single or double quotes.
- A pair of single quotes is equivalent to a pair of double quotes.
- There must be no space between the equals sign (=) and the value.

### 10.1.6 Comments

Comments do not affect the output of a JSP page in any way but are useful for documentation purposes. The syntax of a JSP comment is

```html
<%-- Anything you want to be commented --%>
```

A JSP comment always starts with `<%--` and ends with `--%>`.

We can comment the Java code within scriptlets and declarations by using normal Java-style comments and the HTML portions of a page by using HTML-style comments, as shown here:

```html
<html><body>
Welcome!
<%-- JSP comment --%>
<% //Java comment %>
<!-- HTML comment -->
</body></html>
```

As we mentioned earlier, the exam does not cover comments, but they can be quite useful when you’re debugging JSP pages. The JSP engine drops everything between `<%--` and `--%>`, so it is easy to comment out large parts of a JSP page—including nested HTML and other JSP tags. However, remember that you cannot nest JSP comments within other JSP comments.

**Quizlet**

**Q:** Which of the following page directives are valid?

- a `<% page language="java" %>`
- b `<%! page language="java" %>`
- c `<%@ page language="java" %>`
A: Only option c is correct. Directives use an @ in the opening tag.

Q: What is wrong with the following code?

```
<%! int i = 5; %>
<%! int getI() { return i; } %>
```

A: The opening tag for a declaration is <%! and not <!%>.

Q: Assuming that myObj refers to an object and m1() is a valid method on that object, tell why each of the following are valid or invalid JSP constructs.

a. `<% myObj.m1() %>`
b. `<% myObj.m1() %>`
c. `<% =myObj.m1() %>`
d. `<% =myObj.m1(); %>`

A: The following table explains why an option is valid or invalid.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;% myObj.m1() %&gt;</code></td>
<td>Invalid: It is not an expression because it does not have an = sign. It is an invalid scriptlet because a semicolon is missing at the end of the method call.</td>
</tr>
<tr>
<td><code>&lt;%=myObj.m1() %&gt;</code></td>
<td>Depends: The = sign makes it an expression. But if the return type of the method m1() is void, it is invalid. A method call inside an expression is valid if and only if the return type of the method is not void.</td>
</tr>
<tr>
<td><code>&lt;% =myObj.m1() %&gt;</code></td>
<td>Invalid: There is a space between <code>&lt;%</code> and <code>=</code>. Hence, it is not an expression but a scriptlet. However, the scriptlet construct is not valid because <code>=myObj.m1()</code>, by itself, is not a valid Java statement.</td>
</tr>
<tr>
<td><code>&lt;% =myObj.m1();%&gt;</code></td>
<td>Invalid: Same as previous example except that it has a semicolon.</td>
</tr>
</tbody>
</table>

The valid way to write this as a scriptlet is

```
<% myObj.m1(); %>
```

However, this will just call the method; it will not generate any output. If the method m1() returns a value, then the correct way to write this as an expression is

```
<%= myObj.m1() %>
```

This will print the return value of the method call to the output HTML.

### 10.2 The JSP Page Life Cycle

A JSP page goes through seven phases in its lifetime. These phases are called *life-cycle phases*. The exam requires you to know the sequence of the phases and the activity that takes place in each of the phases. But before we start discussing the life cycle of a JSP page, we need to understand the two important points regarding JSP pages that are explained in the following sections.
10.2.1 **JSP pages are servlets**

Although, structurally, a JSP page *looks* like an HTML page, it actually runs as a servlet. The JSP engine parses the JSP file and creates a corresponding Java file. This file declares a servlet class whose members map directly to the elements of the JSP file. The JSP engine then compiles the class, loads it into memory, and executes it as it would any other servlet. The output of this servlet is then sent to the client. Figure 10.1 illustrates this process.

10.2.2 **Understanding translation units**

Just as an HTML page can include the contents of other HTML pages (for example, when using frames), a JSP page can include the contents of other JSP pages and HTML

```html
<html><body>
<div style="text-align: center;">
<p>Welcome! You are visitor number <%= count %></p>
</body></html>
```

**File counter.jsp**

```
// In Generated Servlet
int count = 0;
// in _jspService()
out.write("<html><body>");
count++;
out.write("Welcome! You are visitor number 
");
out.print(count);
out.write(" 
</body></html>
");
```

**Generated servlet for counter.jsp**

```
<html><body>
Welcome! You are visitor number 1
</body></html>
```

**Output HTML**

**Figure 10.1 A JSP page as a servlet**
pages. This is done with the help of the include directive (see chapter 12 for more information). But an important thing to remember here is that when the JSP engine generates the Java code for a JSP page, it also inserts the contents of the included pages into the servlet that it generates. The set of pages that is translated into a single servlet class is called a translation unit. Some of the JSP tags affect the whole translation unit and not just the page in which they are declared.

Keep in mind these other points regarding a translation unit:

- The page directives explained in section 10.3 affect the whole translation unit.
- A variable declaration cannot occur more than once in a single translation unit. For example, we cannot declare a variable in an included page using the include directive if it is already declared in the including page since the two pages constitute a single translation unit.
- The standard action <jsp:useBean> cannot declare the same bean twice in a single translation unit. We examine the jsp:useBean action further in chapter 14.

### 10.2.3 JSP life-cycle phases

You might have observed that when a JSP page is accessed for the first time, the server is slower in responding than it is in the second, third, and subsequent accesses. This is because, as we mentioned previously, every JSP page must be converted into an instance of a servlet class before it can be used to service client requests. For each request, the JSP engine checks the timestamps of the source JSP page and the corresponding servlet class file to determine if the JSP page is new or if it has already been converted into a class file. Therefore, if we modify a JSP page, the whole process of converting the JSP page into a servlet is performed again. This process consists of seven phases, and you need to understand their order and significance for the exam. Table 10.4 lists the phases in the order in which they occur.

<table>
<thead>
<tr>
<th>Phase name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page translation</td>
<td>The page is parsed and a Java file containing the corresponding servlet is created.</td>
</tr>
<tr>
<td>Page compilation</td>
<td>The Java file is compiled.</td>
</tr>
<tr>
<td>Load class</td>
<td>The compiled class is loaded.</td>
</tr>
<tr>
<td>Create instance</td>
<td>An instance of the servlet is created.</td>
</tr>
<tr>
<td>Call jspInit()</td>
<td>This method is called before any other method to allow initialization.</td>
</tr>
<tr>
<td>Call _jspService()</td>
<td>This method is called for each request.</td>
</tr>
<tr>
<td>Call jspDestroy()</td>
<td>This method is called when the servlet container decides to take the servlet out of service.</td>
</tr>
</tbody>
</table>
Creating the servlet instance

The first four life-cycle phases involve the process of converting the JSP page into an instance of a servlet class.

Translation

During the translation phase, the JSP engine reads a JSP page, parses it, and validates the syntax of the tags used. For example, the following directive is invalid since it uses an uppercase P in `Page` and will be caught during the translation phase:

```jsp
<%@ Page language="java" %>
```

In addition to checking the syntax, the engine performs other validity checks, some of which involve verifying that

- The attribute-value pairs in the directives and standard actions are valid.
- The same JavaBean name is not used more than once in a translation unit.
- If we are using a custom tag library, the library is valid.
- The usage of custom tags is valid.

Once the validations are completed, the engine creates a Java file containing a public servlet class.

Compilation

In the compilation phase, the Java file generated in the previous step is compiled using the normal Java compiler javac (or using a vendor-provided compiler or even a user-specified compiler\(^2\)). All the Java code that we write in declarations, scriptlets, and expressions is validated during this phase. For example, the following declaration tag is a valid JSP tag and will pass the translation phase, but the declaration statement is not a valid Java declaration statement because it does not end with a semicolon and will be caught during the compilation phase:

```jsp
%! int count = 0 %>
```

Scripting language errors (Java, in this case) are caught during the compilation phase. We can force a compilation of a JSP page without actually executing it by using the precompilation request parameter `jsp_precompile`. For example, if we want to compile the `counter.jsp` page without executing it, we must access the page as

```
http://localhost:8080/chapter10/counter.jsp?jsp_precompile=true
```

The engine will translate the JSP page and compile the generated servlet class without actually executing the servlet. This can be quite useful during the development phase if we have complex JSP pages that create database connections or access other J2EE

\(^2\) This varies from container to container. Please consult the servlet container documentation for more information.
services. Also, it is always a good idea to precompile all the pages. In this way, we check for syntax errors and keep the pages ready to be served, thus reducing the response time for the first request to each page.

**NOTE** The parameter jsp_precompile takes a Boolean value, true or false. If the value is false, the precompilation will not occur. The parameter can also be specified without any value, in which case the default is true:

```
http://localhost:8080/chapter10/counter.jsp?jsp_precompile
```

In either case, true or false, the page will not be executed.

Also, this would be a good place to point out that all of the request parameter names that include the prefix jsp are reserved and must not be used for user-defined values. Thus, the following usage is not recommended and may result in unexpected behavior:

```
http://localhost:8080/chapter10/counter.jsp?jspTest=myTest
```

**Loading and instantiation**

After successful compilation, the container loads the servlet class into memory and instantiates it.

**Calling the JSP life-cycle methods**

The generated servlet class for a JSP page implements the HttpServletRequest interface of the javax.servlet.jsp package. The HttpServletRequest interface extends the JspPage interface of the same package, which in turn extends the Servlet interface of the javax.servlet package. The generated servlet class thus implements all the methods of these three interfaces and is also known as the page’s implementation class.

The JspPage interface declares only two methods—jspInit() and jspDestroy()—that must be implemented by all JSP pages regardless of the client-server protocol. However, the JSP specification has provided the HttpServletRequest interface specifically for JSP pages serving HTTP requests. This interface declares one method: _jspService(). Here are the signatures of the three JSP methods:

```java
public void jspInit();

public void _jspService(HttpServletRequest request,
                          HttpServletResponse response)
                          throws
                          javax.servlet.ServletException,
                          java.IO.IOException;

public void jspDestroy();
```

These methods are called the life-cycle methods of the JSP pages. The jspInit(), _jspService(), and jspDestroy() methods of a JSP page are equivalent to the init(), service(), and destroy() methods of a servlet, respectively.
Every JSP engine vendor provides a vendor-specific class that is used as a base class for the page’s implementation class. This base class provides the default implementations of all the methods of the Servlet interface and the default implementations of both methods of the JspPage interface: jspInit() and jspDestroy(). During the translation phase, the engine adds the _jspService() method to the JSP page’s implementation class, thus making the class a concrete subclass of the three interfaces.

jspInit()
The container calls jspInit() to initialize the servlet instance. It is called before any other method, and is called only once for a servlet instance. We normally define this method to do initial or one-time setup, such as acquiring resources and initializing the instance variables that have been declared in the JSP page using <%! ... %> declarations.

_jspService()
The container calls the _jspService() for each request, passing it the request and the response objects. All of the HTML elements, the JSP scriptlets, and the JSP expressions become a part of this method during the translation phase. We discuss the details of this method in chapter 11.

jspDestroy()
When the container decides to take the instance out of service, it calls the jspDestroy() method. This is the last method that is called on the servlet instance, and it is used to clean up the resources acquired in the jspInit() method.

We are not required to implement the jspInit() and jspDestroy() methods, since they have already been implemented by the base class. If we need to override them, we can do so using the JSP declaration tag <%! ... %>. However, we cannot define our own _jspService() method because the engine generates it automatically.

10.2.4 JSP life-cycle example
Let’s modify our counter example to add persistence capabilities to it so that the counter does not start from 1 each time the server is shut down and restarted. Listing 10.2 illustrates how we can

- Use jspInit() to load the previous value of the counter from a file when the server starts.
- Use jspDestroy() to save the final value to the file when the server shuts down.
<%@ page language="java" import="java.io.*" %>

<%! 
// A variable to maintain the number of visits.
int count = 0;

// Path to the file, counter.db, which stores the count 
// value in a serialized form. The file acts like a database.
String dbPath;

// This is the first method called by the container,
// when the page is loaded. We open the db file,
// read the integer value, and initialize the count variable.
public void jspInit()
{
    try
    {
        dbPath = getServletContext().getRealPath("/WEB-INF/counter.db");
        FileInputStream fis = new FileInputStream(dbPath);
        DataInputStream dis = new DataInputStream(fis);
        count = dis.readInt();
        dis.close();
    }
    catch(Exception e)
    {
        log("Error loading persistent counter", e);
    }
}
%

<%--
The main content that goes to the browser.
This will become a part of the generated _jspService() method
--%>
<html><body>
<% count++;
%>
Welcome! You are visitor number
<%= count %>
</body></html>

<%! 
// This method is called by the container only once when the 
// page is about to be destroyed. We open the db file in this 
// method and save the value of the count variable as an integer.

public void jspDestroy()
{
    try
    {
        FileOutputStream fos = new FileOutputStream(dbPath);
        DataOutputStream dos = new DataOutputStream(fos);
        dos.writeInt(count);
        dos.close();
    }
%>
dos.writeInt(count);
dos.close();
}
catch(Exception e)
{
    log("Error storing persistent counter", e);
}

This example illustrates three things: the use of the jspInit() method, the use of the jspDestroy() method, and the use of the getServletContext() method.

When the page is first loaded into the servlet container, the engine will call the jspInit() method. In this method, we initialize the count variable to the value read in from the resource database file "/WEB-INF/counter.db". During its lifetime, the JSP page may be accessed zero or more times, and each time the _jspService() method will be executed. Since the scriptlet `<% count++; %>` becomes a part of the _jspService() method, the expression count++ is evaluated each time, increasing the counter by 1. Finally, when the page is about to be destroyed, the container will call the jspDestroy() method. In this method, we open the resource database file again, and save the latest value of the variable count into it.

Because the JSP page is converted into a servlet, we can call all the methods in a JSP page that we can call on a servlet. Hence we can get the ServletContext object via getServletConfig().getServletContext(). Also, the base class of the page's generated class extends javax.servlet.http.HttpServlet, which gives us access to the log() method. In Tomcat and many other containers, the base class of the page's generated class also implements the ServletConfig interface. Thus, in both methods, jspInit() and jspDestroy(), we get the ServletContext object by using the method getServletContext(), which is actually defined in the javax.servlet.ServletConfig interface. The returned ServletContext object can then be used in a JSP page exactly the way we use it in normal servlets. In our example, we are using the ServletContext object to convert the relative path of a resource into its real path. If the web application is installed in the directory C:\jakarta-tomcat5.0.25\webapps\chapter10, then a call to getServletContext().getRealPath("/WEB-INF/counter.db"); will return C:\jakarta-tomcat-5.0.25\webapps\chapter10\WEB-INF\counter.db.

When the server is started the very first time and the page is first accessed, the file counter.db does not exist and a FileNotFoundException is thrown. We can catch the exception and log the error message with this method.3 When the server is

3 Tomcat uses the <CATALINA_HOME>\logs directory as the default directory to create log files.
shut down the first time, the \texttt{jspDestroy()} method creates a new file, and the current value of the variable is written into it. When the server is started the second time and the JSP page is loaded, the \texttt{jspInit()} method will find the file and initialize the count variable to its previously saved value.

The JSP technology thus combines the best of both worlds: the ease of use offered by the web scripting methodology and the object-oriented features of the servlet technology.

\section*{10.3 Understanding JSP Page Directive Attributes}

A page directive informs the JSP engine about the overall properties of a JSP page. This directive applies to the entire translation unit and not just to the page in which it is declared. Table 10.5 describes the 12 possible attributes for the \texttt{page} directive.

\begin{table}[h]
\centering
\caption{Attributes for the page directive}
\begin{tabular}{|l|l|l|}
\hline
Attribute name & Description & Default value/s \\
\hline
\texttt{import} & A comma-separated list of Java classes and packages that we want to use in the JSP page. & \texttt{java.lang.*; java.servlet.*; java.servlet.jsp.*; java.servlet.http.*;} \\
\hline
\texttt{session} & A Boolean literal specifying whether the JSP page takes part in an HTTP session. & \texttt{true} \\
\hline
\texttt{errorPage} & Specifies a relative URL to another JSP page that is capable of handling errors on behalf of the current page. & \texttt{null} \\
\hline
\texttt{isErrorPage} & A Boolean literal specifying whether the current JSP page is capable of handling errors. & \texttt{false} \\
\hline
\texttt{language} & Any scripting language supported by the JSP engine. & \texttt{java} \\
\hline
\texttt{extends} & Any valid Java class that implements \texttt{javax.servlet.jsp.JspPage}. & Implementation dependent \\
\hline
\texttt{buffer} & Specifies the size of the output buffer. If a buffer size is specified, it must be in kilobytes (kb). If buffering is not required, specify the string \texttt{none}. & Implementation dependent \\
\hline
\texttt{autoFlush} & A Boolean literal indicating whether the buffer should be flushed when it is full. & \texttt{true} \\
\hline
\texttt{info} & Any informative text about the JSP page. & Implementation dependent \\
\hline
\texttt{contentType} & Specifies the MIME type and character encoding for the output. & \texttt{text/html;charset=ISO-8859-1} \\
\hline
\texttt{pageEncoding} & Specifies the character encoding of the JSP page. & \texttt{ISO-8859-1} \\
\hline
\end{tabular}
\end{table}
While the exam requires that you know all of the valid page directive attributes and their values, it focuses more on the usage of the first four: import, session, errorPage, and isErrorPage.

10.3.1 The import attribute
The import attribute of a page directive is similar to the import statement in a Java class. For example, if we want to use the Date class of the package java.util, then we have to either use the fully qualified class name in the code or import it using the page directive. At the time of translation, the JSP engine inserts an import statement into the generated servlet for each of the packages declared using this attribute.

We can import multiple packages in a single tag by using a comma-separated list of package names, as shown here:

```jsp
<%@ page import="java.util.*, java.io.*, java.text.*, com.mycom.*, com.mycom.util.MyClass" %>
```

We can also use multiple tags for readability. For example, the above page directive can also be written as:

```jsp
<%@ page import="java.util.*" %>
<%@ page import="java.io.*" %>
<%@ page import="java.text.*" %>
<%@ page import="com.mycom.*, com.mycom.util.MyClass" %>
```

Since the order of import statements in a Java class does not matter, the order of import tags shown here does not matter, either. A JSP engine always imports the java.lang.*, javax.servlet.*, javax.servlet.jsp.*, and javax.servlet.http.* packages, so we do not have to import them explicitly.

**NOTE** import is the only attribute of the page directive that can occur multiple times in a translation unit. Duplicate values are ignored.

10.3.2 The session attribute
The session attribute indicates whether the JSP page takes part in an HTTP session. The default value is true, in which case the JSP engine declares the implicit variable session. (We will learn more about implicit variables in chapter 11.) If we do not want the page to participate in a session, then we have to explicitly add the following line:

```jsp
<%@ page session="false" %>
```

10.3.3 The errorPage and isErrorPage attributes
During the execution of a page, it is possible that the embedded Java code will throw exceptions. Just as in normal Java programs, we can handle the exceptions in JSP pages using try-catch blocks. However, the JSP specification defines a better approach, which separates the error-handling code from the main page and thus promotes reusability of the exception-handling mechanism. In this approach, a JSP page uses the
errorPage attribute to delegate the exception to another JSP page that has the error-handling code. In listing 10.3, errorHandler.jsp is specified as the error handler.

**Listing 10.3 hello.jsp: Using errorPage to delegate exceptions**

```jsp
<%@ page errorPage="errorHandler.jsp" %>
<html>
<body>
  <%
    if (request.getParameter("name") == null) {
      throw new RuntimeException("Name not specified");
    }
  %>
  Hello, <%=request.getParameter("name")%>
</body>
</html>
```

The JSP page in listing 10.3 throws an exception if the parameter name is not supplied in the request, but it does not catch the exception itself. Instead, with the help of the errorPage attribute, it instructs the JSP engine to delegate the error handling to errorHandler.jsp.

The isErrorPage attribute conveys whether the current page can act as an error handler for any other JSP page. The default value of the isErrorPage attribute is false. For example, the errorHandler.jsp that we used in the previous example must explicitly set this attribute to true, as shown in listing 10.4. In this case, the JSP engine declares the implicit variable exception in the page’s servlet class.

**Listing 10.4 errorHandler.jsp: Handling exceptions**

```jsp
<%@ page isErrorPage="true" %>
<html>
<body>
  Unable to process your request: <%=exception.getMessage()%><br>
  Please try again.
</body>
</html>
```

Notice that this page only extracts the information from the exception and generates an appropriate error message. Because it does not implement any business logic, it can be reused for different JSP pages.

It is not necessary that the errorPage value be a JSP page. It can also be a static file, such as an HTML page:

```jsp
<%@ page errorPage="errorHandler.html" %>
```
Obviously, we cannot write a scriptlet or an expression in the HTML file *error-Handler.html* to generate dynamic messages.

**NOTE** In general, it is always a good programming practice to specify an error page in all the JSP pages. This prevents unanticipated error messages from being displayed on the client’s browser.

### 10.3.4 The language and extends attributes

The `language` attribute specifies the language used by a page in declarations, scriptlets, and expressions. The default value is `java`, which is also the only value allowed by the JSP Specification 2.0. Needless to say, adding the following line to a JSP page is redundant:

```jsp
<%@ page language="java" %>
```

The `extends` attribute specifies that the supplied class be used as a base class of the generated servlet. This is useful only if we want to customize the behavior of the generated servlet class. The default base class is vendor specific and is designed to work efficiently with the rest of the framework. Consequently, this attribute is seldom used. The following line shows the syntax for this attribute:

```jsp
<%@ page extends="mypackage.MySpecialBaseServlet" %>
```

### 10.3.5 The buffer and autoFlush attributes

The `buffer` attribute specifies the minimum size required by the output buffer that holds the generated content until it is sent to the client. The default size of the buffer is JSP engine implementation dependent, but the specification mandates it to be at least `8kb`. The following line sets the buffer size to `32kb`:

```jsp
<%@ page buffer="32kb" %>
```

The value of the buffer is in kilobytes and the suffix `kb` is mandatory. To send the data directly to the client without any buffering, we can specify the value as `none`.

The `autoFlush` attribute specifies whether the data in the output buffer should be sent to the client automatically as soon as the buffer is full. The default value for `autoFlush` is `true`. If it is set to `false` and the buffer is full, an exception is raised when we attempt to add more data to the buffer. Here is the syntax for this attribute:

```jsp
<%@ page autoFlush="false" %>
```

Obviously, the following combinations occurring in a JSP page are invalid and may either cause an error at translation time or have an unknown behavior at runtime:

```jsp
<%@ page buffer="none" autoFlush="false" %>
<%@ page buffer="0kb" autoFlush="false" %>
```
10.3.6 The info attribute
The info attribute allows us to specify the value of the string returned by the getServletInfo() method of the generated servlet. The following line shows one possible use:

```jsp
g%@ page info="This is a sample Page. " %
```

The default value of this attribute is implementation dependent.

10.3.7 The contentType and pageEncoding attributes
The contentType attribute specifies the MIME type and character encoding of the output. The default value of the MIME type is text/html; the default value of the character encoding is ISO-8859-1. The MIME type and character encoding are separated by a semicolon, as shown here:

```jsp
g%@ page contentType="text/html;charset=ISO-8859-1" %
```

This is equivalent to writing the following line in a servlet:

```java
response.setContentType("text/html;charset=ISO-8859-1");
```

The pageEncoding attribute specifies the character encoding of the JSP page. The default value is ISO-8859-1. The following line illustrates the syntax:

```jsp
g%@ page pageEncoding="ISO-8859-1" %
```

Quizlet
Q: Which of the following page directives are valid and which are invalid?

- **a** `<%@ page import="java.util.* java.text.* " %>`
- **b** `<%@ page import="java.util.*", "java.text.* " %>`
- **c** `<%@ page buffer="8kb", session="false" %>`
- **d** `<%@ page import="com.manning.servlets.* " %>
  `<%@ page session="true" %>
  `<%@ page import="java.text.*" %>`
- **e** `<%@ page bgcolor="navy" %>`
- **f** `<%@ page buffer="true" %>`
- **g** `<%@ Page language='java' %>

A: The following table explains why an option is valid or invalid.

<table>
<thead>
<tr>
<th>Page directive</th>
<th>Valid/invalid</th>
<th>Reasons</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;%@ page import=&quot;java.util.* java.text.* &quot; %&gt;</code></td>
<td>Invalid:</td>
<td>A comma is required between the values. <code>&lt;%@ page import=&quot;java.util.*, java.text.* &quot; %&gt;</code></td>
</tr>
<tr>
<td><code>&lt;%@ page import=&quot;java.util.*&quot;, &quot;java.text.* &quot; %&gt;</code></td>
<td>Invalid:</td>
<td>Both packages must be specified in the same string.</td>
</tr>
</tbody>
</table>

continued on next page
10.4 SUMMARY

In this chapter, we examined JavaServer Pages as a web scripting methodology. We learned the basic rules of the six JSP syntax elements—directives, declarations, scriptlets, expressions, actions, and comments—and we examined the first four in depth. We learned that JSP pages are translated into servlet instances before serving the client’s requests, and we reviewed the seven phases of the JSP page life cycle. We then looked at the three life-cycle methods—jspInit(), _jspService(), and jspDestroy()—and how they are used in the initialization, servicing, and destruction of a JSP page.

Through its 12 attributes, a page directive provides information about the overall properties of a JSP page to the JSP engine. We need to understand all of the attributes for writing real-life JSP pages, but in preparing for the exam, it is especially important to understand import, session, errorPage, and isErrorPage.

In the next chapter, we will continue our discussion of JavaServer Pages as we examine some of the more advanced features that form a logical extension of the servlet technology.

10.5 REVIEW QUESTIONS

1. Consider the following code and select the correct statement about it from the options below. (Select one)

```html
<html><body>
  <%! int aNum=5 %>
  The value of aNum is <%= aNum %>
</body></html>
```

a It will print "The value of aNum is 5" to the output.

b It will flag a compile-time error because of an incorrect declaration.
2. Which of the following tags can you use to print the value of an expression to the output stream? (Select two)
   a. `<%@    %>`
   b. `<%!    %>`
   c. `<%    %>`
   d. `<%=` `<%>`
   e. `<%-- --%>`

3. Which of the following methods is defined by the JSP engine? (Select one)
   a. `jspInit()`
   b. `_jspService()`
   c. `_jspService(ServletRequest, ServletResponse)`
   d. `_jspService(HttpServletRequest, HttpServletResponse)`
   e. `jspDestroy()`

4. Which of the following exceptions may be thrown by the `_jspService()` method? (Select one)
   a. `javax.servlet.ServletException`
   b. `javax.servlet.jsp.JSPException`
   c. `javax.servlet.ServletException` and `javax.servlet.jsp.JSPException`
   d. `javax.servlet.ServletException` and `java.io.IOException`
   e. `javax.servlet.jsp.JSPException` and `java.io.IOException`

5. Write the name of the method that you can use to initialize variables declared in a JSP declaration in the space provided. (Write only the name of the method. Do not write the return type, parameters, or parentheses.)
   a. [___________]

6. Which of the following correctly declares that the current page is an error page and also enables it to take part in a session? (Select one)
   a. `<%@ page pageType="errorPage" session="required" %>`
   b. `<%@ page isErrorPage="true" session="mandatory" %>`
   c. `<%@ page errorPage="true" session="true" %>`
   d. `<%@ page isErrorPage="true" session="true" %>`
   e. None of the above.
With the tremendous penetration of J2EE in the enterprise, passing the Sun Certified Web Component Developer exam has become an important qualification for Java and J2EE developers. To pass the SCWCD exam (Number: 310-081) you need to answer 69 questions in 135 minutes and get 62% of them right. You also need $150 and this (completely updated and newly revised) book.

In its first edition, the SCWCD Exam Study Kit was the most popular book used to pass this most desirable web development certification exam. The new edition will help you learn the concepts—large and small—that you need to know. It covers the newest version of the exam and not a single topic is missed.

The SCWCD exam is for Sun Certified Java Programmers who have a certain amount of experience with Servlets and JSPs, but for those who do not, the book starts with three introductory chapters on these topics. Although the SCWCD Exam Study Kit has one purpose, to help you get certified, you will find yourself returning to it as a reference after passing the exam.

What’s Inside

- Expression Language
- JSP Standard Tag Library (JSTL 1.1)
- Custom tags—‘Classic’ and ‘Simple’
- Session management
- Security
- Design patterns
- Filters
- Example code and the Tomcat servlet container
- All exam objectives, carefully explained
- Review questions and quizlets
- Quick Prep section for last-minute cramming

The authors, Deshmukh, Malavia, and Scarpino, are Sun Certified Web Component Developers who have written a focused and practical book thanks to their extensive background in Java/J2EE design and development. They live, respectively, in Iselin, New Jersey, Ardsley, New York, and Austin, Texas.