6

JavaServer Pages

CERTIFICATION OBJECTIVES

- JSP Life Cycle
- JSP Elements
- JSP Directives
- JSP Implicit Objects

✓ Two-Minute Drill

Q&A Self Test

In the previous chapters, you completed a thorough exploration of servlets. In most of the remaining chapters, we will explore a technology that turns servlets on their head: JavaServer Pages, also known as JSP technology, JSP pages, and sometimes simply JSPs. In the pure servlet model, servlet writing is at the front of the development process. JavaServer Pages, by contrast, place servlet creation at the back end of development. Servlets still play a crucial part, but they are, in fact, generated and compiled from JavaServer Page sources.

This is such an important subject that you'll find a big emphasis placed on it in the exam—you're likely to encounter JSP technology in almost half the questions. If you're already familiar with JavaServer Pages at version 1.2, you'll know that they already contain an abundance of neat features. The current exam tests you on version 2.0, which roughly doubles what you need to know about JSPs. If that's the bad news, then the good news is that the enhancements built into version 2.0 make JSP technology an ever more practical and flexible choice for web application development, so the skills you're learning should be both useful and marketable.

CERTIFICATION OBJECTIVE

JSP Life Cycle (Exam Objective 6.4)

Describe the purpose and event sequence of the JSP page life cycle: (1) JSP page translation, (2) JSP page compilation, (3) load class, (4) create instance, (5) call the jspInit method, (6) call the _jspService method, and (7) call the jspDestroy method.

So is JSP technology "better" than servlets? Should you now forget what you learned in the first five chapters (except for passing the exam, of course!) and hone your JSP skills instead? I prefer to view servlets and JavaServer pages as complementary. You might find that a typical web application consists 90 percent of JavaServer pages to encapsulate individual pages or "screens," with 10 percent servlets to control interaction between pages and business components.

The best way to understand when each approach is appropriate to use is to dive into the detail. We'll start with the above exam objective concerning page life cycle: not actually the first in Sun's list, but the one I view as fundamental to getting a handle on JSP technology.

JSP Translation and Execution

So why turn the servlet pattern on its head? Through the exercises, you have written numerous servlets that produce HTML, generally by peppering a liberal sprinkling of `out.write()` statements throughout the `doGet()` or `doPost()` method. By contrast, JavaServer Page technology allows you to write snippets of Java code between HTML statements. Here's an example page that displays the current date and time (note that the line numbers are not part of the source—imagine that you're looking in a text editor that displays line numbers):

```
01 <%@ page language="java" %>
02 <%@ page import="java.util.*" %>
03 <html><head><title>Simple JSP Example</title></head>
04 <body>
05 <h1>Simple JSP Example</h1>
06 <%= new Date() %>
07 </body>
08 </html>
```

Let's just look at the overall "mix" of the lines. Lines 01, 02, and 06 are a bit peculiar, so we'll ignore them for now. However, the remaining five lines are straight HTML. Clearly, this approach confers a huge advantage. You'll find that JavaServer Page source can consist primarily—even wholly—of HTML syntax. So the process of page design is much more natural than writing HTML within a Java servlet source file. And although the majority of JSP pages are designed to produce HTML, you'll see in later chapters how much work has been done to allow JSP pages to create XML output.

JSP pages were intended to be a boon for project management—the web designer could concentrate on the page design (without requiring Java knowledge), and the Java programmer could later insert the dynamic Java elements within the page design. Certainly this approach can work, although it throws up a change control issue, for there will be occasions when both the page designer and the Java programmer want to make changes to the same page at the same time. You'll see through the following chapters that there is more and more emphasis in JSP technology on taking Java out of the page and replacing it with other elements (Expression Language, custom tags) that can—at least in theory—put the dynamic side of page construction in the hands of the nonprogrammer.

The JSP Translation Phase

As it stands, our JSP example above (displaying the date and time) doesn't look much like something that could one day become a Java object. How might this happen? You deploy a JSP page into a JSP container (servlet containers such as Tomcat are invariably JSP containers as well). The JSP container has mechanisms that "translate" the page into something we do recognize as Java code: namely, a servlet. The process is shown in Figure 6-1.

The translation occurs only when necessary, at some point before a JSP page has to serve its first request. Translation doesn't have to happen again—unless the JSP source code is updated and the page redeployed. A JSP container has discretion regarding when the translation occurs. It can occur on demand: as late as the first time a user requests a JSP page. At the other extreme, JSP pages can be translated on installation into a JSP container (a process often referred to as JSP precompilation). Any point in between is possible (though less usual)—for example, when a web application is loaded, a JSP container might choose to translate any new or changed JSP pages.

FIGURE 6-1

The JavaServer Page Translation Phase

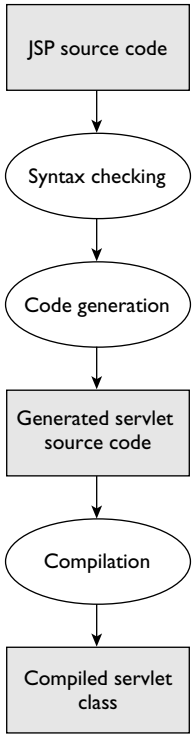


Figure 6-1 shows that there are two outputs within the translation phase. The first is an interim output: a Java source file for a servlet. The second is the compiled class file from the servlet source. The class file is retained for future use, and most JSP containers give you an option whereby you can retain the servlet source for debugging purposes (the default is generally to discard the source within the translation phase). The servlet created isn't just any old servlet—it has some special characteristics. In particular,

- The servlet (or one of its superclasses) must implement the javax.servlet.jsp .HttpJspPage interface, or . . .
- For the tiny minority of non-HTTP, specialist JSP containers, the servlet (or one of its superclasses) must implement the javax.servlet.jsp.JspPage interface.

How does this work? Well, you get an individual servlet based on your Java source—naturally, every JavaServer Page is different. And remember it's the container that provides this, through very clever code generators. Because it's a servlet, the code is likely to inherit from GenericServlet, which incorporates lots of useful servlet behavior. But because JSP servlets have to obey some special rules, a vendor will typically have a specialized JSP base servlet—perhaps extending HttpServlet or GenericServlet, and extended by all generated JSP servlets. In Tomcat, this is called org.apache.jasper.runtime.HttpJspBase. The inheritance tree for the Tomcat-generated servlet from JSP source is shown in Figure 6-2. Although it's "vendor-specific," showing it is instructive. You can see Tomcat makes as much use as possible of existing classes and interfaces in the standard javax.servlet and javax.servlet.http packages.

exam Watch

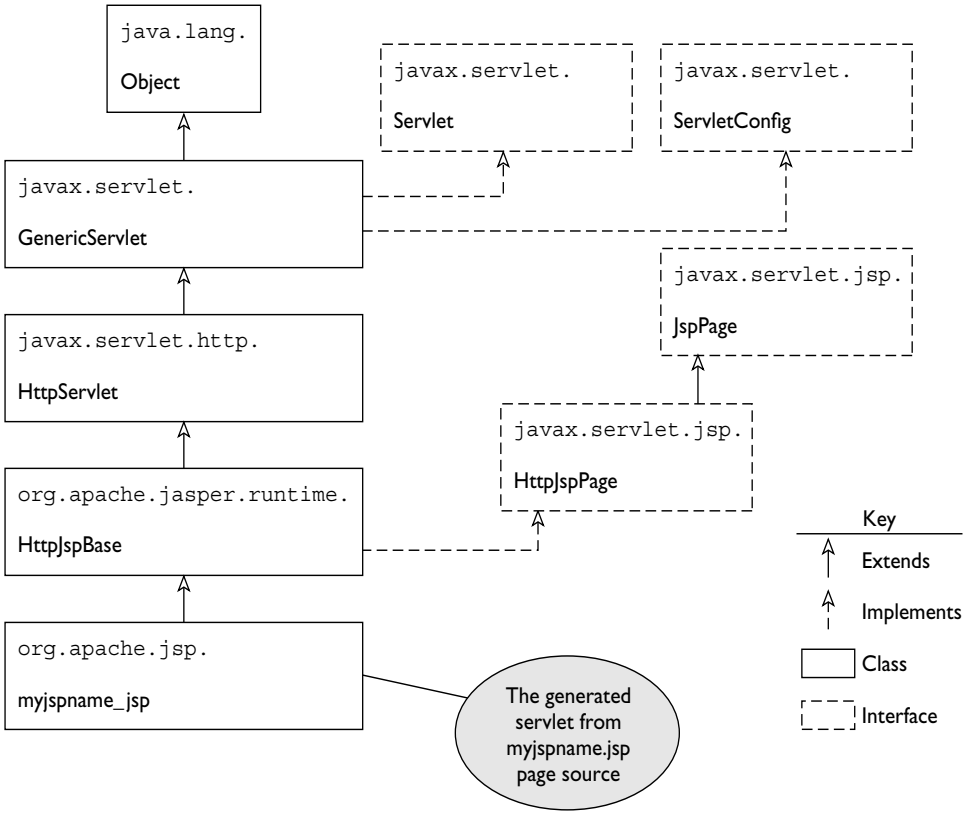
Note that the HttpJspPage interface extends JspPage, so generated JSP servlets (or their superclasses) always implement the defined methods within JspPage.

What did I mean when I mentioned non-HTTP JSP containers? HTTP is by far the main protocol, but the JSP spec is flexible enough to accommodate any request/response protocol you wish to implement. The JspPage interface API documentation says that you have to implement a `_jspService()` method, but it doesn't include the method definition in the interface code. In this way, you are free (as a JSP container designer) to define whatever types you like for the request and response parameters. For the majority of us JSP developers, though, we'll be more than happy that our conventional HTTP containers implement the HttpJspPage interface—which contains the following signature for the `_jspService()` method:

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response);
```

FIGURE 6-2

Class Hierarchy for Generated Servlet from JSP Source (Tomcat)



exam Watch

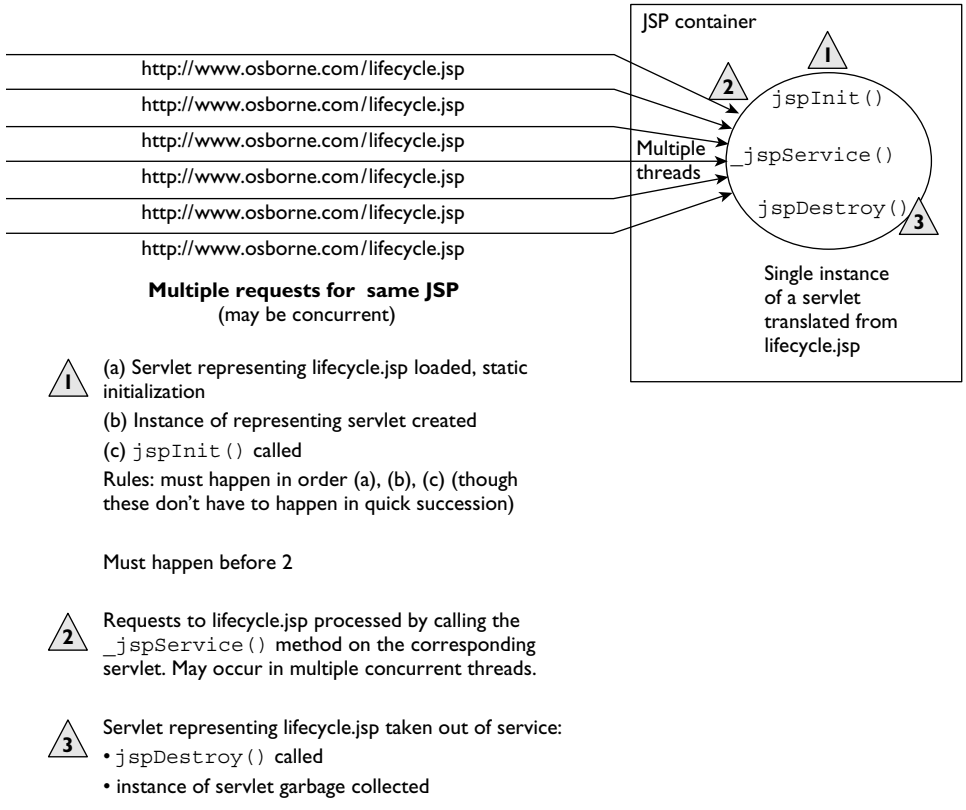
You'll see that the exam objective we're covering talks about translation followed by JSP page compilation as two separate stages. My understanding of the JSP spec tells me that translation* includes *servlet source compilation (as well as syntax checking and servlet source generation). You're unlikely to encounter a question in which this distinction matters—just be aware that "translation" is slightly ambivalent, even within documentation originating from Sun.

As for the exact form the code in your generated servlet takes, that's vendor-specific. The exercise at the end of this section tells you where you might find the Tomcat container's source code.

A final note about translation: If a page fails to translate, an HTTP request for the page should give rise to a 500 (server error) communicated back in the HTTP response.

FIGURE 6-3

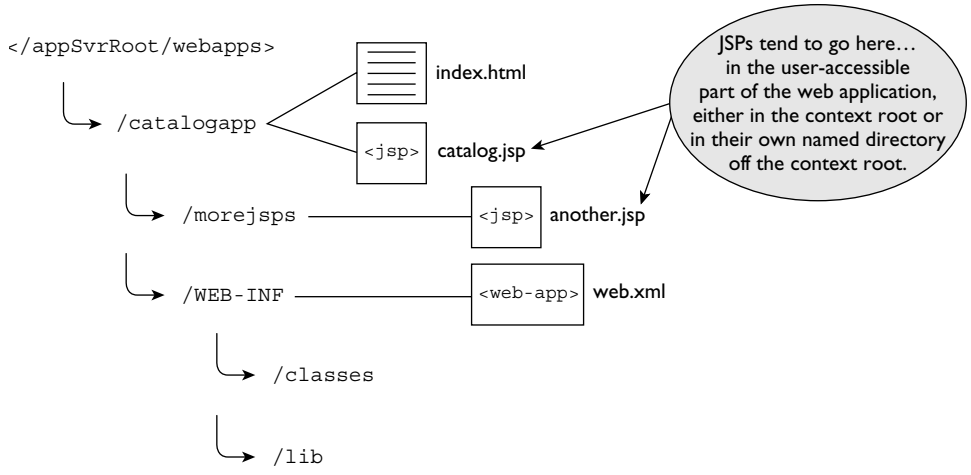
The JavaServer Page Request Phase



The JSP Request/Execution Phase

The request (or execution) phase of a JSP page is remarkably similar to the servlet life cycle. You can remind yourself of this by looking at Figure 1-9 in Chapter 1—then comparing this with Figure 6-3. Of course, because JSP pages are—ultimately—servlets, the request life cycle of a JSP page is controlled from within a servlet's life cycle, and we'll need to discuss how the two relate.

Just as for any other servlet, the JSP page's servlet class is loaded, and an instance of it is created. Generally speaking, there will only be one instance corresponding with a JSP. One of the delights of JSP pages is that they don't have to be registered at all. Just place them in the accessible area of a web application directory structure, as shown in the following illustration.



And, provided they have a file extension (usually .jsp) recognized by your application server, no further work needs to be done. However, you can register a JSP page in the same way as a servlet. You even use the `<servlet>` element, with one vital difference—where the `<servlet-class>` would appear, you substitute `<jsp-file>` instead. Here's part of web.xml file that registers the same JSP twice over, under separate names and with separate mappings:

```
<servlet>
  <servlet-name>JspName1</servlet-name>
  <jsp-file>/instanceCheck.jsp</jsp-file>
</servlet>
<servlet-mapping>
  <servlet-name>JspName1</servlet-name>
  <url-pattern>/jspName1</url-pattern>
</servlet-mapping>
<servlet>
  <servlet-name>JspName2</servlet-name>
  <jsp-file>/instanceCheck.jsp</jsp-file>
</servlet>
<servlet-mapping>
  <servlet-name>JspName2</servlet-name>
  <url-pattern>/jspName2</url-pattern>
</servlet-mapping>
```

So I have a JSP page called "instanceCheck.jsp" installed directly in the root of my web application. Suppose that I install this under Tomcat using a context

directory called examp0601. I have three valid URLs for accessing this JSP. This is the normal way, which ignores any registration details—just use the name of the JSP itself:

```
http://localhost:8080/examp0601/instanceCheck.jsp
```

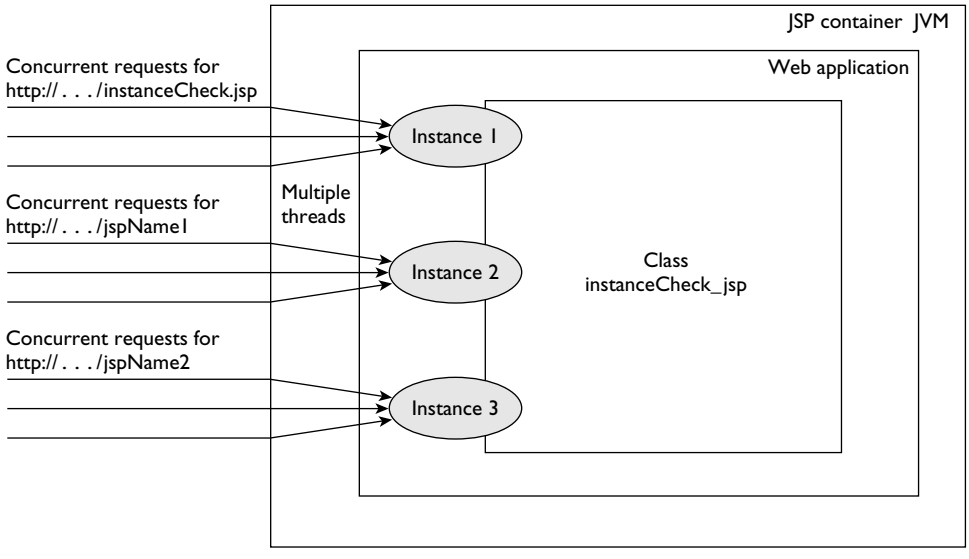
Alternatively, I can use the servlet mapping corresponding to my first registered name for the JSP page, JSPName1:

```
http://localhost:8080/examp0601/jspName1
```

Or, of course, I can use the second mapping:

```
http://localhost:8080/examp0601/jspName2
```

The point to note is that just like servlets (and after all, a JSP page is—ultimately—a servlet), each of these three methods of access establishes three separate instances of the servlet class within the web container. Multiple instances will occur like this only when the JSP page is registered in more than one way. The following illustration shows how this might look inside the JSP container's JVM.



It's worth noting that you can exploit all the other `<servlet>` elements in web.xml for your JSP page. Most usefully, you can set up initialization parameters exactly as you would for a normal servlet.

on the job

If you want to suppress direct access to a JSP (so that users have to go through a registered name and a servlet mapping), locate the JSP page under WEB-INF.

Having established how instances of a JSP's servlet get created, let's return to the life cycle diagram in Figure 6-3. At 1 in the figure, we see that three things happen (very often in succession, but not always). The servlet class is loaded, and any static initialization occurs. Then an instance of the servlet comes into being through normal construction processes. After this, the servlet's `init(ServletConfig config)` method is called by the servlet container (as per normal servlet life cycle rules)—which *must* call the `jspInit()` method. So `jspInit()` becomes a JSP page's equivalent of `init(ServletConfig config)`—it gets called once when the JSP page's servlet instance is created, and never again for that instance. So you are welcome to use this method for any one-off setup processes required for the JSP page.

Then at 2 in Figure 6-3, we see multiple requests being made to the JSP. Under the covers, this calls the generated servlet's `service(request, response)` method, which in turn is obliged to call the method `_jspService()`, passing on the request and response parameters. `_jspService()` constitutes the bulk of your generated JSP servlet. For one thing, all that HTML text in our original JSP source—lines such as

```
<html><head><title>Simple JSP Example</title></head>
```

has to be turned into servlet code equivalent—for example,

```
out.write("<html><head><title>Simple JSP Example</title></head>");
```

The same applies to the dynamic elements of the JSP source as well. So

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

might become

```
out.write(new Date());
```

in the `_jspService()` method.

Multiple threads may access `_jspService()` at the same time, so all the comments about thread safety and the servlet's `service()` method are just as applicable to the `_jspService()` method.

Ultimately, a JSP container will decide to discard your generated servlet instance—the circumstances are the same as for servlets. It could be to save resources, because the web application or entire server is being closed down, or for arbitrary reasons of its own. At this point, the servlet's `destroy()` method is called (nothing new here)—which, because it's a JSP page servlet, *must* call the `jspDestroy()` method. So `jspDestroy()` is called before the instance of the JSP servlet is nullified and garbage collected, giving you the opportunity to reclaim resources in a controlled fashion. The container guarantees that `jspService()` will have completed for all requesting threads on the instance before `jspDestroy()` is called—unless this is overridden by a server-specific timeout.

exam Watch

If you want to harness `jspInit()` and `jspDestroy()` for your own setup and tear-down processes, you can—by overriding either or both in a JSP declaration, which we explore a little later in this chapter. What if you decided to override the servlet equivalents, `init(ServletConfig config)` and `destroy()`? Well, the JSP spec says you can't and mustn't do this. Most JSP container providers prevent this happening by making all Servlet interface methods* final *in the base JSP servlet that they provide—your generated JSP servlet that inherits from this can't possibly override them.

And while on the subject of overriding, you can't override `_jspService()` either. This method represents your page source in Java code form—it's up to the clever container's page generators to worry about the implementation and generation of this method for each individual JSP. It makes no sense for you to override it within the JSP page itself.

EXERCISE 6-1

JSP Life Cycle

This exercise has you write a JSP that documents its own life cycle. We'll also get the JSP to indicate which class your JSP servlet inherits from. You'll meet a couple of concepts documented fully later in the chapter, but don't worry: The steps in the exercise give you enough explanation to follow through without reading ahead.

If you haven't removed any server security settings from the previous chapter, do so now in case they interfere with this and subsequent exercises. The instructions (for Tomcat) can be found with the lab solution at the end of Chapter 5.

For this exercise, create the usual web application directory structure under a directory called ex0601, and proceed with the steps. There's a solution in the CD in the file sourcecode/ch06/ex0601.war—check there if you get stuck.

Create the JSP

1. Create an empty file directly in your newly created context directory, ex0601. Call it lifecycle.jsp.
2. Within lifecycle.jsp, include simple HTML for a complete web page, such as shown below:

```
<html>
<head>
<title>JavaServer Page Lifecycle</title>
</head>
<body bgcolor="#FFFFFF">
<h1>To illustrate JavaServer Page lifecycle</h1>
</body>
</html>
```

3. Now you'll add some Java code. This must be earmarked as a JSP declaration—we learn what that is later in the chapter. Beneath the HTML, type the syntax for opening a JSP declaration:

```
<%!
```

   Then leave some blank lines (this is where you will type Java code, as described in the next steps). Close the declaration after the blank lines with the closing syntax:

```
%>
```

4. As described in the chapter already, your JSP page will be turned into a servlet that extends some base class provided by your application server (and we've been using Tomcat). This base class will implement the interface HttpJspBase, which in turn extends interface JspBase. This means that the

methods `jspInit()` and `jspDestroy()` are available to override. Write the `jspDestroy()` method between the <%! and %> exactly as if it were your Java source editor, and type the Java in just as you would any other method in a normal .java file. Make this method do something to indicate it has fired—`System.out.println("This method is jspDestroy()")` will do fine.

5. Immediately after `jspDestroy()` and before the closing %> marker, write a `jspInit()` method. This should also indicate (through a `System.out.println`) which method has been called. However, also include some code to show the type name of the class generated (a hint if you're not familiar with Java's reflection facilities: `this.getClass().getName()`). Also include some code that shows at least the immediate superclass of this class (another hint: The Class class has a `getSuperClass()` method). Again, make this output visible through `System.out.println()`.
6. Now we'll introduce more Java code, this time as "scriptlet" code. Again, you won't have to wait long for an explanation—the following section of this chapter will furnish you with one. Like declarations, scriptlet code must go between two markers: this time <% at the beginning and (as for declarations) %> at the end. Put in these markers at the end of your lifecycle.jsp file, leaving some blank lines between them.
7. Scriptlet code doesn't include any kind of method declaration, for the resulting generated code is incorporated directly into the `_jspService()` method. Type in code between the <% and %> markers that gives some indication that it is the `_jspService()` method which is being executed. Again, a simple `System.out.println()` statement will suffice.
8. Ensure that you save the file lifecycle.jsp before exiting your text editor.

Deploy and Run the JSP

9. Create and deploy a WAR file that contains lifecycle.jsp to your web server, and start the web server.
10. Use your browser to request lifecycle.jsp using a suitable URL, such as

```
http://localhost:8080/ex0601/lifecycle.jsp
```

11. Refresh your browser a few times to re-request the JSP page. Take a look at your server console window. Here's some (edited) output from the solution code:

```
org.apache.jsp.lifecycle_jsp.jspInit(lifecycle_jsp.java:27)
Class org.apache.jsp.lifecycle_jsp
subclass of org.apache.jasper.runtime.HttpJspBase
  which implements interfaces: interface javax.servlet.jsp.HttpJspPage
subclass of javax.servlet.http.HttpServlet
  which implements interfaces: interface java.io.Serializable
subclass of javax.servlet.GenericServlet
  which implements interfaces: interface javax.servlet.Servlet, javax.servlet.
ServletConfig, java.io.Serializable
subclass of java.lang.Object
org.apache.jsp.lifecycle_jsp._jspService(lifecycle_jsp.java:108)
org.apache.jsp.lifecycle_jsp._jspService(lifecycle_jsp.java:108)
```

12. Your `jspInit()` code should show you the inheritance hierarchy of your servlet, as the solution code does above (the solution code also throws in a bit extra, by showing the interfaces implemented by each class in the hierarchy). You can see from the solution code output above that the generated servlet for the JSP page is called lifecycle_jsp. Its immediate parent is org.apache .jasper.runtime.HttpJspBase—supplied with Tomcat to act as a base class for all generated JSP servlets. Note that this implements the HttpJspPage interface (as per the rules for servlets of this kind). The hierarchy looks more familiar after that. HttpJspBase inherits from HttpServlet, which—as you'll remember from the earlier chapters—inherits from Generic Servlet.
13. You should also see a line of output for each time you refreshed your browser window, to indicate that the `_jspService()` method executed.
14. Causing the `jspDestroy()` method to execute is a little more difficult. You could close your application server altogether, but then you lose your console window! You can probably track down the file containing the log from console output (if your server is configured to save all console output). An alternative—at least in Tomcat—is to remove the context for this exercise but keep the server running. I achieve this by issuing a command to the Tomcat manager application, which should be running by default. The URL to issue the command looks like this, and you enter it as a regular address in your browser:

    ```
    http://localhost:8080/manager/stop?path=/ex0601
    ```

    You may well be prompted to sign on, for the manager application is secured by default. You need to choose a user in the manager role. The list of users (as discussed in Chapter 5) is in the <TOMCAT_INSTALLATION

_DIRECTORY>/conf/tomcat-users.xml. If there isn't a user already present with the manager role ascribed, add a `<user>` entry under an existing `<user>` entry, looking like this:

```
<user username="manager" password="tomcat" roles="manager"/>
```

Restart the Tomcat server, and reissue the stop command. Now sign on as user "manager" with password "tomcat," and you should get this message in your browser:

```
OK—Stopped application at context path /ex0601
```

Check the console. You should see some indication that the `jspDestroy()` method has fired. In the solution code output it looks like this:

```
org.apache.jsp.lifecycle_jsp.jspDestroy(lifecycle_jsp.java:54)
```

Check the Translation Output

15. If you're using Tomcat, the generated servlet Java source and compiled class will, by default, be kept in the following directory:

```
<Tomcat Installation Directory>/work/Catalina/localhost/<context-directory>/
org/apache/jsp
```

16. You are looking for a file called lifecycle_jsp.java. (Tomcat appears to name the generated servlet source file according to the name of the JSP source file, with a _jsp suffix added. The extension becomes .java to denote that this is a Java source file.)
17. Browse lifecycle_jsp.java with a text editor. Find the `jspInit()`, `jspDstroy()`, and `_jspService()` methods. Note how all the `jspInit()` and `jspDestroy()` code comes from what you typed into the JSP page source. Now look at `_jspService()`. You'll see that the code you supplied in the JSP page source is a relatively small percentage of the whole. There's lots of code dedicated to setting up "implicit" objects (we'll come across these at the end of this chapter) and plenty of code that simply writes out the template HTML text to an output stream—just as would happen if you were writing a servlet by hand. Here's an extract:

```
out.write("\n");
out.write("<html>\n");
out.write("<head>\n");
out.write("<title>JavaServer Page Lifecycle</title>\n");
out.write("</head>\n");
out.write("<body bgcolor=\"#FFFFFF\">\n");
out.write("<h1>To illustrate JavaServer Page lifecycle</h1>\n");
out.write("</body>\n");
out.write("</html>\n");
```

18. That's the end of the exercise—you've now seen the JSP life cycle in action and seen firsthand how closely tied it is to the servlet life cycle.

CERTIFICATION OBJECTIVE

JSP Elements (Exam Objective 6.1)

Identify, describe, or write the JSP code for the following elements: (a) template text, (b) scripting elements (comments, directives, declarations, scriptlets, and expressions), (c) standard and custom actions, and (d) expression language elements.

The exam objective above covers almost everything that can legally appear in a JSP page. When JSP technology was first released, it contained only template text and scripting elements—(a) and (b) in the objective in the section head above. This section will focus on these aspects, for they are the most fundamental. Standard and custom actions—(c)—came later, and will have a separate section (and exam objective) of their own in Chapter 7, as will expression language—(d)—which is a powerful but quite recent addition to the JSP repertoire.

Template text is easy to explain: It's any HTML or XML (or indeed any type of content at all) that you care to include in your JSP page. Template text is sent unchanged to response output. This is not so with scripting elements: There are several different sorts, and the JSP container works hard to turn these dynamic elements into something else—usually a coherent string of HTML or XML. Together with the template text, this completes the page response.

These are the topics we investigate in this chapter. In your own work environment, you may find such "traditional" scripting elements deemphasized in

favor of custom tags and expression language. This is even reflected in the exam, which concentrates more on the newer aspects of JSP syntax. However, you must gain a sound knowledge of the older features. The exam still has plenty to say about them (often in combination with the newer JSP features), and you will doubtless be called upon to read, understand, and maintain JSP pages that rely heavily on traditional scripting elements. And although you're encouraged to use the newer features, they are not in any sense deprecated—there are times when these elements are the best or only approach to solving a problem.

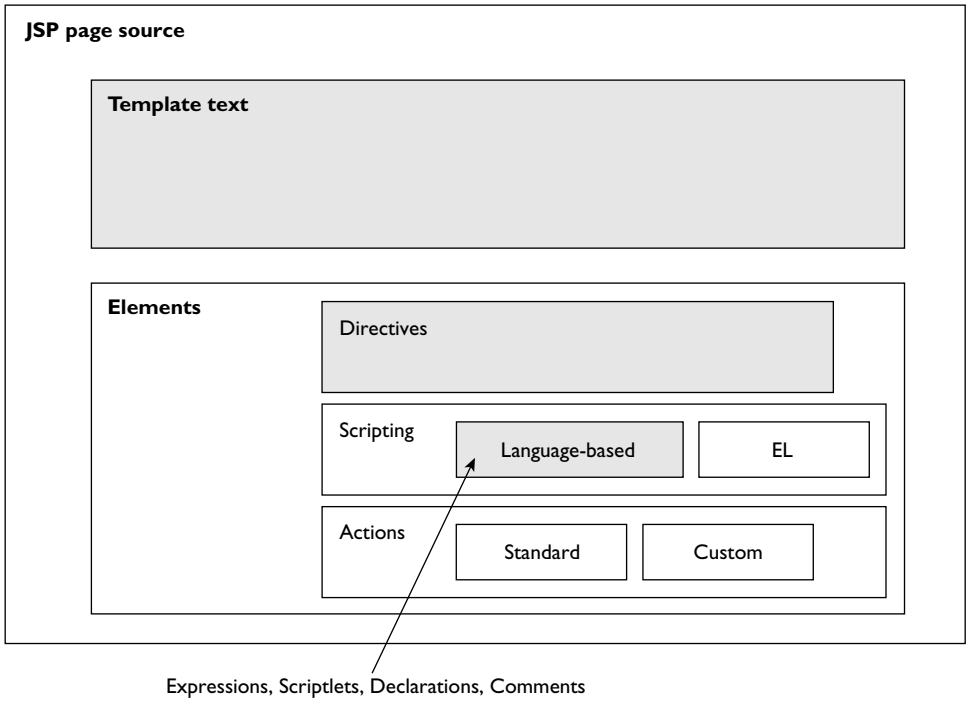
Anatomy of a JSP Page

Before diving into the detail, take a look at Figure 6-4, which breaks down the composition of a JSP page into all its possible constituents.

You can see in Figure 6-4 show a JSP page divides into template text and elements. Elements fall into three types: directive, scripting, and action. There are two forms of scripting (EL or "traditional") and two forms of action element (standard or custom). The lightly shaded areas correspond to the topics covered in this chapter.

FIGURE 6-4

Anatomy of a JSP Page



Template Data

A JSP source page generally starts life as a regular piece of static HTML or XML. As the JSP specification puts it, JSP technology supports the "natural manipulation" of text and XML—you can write it as it would appear in a normal document. And this constitutes the template data in a JSP page—anything that's static: source that the JSP container doesn't have to translate into anything else beyond String parameters to simplistic "out.write()" servlet source code statements.

We saw how this worked at the end of the exercise in the previous section. Here's a further example, this time for XML. This is the opening of some JSP page source, containing only XML template text:

```
<project name="webmodulebuilder" default="deploy" basedir=".">

  <!-- set global properties for this build -->
  <property file="build.properties" />
  <property name="dist" value="../../dist" />
  <property name="web" value="../" />
```

This manifests itself in the generated servlet I'm looking at (from Tomcat) like this:

```
out.write("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n");
out.write("<project basedir=\".\" default=\"deploy\"
name=\"webmodulebuilder\">");
out.write("<property file=\"build.properties\"/>");
out.write("<property value=\"../../dist\" name=\"dist\"/>");
out.write("<property value=\"../\" name=\"web\"/>");
```

So the XML is more or less unchanged—apart from the addition of lots of backslashes in the Java source to escape double quotes. You'll find—particularly with XML—that JSP containers give short shrift to ill-formed documents. We revisit XML generation from JSP page source in Chapter 7.

Having dealt with the constant template aspects, let's turn our attention to the more interesting areas of JSP page source: page elements.

Elements of a JSP Page

In JSP page source, if it's not template text, it must be an element. An element is simply something that can't stand as it is—it needs to be translated. An element is always recognizable through a standard set of characters to mark its beginning and

end. Many elements use XML-style opening and closing tags for this purpose. Other elements use different conventions—here's a scriptlet element:

```
<% System.out.println("A scriptlet"); %>
```

However, the principle is the same—opening characters (`<%`) and closing characters (`%>`).

There are three types of element possible in a JSP page:

1. **Directive elements.** You most often use these to communicate global information to your page, which is independent of any particular request. For example, you might use an appropriate directive for importing classes you need: These will translate to Java source import statement in the generated servlets. We explore directive elements in the next section of this chapter.
2. **Action elements.** These use XML-style tags for the inclusion of dynamic data. You get many as "standard" with JSP containers, but can still build your own "custom" actions. We start exploring actions at the beginning of Chapter 7.
3. **Scripting elements.** There are two kinds. There is a newer, preferred syntax called Expression Language, or EL. We meet this in Chapter 7. Then there is the Java language-based approach. Either way, the purpose is to incorporate dynamic information or execute presentation logic. We explore the "traditional" Java language approach in this section of the chapter.

So let's look at the language-based scripting elements in more detail—all are retained in JSP 2.0, despite the wealth of alternatives. There are four in all:

1. expressions, which exploit Java code to place some output directly in the JSP page. We've met an example already: `<%= new Date() %>`.
2. scriptlets, for more extended pieces of Java code, as long as it's legal Java code that works in the context of the `_jspService()` method. The code doesn't necessarily contribute anything to the page output—for example, you might write a scriptlet like this—`<% System.out.println("in the jspService() method"); %>`—just to log some information to the server console, as we did in the last exercise.
3. declarations, for any piece of Java code that needs to exist in the generated servlet but outside the `_jspService()` method. Declarations usually consist of whole methods, more rarely of instance and class data members (and why

would you use them? A JSP becomes a servlet with its related thread-safety issues—you should be using session or context attributes instead!). Here's a short declaration example: `<%! public void jspInit() { // Do nothing } %>`.

4. comments, to denote any lines you want to be completely ignored by the JSP page translation process. They'll appear in the JSP page source but nowhere else. Example: `<%-- Author: David Bridgewater --%>`.

You can see that all four of these scripting elements have similar syntax—all contain angle brackets, percent signs, and some additional characters as well. Whichever kind you're using, there is a rule that the beginning and end markers (such as `<%` and `%>` for a scriptlet) must appear in the same physical source file. Beyond that, anything goes (well, most things): White space, carriage returns, and tabs are all permissible, just as in normal source code.

With that preliminary survey complete, let's look a little more closely at the niceties of each of these four scripting elements.

Expressions

As we noted before, expressions use the result of evaluating a piece of Java code directly in the page output. You might understand this better if you see how an expression is dealt with in the generated servlet. Let's take the example we've used all along: including a call to the no-argument constructor of the java.util.Date object in the JSP source. The Date object produced by the constructor call represents the current date and time. Here's how it looks when used in the JSP page source:

```
<%= new java.util.Date() %>
```

When this is generated into servlet code, the resulting Java statement will probably look like this:

```
out.print(new java.util.Date());
```

`out` is an instance of a javax.servlet.jsp.JspWriter, and it is associated with the response for the generated servlet. And you can see what happens: The contents of your JSP expression are used directly as the parameter to the `print()` method. You'll see from the API documentation that this is an overloaded method. In this case, we're invoking the version that accepts an object. The object is turned into a String through a call to the `String.valueOf(Object)` method, which ultimately uses the `toString()` method on the object to return a String value. In the case of the

Date object, it will be the current date and time formatted according to the default locale on the server running your JSP container. So the onus on you as the JSP page developer is to ensure that whatever expression you use (and it can be as complex as you like), the result is a String object.

exam Watch

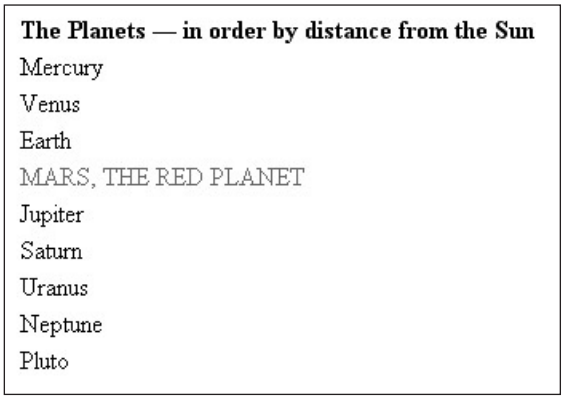
JspWriter's* `print()` *methods are overloaded to accept primitives, so you won't run into difficulties including expressions that evaluate (for example) to an* `int`*. However, you must at all costs avoid* `void`*! If the expression you use is a method with no return value, your JSP page will fail the translation stage.

An expression must begin with <%= and conclude with %>. Whatever Java you place inside the expression, remember not to terminate it with a semicolon! The Java used will be employed as parameter code inside a method call (which is itself inside the `_jspService()` method).

Scriptlets

Scriptlets allow you to include an extended series of Java statements inside the `_jspService()` method that are executed on every request to the page. This time, the Java statements are incorporated "as is," so you must terminate each with a semicolon, unlike expressions. A scriptlet begins with a <% and ends with a %>. Here's an example JSP page, which incorporates four scriptlets and two expressions:

```
<html><head><title>The Planets</title></head><body>
<% /* Scriptlet 1 */
  String[] planets = {"Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn",
"Uranus", "Neptune", "Pluto"};
%>
<table>
  <tr><th><b>The Planets—in order by distance from the Sun</b>  </th></tr>
<% for (int i = 0; i < planets.length; i++) { /* Scriptlet 2 */
    if (i == 3) { // fourth rock from the sun %>
      <tr><td><font color="red"><%=(planets[i] + ", the red planet").toUpper
Case()%></font></td></tr>
<%  } else { /* Scriptlet 3 */ %>
      <tr><td><%= planets[i] %></td></tr>
<%  } /* Scriptlet 4 */
  } %>
</table></body></html>
```



The output from this JSP page is shown in the illustration on the left. You should see that Mars (the fourth planet in the array) shows up in capital letters with some extra description and in a slightly different shade of gray to represent the red writing! Better still, use the electronic version of the book to copy the JSP source into a .jsp file, and deploy it on your server.

Let's analyze what the page source is doing. Scriptlet 1 sets up a String array, containing a list of planets. Some HTML follows, setting up a table with a heading row. Scriptlet 2 begins a loop to iterate through the planet names in the array and a condition to do something different for Mars, while Scriptlet 4—consisting of only two closing braces—ends the condition and the loop. Scriptlet 3 provides the default behavior for all the planets that aren't Mars. Interspersed among the second and third scriptlet we find two pieces of HTML containing expressions—one for normal planets and the other for Mars.

You see here two principal reasons for using scriptlets:

1. To set up data for display (first scriptlet)
2. To control the logic of what is displayed (second, third, and fourth scriptlets)

The template text (such as `<tr><td><font color="red">`) is incorporated as `out.write()` statements in the `_jspService()` method, and expressions (such as `<%= planets[i] %>`) as `out.print()` statements—we've discussed both already. To complete the picture, you must remember that template text, expressions, *and* scriptlets are all translated and generated into `_jspService()` *in order of their appearance in the JSP page source*. The power lies in mixing the three together. So you can dictate—for example—that a particular piece of template text and an expression lie within the body of a "for" loop.

There are some consequences following from this. For one thing, scriptlets don't have to comprise complete blocks of code. In the example, we saw a "for" loop begun in the second scriptlet and ended in the fourth. Indeed, it's mostly undesirable to have scriptlets self-contained, for you can't include expressions and template text within one scriptlet: Within the `<%` and `%>` scriptlet boundaries, you can insert only valid Java source code.

Another consequence: You can declare a local variable in one scriptlet and use that in another scriptlet or expression—as long as that happens at some later point

in the code. So in the example, the first scriptlet declares the local variable *planets* of type String array, then uses this at two points in later expressions. You can imagine (or try out!) what occurs if you move the first scriptlet declaring the String array to the end of the Java page source. Of course, scoping rules apply: If you have a complex nesting structure with your braces, you have to be careful not to create visibility problems for your local variables. The moral is this: Keep it simple!

Before leaving the topic of scriptlets: Note in the example page source that I have been annoyingly inconsistent in my use of the `<%` and `%>` markers. Sometimes these appear in a line to themselves and sometimes on the same line as some Java source code. I have, of course, done this deliberately to make the point—it doesn't matter. The generated servlet code may have a few extra page breaks thrown in to correspond with extra carriage returns, but syntactically it just doesn't matter. However, I would advise choosing or adopting a consistent style to avoid annoying your coworkers.

exam Watch

Any Java code can be placed within scriptlets. Most scriptlet-based questions in the exam are as much a test on Java behavior as they are on JSP syntax. So don't forget all that hard-won SCJP knowledge just yet!

on the job

Of course, because you can use the full power of Java within the JSP page source, the temptation is to do so. There are very real objections to doing this—though I have to say, less real than they used to be. JSP page source was traditionally impossible to debug until the JSP page was deployed and turned into a servlet. Then you would have to find the servlet and debug that—not generally easy, and sometimes impossible if the page translation process didn't result in a legal piece of Java source in the first place. JSP page source editors—again traditionally—had no capability to spot any Java syntax errors you made in your script. However, that position has changed. Page source editors—even free, open source ones (I use a product called Lomboz plugged into the Eclipse IDE)—give instant feedback on compilation problems before you get anywhere near deploying your page. Some commercial editors (IBM Rational Application Developer, for example) even allow real-time debugging of the Java page source: You can place breakpoints in the Java page source and "step through" the page as if it were real Java code.

But with all of that, I would still maintain that if you are writing complex Java code in your JSP page source, there is probably a better place for that code—in a custom tag or even a servlet. Expression Language and JSTL

(coming soon to your JSP page source!—see chapters 7 and 8) can usually provide a more maintainable way of dynamically determining page contents than can Java code. But the choice is still yours—and especially for "quick and dirty" pages to achieve small localized tasks quickly, I can rarely resist the temptation and ease of writing Java source directly into the JSP page.

Declarations

What if you want to place code in the generated servlet outside of the `_jspService()` method? Use a declaration. How do you spot a declaration? It begins with a `<%!` marker. The end marker is still `%>` (the same as it is for expressions and scriptlets).

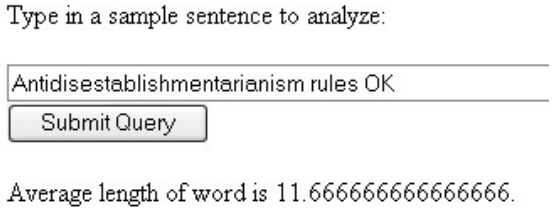
You can place in your declaration any code that can legally appear in a servlet source file: instance methods, static methods, static initialization code, static or instance data members, inner classes — this covers just about everything. You can also use declarations to override some methods that appear further up in the JSP servlet hierarchy — namely `jspInit()` and `jspDestroy()`.

We'll look at an example that analyzes a sentence typed into a simple form, then works out the average length of the words used. Here is the JSP page source:

```
<%
String userInput = (String) request.getParameter("sentence");
if (userInput == null) {
  userInput = "Antidisestablishmentarianism rules OK";
}
%>
<html>
<head><title>Sentence Analyzer</title></head>
<body>
<p>Type in a sample sentence to analyze:</p>
<form method="GET" action="sentenceAnalyzer.jsp">
<input size="80" name="sentence" type="text" value="<%= userInput %>" />
<br />
<input type="submit" />
</form>
<p>Average length of word is <%=avgWordLength(userInput)%>.</p>
</body>
</html>
<%!
private double avgWordLength(String sentence) {
  java.util.StringTokenizer st = new java.util.StringTokenizer(sentence, " ");
  double wordCount = st.countTokens();
  int totalChars = 0;
```

```
  while (st.hasMoreTokens()) {
    totalChars += st.nextToken().length();
  }
  return totalChars / wordCount;
}
%>
```

The output from this JSP page is shown in the following illustration.



You've seen several elements in this page already. The page starts with a scriptlet. This sets up a local variable called *userInput*, whose value is derived from a request parameter called *sentence*. If this parameter is missing (as happens when you first load the page), then local variable *userInput* is loaded with a default value. Some HTML follows, notably a small form with one text field and a button. The text field has a name of *sentence*, providing the request parameter of that name to the JSP. The text field's value is loaded from the local variable *userInput*, which the page user can then overtype as needed.

Beneath the form is an expression. This calls a method, `avgWordLength()`, and passes the method the *userInput* local variable as a parameter. Where is the method defined? Here we come finally to our declaration: at the bottom of the JSP page source. The following shows the beginning of the declaration and the method signature:

```
<%!
private double avgWordLength(String sentence)
```

There's nothing special about the method itself—you have no doubt written dozens like it. The logic parses the sentence, works out the word and character count, and returns a `double` to represent the average number of characters used per word. Because the method call was in an expression, this value is displayed on the web page.

exam Watch

Declaration code has no access to implicit objects; you have to pass these as parameters to the methods you declare. You can feel legitimately ignorant about implicit variables—they haven't yet been introduced—although there is one not very artfully concealed (`request`) at the beginning of the previous code example. Revisit this exam watch when you get to the end of the chapter!

Although I have put the declared method at the end of the example, there's no particular significance to this. All declarations are gathered up by the translation process, and they are placed consecutively in the generated servlet source code.

Comments

You can include two types of comment in your JSP source code. One of these has the following syntax: `<%--` to start the comment and `--%>` to end it. The advantage of this method is that the JSP translation process completely ignores the lines between the comment markers. This is useful as a device during development for temporarily commenting out code, as well as being useful for including comments that should never be reproduced in the web page output. One thing you can't do is to nest one comment of this type within another: As soon as the translation phase reaches the first end marker (`--%>`), which goes with the inner comment, translation recommences, as shown in the following illustration.

```
<%-- <% for (int i = 0; i < 10; i++) {
<%--       if(i==3) System.out.println("i is 3!"); --%>
           System.out.println("i squared is " + i * i);
          } --%>
```

In the code example in the illustration, the grayed-out text is ignored in translation. Translation begins again with the `System.out.println()` statement. Does this cause a compilation error? Actually, no—the translation process sees no marker denoting the beginning of a scriptlet, so it treats the ungrayed Java source as template text—i.e., text that should be directly output in the web page.

However, beware of using this style of commenting within a scriptlet or declaration: It's not allowed. *Only valid Java syntax is allowed within a scriptlet or declaration*—and it doesn't include this style of comment. Of course, you can still use Java's own commenting mechanisms: `//` for single-line comments, `/* ... */` for extended comments, and `/** ... */` for JavaDoc comments. (How or why you would extract JavaDoc comments from a piece of JSP page source or its generated servlet is a different question.)

exam Watch

For the four scripting elements we've talked about—expressions, scriptlets, declarations, and comments—none will nest inside each other or inside themselves. You can't have an expression inside a scriptlet, or a scriptlet inside a declaration, or a scriptlet in a scriptlet—or any other combination.

If you want comments sent within the web page output, you can use regular HTML comment syntax: Open the comment with `<!--` and close with `-->`. Of course, this text is not visible in the displayed web page of most browsers, but it is available if you take your browser's option to view the HTML source code. Why would you do this at all? It can be a useful technique to incorporate debugging or support information. An HTML comment is treated exactly like other HTML template text. So it's perfectly acceptable to include JSP scriptlets, expressions, expression language, or any other legal JavaServer Page syntax within the comment—and it will be processed at translation time.

exam Watch

Here's a summary table that shows the different forms of scripting elements and key facts for each—many exam questions revolve around a grasp of these!

Element Type	Starts with	Ends with	Semicolons on End of Java Source Statements?	Code Generated into the `jspService()` Method?
Expression	`<%=`	`%>`	No	Yes
Scriptlet	`<%`	`%>`	Yes	Yes
Declaration	`<%!`	`%>`	Yes	No
Comment	`<%--`	`--%>`	Not applicable	Not generated at all

EXERCISE 6-2

JSP Elements

Putting this all together, you're now going to write a JSP page that uses scripting elements to work out and display a table that converts a distance in miles to a distance in kilometers. You'll display the results in an HTML table, so the result will look something like that shown in the illustration on the left.

Miles	Kilometers
1	1.6
2	3.2
3	4.8
5	8
10	16
15	24
20	32
50	80
100	160
200	320
500	800

Create the usual web application directory structure under a directory called ex0602, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch06/ex0602.war—check there if you get stuck.

Create the JSP Page Source

1. Create an empty file directly in your newly created context directory, ex0602. Call it milesToKilometers.jsp.
2. At the top of the page source, write a JSP declaration that includes one method with the following signature:

```
private String convert(int miles)
```

   In the method, take the miles value passed as a parameter, and multiply this by a constant 1.6 to obtain a value in kilometers. From the resulting numeric kilometers value, produce a suitably formatted String with a sensible number of decimal places (java.text.DecimalFormat may help you here). You don't have to do the formatting—if you prefer, pass back the double value instead.

3. After this declaration, write a scriptlet that declares a local variable: a primitive **int** array. Load the array with approximately a dozen values to represent values in miles to convert (e.g., 1, 2, 3, 5, 10, 15, 20, . . .).
4. Now write the HTML to display the output. Within the body of the page, declare a table with two heading columns: the first for the miles amount, the second for the converted kilometers amount.
5. Within the table, write a scriptlet to loop through all the elements in the **int** array declared in step 3.

6. Break the scriptlet into two scriptlets—the first for the loop logic and the second for the closing brace.
7. Between these two scriptlets, include HTML to create a table row and two cells. Within the first cell, include an expression to show the miles value—taken straight from an occurrence in your **int** array, this occurrence being the value of your loop counter. Within the second cell, include another expression that calls the `convert()` method and passes in the miles value.

Deploy and Run the JSP

8. Create a WAR file that contains milesToKilometers.jsp, and deploy this to your web server. Start the web server if it is not started already.
9. Use your browser to request milesToKilometers.jsp using a suitable URL, such as

```
http://localhost:8080/ex0602/milesToKilometers.jsp
```

CERTIFICATION OBJECTIVE

JSP Directives (Exam Objective 6.2)

Write JSP code that uses the directives: (a) "page" (with attributes "import," "session," "contentType," and "isELIgnore"), (b) "include," and (c) "taglib."

In the first half of this chapter, you have already delved deeply into the mechanics of JavaServer Pages. The second half of the chapter goes further and shows some features that can make your JSP development tasks more convenient.

This section concentrates on directives, which—like scripting elements—are pieces of JSP tag-like syntax. Like an XML or HTML tag, directives have attributes that dictate their meaning and effect. In almost all cases, the effects produced by directives can't be replicated using expressions, scriptlets, or declarations.

We'll consider the three directives mentioned in the above exam objective in this section: `page`, `include`, and `taglib`.

Directives

The page Directive

You can include a `page` directive anywhere in your JSP page source: beginning, middle, or end. Here's an example of how one looks:

```
<%@ page import="java.util.*" %>
```

The effect that this particular directive achieves is to introduce an **`import`** statement into the source of the generated servlet. We'll discuss the import of *import* very shortly, but for now let's just examine how the directive is made up:

- An opening marker: `<%@`
- The word "`page`," which denotes that this is a *page* directive (as opposed to any other kind—*include* or *taglib*, for example)
- The word "`import`," which is one of the valid attributes for the `page` directive
- An equal sign after the attribute name
- The value of the attribute itself, normally in double quotes
- A closing marker—just like the one for scripting elements: `%>`

exam Watch

Don't be thrown by weird but legal variants for directive syntax. You don't need white space after the opening marker or before the closing marker:

```
<%@page import="java.util.*"%>
```

You can also put extra white space before or after the equal sign for the attribute:

```
<%@ page import = "java.util.*"%>
```

Single quotes are as acceptable as double quotes for attribute values:

```
<%@ page import='java.util.*' %>
```

You're not confined to having only one valid attribute for a directive. For example, there's a (mostly redundant) attribute for the *page* directive called *language*, to denote what kind of scripting language your JSP uses. The only normally valid value is Java, unsurprisingly. You could include this attribute alongside an *import* attribute if you wanted to:

```
<%@ page import="java.util.*" language="Java" %>
```

However, it is common practice to keep one attribute per directive line:

```
<%@ page import="java.util.*" %>
<%@ page language="Java" %>
```

So now that we've seen the syntax for the *page* directive, let's explore some of the valid attributes.

import You use the `import` attribute to create import statements in the generated servlet source produced by the JSP container's translation phase. When we were looking at declarations in the previous section of this chapter, we looked at an example piece of code that used the java.util.StringTokenizer class. Because we hadn't examined the import mechanism for JSP pages at that point, we were forced into some cumbersome source code:

```
<%!
private double avgWordLength(String sentence) {
  java.util.StringTokenizer st = new java.util.StringTokenizer(sentence, " ");
  //... rest of method omitted
```

By including a *page* directive such as the following anywhere in the JSP page source,

```
<%@ page import="java.util.StringTokenizer" %>
```

you can rewrite the source code in a more succinct and normal fashion:

```
<%!
private double avgWordLength(String sentence) {
  StringTokenizer st = new StringTokenizer(sentence, " ");
  //... rest of method omitted
```

If you hunt down the generated servlet source code, you will doubtless find a perfectly normal Java **import** statement:

```
import java.util.StringTokenizer;
```

There are some packages you get for free within the JSP, so it's redundant to import them (although it doesn't matter if you do). There's java.lang, of course: All the classes in that are available to any piece of Java source, generated servlet or otherwise. Then there are these:

- javax.servlet
- javax.servlet.http
- javax.servlet.jsp

If you look closely at the generated servlet source, you'll see that the boilerplate code (i.e., anything you didn't provide in the way of scriptlets, etc.) uses classes that appear in these three packages.

exam Watch

You can't include classes from the default package. Every class name used in a JSP must be qualified (since the advent of the JSP 2.0 specification).

The value for the `import` attribute can be any of the following:

- A fully qualified class name
- A generic package name (e.g., java.util.*)
- A comma-separated list of either of the above (you can mix and match as needed)

This is the only attribute of the *page* directive that can be specified more than once—either across separate *page* directives that contain *import* once or even (silly as it is) using the *import* attribute more than once in the same *page* directive.

`session` The *session* attribute of the *page* directive is used to determine whether an HttpSession object is available within your JSP page source (if available, it's provided through an implicit variable called—surprise, surprise—*session*, which we explore in the next section of this chapter). If you leave this directive out altogether, the session is available—or you can explicitly say

```
<%@ page session="true" %>
```

This will have the equivalent effect of writing the following servlet code:

```
HttpSession session = request.getSession();
```

Truth to tell, the mechanism for getting hold of the session is usually a little more convoluted in generated JSP servlet source. The reason for using this directive is to eliminate the time spent on creating or obtaining an HttpSession object, in which case you write the directive as follows:

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

If your JSP page genuinely doesn't need access to the session (though most will), there's a small performance gain to be made.

Valid values for the session attribute are "true" or "false"—like **boolean** literals in plain Java source. Unlike **boolean** literals, however, these values are case insensitive (so "TRUE" and "FaLsE" are also valid values).

`contentType` In the "Responses" section of Chapter 1, we encountered the `ServletResponse.getContentType(String type)` method. Now you're about to learn the JSP way of achieving the equivalent of this method. By way of reminder, the *type* parameter into this method is a String that specifies the MIME type of the response to be sent back.

Here is the solution code from the ImageLoader servlet in Exercise 1-4, reworked as a JSP. The old servlet and the new JSP simply take an existing .gif from the web application and write this to the response's output stream. Before doing so, both servlet and JSP set the appropriate MIME type for the response, namely "image/gif."

```
<%@ page contentType="image/gif" %>
<%@ page import="java.io.*" %>
<% /* response.setContentType("image/gif"); */
String path = getServletContext().getRealPath("tomcat.gif");
File imageFile = new File(path);
long length = imageFile.length();
response.setContentLength((int) length);
OutputStream os = response.getOutputStream();
BufferedInputStream bis =
  new BufferedInputStream(new FileInputStream(imageFile));
int info;
while ((info = bis.read()) > -1) {
  os.write(info);
}
os.flush(); %>
```

The development of this page source was very simple. I took the code directly from the `doGet()` method of the servlet and pasted this as a scriptlet into the JSP page source. I added a *page* directive to import the java.io package. I also added the *page* directive to set the content type to illustrate what we're talking about here. However, this is a case where scriptlet code works just as well as the *page* directive—the commented-out line of scriptlet code

```
/* response.setContentType ("image/gif"); */
```

is the equivalent of the `page` directive

```
<%@ page contentType="image/gif" %>
```

That's almost true, anyway. You could remove the content-type `page` directive and comment the line of source code back in. What you will then observe in the generated servlet source—near the beginning of the `_jspService()` method—are the following lines of Java:

```
response.setContentType("text/html");
// A few other lines of boiler-plate code, omitted...
response.setContentType("image/gif");
```

The second occurrence of `response.setContentType()` comes about because of your commented-in scriptlet code. The first occurrence comes about because even if you omit the `page` directive for attribute contentType, the JSP container is bound to set a default MIME-type of "text/html." Because no content has been committed, this doesn't matter—the setting of "image/gif" comes later, so it takes precedence—but the moral is that you should use the `page` directive to do a proper job of content-type setting.

`isELIgnored` The "EL" in this attribute name stands for "Expression Language." This is something we look at properly in Chapter 7. However, there's no harm in giving you a small foretaste here. As you have noted a few times now, the scripting elements we have discussed so far—while not being deprecated—have fallen out of favor. There are better options available, and Expression Language is one of those options. Suppose you wanted to manipulate a request attribute that is set up as follows:

```
<% request.setAttribute("squareIt", new Integer(7)); %>
```

To display the square of this number, you could use a scriptlet/expression combination like this later in the JSP page source:

```
<% int i = ((Integer)
request.getAttribute("squareIt")).intValue(); %> <%= i * i %>
```

Using Expression Language, you could achieve the same result like this:

```
${squareIt * squareIt}
```

Whichever approach you use (or prefer), the result is the same: 49 is output on the page.

At the moment, we'll ignore the "magic" that makes EL work. The important thing to note is that the JSP translation phase has a new piece of syntax to cope with: the ${ to begin the EL expression, and the matching } to end it. In the past, these symbols would have been rendered as template text in your JSP page source. It could be that this is still the effect you want—in other words, you want `${squareIt * squareIt}` to appear as just that string of characters instead of the number 49. After all, if `${` appears in pre-JSP 2.0 source (i.e., JSP 1.2 or before), it's unlikely to be intended as EL.

So you can switch off EL evaluation and have the text be treated as template text. There are several ways to do this (a topic we revisit in Chapter 7), but we're only interested in one of those here. If in your JSP page source you have the directive:

```
<%@ page isELIgnored="true" %>
```

then any EL expressions remain unevaluated, and they appear as template text.

The other valid value is "false." You might think this is the default—in other words, this is what you get if you leave out the directive. However, the situation isn't quite that simple. Suppose that you stick an (old) JSP 1.2 application into a JSP 2.0 container. Chances are that the following is true:

- Your web.xml file is at version 2.3.
- You don't want to be bothered including the `isELIgnored` attribute separately in all your JSP page sources.

To deal with this, the JSP 2.0 sets a default of `isELIgnored="true"` so that any occurrences of ${ in your old application will be correctly treated as template text. Only if the deployment descriptor web.xml is at the up-to-date version level of 2.4 is the default of `isELIgnored` set to "false," for an up-to-date web.xml implies an application that is EL-aware.

Besides this directive, there are other ways to control EL-awareness—we'll complete the picture in Chapter 7.

There are several other attributes of the `page` directive that are not on the exam syllabus as such. Following is a list with brief definitions so that you're not thrown when you encounter these attributes in real life.

language	<%@ page language="Java" %> Denotes the scripting language. "Java" is the only value supported by all J2EE-compliant containers. Your container might support something different and specialized, but your page won't in all likelihood be portable to other containers then.
extends	<%@ page extends="com.osborne.webcert.MyBaseJSPServlet" %> If you want to override the base servlet that your container provides when generating servlets, you can—and substitute your own through this directive. Do so at your own peril; your container may not like you for it.
buffer, autoFlush	<%@ page buffer="none" autoFlush="true" %> You can use these attributes to control whether or not you have a buffer (you can specify a size in kilobytes), and how this buffer is flushed. The example above causes the response to be flushed as soon as it is generated.
isThreadSafe	<%@ page isThreadSafe="true" %> Causes the generated JSP servlet to implement the deprecated SingleThreadModel interface: not recommended. The default is, of course, false.
info	<%@ page info="My Clever Hacks Page" %> Use this attribute to publish information about your JSP page, accessible through the `getServletInfo()` method.
errorPage, isErrorPage	<%@ page errorPage="errorPage.jsp" %> <%@ page isErrorPage="true" %> Use errorPage to set a URL pointing to another JSP within the same web application. Should an exception occur, your users will be forwarded to this other JSP. The page you forward to must have "isErrorPage" set to true, and is the only sort of page to have access to the implicit variable exception.

The `include` Directive

We're done with the *page* directive; now let's look at the *include* directive. The good news is that there isn't much to this directive: the directive name itself (include) and one mandatory attribute (file). The purpose of *include* is to merge the contents of one JSP source file into another (the one doing the including). This happens at the point where the including JSP page source goes through *translation*. It's not a dynamic operation happening at run time!

A piece of terminology: A JSP source file and the pages it includes through the `include` directive are collectively referred to as a "translation unit." The file type you include doesn't have to be JSP page source, nor does it have to have a .jsp

extension. Once included, the file contents must make sense to the JSP translation process, but this gives scope to include entire HTML or XML documents, or document fragments.

The value you can attach to the file attribute is a filename together with its path. The path is relative—either beginning with a forward slash: "/" or not. A path beginning with a forward slash starts at the servlet context directory. Paths without the initial forward slash are determined relative to the location of the page doing the including, and they cannot stray beyond the current context. This obeys the same rules as the `ServletRequest.getRequestDispatcher(String path)` method, which you met in Chapter 3.

Here's an example without the initial forward slash:

```
<%@ include file="stubs/header.html" %>
```

Suppose that this directive is found in a JSP located in directory ex0603/jsps . ex0603 is the context directory. The JSP translator will expect to find the following file and path:

```
ex0603/jsps/stubs/header.html
```

To include the same file relative to the context root, the *include* directive would look like this:

```
<%@ include file="/jsps/stubs/header.html" %>
```

Don't spread complex logic across different included files—or any logic at all, if you can help it. Doing so can land you in real trouble. Especially consider the prospect of changing the logic in the included page—but translation not picking this up. These days, most containers will reperform translation if either the including or included file is updated. However, this is not mandated by the JSP specification, even though it states a preference for that behavior. Imagine debugging a problem where you think an updated file has been re-included into a top-level JSP page. Imagine then your frustration when half a day later you discover that the page is in fact executing the old logic, based on the old version of the file. Problems of this sort can be utterly baffling: Plan to avoid them!

At the beginning of Chapter 7, you'll come across a different mechanism for inclusion: the `<jsp:include>` standard action. The key difference between this

and the `include` directive is that `<jsp:include>` executes afresh with every new request to the including JavaServer Page. Don't forget that the `include` directive discussed here happens at *translation* time. You are almost certain to get an exam question that relies on your knowledge of the difference between these two—but we'll save that piece of cruelty for Chapter 7!

The `taglib` Directive

The *taglib* directive makes custom actions available in the JSP page by referencing a tag library. We don't meet tag libraries until Chapter 8 and custom actions not until Chapter 9, so this may not make much sense at this point. We'll just briefly touch on the syntax of the directive at this point—you'll get plenty of practice with it later. There are two mandatory attributes, *prefix* and *uri*. Every custom action you include in your JSP page must use a prefix specified in one of the *taglib* directives in the page. *uri* gives an indication where the tag library file (which defines the custom actions) can be found. The exact retrieval mechanism will be explained later. So here's an example of the directive:

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

EXERCISE 6-3

JSP Directives

This exercise demonstrates a moderately useful application for the *include* directive: providing a standard header and footer for your HTML JSP pages. Along the way, we'll practice with a couple of the *page* directive attributes and do a little bit of preparatory work for the next section of the book, which talks about implicit variables.

Create the usual web application directory structure under a directory called ex0603, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch06/ex0603.war—check there if you get stuck.

Create JSP Page Source for a Standard Header

1. Create an empty file directly in your newly created context directory, ex0603, called header.jsp.
2. In this file, include HTML tags for the start of the document, the start and end of the head section, and the beginning of the body section.

3. Include an HTML title tag within the head section to display some text saying "Exercise 6-3 on " and follow this with an expression showing the date and time. If you use java.util.Date to assist you with this, then you will want a *page* directive to import this class.

Create JSP Page Source for a Standard Footer

4. Create an empty file in ex0603 called footer.jsp.
5. In this file, put in the end of body and end of document HTML tags.
6. Before the end of body, put in a copyright notice. Derive the text for the copyright notice from a context-wide initialization parameter. If you need a refresher on how to set one of these up in web.xml, check out the beginning of Chapter 3. A programming hint for accessing the initialization parameter in your JSP: You'll need to make use of the *application* implicit variable inside an expression scripting element. *application* is an instance variable referencing the ServletContext object for this web application, so it has access to the `getInitParameter()` method.

Create JSP Page Source for a Setup Page

7. Create an empty file in ex0603 called setup.jsp.
8. Use a *page* directive to import the java.util package.
9. In a scriptlet, create a list of things (the solution code uses a TreeMap containing a short list of countries and capitals). Set the object representing the list of things as an attribute of the ServletContext (use the *application* implicit object).

Create the JSP Page Source for a Master Page

10. Create an empty file in ex0603 called master.jsp.
11. Use a *page* directive to import the java.util.package.
12. Use two *include* directives to include header.jsp, followed by setup.jsp.
13. Use a combination of scriptlets and expressions to do the following:
    - retrieve the list you stored in an application attribute in setup.jsp (again, you'll want the *application* implicit variable).
    - display each item (or set of items) in the list as a table row.
14. Conclude the page by including footer.jsp.

Deploy and Run the Master Page

15. Create a WAR file that contains the full directory structure for the exercise context. You've created four JSP files and have amended the deployment descriptor web.xml, so all these resources must be present in the WAR file. Start the web server if it has not started already.
16. Use your browser to request master.jsp using a suitable URL, such as

```
http://localhost:8080/ex0603/master.jsp
```

17. The solution code output is shown in the illustrationon the left—yours may look a little different.

CERTIFICATION OBJECTIVE

JSP Implicit Objects (Exam Objective 6.5)

Given a design goal, write JSP code using the appropriate implicit objects: (a) request, (b) response, (c) out, (d) session, (e) config, (f) application, (g) page, (h) pageContext, and (i) exception.

To finish this chapter, we're going to look at another convenience provided by JSP technology: implicit objects. A couple of these have been introduced by stealth already in examples and exercises in the chapter so far. There are nine you need to know in all.

However, the good news is that with only two exceptions, these implicit objects are variables *that reference classes you have learned about already*. So your learning burden is lighter than you might otherwise presume when presented with the raw exam objective.

JSP Implicit Objects

The best way to get a handle (no pun intended) on implicit objects is to take a look at the source for any generated servlet. You can use any JSP page source as the basis for the generated code, even one that does nothing—it's the boilerplate code at the beginning of the `_jspService()` method that is of interest. Here's how that code looks for the version of Tomcat used in preparation of this book:

```
public void _jspService(HttpServletRequest request,
  HttpServletResponse response)
  throws java.io.IOException, ServletException {
JspFactory _jspxFactory = null;
PageContext pageContext = null;
HttpSession session = null;
ServletContext application = null;
ServletConfig config = null;
JspWriter out = null;
Object page = this;
JspWriter _jspx_out = null;
PageContext _jspx_page_context = null;
try {
  _jspxFactory = JspFactory.getDefaultFactory();
  response.setContentType("text/html");
  pageContext = _jspxFactory.getPageContext(this,
    request, response, null, true, 8192, true);
  _jspx_page_context = pageContext;
  application = pageContext.getServletContext();
  config = pageContext.getServletConfig();
  session = pageContext.getSession();
  out = pageContext.getOut();
  //... etc. rest of jspService() method
```

So there's nothing very taxing about implicit objects—they are just local variables declared at the outset of the `_jspService()` method. Because they have standardized names, they are available for use within your expressions and scriptlets. By the end of this method opening, we have seen eight out of the nine implicit objects. (The ninth—*exception*—is the exception: We'll see how this implicit object gets declared and initialized later.)

Two are parameters to the `_jspService()` method. In an HTTP context (i.e., most of the time), these represent HttpServletRequest and HttpServletResponse objects. Most of the remainder consists of variables with class or interface types known to us already, in familiar packages. The following table lists all nine, with their types:

Implicit Object Name	Type
request	javax.servlet.http.HttpServletRequest interface (rarely—javax.servlet.ServletRequest)
response	javax.servlet.http.HttpServletRequest interface (rarely—javax.servlet.ServletResponse)
application	javax.servlet.ServletContext interface
config	javax.servlet.ServletConfig interface
session	javax.servlet.http.HttpSession interface
out	javax.servlet.jsp.JspWriter abstract class
pageContext	javax.servlet.jsp.PageContext abstract class
page	java.lang.Object class
exception	java.lang.Throwable class

The only really new concepts are contained in the *out* and *pageContext* implicit objects, whose types live in a package we haven't had cause yet to examine: `javax.servlet.jsp`. *out* is the equivalent of the PrintWriter you get from the response in normal servlets. It's not a PrintWriter, but rather another kind of Writer: javax.servlet.jsp JspWriter. *pageContext* is entirely new: a master controlling object for JSP pages. You can see from the tail end of the `_jspService()` source extract above that this object plays a vital role in initializing several of the other implicit objects.

Hoping that this overview convinces you that there isn't much new to take on board, let's look at each implicit object in a little more detail.

request and response

Because you don't control the `_jspService()` method declaration, you need a guarantee that the request and response parameters passed in will always be called by a consistent name. The most obvious names have been guaranteed for you: *request* and *response*. So wherever your scriptlet or expression code appears, you can make reference to the methods on request and response. So, for example, the expression

```
<%= request.getMethod() %>
```

would display the HTTP method (generally GET or POST) used by the request for the JSP. The following scriptlet code, using the *response* implicit object, would return an HTTP 500 error from a JSP page:

```
<% response.sendError
  HttpServletResponse.SC_INTERNAL_SERVER_ERROR); %>
```

You can't dictate exactly what class the request and response objects are going to be, but you will know that 99.99% of the time that class will implement the javax .servlet.http.HttpServletRequest or javax.servlet.http.HttpServletResponse interface as appropriate. The remaining 0.01% will be taken up with JSP containers that don't implement the HTTP protocol. The *request* and *response* implicit variables are still guaranteed to be there, provided the container is JSP-spec compliant. However, the guarantee is that the objects they represent will implement the javax.servlet .ServletRequest or javax.servlet.ServletResponse interfaces (so a subset of their HTTP equivalents).

application

The *application* implicit variable is an object implementing the `javax.servlet .ServletContext` interface. With it, you can save yourself the bother of defining your own context variable using scriptlet code like this:

```
<% ServletContext context = this.getServletContext(); %>
```

You can use *application* to do all the things we saw in the servlet code from Chapter 3. The following expression could be used to display the name of your web application (as defined in the `<display>` element of the deployment descriptor, web.xml):

```
<%= application.getServletContextName() %>
```

session

The *session* implicit variable is an object implementing the `javax.servlet.http.HttpSession` interface. As we saw with `application`, it's a labor-saving device. You can still get hold of the session with a scriptlet line such as

```
<% HttpSession mySession = request.getSession(); %>
```

but there is no need to, because it's provided already. As we discussed in the previous section of the chapter, you can make the *session* implicit object *un*available using the following directive:

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

But by default (or with an explicit *page* directive where the *session* attribute is set to "true"), it will be present in the `_jspService()` method.

As for straight servlets, the most common use of the *session* object is to store and retrieve attributes. You can also control the session status using the same methods as before. So a scriptlet like this:

```
<% session.invalidate(); %>
```

will invalidate your session, as you would expect.

exam Watch

There is one other circumstance in which you don't get a* session *implicit object. That is in the shady hinterland of JSP containers that don't operate with the HTTP protocol.* session *is an HTTP-protocol specific concept.

page

This is a reference to the JSP page object itself—in other words, the generated servlet. You can see from the generated servlet source that began this section on implicit objects that it's even set from the Java implicit variable **this.**

Somewhat surprisingly, the type of the variable is java.lang.Object. So in itself it is pretty useless—you can call Object methods only on `page`—nothing servlet-specific! You could safely downcast the reference within the Tomcat container like this:

```
<%!
public String getServletInfo() {
  return "My Downcast Page :-(";
}
```

```
%>
<% HttpServlet servlet = (HttpServlet) page; %>
<%= servlet.getServletInfo() %>
```

It might not be safe to cast *page* to an HttpServlet reference variable in all JSP container environments: There's always non-HTTP JSP containers to consider. Even though nearly every servlet in the world derives from GenericServlet (even non-HTTP ones), you couldn't safely cast to this either—as a JSP container author, you could still write a set of JSP-implementing servlet classes from scratch. Consequently, *java.lang.Object* is the only safe choice for the page reference variable!

Not that this causes any practical difficulty. You could equally well substitute the following for the last two lines of the above mini-JSP:

```
<%= this.getServletInfo() %>
```

Or, of course, just:

```
<%= getServletInfo() %>
```

In summary, *page* is not—as most JSP books end up by telling you—terribly useful to page authors.

config

While on the subject of not very useful implicit objects, let's consider *config* next. You may never have to use it. The only practical reason for using the *config* object, which implements javax.servlet.ServletConfig, is to get hold of initialization parameters associated with the JSP page. If there are any, they will be in the deployment descriptor, as `<init-param>` elements associated with the `<servlet>` element for this JSP page.

Howerer, in a JSP, you can already inside an instance of a servlet--which is very likely to implement the SerletConfig interface, So under most circumstances your can use the method `getInitParameter()` directly in a scriptlet. If you are feeling ultracautionus, use `config.getInitParameter()` insted.

pageContext

Now to the *pageContext* implicit object. This will be your JSP container's implementation of the javax.servlet.jsp.PageContext abstract class, which in turn inherits from the java.servlet.jsp.JspContext class. The PageContext class is only really exciting to developers who implement JSP containers, not so much page

authors. For example, *pageContext* provides a mechanism for handling exceptions on a page, and forwarding to another page. The workings of this mechanism can be seen in the generated servlet code, but as a page author and exam taker, you need only to know how to take advantage of the mechanism (more on this when we examine the `exception` implicit object).

The `pageContext` object knows about all the other implicit objects, and it has methods to get hold of them. Again, as a page author, you don't usually bother with these, for the implicit variable names are available for use directly. However, you can see several of these methods in action—`getServletContext()`, `getServletConfig()`, `getSession()`, and `getOut()`—in the generated servlet source that begins this section.

Most usefully, `pageContext` provides a new scope for attributes: unsurprisingly called page scope. You've encountered three sorts of attributes already, here listed from most local to most global:

- Request attributes
- Session attributes
- Context attributes

Page scope attributes are more local still—confined as they are to the JSP page. You'll already have realized that *request*, *session*, and *application* implicit objects have the requisite attribute methods that go with their types (`HttpServletRequest`, `HttpSession`, and `ServletContext`). The *pageContext* implicit object goes further. As you would expect, it has methods to access attributes in its own page scope. However, *pageContext* also gives access to attributes in any scope, using slightly modified versions of the `get`, `set`, and `removeAttribute()` methods. Table 6-1 summarizes the operation of those methods.

exam Watch

Get clear in your mind which methods belong to which class. The `pageContext` object inherits from PageContext, which inherits from JspContext. The attribute methods actually reside in JspContext. And while you're checking out the API documentation for these two classes (javax.servlet.jsp package), do take a look at the exceptions that can be thrown by `pageContext` attribute methods—don't take for granted that they are exactly the same as attribute methods you have met before.

TABLE 6-1

Some of the `pageContext` Attribute Methods

Method Signature	Explanation

Object getAttribute (String name, int scope)	This is like the other `getAttribute()` methods you have encountered—you supply a named attribute, and get back an object in return. The difference is the addition of a second parameter, scope. The allowed scopes are defined as **public**, **final**, **static** constants on the PageContext class:
	`PageContext.PAGE_SCOPE`
	`PageContext.REQUEST_SCOPE`
	`PageContext.SESSION_SCOPE`
	`PageContext.APPLICATION_SCOPE`
	The method will search only the scope supplied as a parameter. If the named attribute isn't found, **null** is returned (there's no other penalty, such as an exception).

void setAttribute (String name, Object value, int scope)	Again like other `setAttribute()` methods, in that you pass in a String name and an Object value. Other things in common: ■ If you pass in a null Object for the value, this has the same effect as calling the equivalent `removeAttribute()` method. ■ Attribute names are unique, so if a name already exists (*in a given scope, of course*), a later value will overwrite the earlier value. The only difference is that you must pass the scope as well, which uses the same set of PageContext constants as before.
void removeAttribute (String name, int scope)	Removes the named attribute in the given scope. Has no effect if no attribute of this name exists in the given scope.
Object findAttribute (String name)	This is the most interesting method. You merely pass in an attribute name, and the method works through the scopes in order: page, request, session, and application. The first attribute found is returned; consequently, an attribute in a more local scope (such as page) will "shield" one of the same name in a less local scope (such as session), so you can use this method only to get the most local attribute value when names are duplicated across scopes. For this among other reasons, I try to keep my attribute names unique across scopes unless there is a very good reason to do otherwise. If no attribute of the supplied name is found in any scope, null is returned—as is the case for `getAttribute()`.
get, set, remove Attribute() without the scope parameter. . .	`get`, `set`, and `removeAttribute()` are overloaded methods. The signatures *without* the scope parameter specifically target *page* scope and nowhere else.

out

The *out* implicit variable represents a writer associated with the response for your JSP page. You would automatically think this is the same as the PrintWriter you can get directly from the response object, but you would be wrong. Instead, `out` is a javax.servlet.jsp.JspWriter. The main thing that JspWriter gives you is buffering: The content in your page (template text, expressions, whatever) isn't—at least by default—committed to the response straightaway. What advantages does this confer? As we learned earlier, there are things you can't do once output has been committed, such as setting response headers. So to absolve you from having to take exceptional care—having to set all your response headers before even the tiniest fragment of template text appears in your JSP page—the JspWriter is buffered.

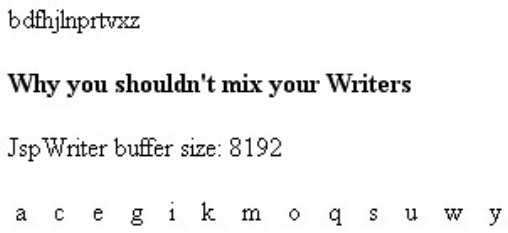
You can control the buffering through page directives, but that's a bit beyond the scope of the exam. For amusement and instruction, though, let's consider what happens if you try to mix and match use of the `out` implicit object and the regular PrintWriter from the response. Here's a complete piece of JSP page source (use the electronic version of the book to copy and paste into a file editor and deploy the JSP):

```
<%@ page import="java.io.PrintWriter" %>
<html><head><title>Wacky Alphabet</title></head>
<body><h4>Why you shouldn't mix your Writers</h4>
<p>JspWriter buffer size: <%= out.getBufferSize() %></p>
<% PrintWriter direct = response.getWriter();
char[] alphabet = {'a','b','c','d','e','f','g','h',
'i','j','k','l','m','n','o','p','q','r','s','t','u',
'v','w','x','y','z'};
for (int i = 0; i < alphabet.length; i++) { %>
 <% out.print(alphabet[i]); %>
<%i++;%>
 <% direct.print(alphabet[i]);
}// end loop%>
</body></html>
```

Here are the main things the JSP does:

- Uses the *out* implicit variable to display the buffer size—just to prove that there is a buffer involved.
- Gets hold of the response's PrintWriter into a local variable called *direct*.
- Puts the letters of the alphabet into a char array called *alphabet*.
- Loops around the array, printing all the letters of the alphabet with spaces in between, alternately using the implicit object *out* followed by the forbidden PrintWriter *direct*.

The page compiles and runs just fine, but the output is surprising, as shown in the following illustration.



As you can see, every other letter (the ones written using the forbidden PrintWriter) has appeared at the very beginning of the web page! This is because PrintWriter is unbuffered, so the response output is sent to the page directly. The buffered JspWriter output is appended to the end once the page is complete.

The other thing to be aware of is that every time you use an expression, or template text, the generated servlet uses the heavily overloaded `out.print()` method—so even though you may make no explicit use of *out*, chances are that it's still the most often used of all the implicit objects.

exception

The final implicit object to consider is *exception*, whose type is java.lang.Throwable—the parent of the Exception class hierarchy in Java. This is present only in the generated servlet for a JSP page designated as an error page, meaning that it contains the following directive:

```
<%@ page isErrorPage="true" %>
```

Furthermore, you never call such a page directly. Instead, you include another directive in all those JSP pages that might give rise to an error. This is in the form:

```
<%@ page errorPage="/jsps/myErrorPage.jsp" %>
```

The value for the errorPage attribute is the name of a JSP page where *isErrorPage* is set to true, and can give a path to that page. The path can begin with a forward slash, in which case it is relative to the context root.

This gives you more control over the presentation of errors to the user. You can make as much or as little use of the *exception* implicit object as you wish. The following error page uses the *exception* object to print the main error message in a

visible way, and again to print the stack trace. However, the stack trace is confined to an HTML comment. In this way, the error may appear not so scary for an application user, but support staff can still view the HTML source of the output for details.

```
<%@ page isErrorPage="true" %>
<%@ page import="java.io.*" %>
<html><head><title>Error Page</title></head>
<body><h1>You're here because an error occurred</h1>
<p>The main error message is: <%= exception.getMessage() %></p>
<p>To see the error message detail, view the source of this web page.</p>
<!--<% exception.printStackTrace(new PrintWriter(out)); %>--!>
</body></html>
```

EXERCISE 6-4

JSP Implicit Objects

In this exercise, we'll use Java reflection techniques to investigate all the implicit objects and display information about them on a web page. You'll write a Java class called Reflector to do most of the hard work, and separately a JSP page called implicitObjects.jsp to display the results.

Create the usual web application directory structure under a directory called ex0604, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch06/ex0604.war—check there if you get stuck. For this exercise, there's also a halfway-house cheat. If you're not comfortable writing Java reflection code (and you won't be asked about that in the exam!), you could take the webcert .ch06.ex0604.Reflector class ready-made from the solution package—this leaves you free to concentrate on writing the JSP (which is more germane to your aspirations to be a web component developer).

Create the Reflector Class

1. Create a source file called Reflector.java, under your WEB-INF/src directory in an appropriate package directory. It must be in some named package, for the JSP that is going to use it can't use classes in the default package. You can choose any package name you like, or use the solution package name, which is webcert.ch06.ex0604.

2. Define three private instance variables in the class:
   - a String called *className*
   - a Set called *interfaces*
   - a Set called *methods*

   Write "getter" methods for each of these three variables (no need for "setters").
3. Define a constructor for Reflector.java that accepts an Object as a parameter.
4. Within the constructor (or using linked methods from the constructor), write code that will
   - Extract the Class object from the Object passed in (using the `Object.getClass()` method).
   - From the Class object, extract the name (`Class.getName()`) and store this in the *className* instance variable.
   - From the Class object, extract the interfaces implemented by the class, and add the names of each of these to the *interfaces* instance variable. (There's a `Class.getInterfaces()` method that returns an array of classes.)
   - From the Class, extract the methods implemented by the class, and add the names of each of these to the *methods* instance variable. (There's a `Class.getMethods()` method that returns an array of methods.)
5. Compile Reflector.java just as you would compile servlet code (so that Reflector.class ends up under the WEB-INF/classes directory—see Appendix B if you need refreshing on this).

Create the implicitObjects.jsp JSP Page Source

6. Create an empty file called implicitObjects.jsp directly in the context directory ex0604.
7. Use a *page* directive to import your Reflector class and the java.util package.
8. Next, include a scriptlet that defines a String array initialized with the names of the eight implicit objects (all except *exception*, which we're not going to bother with). In the same scriptlet, define an Object array initialized with the eight implicit objects—make sure that these appear in the same order as the names in the String array.

9. Define the appropriate elements to begin your HTML document. Then include a table with four headings: Implicit Object Name, Class Name, Interfaces Implemented, and Methods Available.
10. Write the beginning of a **for** loop to iterate around the eight implicit objects in the Object array. At the beginning of the loop, instantiate a Reflector object, passing the implicit object from the Object array into its constructor. Now you have a Reflector object loaded with the information you want to display. Close this scriptlet.
11. Within the **for** loop you will write out the rows of your table, each containing four cells.
12. In the first `<td>` cell, use an expression to display the name of the implicit object. This is taken from the String array you established earlier, using the loop counter for the appropriate occurrence.
13. In the second cell, use an expression to display the name of the class implemented by the implicit object. Use the `getClassName()` method from your Reflector object.
14. In the third cell, use a combination of a scriptlet and an expression to display the names of all the interfaces implemented by the implicit object. You'll want to use the `getInterfaces()` method from your Reflector object. This returns a Set, from which you can derive an Iterator—structure the loop around this.
15. In the fourth cell, use a similar technique to display the names of all the methods implemented by the implicit object. This time, use `getMethods()` on Reflector, and again derive an Iterator from the Set returned.
16. Finally, don't forget a separate scriptlet to close off the **for** loop with a terminating curly brace.
17. Round off the table with `</table>` and any other closing HTML document tags as appropriate

Deploy and Run the JSP Page

18. Create a WAR file that contains the contents of ex0604.
19. Start the web server, if it is not started already.
20. Use your browser to request implicitObjects.jsp using a suitable URL, such as

```
http://localhost:8080/ex0604/implicitObjects.jsp
```

21. A truncated extract from the solution code output is shown in the following illustration, for the `request` implicit object. You can see how the implementing class is Tomcat-proprietary: something called CoyoteRequestFacade! However, it's J2EE-standard in that the one interface this class implemented is javax.servlet.http.HttpServletRequest. The methods implemented are legion (I cut the list short)—all the methods from superclasses (including Object) can be found in the list. There are plenty of refinements you could make to this code—showing all the superclasses, the interfaces they implement, and so on.

Implicit Object Name	Class Name	Interfaces Implemented	Methods Available
request	org.apache.coyote.tomcat5.CoyoteRequestFacade	javax.servlet.http.HttpServletRequest	clear equals getAttribute getAttributeNames getAuthType getCharacterEncoding getClass getContentLength getContentType getContextPath getCookies getDateHeader getHeader getHeaderNames

CERTIFICATION SUMMARY

This chapter gave you a lot of basic grounding in JSP technology. In the four major sections of the chapter, you first learned about the life cycle of JSP pages within a JSP container, then about scripting elements, then page directives you can use to influence mainly global aspects of your page, and finally the nine implicit objects available to you in JSP source code.

You first saw how JSP pages turn servlets on their head. You learned that instead of putting HTML inside Java code (the servlet pattern), you place Java code (and other bits of syntax) inside an otherwise normal HTML page. You also saw that JSP technology is not limited to HTML: XML is viable to include in a JSP page. You learned that a JSP container (usually just another facet of the servlet container

you use, as with Tomcat) takes JSP page source and turns this into a working program—in fact, a generated and compiled servlet. This puts the HTML (or XML) back inside Java code. You learned that this phase is called "translation" of a JSP page and that it incorporates the code generation and compilation. You experienced translation firsthand in the exercise and saw that it usually occurs on first request to a JSP (or on request to a JSP that has changed), but can in fact happen at any time a JSP container chooses before the first request is received.

You saw that generated servlets implement the interface JspPage or, much more likely, its subinterface, HttpJspPage. You learned that this involves providing three methods:

- `jspInit()`
- `jspService()`—with request and response parameters
- `jspDestroy()`

You learned that the JspPage interface keeps the request and response parameters nonspecific, whereas the HttpJspPage interface narrows these down to HttpServlet Request and HttpServletResponse. You saw that this means that the `_jspService()` method ends up looking very much like the signature of a servlet's `service()` method—and it came as no surprise to you that the generated servlet's service method does indeed call `_jspService()`. You learned that the parallels between servlet life cycle and JSP life cycle go further: `init(ServletConfig config)` calls `jspInit()`, `service()` calls `_jspService()`, and `destroy()` calls `jspDestroy()`.

Having explored the JSP page life cycle, you went to look at the composition of JSP page source. You learned that the source subdivides into template text and elements. Of elements, there are three kinds: directive, scripting, and action. Of actions, you confined your knowledge to the fact that they divide into standard or custom types. You learned that there is a recent innovation in scripting called Expression Language, but that the traditional language-based scripting you explored in this chapter is still retained in JSP 2.0. You met the four traditional scripting elements:

- expressions (for displaying data)
- scriptlets (for logic embedded in the request method, `_jspService()`)
- declarations (for separate code outside of `_jspService()`)
- comments (for the translation phase to ignore)

You saw that expressions are demarcated with `<%=` at the beginning, and `%>` at the end. You learned that any Java expression can be inserted in the middle of this, provided that it evaluates to any primitive or object. You learned that an expression, although being Java source, cannot terminate in a semicolon, for expressions are actually incorporated inside existing statements in the generated Java source (usually `out.print(your expression goes here);` statements). You also learned that expressions are inserted into the `_jspService()` method and so are potentially executed on every request to a page.

You saw that scriptlets can be long or short, and contain complete Java statements—as many as required. You learned that <% denotes the beginning of a scriptlet and %> the end. You saw that scriptlets can be used for several purposes, but typical ones include local data setup, presentation logic (e.g., dynamically generating HTML table rows), and, more occasionally, direct output to the page using the implicit object `out`. You also learned that scriptlet code, like expression code, is incorporated directly into the `_jspService()` method at the point of insertion into the JSP page source.

You then explored declarations and found that the Java code in them goes into the generated servlet but *outside* the `_jspService()` method. You discovered that you can include all sorts of things in declarations—instance and class variables, for example—but that their best use is for defining complete methods. You learned that you can use declarations to override the life cycle methods `jspInit()` and `jspDestroy()`. You found that a declaration begins with `<%!` characters and ends the same as expressions and scriptlets—with `%>`.

Finally, in scripting elements, you looked at comments. You learned about two styles—the first beginning with `<%--` and ending with `--%>`. You saw that the translation phase completely ignores everything between these two markers and that anything can go within such a comment—except `<%` (in other words, you can't nest comments of this style). You also learned that you can use HTML/XML-style comments (`<%--` commented out words `--%>`) and that anything within such a comment (expresson, scriplet, declaration) is translated normally. You saw that the advantage of this style is for sending back comment text in the response.

In the next section, you learned about directives. You saw that they have a style that is similar to scripting elements (`<%@` opening characters and `%>` closing characters) but play no direct role in producing response output. You met the three sorts of directive: *page*, *include*, and *taglib*. You saw that every directive has one or more attributes, consisting of name/value pairs—very much like HTML or XML tags. You found that *page* is the most complex directive, having a dozen or so possible attributes (of which you only need know four in detail for the exam). You

used the *import* attribute to have import statements in your generated servlet code. You learned that the *session* attribute can be used to make the `session` implicit object unavailable in a page (`<%@ page session="false" %>`). You saw that the *contentType* attribute determines the MIME type of the response sent back. And you learned that the *isELIgnored* attribute can be used to stop translation evaluating Expression Language.

You then met the *include* directive and saw that it is far simpler—having only one mandatory attribute, file. You learned that you can use *include* to incorporate entire files within your JSP page source—crucially, at *translation* time. You then saw the third and final directive—*taglib*. You learned that this is used to allow the page access to custom actions defined in a tag library, whose location is hinted at in the *uri* attribute. You also learned that custom actions used within the page use the value for the prefix attribute specified in the *taglib* directive.

In the final section of this chapter, you explored implicit objects. You saw that these are nothing more or less than local variables defined at the beginning of the `_jspService()` method and that since the names of these variables are standardized, you can rely on them in your expressions and scriptlets. You were able to greet many of these as old friends from the servlet chapters: *request* (almost always an HttpServletRequest object), *response* (almost always HttpServletResponse), *application* (invariably ServletContext), *session* (HttpSession), and *config* (ServletConfig). You also learned that *out* is nothing more than a writer, albeit a buffered JspWriter—not the PrintWriter directly available from the response. You saw that *exception* is just a Throwable object, available only in `page` designated for error handling (having a *page* directive where the attribute *isErrorPage* is set to true). This left only *page* and *pageContext*. You saw that `page` represents the generated servlet (so as a page author, you don't really need it). You found that *pageContext* supplies a very localized set of attribute functions for `page` scope but can also be used to access the attributes in request, session, or application scope. You learned that `pageContext` performs other vital functions that are generally of more use to JSP container designers than to page authors.

TWO-MINUTE DRILL

JSP Life Cycle

- ❏ Servlets are typically Java programs containing HTML elements. JavaServer Pages are HTML or XML documents containing Java elements.
- ❏ JavaServer Page source is "translated" into a servlet class file, thanks to the JSP container (part of a J2EE application server such as Tomcat).
- ❏ If a page fails to translate, an HTTP 500 error is given back to the requester.
- ❏ HTTP is not the only protocol: It is possible (but rare) to have JSP containers that implement other request/response protocols.
- ❏ Every servlet generated in this way must contain a `_jspService()` method, which receives two parameters: a request object and a response object.
- ❏ In a typical HTTP implementation, the two parameters are of types HttpServletRequest and HttpServletResponse.
- ❏ Every generated servlet must contain a `jspInit()` and `jspDestroy()` method (or inherit these through one of its superclasses).
- ❏ Every generated servlet must implement (itself or in superclasses) at least the JspPage interface.
- ❏ Most implement HttpJspPage (whose super-interface is JspPage).
- ❏ A JSP container can translate a page at any time but often does so at the point when a page is first requested.
- ❏ Once requested, a JSP page enters the request or execution phase.
- ❏ Class loading and initialization of static information occur, as for any other Java class.
- ❏ An instance of the JSP page servlet is made.
- ❏ `jspInit()` is called once, before any requests for an instance of the JSP page are serviced.
- ❏ `_jspService()` is called for every request made to the page.
- ❏ Concurrent container threads can call the same `_jspService()` method on the same JSP page instance at the same time.
- ❏ `jspDestroy()` is called once after all requests to the JSP page have completed—no further requests are admitted once `jspDestroy()` has been called.

- There is usually only one instance of a JSP page servlet.
- A JSP page can be explicitly registered in the `<servlet>` element of a deployment descriptor, but it doesn't have to be.
- The same JSP page can be registered more than once, under a different `<servlet-name>`.
- A different instance of the same JSP page servlet is created for each separate `<servlet-name>`.
- You must not override servlet life cycle methods (`init(ServletConfig config)`, `service()`, `destroy()`) in your JSP pages.

JSP Elements

- A JSP page is composed of template text and elements.
- Template text is any content in a JSP page that is not a scripting element, standard or custom action, or expression language syntax.
- Template text normally comprises HTML or XML, but there are no constraints on the type of content.
- Template text doesn't require "translation," beyond the insertion of some escape characters when necessary.
- There are three sorts of elements: directive, scripting, and action.
- There are two sorts of action: custom and standard.
- There are two families of scripting element: "traditional" and "modern" (Expression Language).
- There are four sorts of "traditional" scripting element: expressions, scriptlets, declarations, and comments.
- An expression displays output in the response and looks like this:

```
<%= expression %>
```

- A scriptlet is generally for more extended Java logic and looks like this:

```
<% scriptlet statement 1; scriptlet statement 2; %>
```

- A declaration is typically used to include complete methods in the generated servlet and looks like this:

```
<%! methodSignature() { method content } %>
```

- A comment is for any area translation should ignore and looks like this:

```
<%-- Anything at all goes here --%>
```

- ❏ Expressions can contain any valid Java code that returns something, primitive or object (i.e., not **void**).
- ❏ Expression code is incorporated in `out.print()` statements in the `_jspService()` method.
- ❏ Expression code must not end in a semicolon (expression code forms part of a statement).
- ❏ Scriptlets can contain most valid Java code.
- ❏ Scriptlet code is generated into the `_jspService()` method, at the point of insertion into JSP page source, so don't use scriptlets to define whole methods.
- ❏ Scriptlet code can contain multiple statements, a single statement, or even a single curly brace ({ or }—block delimiter).
- ❏ Local variables declared in one scriptlet can be accessed in another later in the page source.
- ❏ Declarations are for Java syntax that appears in the generated servlet source outside of the `_jspService()` method.
- ❏ Declarations can be used to override the life cycle methods `jspInit()` and `jspDestroy()`.
- ❏ Comments of the `<%-- lines commented out --%>` variety must not be nested.
- ❏ Use the `<!-- comment lines -->` HTML/XML style of commenting for a comment that is returned in the response.

JSP Directives

- ❏ Directives have similar syntax to scripting elements: opening characters `<%@` and closing characters `%>`.
- ❏ Within these delimiters, directives consist of a name and attributes.
- ❏ Attributes consist of *name*="*value*" (or *name*='*value*') pairs, as in HTML or XML.
- ❏ There are three sorts of directive: *page*, *include*, and *taglib*.
- ❏ The *page* directive has several possible attributes.
- ❏ There can be many *page* directives within a piece of JSP page source.
- ❏ Each *page* directive can have one or several attributes.
- ❏ The *import* attribute produces import statements in the generated servlet code. Example: `<%@ page import="java.util.*, java .io.PrintWriter" %>`.

- The *session* attribute can be used to make the *session* implicit object unavailable. Example: `<%@ page session='false' %>`.
- The *contentType* attribute specifies the MIME type of the response. Example: `<%@ page contentType="image/gif" %>`.
- The *isELIgnored* attribute determines if expression language should be interpreted, or just treated as template text. Example: `<%@ page isELIgnored="true" %>`.
- There are several other `page` attributes that aren't on the exam objectives.
- The *include* directive is used to incorporate files into JSP page source at translation time.
- Files included do not have to be JSP page source (with a .jsp extension)—anything that will translate is permitted.
- A JSP page and its included files are referred to as a translation unit.
- The *include* directive has one mandatory attribute—file.
- The *file* attribute gives an absolute or relative path to the file to be included. Example: `<%@ include file="stubs/header.html" %>`.
- The *taglib* directive makes custom actions from a tag library available in a JSP page.
- The *taglib* directive has two mandatory attributes—*prefix* and *uri*.
- The *uri* attribute indicates how the container should find the tab library.
- The *prefix* attribute is used to preface all tags from the tab library used within the JSP page. Example:

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

JSP Implicit Objects

- There are nine implicit objects: *request*, *response*, *application*, *session*, *config*, *page*, *pageContext*, *out*, and *exception*.
- Implicit objects are just local variables with standard names in the `_jspService()` method.
- *request* and *response* are passed as parameters into `_jspService()`.
- For HTTP containers, *request* must implement the HttpServletRequest interface.
- Otherwise, *request* must implement ServletRequest.

- ❏ For HTTP containers, *response* must implement the HttpServletResponse interface.
- ❏ Otherwise, *request* must implement ServletResponse.
- ❏ *application* is the web application's context object (of ServletContext type).
- ❏ *session* is a web application session object (of HttpSession type).
- ❏ *config* is the ServletConfig object associated with the generated servlet, so it can be used to access initialization parameters for the servlet/JSP page.
- ❏ *out* is a JspWriter, used within JSP pages to write to the response instead of the standard PrintWriter from the response.
- ❏ *out* is—by default—buffered.
- ❏ *exception* is available only in designated error pages, which have the directive `<%@ page isErrorPage="true" %>`
- ❏ Error pages are not called directly. Another JSP with this kind of directive is required:
  ```
  <%@ page errorPage="myErrorPage.jsp" %>
  ```
- ❏ if the page containing this directive throws an exception, the JSP container forwards to the named error page.
- ❏ In the named error page, *exception* (a Throwable object) can be used to execute methods (such as `printStackTrace()`, `getMessage()`) on the exception.
- ❏ *page* is synonymous with *this:* the generated servlet itself.
- ❏ *pageContext* provides "page scope" attributes for the JSP page.
- ❏ *pageContext* has methods that can set, get, and remove attributes in any of the four scopes: page, request, session, or application.
- ❏ The `set`, `get`, and `remove` methods of *pageContext* are overloaded—some signatures accept an **int** parameter to represent scope.
- ❏ Valid values for the scope parameter are defined as constants in the Page Context class: PAGE_SCOPE, REQUEST_SCOPE, SESSION_SCOPE, and APPLICATION_SCOPE.

SELF TEST

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

JSP Life Cycle

1. What will be the most likely outcome of attempting to access the following JSP for the second time? (Choose one.)

```
<%@ page language="java" %>
<%@ page import="java.util.*" %>
<html><head><title>Chapter 6 Question 1</title></head>
<body>
<h1>Chapter 6 Question 1</h1>
<%!
public void jspInit() {
  System.out.println("First half of jspInit()");
%>
<%> new Date() %>
<%!
  System.out.println("Second half of jspInit()");
}
%>
</body></html>
```

   A. Translation error (HTTP response code of 500)
   B. Page not found error (HTTP response code of 404)
   C. Web page returned showing a heading and the current date; two lines of output written to the server log
   D. Web page returned showing "First half of jspInit()," "Second half of jspInit()," a heading, and the current date
   E. Web page returned showing a heading and the current date

2. What is output to the web page on the second access to the same instance of the following JSP? (Choose one.)

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

```
<head><title>Chapter 6 Question 2</title></head>
<body>
<h1>Chapter 6 Question 2</h1>
<%! int x = 0; %>
<%!
public void jspInit() {
  System.out.println(x++);
}
%>
<%= x++ %>
<% System.out.println(x); %>
<% jspInit(); %>
</body></html>
```

A. 0
B. 1
C. 2
D. 3
E. 4
F. Page does not translate.
G. Page translates, but there is another runtime error.

3. For what reason does the following JSP fail to translate and compile? (Choose one.)

```
<%@ page language="java" %>
<html>
<head><title>Chapter 6 Question 3</title></head>
<body>
<h1>Chapter 6 Question 3</h1>
<%! int x; %>
<%!
public void jspDestroy() {
  System.out.println("self-destructing");
} %>
<%!
public void jspInit() {
  System.out.println(<%= x %>);
}
%>
</body></html>
```

A. Expression embedded in declaration.
B. Data member *x* not initialized before use.
C. Local variable *x* not initialized before use.
D. Placement of `jspDestroy()` method before `jspInit()` method.
E. The page actually compiles and translates without any problem.
F. None of the above.

4. Which of the following are true statements about the JavaServer Page life cycle? (Choose two.)

A. The `_jspService()` method is called from the generated servlet's `service()` method.
B. `jspInit()` is only ever called on the first request to a JSP instance.
C. `jspDestroy()` is only ever called on the last request to a JSP instance.
D. All servlet methods are accessible from the `jspInit()` method.
E. You cannot override or provide a no-parameter `init()` method in a JSP page.

5. What is the consequence of attempting to access the following JSP page? (Choose two.)

```
<%@ page language="java" %>
<html>
<head><title>Chapter 6 Question 5</title></head>
<body>
<h1>Chapter 6 Question 5</h1>
<%!public void _jspService(HttpServletRequest request,
  HttpServletResponse response) {
  out.write("A");
} %>
<% out.write("B"); %>
</body>
</html>
```

A. Cannot resolve symbol compilation error.
B. "A" is output to the response.
C. "B" is output to the response.
D. "A" is output to the response before "B."

E. Duplicate method compilation error.
F. "B" is output to the response before "A."

JSP Elements

6. What is the result of attempting to access the following JSP page? (Choose one.)

```
<html>
<head><title>Chapter 6 Question 6</title></head>
<body>
<h1>Chapter 6 Question 6</h1>
<%! public String methodA() {
      return methodB();
   }
%>
<%! public String methodB() {
      return methodC();
   }
%>
<% public String methodC() {
      return "Question 6 Text";
   }
%>
<h2><%= methodA() %></h2>
</body>
</html>
```

A. "Question 6 Text" is output to the resulting web page.
B. A translation error occurs.
C. A runtime error occurs.
D. The text between the `<h1></h1>` HTML tags appears, followed by a Java stack trace.
E. The web page is blank.

7. (drag-and-drop question) The following illustration shows a complete JSP page source. Match the lettered values, which conceal parts of the source, with numbers from the list on the right, which indicate possible completions for the source.

```
<html>
<head><title>Chapter 6 Question 7</title></head>
<body>
<h1>Chapter 6 Question 7</h1>
<form action="Question7.jsp">
Type distance abbreviation here:
<input type="text" name="abbrev" />
Request made at: [A] Date d = new Date();
java.text.DateFormat fmt =
java.text.DateFormat.getDateInstance(DateFormat.S
HORT);
String s = fmt.format(d); [B] s%>
<input type="submit" [C]
</form>
<%=
fullTextOfUnits(request.getParameter("    [D]    ")[E])
%>
[F] page import="java.text.*,java.util.*" %>
[G] public String fullTextOfUnits(String key) {
      if ("km".toLowerCase().equals(key)) {
        return "kilometers";
      }
      if ("m".toLowerCase().equals(key)) {
        return "miles";
      }
      return "";
    }
%>
</body>
</html>
```

1	<%
2	<%=
3	%>
4	<%@
5	<%#
6	<%--
7	--%>
8	#%>
9	;
10	' ' (a single blank space)
11	/>
12	%><%=
13	<%!
14	;%>
15	abbrev

8. What true statements can you make about the following JSP page source? The line numbers are for reference only and should not be considered part of the source. (Choose two.)

```
01 <%@ page import="java.io.*" %>
02 <html>
03 <head><title>Chapter 6 Question 8</title></head>
04 <body>
05 <%
06 PrintWriter out = response.getWriter();
07 out.write("P");
08 %>
09 <% out.write("Q"); %>
10 </body>
11 </html>
```

A. In line 09, the scriptlet markers should not be on the same line as the Java source statement.
B. In JSP technology, it's a bad idea to get hold of the PrintWriter directly from the response.
C. "P" will be written to the response, followed by "Q."
D. "Q" will be written to the response, followed by "P."
E. Only "Q" will be written to the response.
F. The page has a compilation error because of a directive syntax error.
G. The page has a compilation error because the *import* in the directive syntax is for the wrong Java package.
H. The page has a compilation error for other reasons.

9. Which of the following are false statements to make about JSP scripting elements? (Choose three.)

A. It is legal to embed a `<%--` style comment inside another comment.
B. It is legal to embed an expression inside a scriptlet.
C. It is legal to embed an expression inside a declaration.
D. It is legal to embed an expression inside a directive.
E. It is legal to include a declaration at any point in the JSP page source, provided that it appears outside of other elements.
F. It is legal to embed a scriptlet inside an expression inside a `<%--` style comment.

10. What is the result of attempting to access the following JSP page source? (Choose one.)

```
<% <%-- for (int i = 0; i < 10; i++) {
<%-- if(i==3) System.out.println("i is 3!");--%>
    System.out.println("i squared is " + i * i);
} %> --%>
```

A. Doesn't translate because the page source is incomplete.
B. Doesn't compile because nesting comments in this way is illegal.
C. The JSP page would compile if the terminating curly brace were removed.
D. The JSP page would compile if one of the percent (%) signs were removed.
E. Runs as is and produces output (possibly not the output intended).

JSP Directives

11. Which of the following constitute valid ways of importing Java classes into JSP page source? (Choose two.)

A.
```
<%! import java.util.*; %>
```

B.
```
<%@ import java.util.* %>
```

C.
```
<%@ page import="java.util.StringTokenizator" %>
```

D.
```
<%@ page import='java.util.*, java.io.PrintStream'
import="java.text.*" %>
```

E.
```
<%@page import = " java.util.* "%>
```

12. What is the outcome of accessing the first JSP page, includer12.jsp, shown below? (Choose one.)

```
<%-- file includer12.jsp begins here --%>
<% for (int i = 0; i < 10; i ++) { %>
<%@ include file="included12.jsp" %>
<% } %>
<%-- End of file includer12.jsp --%>

<%-- Beginning of file included12.jsp --%>
<html>
<head><title>Chapter 6 Question 12</title></head>
<body>
<h1>Chapter 6 Question 12</h1>
For the <%=i%>th time<br />
</body>
</html>
<%-- End of file included12.jsp --%>
```

A. An ill-formed HTML page will be the output.

B. The call will fail because variable *i* is not declared in included.jsp.

C. Translation will fail because elements denoting the beginning and end of an HTML document must be in the including JSP document, not the included.

D. "For the 10th time" appears in the output.

E. Translation fails for other reasons.

13. What statements are true about the following two JSP page sources, given the intention of always requesting includer13.jsp? (Choose two.)

```
<%-- file includer13.jsp begins here --%>
<%@ page import="java.util.*" contentType="text/html" session="true"%>
<html>
<head><title>Chapter 6 Question 13</title></head>
<body>
<h1>Chapter 6 Question 13</h1>
<%ArrayList al = new ArrayList();
al.add("Jack Russell");
al.add("Labrador");
al.add("Great Dane");%>
<%@ include file="included13.jsp" %>
</body>
</html>
<%-- file includer13.jsp ends here --%>

<%-- file included13.jsp begins here --%>
<%@ page import="java.util.*" contentType="text/html" %>
<table>
  <%for (int i = 0; i < al.size(); i++) {%>
  <tr><td><%= al.get(i) %></td></tr>
  <%}%>
</table>
<%-- file included13.jsp ends here --%>
```

A. Translation will fail because the *import* attribute of the *page* directive is repeated across the page sources.

B. Translation will fail because the *contentType* attribute of the *page* directive is repeated across the page sources.

C. Removing the *Session* attribute from includer13.jsp will make no difference to the generated servlet code.

D. The local variable *al* in included13.jsp will not be recognized.

E. The *import*, *contentType*, and *session* attributes should appear in separate *page* directives.

F. The order of the *import* and *contentType* attributes in both JSP page sources is immaterial.

14. Which of the following are invalid directives? (Choose three.)

    A. <%@page isELignored = “false” %>

    B. <%@ page session=the“/’ is okay true’ %>

    C. <%@ page contentType=“image/music/text” %>

    D. <%@include uri=“header.jsp” %>

    E. <%@ taglib prefix=“mytags” uri=“http://www.osborne.com/taglibs/mytags” %>

15. Given the beginning of the JSP page source below, which set of lines should be used to complete the JSP page source in order to print out all the song lyrics? (Choose one.)

```
<%! static String[] suedeShoes = new String[4];
static { suedeShoes[0] = "One for the Money,";
suedeShoes[1] = "Two for the Show,";
suedeShoes[2] = "Three to Get Ready,";
suedeShoes[3] = "And Go, Cat, Go!";} %>
<% pageContext.setAttribute("line1", suedeShoes[0]);
request.setAttribute("line2", suedeShoes[1]);
session.setAttribute("line3", suedeShoes[2]);
config.getServletContext().setAttribute("line4", suedeShoes[3]);
%>
```

    A.

```
<%@ page contentType="text/plain"
info="Blue Suede Shoes" session="false" %>
<%for (int i = 0; i < suedeShoes.length; i++) {
  String songLine =
  (String) pageContext.findAttribute("line" + (i + 1));%>
  <%= songLine %>
<%}%>
```

    B.

```
<%@ page contentType="text/plain"
info="Blue Suede Shoes" session="true" %>
<%for (int i = 0; i < suedeShoes.length; i++) {
  String songLine =
  (String) pageContext.findAttribute("line" + (i + 1));%>
  <%= songLine %>
<%}%>
```

    C.

```
<%@ page contentType="text/plain"
info="Blue Suede Shoes" session="true" %>
```

```
<%for (int i = 1; i < suedeShoes.length; i++) {
  String songLine =
  (String) pageContext.findAttribute("line" + i);%>
  <%= songLine %>
<%}%>
```

D.

```
<%@ page contentType="text/plain"
info="Blue Suede Shoes" session="true" %>
<%for (int i = 0; i < suedeShoes.length; i++) {
  String songLine =
  (String) pageContext.getAttribute("line" + (i + 1));%>
  <%= songLine %>
<%}%>
```

E.

```
<%@ page contentType="text/plain"
info="Blue Suede Shoes" session="true" %>
<%for (int i = 0; i < suedeShoes.length; i++) {
  String songLine =
  (String) pageContext.getAttribute("line" + (i + 1));%>
  <%= songLine; %>
<%}%>
```

JSP Implicit Objects

16. Which of the following techniques is likely to return an initialization parameter for a JSP page? (Choose two.)

A. <%= request.getParameter(“myParm”) %>

B. <% String s = getInitParameter(“myParm”); %>

C. <% = application.getInitParameter(“myParm”) %>

D. <%= config.getInitParameter(“myParm”); %>

E. <%= getParameter(“myParm”) %>

F. <% Object o = config.getParameter(“myParm”); %>

G. <% String s = config.getAttribute(“myParm”); %>

H. <% String s = getAttribute(“myParm”); %>

17. (drag-and-drop question) The following illustration shows a complete JSP page source. The desired output on the web page is 0 0 1 2 3. Match the lettered values, which conceal parts of the source, with numbers from the list on the right. The numbered fragments indicate possible completions for the source that will achieve the desired output. The numbered fragments may be used more than once, and not all of them are needed.

```
<%@ page session="[ A ]" %>
<%
[       B       ].setAttribute("attr", new Integer(0));
request.setAttribute("attr", new Integer(1));
session.setAttribute("attr", new Integer(2));
[       C       ].setAttribute("attr", new Integer(3));
%>
<%= [       D       ].findAttribute("attr") %>
<%= pageContext.getAttribute("attr",
 [       E       ].[       F       ]) %>
<%= pageContext.getAttribute("attr",
[       G       ].[       H       ]) %>
<%= pageContext.getAttribute("attr",
[       I       ].[       J       ]) %>
<%= pageContext.getAttribute("attr",
[       K       ].APPLICATION_SCOPE) %>
```

1	yes
2	no
3	page
4	PageContext
5	Page
6	PAGE_SCOPE
7	true
8	CONTEXT_SCOPE
9	pageContext
10	false
11	JSP_SCOPE
12	application
13	context
14	REQUEST_SCOPE
15	SESSION_SCOPE
16	APPLICATION_SCOPE

18. What is the result of requesting errorProvoker.jsp for the first time? Assume that neither of the JSP pages below has yet been translated. (Choose one.)

```
<%--Beginning of errorProvoker.jsp page source --%>
<%@ page errorPage="/errorDisplayer.jsp" %>
<% request.setAttribute("divisor", new Integer(0)); %>
<html>
<head>
<% int i = ((Integer) request.getAttribute("divisor")).intValue(); %>
<title>Page Which Terminates In Error</title>
```

```
</head><body>
<%= 1.0 / i %>
</body></html>
<%-- End of errorProvoker.jsp page source --%>
<%-- Beginning of errorDisplayer.jsp page source --%>
<%@ page isErrorPage="true" %>
<%@ page import="java.io.*" %>
<html><head><title>Divide by Zero Error</title></head>
<body><h1>Don't divide by zero!</h1>
<pre><% exception.printStackTrace(new PrintWriter(out)); %></pre>
</body></html>
<%-- End of errorDisplayer.jsp page source --%>
```

A. errorProvoker.jsp will not translate and compile because of faults in the page source.
B. errorDisplayer.jsp will not translate and compile because of faults in the page source.
C. errorDisplayer.jsp will not be translated.
D. A stack trace will be displayed in the requesting browser.
E. The browser's title bar will display "Divide by Zero Error."
F. errorProvoker.jsp displays output.

19. Which of the following are false statements about implicit objects and scope? (Choose four.)

A. *out* is of type java.io.PrintWriter.
B. *config* can be used to return context initialization parameters.
C. `PageContext.findAttribute()` can't be used to return a session scope attribute if an attribute of the same name exists in page scope.
D. *page* is of type java.lang.Object.
E. *application* can't be used to access other web application resources.
F. It is illegal to have attributes of the same name existing in more than one scope.

20. What is the result of attempting to access attributeFinder.jsp below, typing text into the input field, and pressing the submit button? (Choose one.)

```
<%-- Beginning of attributeFinder.jsp page source --%>
<%@ include file="fieldSetter.jsp" %>
<html><head><title>Echo Input</title></head>
<body><h5>Type in the field below and press the button to echo
input...</h5>
<form>
<input type="text" name="<%= session.getAttribute("echoFieldName")%>" />
<input type="submit" />
```

```
</form>
<h3>Echoed Text: <%= request.getAttribute("echoInput") %></h3>
</body></html>
<%-- End of attributeFinder.jsp page source --%>
<%-- Beginning of fieldSetter.jsp page source --%>
<% session.setAttribute("echoFieldName", "echoInput"); %>
<%-- End of fieldSetter.jsp page source --%>
```

A. Can't be accessed: translation error in attributeFinder.jsp.
B. Can't be accessed: translation error in fieldSetter.jsp.
C. Can't be accessed: translation errors in both attributeFinder.jsp and fieldSetter.jsp.
D. **null** is displayed for the echoed text.
E. Echoed text is displayed as intended.

LAB QUESTION

This lab encourages you to use techniques from across this chapter, plus one or two from previous chapters. You're going to write a simple JSP version of the game tic-tac-toe (or as we know it in England, noughts and crosses).

This can be accomplished in a single JSP file. For the playing "grid," construct a table with three rows, each with three cells. Place an input text field in each of the nine cells, to receive input of "X" or "O." Include a button beneath the table that submits the enclosing form. The action associated with the form should be the same JSP file again. However, because you have made a server request, you will have to recall the state of play and reload the table cells with the appropriate value—"X," "O," or nothing at all, if nothing has been marked in the cell.

There are plenty of refinements you can make to the above specifications. The solution code prevents a cell from being "input capable" once an "X" or "O" has been placed within it and the confirm button pressed. You might like to have a go at that and introduce bells and whistles absent from the solution: validating that "X" and "O" are placed alternately, perhaps, or spotting a winning line and presenting an appropriate message.

SELF TEST ANSWERS

JSP Life Cycle

1. ☑ **E**. A web page is returned showing a heading and the current date.
   ☒ **A** is incorrect. It certainly looks strange splitting a declared method (`jspInit()`) in two halves (it's pretty pointless, and I don't recommend it as good style!), but as declarations are imported contiguously into the generated servlet, the page translates just fine. **B** is wrong—had there been a translation error, a 500 error should result in any case, not a page not found 404 error. **C** could be right under unusual circumstances, but for the most part, if this is the *second* access to this JSP, and so the same instance is used again, then `jspInit()` won't be executed (as for the same instance, it would have fired on the *first* access to the JSP, writing two lines to the server log). **D** is definitely wrong: The suggestion here is that the `System.out.println` output goes to the returned response, whereas it goes to the server log.

2. ☑ **D**. 3 is output to the web page. The sequence of events is this: On or before the first access to the JSP, the page is instantiated. The initialize event fires, so the JSP container calls `jspInit()`. The declared instance variable **int** *x* is incremented from 0 to 1 within the method. Then the web page takes this current value of *x* for display through the expression `<%= x++ %>`; only afterward is *x* incremented again from 1 to 2. `jspInit()` is called again in the last scriptlet in the JSP. This causes *x* to be incremented from 2 to 3. On the second request to the web page, the JSP container doesn't call `jspInit()` again: It happens only once per life cycle. So the first thing that happens is that the expression `<%= x++ %>` is evaluated—for a current value of 3, which is then displayed in the web page. Any subsequent increments to *x* don't matter for the purposes of the question.
   ☒ **A**, **B**, **C**, and **E** are incorrect because of the reasoning in the correct answer. **F** and **G** are incorrect because the page is correct syntactically, and translates and runs just fine. The manual call to `jspInit()` may have given you pause for thought, but outside of the fact that this is a method called automatically by the JSP container, it can be treated just as any other regular method. It's not usual, it's not good style, and I hope you'll encounter this kind of obscure coding only in exam questions, but the important thing is to recognize it as legal!

3. ☑ **A**. An expression belongs in the middle of the `_jspService()` method of the JSP's generated servlet. So it makes no sense to locate an expression inside of another declared method in the JSP, whether `jspInit()` or any other.
   ☒ **B** and **C** are incorrect. You need to recognize for one thing that `<%! int x; %>` causes the variable *x* to be declared as a data member, not a local variable. Because data members are always initialized to a value (in this case 0 for the primitive type **int**), there is no initialization problem. **D** is incorrect because the placement of the life cycle methods `jspInit()` and

`jspDestroy()` within the JSP doesn't matter. You can put them in any order, and the result will be the same. **E** is incorrect because there is a fault with the page, and **F** is ruled out because there is a correct explanation for the fault.

4. ☑ **A** and **D**. **A** is true because `_jspService()` does originate from a servlet's `service()` method. **D** is also true: All servlet methods are accessible at this point. For a couple of them (`destroy()`, `service()`), it would make no sense to use them at this point, but they are accessible.
   ☒ **B** is probably correct nearly all of the time but can't be taken for granted: A JSP servlet could be instantiated and the `jspInit()` method called before any request reaches the JSP container. The specification leaves room for the container vendor to do what seems best—as long as the call to `jspInit()` has occurred before the first request. **C** is incorrect: You can't determine (from within the server) when the last request to a JSP will occur. **E** is also incorrect, because (surprisingly) you can override or supply an `init()` method. It's likely that your JSP's implementing servlet will inherit an `init()` method with no parameters from GenericServlet. However, this method is *not* defined in the Servlet interface—rather, there's one that receives a parameter: `init(ServletConfig config)`. This one you can't override in JSPs, and that's true for all the methods defined in the Servlet and ServletConfig interfaces (indeed, most vendors supply a parent JSP servlet—from which your generated servlets inherit—which has the Servlet and ServletConfig methods marked as **final**).

5. ☑ **A** and **E**. Two compilation errors occur. Because the servlet generated from JSP source already contains a `_jspService()` method, any attempt to include one (with the same parameters) will fail with a duplicate method error (can't have two methods with the same names and same parameters in the same piece of Java source). Furthermore, the implicit variable *out* (for the JspWriter) isn't implicitly available within any method of your own that you define in a declaration—which gives rise to the "cannot resolve symbol" error.
   ☒ **B**, **C**, **D**, and **F** are incorrect; because the JSP as it stands makes no sense and won't compile, you certainly won't get any kind of output to the response.

JSP Elements

6. ☑ **B** is correct. There is a translation error because `methodC()` is inside a scriptlet element, whereas it should be inside a declaration. Consequently, `methodC()`'s source is generated directly into the middle of the `jspService()` method, so compilation (part of the translation process) fails.
   ☒ **A** is incorrect because there will be no output to the browser apart from an HTTP 500 error—this rules out **E** as well. **C** is incorrect because there is no compiled code to run. **D** is

incorrect because only the HTTP 500 error is visible in the browser. This may indeed include a Java stack trace—but this is from the translation phase classes, not from the generated servlet, which is the only code that could produce the HTML heading 1 output.

7. ☑ **A** maps to 1: It's the beginning of a scriptlet. **B** maps to 12, for it denotes the end of the scriptlet and the beginning of an expression. **C** maps to 11, for this denotes the end of tag for an HTML element with no body. **D** maps to 15: *abbrev* is the name of the parameter passed on the input text field. **E** maps to 10—only a single white space will do here! (It's an expression—this was an attempt to fool you into picking the semicolon. Expressions don't terminate with a semicolon.) **F** maps to 4, for it's the beginning of a directive (admittedly placed in a stupid position in the middle of the source, but it's still legal and works). **G** maps to 13, for it marks the beginning of a declaration for the `fullTextOfUnits()` method.
☒ There are no other correct permutations, to the best of my knowledge.

8. ☑ **B** and **H** are the correct answers, for both are true statements about the Java page source. **B** is correct because it is a bad idea to use the response's PrintWriter directly. You should instead use the JspWriter associated with the *out* implicit variable. The effects of buffering can jumble the output if *out* and the response's PrintWriter are mixed. **H** is correct: The page in fact fails to compile because a duplicate local variable is defined. *out* is automatically provided in generated JSP source—but this is also the name chosen in this case for the local variable associated with the PrintWriter at source line 06.
☒ **A** is incorrect, for scriptlet markers can share a line with Java source statements entirely as needed. **C**, **D**, and **E** are incorrect because the page doesn't compile, so there is no response. I find if I correct the compilation problem, I do get the output described in answer **C** (P followed by Q—as you might expect. If you have time, though, correct the question source and try swapping the two scriptlets around—observe what happens). **F** and **G** are incorrect answers because the directive is the correct syntax and java.io is the correct package for the PrintWriter class (which is the only unqualified class name in the source).

9. ☑ **A**, **C**, and **D** are the correct answers, for they are all false statements. **A** is correct because you can legally embed one `<%--` comment inside another—it's just not a sensible thing to do, for the translator will think the outer comment has ended as soon as it encounters the inner comment end marker. Doing this may lead to compilation errors, but is not in itself a cause for a translation failure. **C** and **D** are correct because you can never embed an expression inside a declaration or a directive (so **C** and **D** are false statements).
☒ **B** is incorrect because it is a true statement: Expressions can't be embedded inside scriptlets. Although both scriptlets and expressions are generated to the `_jspService()` method, it would almost never lead to legal Java to include one inside the other; hence, the translator won't allow it. **E** is incorrect because it's also a true statement—you can include declarations

anywhere in the JSP page source (they don't have to appear at the beginning or end). And **F** is also true (and so an incorrect answer)—it's legal to embed illegal syntax (a scriptlet inside an expression) or anything else inside a `<%--` style comment; the translator simply ignores anything inside the comment.

10. ☑ **D** is the correct answer. If the percent sign associated with opening of the scriptlet were removed, the JSP page would compile. The angle bracket would be treated as template text—the opening of an HTML tag. The translator would ignore everything from the first comment sign to the first comment end sign. The rest of the page (from "System.out.println" onward) would be interpreted as template text. The browser rendering the dubious output (unbalanced angle brackets in the HTML) might have problems, but the JSP page would compile.
☒ **A** is incorrect: JSP page sources can have no characters in them at all and still be "complete" as far as the translation process is concerned. This is part of their convenience. So don't feel that this fragment is an incomplete page. **B** is incorrect—as discussed in the previous question, you can have nested comments as far as the compiler is concerned; it's just not a sensible thing to do. **C** is incorrect because even if you did remove the curly brace—thus making the scriptlet almost legal Java—compilation would still fail because the declaration for variable *i* is still commented out. And for all the reasons discussed so far, **E** must be incorrect: The page won't run and produce output if left unaltered.

JSP Directives

11. ☑ **D** and **E** are the correct answers. Although **D** looks strange, because it includes two import attributes in the same directive and uses a different style of quote for each set of attribute values, this translates, compiles, and works. Again, **E** looks strange because of the surfeit of white space in places and absence in others—but again, it works fine.
☒ **A** is incorrect: You might think it is reasonable to include an import statement in a declaration element, but it simply fails to translate. **B** is incorrect: *import* is not a directive in its own right (it's an attribute of the *page* directive). Finally, **C** is incorrect because although the JSP syntax is legal, the class java.util.StringTokenizator does not exist—consequently, compilation will give rise to an unresolved import error (and yes, it is fair to test such knowledge in a web application exam!).

12. ☑ **A** is the correct answer. An ill-formed HTML document will be the output—including ten beginning of document `<html>` and end of document `</html>` tags. Everything else about the inclusion process works. On most browsers (which aren't too fussy about well-formed HTML), the page will display.

☒ **B** is incorrect—the *include* directive causes the source of included12.jsp to meld into includer12.jsp. Together they form a single "translation unit"; it is as if there is only one JSP to translate. So the declaration of variable *i* in includer12.jsp makes *i* available to the source of included12.jsp. **C** is incorrect—however incorrect the template HTML, the translation process won't mind (you might design a JSP container to do additional checks, but the JSP spec doesn't mandate this, and the exam questions are according to the JSP spec). **D** is incorrect: The loop is zero-based, so "For the 9th time" is the highest value in the output. And because translation doesn't fail, **E** must also be incorrect.

13. ☑ **C** and **F** are the correct answers. **C** is correct because the *session* attribute for the `page` directive is set to "true"—which is the default—so removal would have no impact. **F** is correct because it's OK for more than one attribute to appear in the same *page* directive, and the order of attributes won't cause an error.
☒ **A** is incorrect because it's always fine to repeat the *import* attribute over the same *page* directive or over several *page* directives. The translator will even forgive you importing the same package or class twice. **B** is incorrect because while it isn't wise to repeat the *contentType* attribute more than once in the same translation unit, as long as the values are the same wherever it appears, then no error will occur. **D** is incorrect: *al* would not be recognized if included13.jsp was translated directly, but we're told in the question that the intention is to call (and translate) includer13.jsp, which declares the variable. **E** is incorrect because there is no issue with having a *page* directive with several attributes. Stylistically, you may want to spread different attributes over different directives—but the exam is a syntax test, not a style test!

14. ☑ **A**, **B**, and **D** are the correct answers, for all illustrate invalid directives. **A** is mistaken in the case for an attribute name, which should be isELIgnored (the second "I" must be a capital letter). **B** mixes double quotes and single quotes to hold an attribute value. Although different attribute values—even in the same directive—can differ in quote style, a single value must stick to the same style. **D** is wrong (and so a correct answer) because *uri* is an attribute of the *taglib* directive, not the *include* directive (which has one mandatory attribute of *file*).
☒ **C** is an incorrect answer because it is a valid directive. Even though the MIME-type ("image/music/text") is rubbish, this won't throw any JSP translation or runtime error. The receiving browser might have a hard time coping with the response, though! **E** is an incorrect answer, for it perfectly expresses a tag library directive.

15. ☑ **B** is the correct answer. This sets up the session variable, finds the attributes for the song lines in the right scope, and prints out all the lines.
☒ **A** is incorrect because the *session* attribute for the `page` directive has a value of false, yet the implicit variable *session* is accessed. The page will not compile in this state. **C** is incorrect

because the logic for finding the attributes is slightly flawed. The loop has been adjusted so the counter starts at 1 instead of 0 (making it simpler to compose the attribute names) but will now only loop through the first three lines. **D** is wrong because the `getAttribute` method for the *pageContext* implicit variable will find attributes only in page scope. So the logic will only find one of the lines. **E** is wrong for the same reason as **D**, and also because the expression to print out the song line terminates incorrectly in a semicolon (so the page will not compile successfully).

JSP Implicit Objects

16. ☑ **B** and **D** are the correct answers. **D** is correct because the *config* implicit object (which must be available in your JSP page) has a `getInitParameter()` method that accepts a String for the parameter name and returns a String representing the parameter object. **B** is correct because you are in the middle of servlet code, and the servlet you inherit from almost certainly implements the `ServletConfig.getInitParameter()` method.
☒ **A** is incorrect because the parameters you retrieve from requests are tied to the request, usually from fields on an HTML form—so have nothing to do with servlet (JSP) initialization parameters. **C** is incorrect because while *application* does have a `getInitParameter()` method, the parameter values returned are for the ServletContext (i.e., the web application as a whole). **E** is incorrect because there is no `getParameter()` method available in the generated servlet. **F** is incorrect because *config*'s method for returning servlet parameters is `getInitParameter()`, not `getParameter`. **G** and **H** are incorrect because there are no attribute methods associated with ServletConfig or servlets directly—and besides, attributes are not parameters.

17. ☑ **A** maps to **7** (because session scope is used, it must be "true" that the *session* implicit object is available). **B** maps to **9** (the Integer 0 must be in page scope because the later expression containing the method `findAttribute()` will then find a value of 0 to print at the right time). **C** maps to **12** (*application* matches APPLICATION_SCOPE in the last line of the page source, for printing out the number 3). **D** maps to **9** again (only *pageContext* has a `findAttribute()` method). **E**, **G**, **I**, and **K** all map to **4** (the class PageContext—required for the final, static constants used as the second parameter in each occurrence of the `getAttribute()` method). **F** maps to **6** (must be PAGE_SCOPE—no other scope will do), **H** maps to **14** (must be REQUEST_SCOPE), and **J** maps to **15** (must be SESSION_SCOPE).
☒ No other combinations will work to produce the correct output.

18. ☑ **C** and **F** are the correct answers. **C** is correct because errorDisplayer.jsp will not be translated, for no error occurs to transfer processing from errorProvoker.jsp. **F** is correct because errorProvoker.jsp does display output: the word "Infinity." The key to answering the question is the knowledge that division involving a double (and 1.0 is a double constant) does not result in an ArithmeticException, as integer division does.

☒ **A** is incorrect, for the page source for errorProvoker.jsp is absolutely fine, as is that for errorDisplayer.jsp , so B is incorrect as well. You might have thought the scriptlet

```
<% exception.printStackTrace(new PrintWriter(out));>
```

strange—but this use of the `out` implicit variable is perfectly acceptable (I'm indebted to Phil Hanna's book—*JSP 2.0: The Complete Reference*, McGraw-Hill/Osborne—for pointing out this handy technique for printing a stack trace to the web page). **D** is incorrect, for no error occurs—so no stack trace. **E** is incorrect—the title "Divide by Zero Error" would occur only if processing was diverted to errorDisplayer.jsp.

19. ☑ **A**, **B**, **E**, and **F** are the correct answers, for all of these are false statements. **A** is false (and so a correct answer) because `out` is a `javax.servlet.jsp.JspWriter`, not a `java.io.PrintWriter`. **B** is false (so a correct answer) because `config` can be used to return initialization parameters for an individual JSP, but not for the entire context (for which you would use the `application` implicit object). **E** is false (so a correct answer) because `application` is just a `ServletContext` object—and this provides the `getContext()` method for accessing different contexts outside of the current web application. **F** is false (so a correct answer) because it is absolutely possible to have attributes of the same name in different scopes (as you would have understood from Question 17 if nowhere else!).
☒ **C** and **D** are incorrect, for both are true statements. **C** is true (so an incorrect answer) because `findAttribute()` will indeed look to the innermost scope (`page`) first for attributes, and will look no further once it has found an attribute for the sought name. **D** is true (so an incorrect answer), for `page` is surprisingly defined as an `Object` reference—not a `Servlet`, or `GenericServlet` or `HttpJspPage`. It is likely that you can *cast* the `page` reference to something more useful, such as one of those listed, but you probably wouldn't bother, for you have `this` to use as a more useful alternative.

20. ☑ **D** is the correct answer. The constant **null** is invariably displayed as the echoed text. Everything translates fine—there is nothing syntactically incorrect in the page sources. However, in attributeFinder.jsp, the developer should have used `request.getParameter("echoInput")` instead of `request.getAttribute("echoInput")`. Field values from HTML forms are made available as parameters, not attributes.
☒ **A**, **B**, and **C** are incorrect because no translation errors occur. You may have thought that using an expression for an HTML form field is illegal—but no, it's fine, even with the peculiar logic here, with the name set in an included JSP page source. **E** is incorrect because you have to make the adjustment described in the correct answer to get the input echoing properly.

LAB ANSWER

Deploy the WAR file from the CD called lab06.war, in the /sourcecode/chapter06 directory. This contains a sample solution. You can call the JSP using a URL such as

```
http://localhost:8080/lab06/tictactoe.jsp
```

This is a very basic version of the game—you may find it hard to stop yourself from improving the code!