7

JSP Standard Actions, XML, and EL

CERTIFICATION OBJECTIVES

- Standard Actions
- Dispatching Mechanisms
- JSPs in XML
- Expression Language

✓ Two-Minute Drill

Q&A Self Test

In the last chapter you covered the fundamentals of JavaServer Pages and did a lot of work with the "traditional" scripting elements. In this chapter you start by learning about standard actions. These are interesting in that they have been available for a long while and—like "traditional" scripting—were common topics in the previous version of the exam. Yet they also foreshadow the more recent innovations in JSP technology, which continues to push in an XML direction.

You'll also explore in the first half of the chapter how standard actions play a role in dispatching mechanisms: forwarding to and including other resources (you'll remember that you have come across one inclusion mechanism already—the `<%@ include %>` directive—and this chapter rounds out that topic).

In the second half of the chapter, you are introduced to two topics that are recent additions to JSP technology, and so also to the latest syllabus of the web component developer exam:

- JSP documents, which are JSPs written entirely in well-formed XML
- Expression Language, the "modern" alternative to language-based scripting

You've seen Figure 7-1 before, which shows the makeup of a JSP page—it's been slightly modified to indicate the topics you covered previously and which JSP elements are explained in this chapter.

CERTIFICATION OBJECTIVE

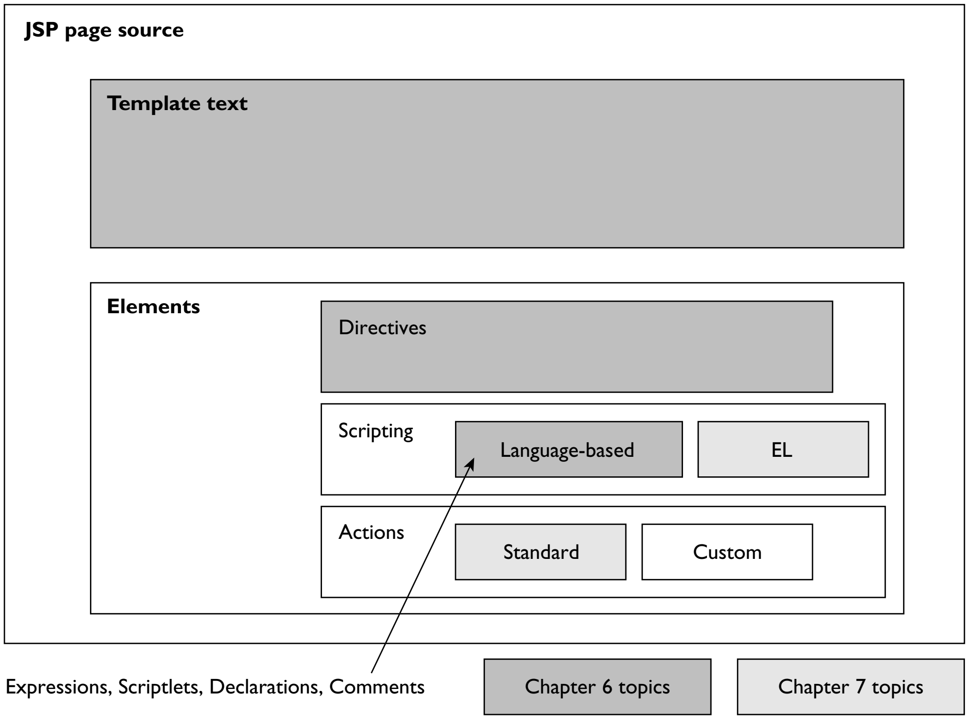
JSP Standard Actions (Exam Objective 8.1)

Given a design goal, create a code snippet using the following standard actions: jsp:useBean (with attributes: "id," "scope," "type," and "class"), jsp:getProperty, and jsp:setProperty (with all attribute combinations).

Standard actions are used for the same purpose as Java language-based scripting: Most if not all the goals that you can achieve with standard actions are achievable with other scripting elements. But in contrast to the techniques we've used so far, they are written using entirely conventional XML syntax.

FIGURE 7-1

Anatomy of a JSP Page Revisited



So why use them? The answer is that they get the job done more elegantly. They often provide an alternative to inserting screeds of Java logic into your neatly designed presentation page. Standard actions are also—arguably—tools that can be used by the nonprogrammer to introduce dynamic behavior that would otherwise entail Java language knowledge. (Actually—given the range of standard and custom actions available, combined with Expression Language and JSTL, which we cover later—you probably need a programmer mentality to embrace even the "non-Java-language" tools that are available in JSP page source these days.)

Seven standard actions are provided from JSP 1.2 onward: Three are considered here; three are in the next section, on dispatching; and one is out of scope for the exam (`<jsp:plugin>`). Although a few more have been added in JSP 2.0, they are not directly on the exam syllabus.

The trio of standard actions in this section (`<jsp:useBean>`, `<jsp:getProperty>`, `<jsp:setProperty>`) work together to incorporate information from existing Java objects into your JSP page. These existing Java objects must be written to rules within the JavaBean specification, so before we approach the standard actions themselves, we'll do some preliminary work exploring the least you need to know about beans.

Why are they *standard* actions? The reason is that you can have *custom* actions as well, which you build yourself using tag libraries, a subject that we fully explore in Chapters 8 and 9. Both standard and custom actions are similar in appearance: XML elements that encapsulate functionality on a JSP page. The difference is that you can rely on any J2EE-compliant JSP container to provide support for all the standard actions defined in the JSP spec. Not so custom actions, because they are—well—customized for your web applications. Not that you can't use custom actions across several projects, just that the onus is on you to deliver all the apparatus to make them work within the web application (you can't rely on the JSP container to have the parts required).

Beans

The standard actions we explore first are designed to instantiate Java objects, then write data to or read data from those objects. Java objects come in many shapes and sizes, so it's little wonder that standard actions can work only if those objects obey at least some minimal conventions. Enter the JavaBeans Specification, which has been around almost as long as Java itself. The idea of JavaBeans is that you can have Java components (typically classes) that can be interrogated by interested software, using the reflection techniques we have employed several times in the exercises up to now. "Interested software" includes the web container code that implements the standard actions we're about to discuss. By interrogating the methods available on a JavaBean, a standard action can obtain information about the properties that the bean supports—in other words, the data that it stores.

This process works in a remarkably simple way. All your bean has to do is to provide "getter" and "setter" methods. Here's a short class that defines information about a dog:

```
public class Dog {
  private String name;
  private float mass;
  private boolean insured;
  private char sex;
  private String barkVolume;
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }
  public float getWeight() { return mass; }
```

```
    public void setWeight(float weight) { mass = weight }
    public boolean isInsured() { return insured; }
    public void setInsured(boolean insured) {
        this.insured = insured;
    }
    public char getSex() { return sex; }
    public void setSex(char sex) { this.sex = sex; }
    public String getBarkVolume() {
        return barkVolume;
    }
    public void setBarkVolume(String barkVolume) {
        this.barkVolume = barkVolume;
    }
  }
```

You can see that the Dog class contains five pieces of information pertaining to the dog. The first is the data member *name*. Because *name* is private, the class provides a public `setName()` method to update the dog's name within a dog instance and, of course, a `getName()` method to read the data. Note how the method names use the instance variable name but capitalize the initial letter: `setName()`. In this way, the standard Java naming conventions for instance variables and method names remain unbroken. This is such a standard convention in Java that you're probably wondering why I've wasted a precious paragraph on the topic.

However, there are a couple of twists if you've never encountered beans before. Tools that use beans (which include JSP standard actions) care only about the methods on the bean. From `setName()` and `getName()`, a bean-literate tool understands that there is a property on any dog "bean" called *name* (i.e., what you would expect the instance variable to be called after allowing for the capitalization difference). But let's look at the next pair of methods—`setWeight()` and `getWeight()`. From this, we infer there is a property called *weight.* And this is correct—even though the instance variable connected to these methods is called something quite different: *mass,* in this case. How we represent the data in the bean (and we may not bother at all) doesn't matter.

The only other convention to mention is that for primitive **boolean** properties, such as *insured* in our dog bean, you have the option (as a bean developer) to supply a method called `isInsured()` instead of (or as well as) `getInsured()` for reading the property value.

Another thing about a bean is that you—as a developer—don't normally have control over creation of your beans. You don't write code like this:

```
Dog d = new Dog();
```

Instead, you leave the instantiation to the bean tool you are using— in our case, standard actions. Consequently, your bean must have a no-argument constructor— either one you provide or the default one the compiler provides in the absence of others. This is the only kind of constructor that your bean tool can assume as universal across all the beans it has to deal with.

on the job

If you carry on down the J2EE road beyond the SCWCD, you'll go on to learn about Enterprise JavaBeans (EJBs). These do quite often have getter and setter methods, but beyond that, the resemblance to the "normal" JavaBeans we have just discussed comes to an end. EJBs are a completely different ball game—they require a specialized container, much as servlets and JSPs do. So don't try to connect your `<jsp:useBean>` standard action to an EJB, because you are doomed to fail!

Standard Actions

After what may seem like a digression, we can return to the core syllabus matter of standard actions. Let's first look at the general syntax of actions, whether standard or custom. As we've said, they adhere to strict XML syntax. Here's a generalized picture:

```
<prefix:tagname firstAttribute="value" secondAttribute="value"> ...
</prefix:tagname>
```

Each standard action element consists of a start tag, `<prefix:tagname>`, and an end tag of the same name (with a forward slash inserted after the first angle bracket), `</prefix:tagname>`. The start tag may contain named attributes, separated from their corresponding value by equal signs. The value is typically surrounded by double quotes or by single quotes (which is sometimes convenient). After so much exposure to the deployment descriptor, web.xml, and HTML web pages, this syntax must feel refreshingly familiar.

But just in case you didn't know, the area between the start and end tag (represented by the ellipsis [. . .] above) is termed the body of the element. A standard action may have a body, but it often has no body at all. This can be represented in one of two ways:

1. By having start and end tag touching, thus: `<prefix:tagname attr="value"></prefix:tagname>`
2. By omitting the end tag but using a special /> terminator for the start tag, thus: `<prefix:tagname attr="value" />`

It makes no difference which of the above forms you use; the JSP container will interpret both identically. So with that in mind, let's look at the three standard actions for the exam objective:

- `<jsp:useBean>`
- `<jsp:setProperty>`
- `<jsp:getProperty>`

You can see three tag names here: `useBean`, `setProperty`, and `getProperty`. You can also see a common prefix—`jsp`—separated from the tag name by a colon. Indeed, the prefix for all standard actions is `jsp`. When you come to write your own custom actions later, you'll have to supply a prefix—it may come as no surprise to learn that `jsp` is reserved, even if your page eschews all standard actions in favor of your own custom ones.

`<jsp:useBean>`

The `<jsp:useBean>` standard action declares a JavaBean instance and associates this with a variable name. The instance is then available for use elsewhere in your JSP page: either in Expression Language (highest grades), other standard actions (still good practice), or in Java language scripting (frowned upon—but still legal! You'll see plenty of to-ing and fro-ing between standard actions and scriptlets and expressions in this chapter—all in the name of education, of course).

In general, you'll use `<jsp:useBean>` to set up your bean and two more standard actions to write and read properties on your bean. These standard actions are called `<jsp:setProperty>` and `<jsp:getProperty>`—which makes pretty good sense in light of our bean discussion a little earlier.

So how exactly do you use `<jsp:useBean>` to do this? The answer is to get to work with its attributes. The simplest approach is this:

```
<jsp:useBean id="theDog" class="animals.Dog" />
```

For this to work, several things have to be true:

- The class attribute must specify the fully qualified name of a class (the import attribute of the page directive will be no help to you, unfortunately).
- `animals.Dog` must obey JavaBean conventions.

- `animals.Dog` must be visible somewhere in the web application—mostly this means it will exist as a class in WEB-INF/classes or within a JAR file in WEB-INF/lib.
- An *id* with a value of *theDog* must not have been used in `<jsp:useBean>` already; in other words, all *ids* for beans on a page must be unique.

Inserting this in your JSP page source will result in some quite complex code in your generated servlet's `_jspService()` method. Ultimately, the code will create an instance of the bean `animals.Dog`. How will the code reference the object? In two ways:

1. As a local variable in the method, whose name comes from value of the *id* attribute (so *theDog* becomes a local variable).
2. As an attribute in some scope or other—page, request, session, or application. In this case, *theDog* is the name of the attribute.

On this second point: Because we didn't specify the scope anywhere in our example, where did `theDog` go? The answer is into page scope. The following cumbersome combination of directive and scriptlet code—following on from the `<jsp:useBean>` standard action declaring *theDog* JavaBean above—will get hold of our bean object into the local variable *myDog*:

```
<%@ page import="animals.Dog" %>
<% Dog myDog=(Dog)
pageContext.getAttribute("theDog"); %>
```

I'm not suggesting that you should ever do this—this code is only here to unravel the mystery of bean location.

If we want our bean to be a little more permanent than page duration, we need to use another attribute of `<jsp:useBean>`: namely, *scope*. To put *theDog* into session scope, it really is this simple:

```
<jsp:useBean id="theDog" class="animals.Dog" scope="session" />
```

The valid values for *scope* are page, request, session, and application—exactly as expected. So now we can access *theDog* across a series of requests from the same user. What if the *theDog* already exists in session scope when the standard action above is encountered? That's OK—`<jsp:useBean>` recycles the existing bean; it doesn't create a new one.

We haven't explored all the ramifications of `<jsp:useBean>` just yet, in particular a fourth attribute named *type*. But before we do that, we should look at the two standard actions inevitably used in conjunction with `<jsp:useBean>`, which are of course `<jsp:setProperty>` and `<jsp:getProperty>`.

on the job

There is an attribute of* `<jsp:useBean>` *called* beanName*, off scope for the exam. The main functionality this offers (over and above the* class *and* type *attributes) is the possibility of using a serialized bean from your file system. To learn more, take a careful look at section 5.1 of the JSP specification, and the J2SDK API for the* `instantiate()` *method on the java.beans.Beans class.

`<jsp:setProperty>`

The purpose of `<jsp:setProperty>` is to set the values of one or more properties on a bean previously declared with `<jsp:useBean>`. The most obvious way to use it is as follows:

```
<jsp:setProperty name="theDog" property="weight" value="6.4" />
```

The first thing to watch is the attribute *name*. This is the name of the bean itself. The value for this attribute has to be the same as a previous value for a `<jsp:useBean>` *id* attribute. It's a pity that the attributes don't match—it's another thing you have to remember for the exam: *id* on `<jsp:useBean>` = *name* on `<jsp:setProperty>` (and `<jsp:getProperty>` as well, when we get to it).

exam watch

Actually, the truth is that you can use* `<jsp:setProperty>` *and* `<jsp:getProperty>` *without a previous* `<jsp:useBean>`*. All* `<jsp:setProperty>` *and* `<jsp:getProperty>` *do is to use* `PageContext.findAttribute()`*—so if an attribute of the right name exists—set up, perhaps, in a previous servlet—these standard actions will find it. However, it's good practice to include* `<jsp:useBean>` *before these actions in the same JSP page. After all, it won't replace beans of the same name that you have set up by other means, and it will create beans of the right name that don't exist already. Furthermore, if your* `<jsp:setProperty>` *and* `<jsp:getProperty>` *standard actions try to access an attribute that doesn't exist, they are liable to die a horrible death with HTTP 500 errors returned to the requester.

The *property* attribute specifies a property on the bean. Because the property here is "weight," then the underlying code will assume the existence of a `getWeight()` method on the *theDog* bean. The *value* attribute supplies the data for the property—or in code terms, the parameter that is passed into the getter method. In our example, `getWeight()` expects a float parameter, yet the value for the value attribute looks very like a String constant: `value="6.4."` Yet we don't have to worry—it's the responsibility of the JSP container to handle the type conversions involved.

The *value* attribute has another feature not shared by any other attributes in `<jsp:useBean>`, `<jsp:setProperty>`, or `<jsp:getProperty>`. Instead of supplying a literal value, you can substitute an expression. Here are two examples:

```
<% float w = 6.4f; %>
<jsp:setProperty name="theDog" property="weight" value="<%=w%>" />
```

Another:

```
<% String dftWeight = config.getInitParameter("defaultDogWeight"); %>
<jsp:setProperty name="theDog" property="weight" value="<%= dftWeight %>" />
```

All these are viable ways to "soft-code" the value of *value*, so as to set a property in your application. The first sets up a float variable called *w* in a scriptlet, and uses this directly in an expression embedded into the following standard action. Note how the double quotes are retained to demarcate the beginning and end of the value, and the expression plugs between them: `value="`**`<%=w%>`**`."` The second example uses a scriptlet to obtain an initial parameter called defaultDogWeight associated with the JSP, and plugs this into the *value* expression. Later, you'll see that you're not stuck with Java language expressions to supply values. Expression Language also fits the bill, and we'll revisit standard actions with Expression Language before the end of this chapter.

You'll very often want to use request parameters (say from an HTML form) to set properties. You could follow on from the examples above and write code like this:

```
<jsp:setProperty name="theDog" property="weight"
value="<%= request.getParameter("dogWeight") %>" />
```

The expression plugged into the value attribute this time uses the *request* implicit object to retrieve the parameter "dogWeight." In fact, this is such a common thing to want to do that `<jsp:setProperty>` provides some convenient syntax to avoid ungainly code like this. This is how it looks:

```
<jsp:setProperty name="theDog" property="weight" param="dogWeight" />
```

This is shorthand for saying take the request parameter called "dogWeight," and use the value for this to set the property called "weight" on the bean called "theDog." Very neat, and it can get neater still. It could well be that name of a request parameter (from your HTML form) matches the corresponding property name. In that case, you can omit the param attribute altogether:

```
<jsp:setProperty name="theDog" property="weight" />
```

This time, the underlying code will look for a parameter (from the ServletRequest or HttpServletRequest) called "weight" and use this to set the property value for "weight" on "theDog." This shorthand goes further still. If you have a number of request parameters that match the names of several properties on your target bean, you can simply write

```
<jsp:setProperty name="theDog" property="*" />
```

Now *any* property whose name matches a request parameter name will have its value preloaded from that request parameter.

exam watch

The* param *and* value *attributes are mutually exclusive. After all, it doesn't make sense to have both on the same* `<jsp:setProperty>`*. Either you use the convenience of* param *to set a property from a request parameter or you take a more explicit approach with the* value *attribute. Note that you can also omit both the* param *and* value *attributes: This is equivalent to supplying* param *with the same value as the* property *attribute.

`<jsp:getProperty>`

That leaves `<jsp:getProperty>`. This is the easiest of the three standard actions we've used. You use it to output the value of a bean's property to the response (for display on a web page, inclusion in an XML document—whatever). There are two attributes to supply: *name* (the name of the bean) and *property* (the name of the property). They're both mandatory (as are the identical attributes on `setProperty`, by the way). So to get hold of our dog's weight property, you simply write the following:

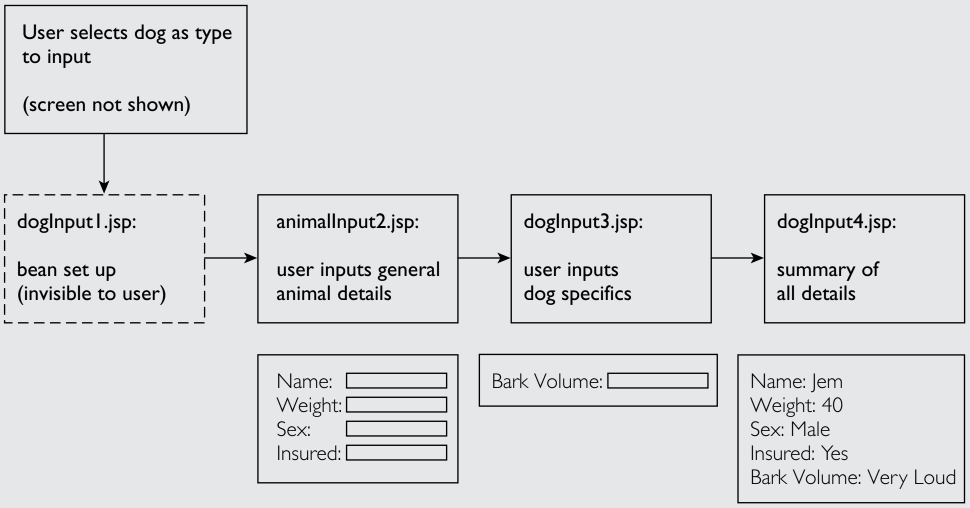
```
<jsp:getProperty name="theDog" property="weight" />
```

Like `<jsp:setProperty>`, *theDog* bean has to be there—preferably declared with `<jsp:useBean>` earlier in the page.

INSIDE THE EXAM

Before leaving these three standard actions, let's put them together in a bigger example. This is here to reinforce some of the points made earlier, but also to illustrate some of the subtler aspects of `<jsp:useBean>` that you need to know for the exam. We'll be working with the Dog JavaBean that started this chapter, in a slightly modified form.

Let's suppose that you're writing a system to register animals at your local veterinary clinic. The clinic copes with all kinds of animals: dogs, cats, parakeets, and rattlesnakes. But whatever the animal, there are some details that will be common. Others may be animal-specific. We'll look at a portion of the system: that deals both with some general animal characteristics and also ones that are dog-specific. Here's an illustrative screen flow:



First of all, the vet's receptionist selects the type of animal to be registered—in our case, dog. In the background, a Dog JavaBean is set up with some defaults (this happens in dogInput1.jsp). The receptionist is presented with a screen to fill in some details about the animal in general (animalInput2.jsp), followed by a screen for specific dog details (dogInput3.jsp). On pressing ENTER, the receptionist finally sees a summary screen of all the dog details that have been saved to the database (dogInput4.jsp). (You'll have to imagine the database in the example code that follows—the intention here is just to show the JSP aspects.) You can run this code (and look at the full source) from the WAR file /sourcecode/ch07/examp0701.war. Start at the following URL:

INSIDE THE EXAM (continued)

```
    http://localhost:8080/examp0701/dogInput1.jsp
```

The point about this design is that the general animal details screen—animal Input2.jsp—will work regardless of animal type, for there is nothing dog-specific within it. Yet it makes use of a Dog JavaBean set up with `<jsp:useBean>`. How can this be? We'll solve this mystery in a page or two, as we step carefully through the code. Let's first take a look at dogInput1.jsp:

```
<jsp:useBean id="currentAnimal"scope="session"
class="webcert.ch07.examp0701.Dog">
 <jsp:setProperty name="currentAnimal" property="name" value="Fido" />
 <jsp:setProperty name="currentAnimal" property="weight" value="6.5" />
 <jsp:setProperty name="currentAnimal" property="sex" value="F" />
 <jsp:setProperty name="currentAnimal" property="insured" value="false" />
 <jsp:setProperty name="currentAnimal" property="barkVolume" value="Loud" />
</jsp:useBean>
<%session.setAttribute("animalSort", "dog");
  RequestDispatcher rd =
  application.getRequestDispatcher
  ("/animalInput2.jsp");
  rd.forward(request, response); %>
```

Recall at this point that the receptionist has made a choice of animal type. First of all, a Dog bean called "`currentAnimal`" is set up in session scope. Nothing unusual there, but notice that the `<jsp:useBean>` tag is not "self-closing" as we've seen before—like this:

```
<jsp:useBean id="currentAnimal" scope="session"
class="webcert.ch07.examp0701.Dog" />
```

In this case, there is an end tag a few lines further on: `</jsp:useBean>`. So this tag has a body—in this case, filled with five `<jsp:setProperty>` standard actions. The presence of a body signifies that some logic will execute:

- If the currentAnimal bean doesn't exist, it will be created, and the `<jsp:setProperty>` tags will execute to set up some default values.
- If the currentAnimal bean exists already, it will be *left alone*, and the `<jsp:setProperty>` tags will not execute, so any property values already set will remain unchanged.

INSIDE THE EXAM (continued)

Not that it does, but if our screen flow came back through dogInput1.jsp, currentAnimal would stay unaffected.

The scriptlet at the end of dogInput1.jsp does two things: First, it sets up a session attribute called `animalSort` with a value of "dog," to indicate to future screens that it's a dog we're dealing with (not a cat or a hamster). Secondly, it uses standard Request Dispatcher code to forward to the next screen in sequence—animalInput2.jsp (you'll see how to replace this code with a standard action a bit later in the chapter). So notice that all you ever do in dogInput1.jsp is "pass through"—the response isn't returned to the user.

The code for animalInput2.jsp is shown below. Notice that there isn't anything dog-specific anywhere in the source code:

```
<html><head><title>General Animal Information</title></head>
<body>
<h2>Fill in general animal information here, regardless of what sort of
animal...</h2>
<jsp:useBean id="currentAnimal" scope="session" type="webcert.ch07
.examp0701.Animal" />
<p>Overtype the defaults in the form below...</p>
<form action="<%= session.getAttribute("animalSort")%>Input3.jsp">
<br />Name: <input type="text" name="name" value="<jsp:getProperty
name="currentAnimal" property="name" />" />
<br />Weight: <input type="text" name="weight" value="<jsp:getProperty
name="currentAnimal" property="weight" />" />
<br />Sex: <input type="text" name="Sex" value="<jsp:getProperty
name="currentAnimal" property="sex" />" />
<br />Insured: <input type="text" name="insured" value="<jsp:getProperty
name="currentAnimal" property="insured" />" />
<br /><input type="submit" value="Continue..." />
</form></body></html>
```

After the template text at the beginning, inviting you to fill in general animal information, you find the `<jsp:useBean>` standard action, requesting the same bean called currentAnimal in session scope. But instead of the class attribute, we find another attribute called type instead:

`type="webcert.ch07.examp0701.Animal"`

It's the same bean that we get hold of, which is a Dog object. However, if we look in the generated servlet code, any reference to this bean will be of Animal type. This can only work under the following circumstances:

INSIDE THE EXAM (continued)

- Dog is a subclass of Animal.
- Or Dog implements an interface called Animal.

In other words, the type you choose must be compatible with the actual class of the object. Let's suppose that Dog implements an Animal interface and that its class declaration now looks like this:

```
public class Dog implements Animal
```

And the Animal interface declares all the methods about general animal characteristics:

```
    public interface Animal {
    public abstract String getName();
    public abstract void setName(String name);
    public abstract float getWeight();
    public abstract void setWeight(float weight);
    public abstract boolean isInsured();
    public abstract void setInsured(boolean insured);
    public abstract char getSex();
    public abstract void setSex(char sex);
}
```

Dog implements all these methods as we've already seen, plus a couple that are dog-specific—to set and get the barkVolume property.

Using type in `<jsp:useBean>` without the `class` attribute relies on the fact that the bean has already been created. You can simultaneously create a bean object and type it to something else for use in the current JSP page by using both attributes at the same time, like this:

```
<jsp:useBean id="currentAnimal"
scope="session" class="Webcert.ch07.examp0701.Dog"
type=""webcert.ch07.examp0701.Animal" />
```

The form in animalInput2.jsp uses `<jsp:getProperty>` standard actions to display the default values already set up on the Dog bean, which can be overtyped in the form. Of course, *barkVolume* is missing from the list. The only other point to note is that—to keep

INSIDE THE EXAM (continued)

animalInput2.jsp generic—the `<form>` *action* attribute uses an expression to complete the name of the next JSP in sequence. The start of the name comes from the session attribute animalSort, which you'll recall was set up as "dog" way back at the beginning. So the user will navigate to dogInput3.jsp, but you can see that a different initial choice of animal might have led to catInput3.jsp or budgerigar3.jsp.

In dogInput3.jsp, not a great deal happens that we haven't seen already. Here's the code:

```
<html><head><title>Specific Dog Information</title></head>
<body>
<h2>Fill in specific dog information here...</h2>
<jsp:useBean id="currentAnimal" scope="session" class="webcert.ch07
.examp0701.Dog" />
<jsp:setProperty name="currentAnimal" property="*" />
<p>Overtype the defaults in the form below...</p>
<form action="dogInput4.jsp">
<br />Bark Volume:<input type="text" name="barkvolume"
value="<jsp:getProperty
name="currentAnimal" property="barkVolume" />" />
<br /><input type="submit" value="Continue..." />
</form></body></html>
```

The receptionist uses this screen to type in vital dog-specific properties—we have only the one, *barkVolume*. There's a vital line here:

```
<jsp:setProperty name="currentAnimal" property="*" />
```

This line has nothing to do with the setup of the current page, in fact. Its purpose is to take the request parameters typed in to the previous animalInput2.jsp, and save these to properties on the bean. Without this, the original default values would stick and the receptionist's overtyping would be in vain. This time, clicking the submit button navigates to the last page in sequence, dogInput4.jsp:

```
<html><head><title>Your Completed Dog</title></head>
<body>
<h2>The animal database has been updated with these DOG details:</h2>
<jsp:useBean id="currentAnimal"
scope="session"
type="webcert.ch07.examp0701.Dog" />
```

INSIDE THE EXAM (continued)

```
<jsp:setProperty name="currentAnimal" property="*" />
<br />Name: <jsp:getProperty name="currentAnimal" property="name" />
<br />Weight: <jsp:getProperty name="currentAnimal" property="weight" />
<br />Sex: <jsp:getProperty name="currentAnimal" property="sex" />
<br />Insured: <jsp:
getProperty name="currentAnimal" property="insured" />
<br />Bark Volume: <jsp:
getProperty name="currentAnimal" property="barkVolume" />
</body></html>
```

All this page does is to display all the bean properties, whether dog-specific or general. In a real system, the receptionist might scan the details and confirm the database update. Notice one thing here: The `<jsp:useBean>` standard action uses the *type* attribute but actually names the class Dog instead of the interface Animal: `type="webcert.ch07.examp0701.Dog."` Only a Dog will do if we want to get hold of the *barkVolume* property, not present on Animal. And indeed, the code could just as well have used the class attribute:

```
<jsp:useBean id="currentAnimal" scope="session"
class="webcert.ch07.examp0701.Dog" />
```

The point this makes is that the value for the *type* attribute can be the same as the *class*—there's no compulsion to make it different (although it normally makes sense to do so).

EXERCISE 7-1

ON THE CD

JSP Standard Actions

In this exercise you're going to put together two web application pages: an HTML page with a form whose action takes you to a JSP page. The HTML page will invite you to put in details about a music CD. On clicking a Continue button, you're taken to a summary form (the JSP) that confirms the details you entered.

Not the world's most exciting application—but the first in this book in which the JSP component is *completely free of Java code*! The only building blocks required are HTML and the standard actions you have just learned. That's not to say you won't write any Java, though—as you still need a JavaBean on which the standard actions can operate.

Create the usual web application directory structure under a directory called ex0701, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch07/ex0701.war—check there if you get stuck.

Create the HTML Page

1. Create an empty file directly in your newly created context directory, ex0701. Call it musicCDform.html.
2. Provide a form with four text fields for title, artist, year of release, and favorite track. Give names to the input fields as follows: title, artist, year, and track.
3. Don't forget a submit button. Make the action of the form "musicCDsummary.jsp."

Create the MusicCD JavaBean

4. Create a package directory in ex0701/WEB-INF/src, and within it create a Java source file called MusicCD.java.
5. Include four private instance variables as follows:
   - `String title`
   - `String artist`
   - `int yearOfRelease`
   - `String favoriteTrack`
6. Provide a no-argument do-nothing constructor (you could leave this out and let the compiler provide it) and getters and setters for the instance variables. Make sure these are public and that they exactly follow the bean convention (e.g., `get`**`T`**`itle()`).
7. Compile the source into ex0701/WEB-INF/classes/<package directory>.

Create the JSP Page Source

8. Create an empty file in ex0701, called musicCDsummary.jsp.
9. Use the `<jsp:useBean>` standard action to create a MusicCD bean in page scope.
10. Set the properties of the bean from the request parameters passed in from the HTML form. In two cases, the request parameter names match the bean property names (for title and artist). So use `<jsp:setProperty>` with the "*" setting for the property attribute to take advantage of this.

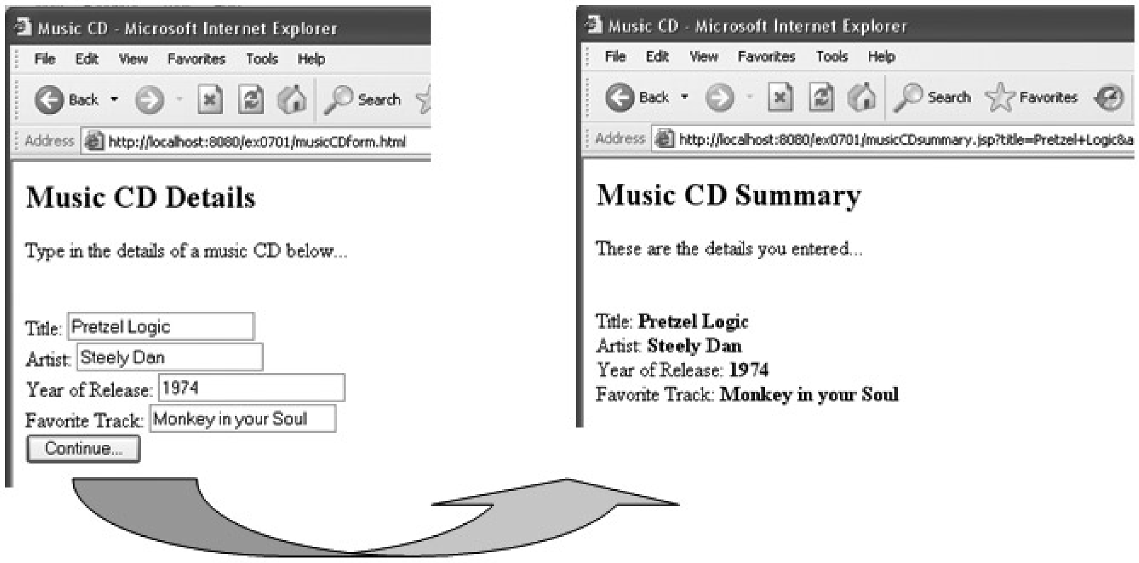
11. The other two request parameters have different names from their corresponding bean properties: Request parameter "year" must map on to bean property "yearOfRelease," whereas "track" needs to map on to "favoriteTrack." So use two invocations of the `<jsp:setProperty>` standard action to achieve this mapping (you'll need to set the `property` *and* `param` attributes).
12. Finally, display the four properties on the page, using four separate occurrences of `<jsp:getProperty>`.

Deploy and Run the Application

13. Create a WAR file that contains the contents of ex0701, and deploy this to your web server. Start the web server if it has not started already.
14. Use your browser to request musicCDform.html, with a URL such as

```
http://localhost:8080/ex0701/musicCDform.html
```

15. Enter some details (note that the year field must be numeric), click the button to submit the form, and check that the output is correct on musicCDsummary.jsp. The following illustration shows the screen flow for the solution.


Music CD - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address

http://localhost:8080/ex0701/musicCDform.html

Music CD Details

Type in the details of a music CD below...

Title: Pretzel Logic
Artist: Steely Dan
Year of Release: 1974
Favorite Track: Monkey in your Soul
Continue...

http://localhost:8080/ex0701/musicCDsummary.jsp?title=Pretzel+Logic&a

Music CD Summary

These are the details you entered...

Title: **Pretzel Logic**
Artist: **Steely Dan**
Year of Release: **1974**
Favorite Track: **Monkey in your Soul**

CERTIFICATION OBJECTIVE

Dispatching Mechanisms (Exam Objectives 6.7 and 8.2)

Given a specific design goal for including a JSP segment in another page, write the JSP code that uses the most appropriate inclusion mechanism (the include directive or the jsp:include standard action).

Given a design goal, create a code snippet using the following standard actions: jsp:include, jsp:forward, and jsp:param.

In this section of the chapter, we're going to explore three more standard actions. These give you the equivalent of the `forward` and `include` RequestDispatcher mechanisms, which we explored in Chapter 3 in servlet code. The two main standard actions are called `<jsp:include>` and `<jsp:forward>`, which serve to include content into a JSP or forward on to another resource altogether (JSP, servlet, or any resource that can be described with a URL). We'll see how these are a bit easier to set up than the coding equivalent, and also how there are one or two differences between the standard actions and a naked RequestDispatcher.

We'll also look at the `<jsp:param>` standard action and see how this can be embedded into either of `<jsp:include>` and `<jsp:forward>`. It provides an easy way to graft on additional parameters to the request.

You'll recall that we encountered the include directive (`<%@ include file="..." %>`) in the last chapter. So we devote some time in this chapter to understanding the differences between this directive and the `<jsp:include>` standard action. That way we can tick off both the exam objectives above for the price of one section in the book.

Including

The standard action `<jsp:include>` can be used to include the response from another file within your JSP page output. You specify the file whose response should be included with the page attribute, like this:

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

The file whose response should be included has to reside somewhere in your web application but doesn't have to be present until the page is actually requested.

on the job

You might use `<jsp:include>` to include files that don't exist at the point where you deploy your including pages. This could be for a number of reasons: Perhaps the included files are produced as output from other systems and are uploaded to your web application directory structure only at scheduled intervals. This doesn't stop you from precompiling your including JSPs when you deploy them, however. There's no check on the existence of the page specified in `<jsp:include>` during the translation phase. Obviously, you'll get a run-time error if you let your users access JSPs that try to include a page that doesn't exist—it's then up to you to introduce controls that prevent access to the including JSP until the files needed for inclusion are actually present in your web application directory structure.

The value for the *page* attribute is a URL pointing to a resource within the current web application (you can't go outside the web application with `<jsp:include>`). The URL used follows rules similar to those we've seen many times elsewhere:

- If the page URL begins with a slash, this is interpreted as starting from the context directory for the web application.
- If the page URL doesn't begin with a forward slash, this is interpreted as relative to the directory containing the including page.

Any kind of file can be the target of the page attribute. It's typical to target other JSP pages but by no means mandatory—you can include any file of any MIME type (though bear in mind that if this isn't compatible with the MIME type for the rest of the response, you may well run into run-time issues).

A delightful feature of `<jsp:include>` is that it runs at request time. This may not sound like much, but what it means is that the value for the *page* attribute can be an expression embedded within the standard action. So code like this is perfectly legitimate:

```
<jsp:include page='<%= request.getParameter("thePage") %>' />
```

You can see from this that I can nominate the page whose response should be included from a parameter value passed in my request. This gives you a great deal of flexibility.

The `<jsp:include>` standard action has a second (optional) attribute, *flush*. This can have the values "true" or "false," and if you leave off the attribute, the default is "false." To understand this, recall that JSP page output is buffered as a rule—not immediately committed to the response. If you set the value to "true,"

```
<jsp:include page="aPageToInclude.html" flush="true" />
```

this has the effect of flushing the buffer in the including page (i.e., committing the response so far) before anything is done about including the target page.

Even if you set the *flush* attribute to "false," and both the including and included page have unfilled, unflushed buffers, there are still restrictions on included pages. Included pages can't do anything to the response header—in just the same way that servlets can't if anything has been written to the response. The assumption is that somewhere along the chain to the included page, some part of the response has been written (and don't forget that you may well have a long chain: through several filters and servlets before you get to the including, and then the included, JSP pages). So code like the following:

```
<% response.addCookie(myCookie); %>
```

Or:

```
<% response.setHeader("Date", utcFormatDate); %>
```

is simply ignored in the *included* JSP page. This is no different from the world of servlet code: A servlet that has been included from another servlet by a RequestDispatcher object is treated in the same way.

`<jsp:include>` vs. `<%@ include %>`

It's very hard to talk about `<jsp:include>` without comparing it with the include directive, so let's not put that off anymore. After all, it's an exam objective in its own right!

When I introduced `<jsp:include>`, you may have noticed the pedantic phrase "include the response from another file" several times. Why didn't I just say "include the file"? Because that's not quite true—not in the sense we mean, for example, when we talk about the include directive (`<%@ include file="..." %>`). Let's say I have a JSP page a.jsp that includes b.jsp with the standard action. Neither has been translated yet. Figure 7-2 shows what happens when I request a.jsp for the first time.

You can see from Figure 7-2 that the two JSPs a.jsp and b.jsp remain independent and that b.jsp provides a service to a.jsp. a.jsp requests a response from b.jsp and incorporates this in its own output. The situation is fundamentally different with the `include` directive. Figure 7-3 shows what happens when a.jsp includes b.jsp with the `include` directive.

FIGURE 7-2

a.jsp Using `<jsp:include>` to Include b.jsp

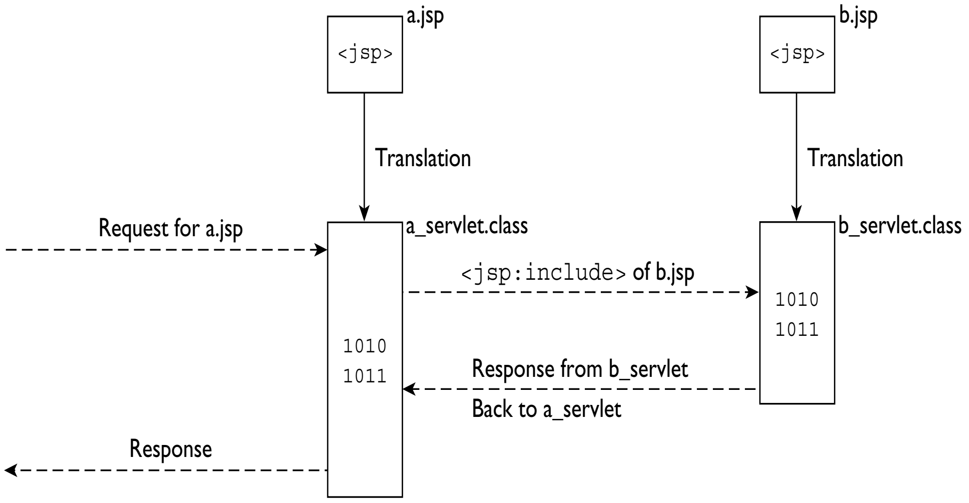


FIGURE 7-3

a.jsp Using `<%@ include %>` to Include b.jsp

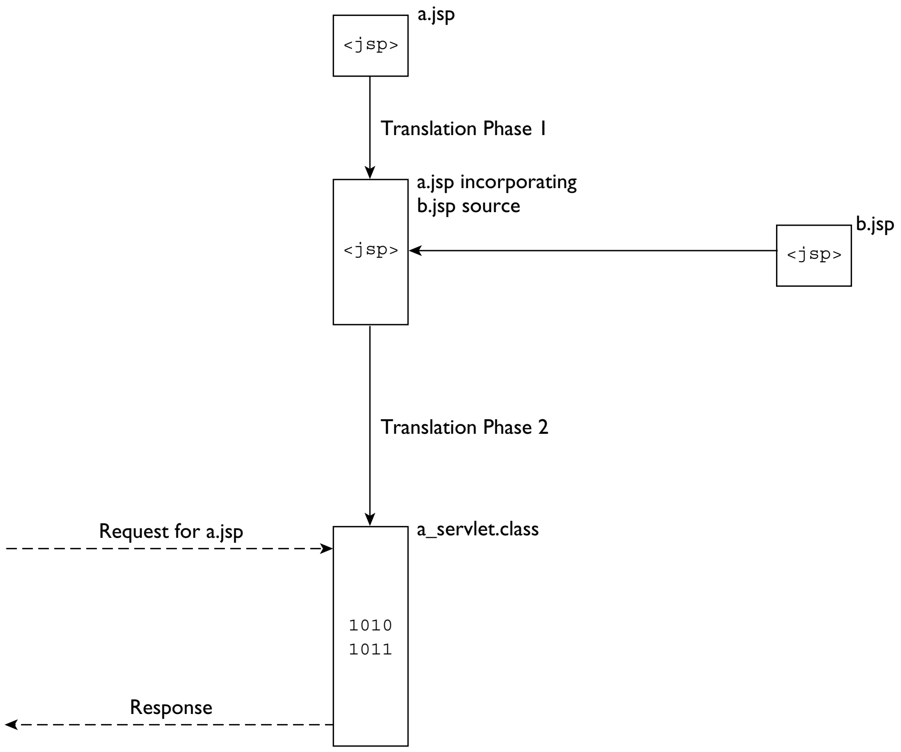


Figure 7-3 shows how the lines of b.jsp are first incorporated into a.jsp. It's as if a.jsp is a composite of its own page source and b.jsp's as well. Only after this has happened does the translation to generated servlet occur. This has some interesting consequences. Let's look at an example—and please, please note that this is to illustrate a point: Don't write your JSPs this way! Suppose that you declare a local variable in one JSP (we'll call it declaration.jsp), then use that local variable in another JSP (display.jsp) that includes declaration.jsp. Here's the code for each:

```
<!-- declaration.jsp -->
<% int aNumber = (int) (Math.random() * 100); %>
```

So in declaration.jsp, we have a local variable called *aNumber* that is initialized to a random value between 0 and 99. Here's display.jsp, which uses an expression to show the random number on the page output:

```
<!-- display.jsp -->
<%@ include file="declaration.jsp" %>
Think of a number: <%= aNumber %>
```

This works absolutely fine (despite coming with a massive design health warning!). The `include` directive causes the amalgamation of the JSPs—the result is a composite JSP looking like this:

```
<!-- display.jsp -->
<!-- declaration.jsp ->
<% int aNumber = (int) (Math.random() * 100); %>
Think of a number: <%= aNumber %>
```

This is the whole "translation unit." If neither JSP page has been accessed, and our container translates and compiles only when a JSP is requested, then a request to display.jsp will result in only one generated servlet for your web application that represents the composite page source. So what happens if we change display.jsp to use a `<jsp:include>` standard action?

```
<!-- display.jsp -->
<jsp:include page="declaration.jsp" />
Think of a number: <%= aNumber %>
```

Result: misery. The functionality of `<jsp:include>` isn't invoked until post-translation, and display.jsp won't get past translation. If you try to request it, you're likely to get an HTTP 500 error accompanied by a stack trace that informs you that the local variable *aNumber* is used but not declared. So if you're using

TABLE 7-1

Comparing and Contrasting the Two JSP Inclusion Mechanisms

`<jsp:include>` Standard Action	`<%@ include %>` Directive
Attributes: page (and flush)	Attribute: file
Page attribute accepts relative and absolute URLs.	File attribute accepts relative and absolute URLs.
Response from target page included at *request* time.	Target file included during *translation* phase.
Target page to include can be *soft-coded* as an expression.	Target file must be a *hard-coded* literal value.
Can execute *conditionally* in the middle of page logic.	Will be processed *unconditionally*—can't be embedded in page logic.
Target page doesn't have to exist until request time.	Target file must exist at translation time.
Always includes the latest version of the target page.	Does not necessarily include the latest version of the target file: depends on your container (not mandated by the JSP specification).

`<jsp:include>`, each of your JSPs must be able to "stand alone"—at least in translation terms.

With `<jsp:include>`, you are always guaranteed to get the latest versions of responses from included files because the included files are still accessed at run time. They have to be there to complete the picture. This isn't necessarily the case for files that are included through the `include` directive. After all, once a file has been incorporated through the `include` directive, the resulting composite servlet is whole and complete. This leads to an interesting question: If a file included by the `include` directive is updated, will the JSP container spot the fact and re-do the file inclusion when the JSP doing the including is next accessed? The answer is that the JSP specification recommends that this should happen, but doesn't say that it has to be this way.

In conclusion, take a look at Table 7-1, which summarizes the differences between the two approaches. Do some experiments so you're comfortable with the difference—it's a favorite topic on the exam!

Forwarding

After `<jsp:include>`, `<jsp:forward>` is moderately straightforward. As the name implies, the purpose of this standard action is to forward processing to another resource within the web application. There is only one mandatory attribute, which (as with `<jsp:include>`) is `page="URL."` Consider the following example—a complete page source called doThis.jsp:

```
<!- doThis.jsp -->
<p>You won't see this in the response</p>
<jsp:forward page="doThisInstead.jsp" />
<p>You won't see this either</p>
<% /* Will the following line of code be executed? */
session.setAttribute("doThis", "isDone"); %>
```

The effect of accessing doThis.jsp is to transfer responsibility for the output to doThisInstead.jsp. The template text before the `<jsp:forward>` action is effectively ignored, for anything that doThis.jsp writes to the output buffer is cleared. What happens after the `<jsp:forward>`, though? Were this a hand-coded servlet, the code following a `RequestDispatcher.forward()` method would still be executed (if we wanted it to be). But this isn't a hand-coded servlet: The corresponding servlet code is generated by the JSP container, as we well know. And to respect the fact that the JSP specification says that a "`<jsp:forward>` effectively terminates the current page," the reference implementation—Tomcat—returns from the `_jspService()` method. Consequently, the last line of page source in doThis.jsp—which sets an attribute in session scope—will not be executed, even though it has nothing to do with writing output to the response.

There's no *flush* attribute as there is for `<jsp:include>`—instead, certain things have to be true about the state of the response for the `<jsp:forward>` to be processed successfully. It all comes down to the thorny question of whether any part of the response has already been committed—in other words, written back to the client. Responses are considered uncommitted when anything written to them already is still in the memory buffer and the buffer has never been flushed. So a `<jsp:forward>` won't work if

- There is no buffer (in a JSP, this can be achieved with a `page` directive, setting the `buffer` attribute to "`none`"), and even one character has been written to the response.
- The buffer has been explicitly flushed (`response.flushBuffer()`).
- The buffer has been automatically flushed on filling up (in a JSP, this will happen by default—see the `page` directive attribute *autoFlush* for more information).

If you try to do any of the above, you'll get an IllegalStateException.

Parameters

Whether you are including or forwarding, you can add in additional parameters to the request. For this, you use the `<jsp:param>` standard action and include it in the body of a `<jsp:include>` or `<jsp:forward>`. This is the only reason for including a body in these two standard actions, which as you have probably noticed have been expressed as self-closing tags without any body up until now.

There are three important things to take account of when you add parameters into a request using the `<jsp:param>` standard action:

1. They only last for the duration of the `include` or `forward`. Once you're back in the including or forwarding JSP page, the parameters disappear.
2. They don't replace existing parameters of the same name—they merely augment the list of values. (Recall that parameters—unlike attributes—can have multiple values for the same name.)
3. When they do augment the list of values, their values come at the front of the list.

To illustrate these points, suppose that you make the following HTTP request to forwarder.jsp:

```
http://localhost:8080/examp0702/forwarder.jsp?animaltypes=dog
```

forwarder.jsp does nothing other than forward the request with a supplementary value for the animaltypes request parameter, so

```
<jsp:forward page="animalHouse.jsp">
  <jsp:param name="animaltypes" value="cat" />
</jsp:forward>
```

Now, if the forwarded-to JSP page, animalHouse.jsp, has code like the following:

```
<% String[] a = request.getParameterValues("animalTypes");
for (int i = 0; i < a.length; i++) {
    out.write(a[i] + ";");
}
%>
```

then the output will be

```
cat;dog;
```

exam watch

The value for `<jsp:param>`'s value attribute can come from a run-time expression—for example: `value="<%=calcValue%>."` That's not the case for the name attribute, which must be a literal in the page source.

You can see from this that the original parameter value (dog) is not lost but that the value added with `<jsp:param>` (cat) has taken precedence, and now comes first in the list.

Meanwhile, once you return to forwarder.jsp, you will find the request parameter animalTypes has reverted to having only the one value, dog.

All the points described above hold just as true for `<jsp:param>` standard actions embedded in the body of a `<jsp:include>` standard action.

EXERCISE 7-2

ON THE CD

Dynamic Inclusion

This exercise shows you how to include a JSP dynamically from a request parameter. The idea is to have a small application that chooses and displays a poem at random. Each poem is kept in a separate static HTML page of its own. There are two JSP pages involved. One, "poemOfTheDay.jsp," receives the name of one of these static web pages as a request parameter and includes this into its body. The other JSP page, "choosePoemOfTheDay.jsp," chooses one of the available filenames at random and forwards it to "poemOfTheDay.jsp," setting up the chosen filename within the body of the `<jsp:forward>` standard action.

Create the usual web application directory structure under a directory called ex0702, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch07/ex0702.war—check there if you get stuck.

Write Poems

1. Create a number of HTML pages directly in ex0702 (three is a good number, but make as many as you like). Call these poem1.html, poem2.html, and so on.
2. In each page, include the title (between `<h1>` tags) and the text of any poem you like (between `<pre>` tags). You can write your own original poems as a diversion from exam cramming, but I'd recommend doing what I did:

copying and pasting a few of your favorites from the Web. Of course, if poems don't appeal, you can use any text you like—as long as you can tell your HTML pages apart.

Create the Including JSP Page Source

3. Create a file called poemOfTheDay.jsp in ex0702.
4. Put in template text for a skeleton web page—`<html>`, `<head>`, `<title>`, and `<body>` tags (and the corresponding end tags).
5. In the body, place a `<jsp:include>` standard action. Use an expression as the value for the page, taken from a request parameter called *poem*.

Create the Forwarding JSP Page Source

6. Create a file called choosePoemOfTheDay.jsp in ex0702.
7. Have a scriptlet at the beginning of the file that sets up a String array, and initialize this with the filenames of each of your poem HTML pages. (If you want more of a challenge, use `ServletContext.getRealPath()` and recover the names of any filenames beginning with the characters "poem" in the context directory. From this, load a String array or collection class: harder certainly!)
8. In the same scriptlet, set up a local variable to hold the name of one of the files. Use `Math.random()` as the basis for picking a name randomly from the String array (or collection class).
9. Set up a `<jsp:forward>` standard action following the scriptlet, to forward to the poemOfTheDay.jsp page. Ensure that this standard action has a body (and so has both a beginning and end tag; it should not be a self-closing tag).
10. In the body of the <jsp:forward> standard action, place a `<jsp:param>` action. Name the parameter poem, and set the value of the parameter using an expression—this should use the local variable set up in step 8.

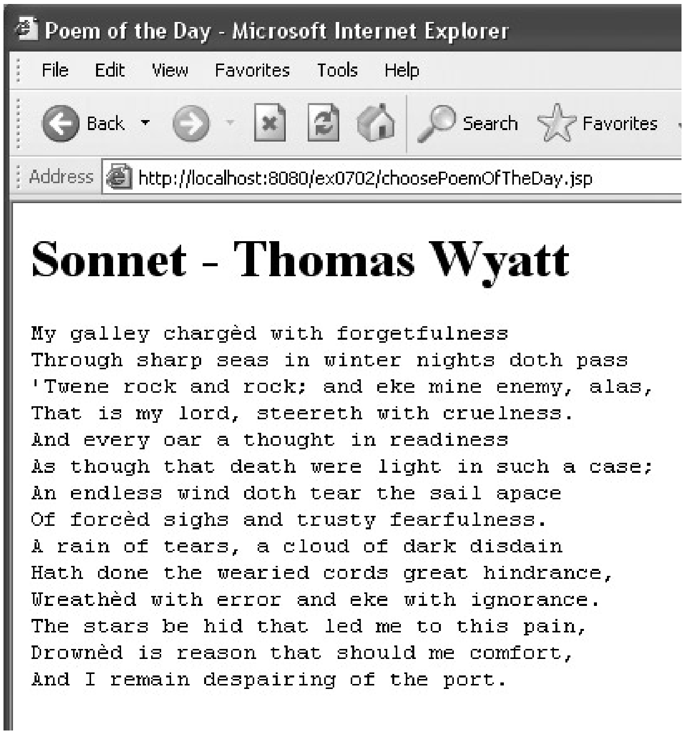
Deploy and Run the Application

11. Create a WAR file that contains the contents of ex0702, and deploy this to your web server. Start the web server if it has not started already.

12. Use your browser to request musicCDform.html, with a URL such as

    ```
    http://localhost:8080/ex0702/choosePoemOfTheDay.jsp
    ```

13. With luck, one of your poems will be displayed in the browser. Press the refresh button a few times to check that other poems are selected randomly (another extra challenge is to consider how you might prevent the same poem from being selected twice in succession). Here's some sample output from the solution code.


Poem of the Day - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
http://localhost:8080/ex0702/choosePoemOfTheDay.jsp

Sonnet - Thomas Wyatt

```
My galley chargèd with forgetfulness
Through sharp seas in winter nights doth pass
'Twene rock and rock; and eke mine enemy, alas,
That is my lord, steereth with cruelness.
And every oar a thought in readiness
As though that death were light in such a case;
An endless wind doth tear the sail apace
Of forcèd sighs and trusty fearfulness.
A rain of tears, a cloud of dark disdain
Hath done the wearied cords great hindrance,
Wreathèd with error and eke with ignorance.
The stars be hid that led me to this pain,
Drownèd is reason that should me comfort,
And I remain despairing of the port.
```

CERTIFICATION OBJECTIVE

JSPs in XML (Exam Objective 6.3)

Write a JSP Document (XML-based document) that uses the correct syntax.

This innocently short objective encompasses a large number of things you need to know. What is a JSP document? Well, it's JSP page source that's written in XML. Quite often (but not by any means always), you use a JSP document to produce XML as well.

XML is a big and scary topic in its own right. Fortunately, only a basic knowledge is required for the SCWCD exam. So if you're relatively new to XML, don't worry. In any case, you've already handled plenty of XML in the course of this book. The standard actions you covered in the previous section of this chapter are XML. And whereas HTML isn't necessarily XML, all the examples and exercises in this book have been using an XML-compliant version called XHTML. So although there's no room to turn this part of the chapter into a full-blown XML tutorial, I'll be taking a "least you need to know" approach. Fortunately, full-blown XML tutorials litter the Web, as do excellent books on the topic.

JSP documents also have more facets than can be covered in just one section of one chapter. You'll meet the basics in this section, and from this point in the book onward, most of the JSP examples and exercises (and quite a few of the questions) will use XML syntax.

XML for JSPs

Why are JSPs moving to XML-style page source at all? After all, they have a perfectly viable syntax all their own. Likely reasons include (but are not limited to) the following list:

- It makes a lot of sense if you are in the business of producing XML anyway. You've already encountered the idea that template text in a JSP page isn't limited to HTML, and XML is the most usual alternative. If you have an XML file you want to produce, it can immediately become the template text for a piece of JSP Page Source—all that remains is to mark it up with some more XML for the dynamic parts.
- You can check that your page source is valid in XML terms, using proper XML validators (and that's something your JSP container does during the translation phase for JSP documents).

- If you use XML-authoring tools (such as XML Spy), then those same tools can handle the production and validation of your JSP page source as well as other XML files you write.
- Arguably, XML-style source is easier to write and read than a mishmash of template text and Java language. So XML might mark a step along the way toward production of JSP page source by nonprogrammers.
- You certainly get the impression from the JSP specification that Sun would like page authors to move in an XML direction—that in itself might be a good reason to make the switch. While support for the `<% .. %>` way of doing things is bound to last for a long while, you might find yourself excluded from newer, trendier tool developments if you stick doggedly to the old-style syntax.

Given that you're persuaded that XML-style JSPs are a good thing, let's make a few top-level statements about XML itself. I won't assume any previous knowledge, though the chances are you've heard this before.

- XML is a tag language. Its structure is often compared to HTML: Both contain opening and closing tags. But while HTML is narrowly focused on marking up text, XML is much more general purpose. XML can be used for marking up text (as in XHTML), but it has a pretty much infinite set of other possible uses.
- You can define your own tags in XML, calling them whatever you like. You just have to make sure that for every opening tag of a particular name, you have a balancing closing tag:

```
<painting>
  <artist>Leonardo da Vinci</artist>
  <title>Mona Lisa</title>
  <location museum="Louvres"></location>
</painting>
```

- There may be some data content between the opening and closing tags ("Mona Lisa" for the `<title>` tag above), or simply other tags (`<painting>`), and sometimes nothing at all (`<location>`). The area between the opening and closing tags is referred to as the "body."
- Tag names can have a prefix, separated from the main name by a colon. We met that in standard actions, such as `<`**`jsp`**`:useBean>`. The prefix ties the tag to a "namespace," which you can think of as a signpost with information about the tag.

- Tags can have attributes. Again, we've seen this in standard actions. Attributes are included in the opening tag, and they form name/value pairs, in the form *name*=*"value"* (or with single quotes—*name*=*'value'*). You'll recall `<jsp:getProperty>`:

```
<jsp:getProperty name="beanName" property="beanProperty" />
```

- The `<jsp:getProperty>` example shows us another common XML feature. Sometimes it doesn't make a lot of sense to have a closing tag, because there's no appropriate data content to insert in the tag body. Under those circumstances, an opening tag can be "self-closing" by including a slash before the final angle bracket: />.
- Tags must be properly nested. Here is a rogue version of the painting example—you can see that the closing tag `</artist>` comes within the body of the `<title>` tag, which just won't do:

```
<painting>
  <artist>Leonardo da Vinci<title></artist>Mona Lisa</title>
  <location museum="Louvres"></location>
</painting>
```

- There must be one root tag at the top of the document. For the deployment descriptor, web.xml, the root tag (as we've seen throughout) is `<web-app>`.

Of course, there's a lot more to XML than that, but these simple rules will get you most of the way there. All we need to consider now is how XML exactly applies in JSP documents.

XML-Friendly Syntax

When we talk about a JSP document, we mean a JSP page source that obeys all the rules of XML. However, the pseudo-tag-like structure of some JSP syntax will wreak havoc with any XML validator. Certain bits of syntax have to go—especially arbitrary angle brackets. Any XML parsing code sees an angle bracket as the beginning or end of a tag, so it won't know what to make of `<%`, `<%!`, `<%@`, `<%=`, `<%--`, `--%>`, and `%>`. All of these have to be replaced. So the JSP specification provides an XML equivalent for all of the above. In the main, these look like standard actions, behave like standard actions, and are standard actions. They extend the set of those we have already looked at, such as `<jsp:include>`. There are only one or two syntax differences implemented

TABLE 7-2 XML Equivalents for JSP Syntax

Scripting Elements	Original JSP Syntax	XML Syntax
Scriplets	`<% ... %>`	`<jsp:scriptlet>... </jsp:scriptlet>`
Expressions	`<%= ... %>`	`<jsp:expression>... </jsp:expression>`
Declarations	`<%! ... %>`	`<jsp:declaration>... </jsp:declaration>`
Directives		
page	`<%@ page attr="value" %>`	`<jsp:directive.page attr="value" />`
include	`<%@ include file="abc.txt" %>`	`<jsp:directive.include file="abc.txt" />`
taglib	`<%@ taglib prefix= "abc" uri="..." %>`	`xmlns:abc="..."`
Comments		
Exclude from translation (and output)	`<%-- ... --%>`	`<!-- ... -->`
Include HTML comment in HTML output	`<!-- ... -->`	`&lt;!-- ... --&gt;`

without the use of standard actions, however. Table 7-2 lists the different kinds of JSP syntax and shows the original syntax alongside its XML-friendly equivalent.

Scripting Elements

Note that this change to XML syntax doesn't—in itself—mean abandoning Java code in your JSP pages. All we're doing at the moment is making the XML well-formed, and for the most part, any old Java code can be dumped into the body of an XML element. Let's look at a few short examples.

Here's a scriptlet in the old style of syntax:

```
<% String s;
  s = request.getParameter("user"); %>
```

Here it is again as the body of a standard action:

```
<jsp:scriptlet>String s;
         s = request.getParameter("user"); </jsp:scriptlet>
```

There's nothing remotely difficult here—just a straight swap of opening markers (`<jsp:scriptlet> for <%`) and closing markers (`</jsp:scriptlet> for %>`). The contents of the scriptlet remain as before: valid Java syntax (with a slight modification we'll soon see). You can't embed any other sort of tag within the body of the `<jsp:scriptlet>` tag.

Declarations are no different. Again, here's a declaration before:

```
<%! public void jspInit() {
      System.out.println("My JSP is initialized");
    } %>
```

And the same declaration after:

```
<jsp:declaration> public void jspInit() {
              System.out.println("My JSP is initialized");
            } </jsp:declaration>
```

Note that the inclusion of indentation and white space is entirely my own choice, just as it is in normal Java source code. It will come as no surprise that a converted expression follows a similar pattern. Here's one before:

```
<%= session.getAttribute("user") %>
```

And one after XML-ification:

```
<jsp:expression>session.getAttribute("user")</jsp:expression>
```

Again, the choice of white space is mine. This is equally valid:

```
<jsp:expression> session.getAttribute("user") </jsp:expression>
```

As is this:

```
<jsp:expression>
  session.getAttribute("user")
</jsp:expression>
```

However, there are some things within the Java language itself that are anathema to XML validators. Take the following source code, showing the beginning of a **for** loop in a scriptlet:

```
<jsp:scriptlet>for (int i = 0; i < 10; i++) { </jsp:scriptlet>
```

The "less than" sign (<) looks like the beginning of an opening or closing tag, and an XML validator will assuredly treat it as such. However, it's difficult to write Java code without using < anywhere!

You have two options to deal with this. The first is to escape the source code, using what's referred to in XML as an "entity"—beginning with an ampersand (`&`) and ending in a semicolon (`;`). If you've written any amount of HTML, you'll recognize this device. The offensive < sign is replaced with the entity `&lt;`. It doesn't make your Java code very readable, but it'll get through the XML validation process—and your JSP container will turn it back into valid code for your generated servlet. Here's how it looks in our example:

```
<jsp:scriptlet>for (int i = 0; i &lt; 10; i++) { </jsp:scriptlet>
```

You should treat the > sign in the same way, replacing it with `&gt;` when it comes up in your code.

The second option is to mark up the offensive part as XML character data. This is part of an XML file that the XML validator treats as off-limits—it lets the characters stand just as they are. The syntax for this is more intrusive than entities, even though our < sign stays intact. Here's how this second solution might look:

```
<jsp:scriptlet>for (int i = 20; i <![CDATA[<]]> 30; i ++) {</jsp:scriptlet>
```

In this case, it's pretty hard to find the < sign in the middle buried in the middle of the syntax. A variation on this approach is to demarcate the whole body of the scriptlet as character data:

```
<jsp:scriptlet><![CDATA[for (int i = 10; i < 20; i ++) { ]]></jsp:scriptlet>
```

This at least keeps the Java code more integral, and an entire longer scriptlet or declaration consisting of multiple statements can be "wrappered" in this way.

Directives

Directives are dealt with in much the same way as scripting elements, by substituting standard actions.

There's a JSP directive for page directives called `<jsp:page.directive ... />`, which is a substitute for `<%@ page ... %>`. The syntax represented by the ellipsis (. . .) is identical in both cases. So

```
<%@ page import="java.util.*, a.b.MyClass" %>
```

becomes

```
<jsp:directive.page import="java.util.*, a.b.MyClass" />
```

Because the attributes (such as `import`) within the original directive follow XML syntax, they can be transferred directly into the XML tag. Note that the XML tag closes itself—there's no requirement for a directive to have a body. Other than that, the functionality is no different from the JSP syntax original.

The `include` directive follows exactly the same pattern, and it works identically in its XML form. So

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

becomes

```
<jsp:directive.include file="myFile.html" />
```

The one directive that doesn't follow this norm is the `taglib` directive for referencing tag libraries. This uses a namespace instead of a standard action to define a tag library in use in the JSP document. We'll soon learn more about namespaces, and a lot more about tag libraries in Chapter 8, where we revisit the `taglib` directive in both its guises: JSP and XML syntax.

Comments

Finally from Table 7-2, what happens about comments? You'll recall that JSP syntax allows for two styles—one that removes source text from the translation phase entirely (`<%-- not translated --%>`) and the other that causes an HTML-style comment to be buried in the output (`<!-- The user won't see this in a normal browser, but will when viewing source. -->`).

The first style of comment is XML-unfriendly, so it can't be used in a JSP document. The second (HTML-style) comment is, in fact, an XML-style comment. So it's fine to carry on using the second style, except that it acts slightly differently. It's like this:

```
<!-- In a JSP document, this style of comment will NOT be included
in the generated output; it's ignored by translation. -->
```

This begs the question: What if I am generating HTML-output from my JSP document and want to include an HTML-style comment? Because these are hijacked by XML, they'll never appear! Escape conventions come to our rescue. If you hide the < and > signs with their entity equivalents, you'll get your HTML comment:

```
&lt;!-- This comment will appear in the HTML output. --&gt;
```

exam watch

There is a standard action called `<jsp:text>` that exists solely to dig you out of trouble when writing JSPs in XML syntax. More or less all the content you put into an XML document (aside from white space) must exist in the body of a tag. Of course, you're always in the body of a tag in that there's a root tag that encloses everything. However, some tags (normally including the root tag) don't allow any content in their bodies—only other tags. If you have some content that needs to be placed in the "no-man's land" of a bodiless tag, then wrap it up with `<jsp:text> ... </jsp:text>`.

Namespaces

The conversions to JSP document syntax that we've seen are easy enough, mostly involving the substitution of XML standard actions for their traditional counterparts. However, whereas standard JSP syntax takes for granted that standard actions are simply available in your JSP page source, XML syntax demands more than that. If your tags use a prefix—as all standard actions do—that prefix must be associated with something that XML terms a "namespace."

Any opening tag within XML can define a namespace. Here's a very short JSP document example that includes a directive to specify the MIME type of the output:

```
<html>
<head><title>Namespaces</title></head>
<jsp:directive.page xmlns:jsp="http://java.sun.com/JSP/Page"
contentType="text/html" />
<body><h1>Namespace Demonstration</h1></body>
</html>
```

You can see that the `<jsp:directive.page>` element now contains an additional attribute:

```
xmlns:jsp="http://java.sun.com/JSP/Page"
```

The `xmlns` stands for XML namespace (unsurprisingly) and—after the colon—uses a name/value pair. The name (`jsp`) is the prefix you use for any elements belonging to this namespace—such as `<jsp:directive.page>`. The value is—more often than not—a URL, though it can be any text at all. Sometimes, the URLs actually correspond to pages on the Internet. Mostly—and http://java.sun.com/JSP/Page is a case in point—they don't. There's no technical need for the resource at the end of the URL to exist; a URL is often used because it has a good chance of being unique. So when you see the namespace http://java.sun.com/JSP/Page, you can safely assume that this is uniquely associated with a set of elements that have to do with JavaServer Page standard actions.

In our example above, the namespace is associated only with `<jsp:directive.page>`. This is because namespaces apply only to the element in which they are defined, plus any elements contained within that element. Because `<jsp:directive.page>` can't contain other elements, the namespace applies only to that element. So it's much more usual to place your namespace declarations farther up the XML document's containment ladder—usually, in fact, right in the root element. Here is the same example again with the namespace transferred to `<html>`, the root element for XHTML documents:

```
<html xmlns:jsp="http://java.sun.com/JSP/Page" >
<head><title>Namespaces</title></head>
<jsp:directive.page contentType="text/html" />
<body><h1>Namespace Demonstration</h1></body>
</html>
```

Now any standard action can be used anywhere in the document without repeating the namespace, for it's available throughout—the prefix `jsp`: is sufficient.

XML and the JSP Container

What tells the JSP container that it's dealing with a JSP document, as opposed to a page in normal JSP syntax? You might think that a page written in bona fide XML is a JSP document and will be treated as such, but actually, it isn't. The page will continue to work, but the JSP container is likely to treat it as a standard syntax page. You can include as much or as little of the XML syntax in a normal JSP page as you like—this is to encourage you to migrate your JSP pages to XML syntax at a pace to suit. Take this example:

```
<% String s = "Mixed syntax"; %>
<jsp:expression>s</jsp:expression>
```

This is perfectly viable JSP page syntax, though it has the makings of a maintenance nightmare—why use two syntaxes when you can stick to one?

There are three approaches that identify a page as a JSP document:

1. Ensure that your web application deployment descriptor web.xml is at version level 2.4 and that the file with your JSP page source has the extension `.jspx`.
2. Ensure that your web application deployment descriptor web.xml is at version level 2.4, and include some appropriate settings in deployment descriptor's `<jsp-config>` element (we'll see what these are in a moment).
3. Enclose your page source with the root element `<jsp:root>`. This element is backward-compatible with previous versions of the JSP container, so it doesn't rely on a particular version level for web.xml.

Method 1 is certainly the most straightforward. Assuming that your deployment descriptor is at version level 2.4 (and why wouldn't it be?), you should suffix your JSP documents with .jspx instead of .jsp. Method 2 is still straightforward. You just need to know how to configure the relevant element in web.xml. Here's an example configuration:

```
<jsp-config>
  <jsp-property-group>
  <url-pattern>/jspx/*</url-pattern>
  <is-xml>true</is-xml>
  </jsp-property-group>
</jsp-config>
```

This says that any for any file accessed with a URL ending in /jspx/anythingatall.any within the web application, treat this as a JSP document. The `<is-xml>` element takes two valid values: true (treat these as JSP documents with XML syntax) or false (treat these documents as JSP pages with standard syntax). The `<url-pattern>` element works in just the same way we saw within the `<servlet-mapping>` element way back in Chapter 2.

exam watch

Notice that the element `<jsp-property-group>` nests inside `<jsp-config>` and that the elements which do the work (such as `<url-pattern>` and `<is-xml>`) nest inside `<jsp-property-group>`.

This leaves method 3, which is to make `<jsp:root>` your document's root element, not forgetting to include the namespace in the opening tag:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page">
```

This approach can have advantages if you need to remain compatible with older containers, or older applications in newer containers—this is how JSP documents were identified in the past, at JSP specification level 1.2. An older-style web.xml won't matter. Even if your application and container are bang up-to-date, `<jsp:root>` can be handy if your source files can't have the `.jspx` extension for some reason, and if their URL patterns are too diverse to warrant defining inside the `<jsp-config>` element.

on the job

We know now that a JSP document is written in XML. What dictates what it outputs? Well, by default, a JSP document wants to produce XML. This is regardless of the MIME type that you set with `<jsp:directive.page contentType="..." />`. If you do nothing, an XML header statement appears at the very beginning of the page output, looking like this: `<?xml version="1.0" encoding="UTF-8"?>`. If what you're producing is not XML, you really ought to suppress this. There are a couple of approaches:

- ***Use `<jsp:root>` as your root element. This suppresses the XML header statement by default (if you're using `<jsp:root>` for some other reason, there are ways to retain the XML header statement if you actually need it).***
- ***Include a `<jsp:output>` element as follows: `<jsp:output omit-xml-declaration="true" />`.***

There is quite a bit more to `<jsp:output>` than this one attribute—to find out more, take a look at JSP Specification section 5-16.

EXERCISE 7-3

ON THE CD

JSP Syntax to XML Syntax

This exercise differs a little from most of its predecessors, for you deliberately start with the solution code. Your mission is to take a moderately complex JSP page, written in JSP syntax, and convert this to XML syntax. The page works in combination with a servlet and a JavaBean. The servlet receives the name of a comma-separated values file and parses the contents of this, placing the results in the JavaBean. The JSP page uses the JavaBean to display the results in an HTML table.

Any comma-separated values (.csv) file will do. There is one provided called Timesheet.csv, but you can supply one of your own. The expectation is that the first row of the file contains header information—here's the first row of Timesheet.csv:

```
Date,Start Time,End Time,Duration,Description,Code
```

And each subsequent row contains data corresponding to the heading fields. Here's an example data row from Timesheet.csv:

```
Mon-30-Jun,09:00,11:00,2:00,Certification article,ARTICLE4
```

Having checked you can run the application, you'll take the regular JSP version of the file and duplicate that *in situ* to a new file that will be a JSP document. You'll work through this document, making alterations to remove source that works only in JSP syntax terms. On the way, you'll try accessing the document, and observe the syntax errors you get. The result will be a genuine JSP document that works as the solution code does.

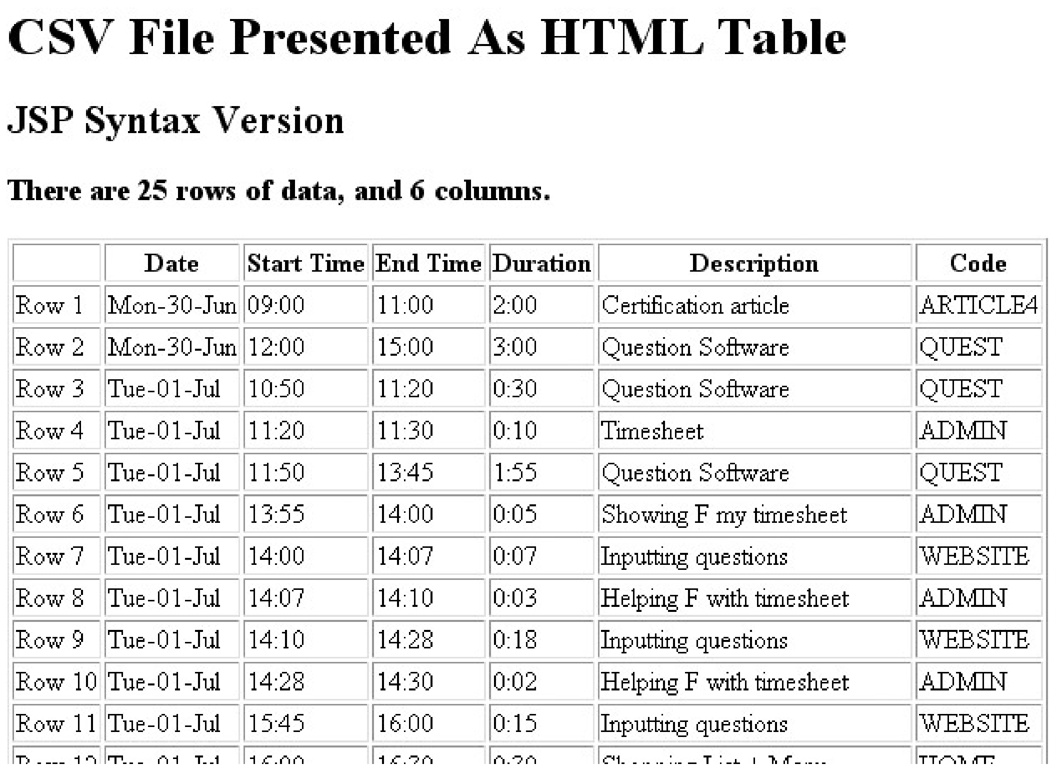
So this time, start by finding the solution file from the CD (which is sourcecode /ch07/ex0703.war), and then follow the instructions below.

Deploy and Test the Application

1. Deploy ex0703.war on your server—start the server if it's not started already.
2. Use your browser to run the application, using a URL such as

   ```
   http://localhost:8080/ex0703/CSVReader/Timesheet.csv
   ```

   CSVReader is the mapping for a servlet in the ex0703 context. The servlet uses the path information that appears after the servlet mapping in the URL—in this case, Timesheet.csv. The servlet looks for the named file in the context directory. So if you are using your own CSV file, place this in the context directory (e.g. ex0703) on your server, and change the name in the URL from "Timesheet.csv" to the name of your file. Make sure to respect upper and lower case.
3. Make sure that you get output like that shown in the following illustration.



CSV File Presented As HTML Table

JSP Syntax Version

There are 25 rows of data, and 6 columns.

	Date	Start Time	End Time	Duration	Description	Code
Row 1	Mon-30-Jun	09:00	11:00	2:00	Certification article	ARTICLE4
Row 2	Mon-30-Jun	12:00	15:00	3:00	Question Software	QUEST
Row 3	Tue-01-Jul	10:50	11:20	0:30	Question Software	QUEST
Row 4	Tue-01-Jul	11:20	11:30	0:10	Timesheet	ADMIN
Row 5	Tue-01-Jul	11:50	13:45	1:55	Question Software	QUEST
Row 6	Tue-01-Jul	13:55	14:00	0:05	Showing F my timesheet	ADMIN
Row 7	Tue-01-Jul	14:00	14:07	0:07	Inputting questions	WEBSITE
Row 8	Tue-01-Jul	14:07	14:10	0:03	Helping F with timesheet	ADMIN
Row 9	Tue-01-Jul	14:10	14:28	0:18	Inputting questions	WEBSITE
Row 10	Tue-01-Jul	14:28	14:30	0:02	Helping F with timesheet	ADMIN
Row 11	Tue-01-Jul	15:45	16:00	0:15	Inputting questions	WEBSITE

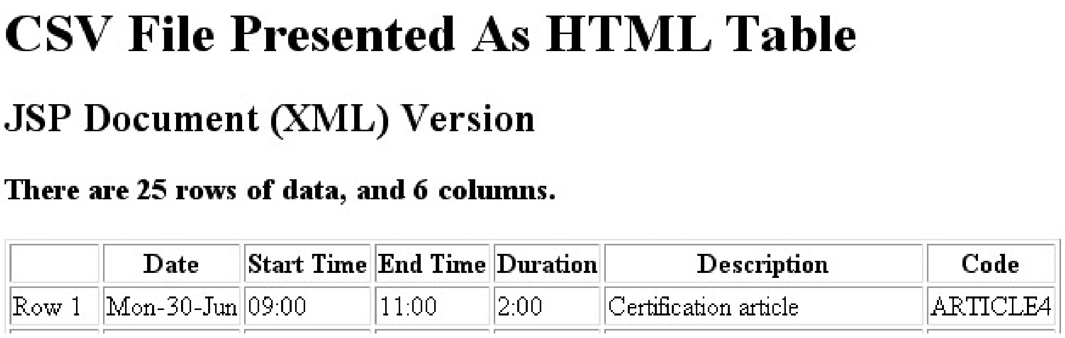
4. Now use a text editor to change the deployment descriptor web.xml directly in the server directory—which for me is

```
<Tomcat installation directory>/webapps/ex0703/WEB-INF/web.xml
```

5. You'll see that the servlet CSVReader has an initialization parameter, which is the name of the JSP page to forward to. Change the extension on the name of the file to .jspx, which points to the JSP Document solution file. The whole line looks like this:

```
<param-value>/csvRenderer.jspx</param-value>
```

6. Save and close web.xml. Restart your server. Use your browser to access the servlet, using exactly the same URL as in step 2. The output should look identical, as shown in the following illustration—just the heading has changed to indicate that the output came from a JSP document.



CSV File Presented As HTML Table

JSP Document (XML) Version

There are 25 rows of data, and 6 columns.

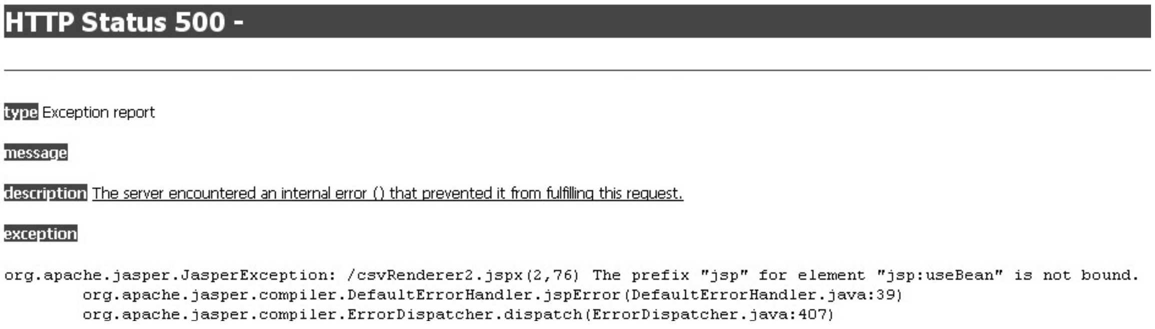
	Date	Start Time	End Time	Duration	Description	Code
Row 1	Mon-30-Jun	09:00	11:00	2:00	Certification article	ARTICLE4

Create Your Own JSP Document

7. Now find the original JSP syntax file in the context directory, the one called csvRenderer.jsp (i.e., *not* the version with the .jspx extension). Copy this into the same directory, and call it csvRenderer2.jspx (*with* a .jspx extension—very important!). This is going to be the file you'll work on—you'll change all the JSP syntax within it to JSP document XML syntax, stage by stage.
8. Edit web.xml again as you did in step 5—change the name of the parameter value to that of your copied file, csvRenderer2.jspx:

```
<param-value>/csvRenderer2.jspx</param-value>
```

9. Stop and start your web server again. Again, invoke the application with the same URL as in step 2. This time, the application should fail—the target JSP, csvRenderer2.jspx, won't translate until its syntax is corrected. You may well get an error in your browser like the one shown in the following illustration, which complains that "The prefix 'jsp' for element 'jsp:useBean' is not bound." Don't worry if you see something different; the point is that we need to fix the errors.



HTTP Status 500 -

type Exception report

message

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

exception

```
org.apache.jasper.JasperException: /csvRenderer2.jspx(2,76) The prefix "jsp" for element "jsp:useBean" is not bound.
	org.apache.jasper.compiler.DefaultErrorHandler.jspError(DefaultErrorHandler.java:39)
	org.apache.jasper.compiler.ErrorDispatcher.dispatch(ErrorDispatcher.java:407)
```

10. Now open csvRenderer2.jspx with a text editor. Change the `<h2>` heading on line 5 to read

    ```
    <h2><b>My</b> JSP Document Version</h2>
    ```

11. The JSP page source uses standard actions. When these appear in JSP documents, a namespace must be supplied to say where the XML elements for the standard actions are defined. You can supply this by altering the `<html>` tag exactly as shown below:

    ```
    <html xmlns:jsp="http://java.sun.com/JSP/Page" >
    ```

    Save the file, but leave your text editor open ready to make more changes.
12. Refresh your browser for the same URL. The application should still fail, but this time the error should be deferred to later in the page source. The next error I get is located at the second character on the third line of page source: "The content of elements must consist of well-formed data or markup" (the location is given in the stack trace).
13. The issue is the `page` directive doing the import, with its `<%` syntax. Change this to use XML `page` directive syntax:

    ```
    <jsp:directive.page import="java.util.*" />
    ```

14. Save the file, and again refresh your browser—note the next error. For me, this is on line 14, and it says that "the entity 'nbsp' was referenced, but not declared." This is a pure XML problem. In HTML, `&nbsp;` denotes a nonbreaking space. This style of denoting a special character (beginning with an ampersand and ending in a semicolon) is fine for XML; a character denoted this way is called an entity. However, it needs to be defined to the XML file in order to be respected. We're not going to do this, but instead cheat. There's a "get out of jail free" card to counter any dodgy text in an XML file, which is to describe the text as "character data." This uses some involved syntax. Replace `&nbsp;` with exactly what is written below:

    ```
    <![CDATA[&nbsp;]]>
    ```

15. Save the file, and refresh the browser. My latest error is now on line 15—again complaining about the lack of well-formed character data or markup. As you've probably guessed, it's the scriptlet syntax. Go through the

code, replacing every occurrence of `<%` with `<jsp:scriptlet>`, and every corresponding `%>` with `</jsp:scriptlet>`.

16. Save and refresh. I get the same error, now transferred to line 16. This time, it's an expression causing the grief. Replace every occurrence of `<%=` with `<jsp:expression>`, and every *corresponding* `%>` with `</jsp:expression>`.
17. Save and refresh. There's still a problem! Look at the **for** loop in the first scriptlet—the condition test contains the "<" sign. This is—of course—good Java but lousy XML. To the XML parser, it looks like the beginning of a tag that never ends. Change the "<" sign to escape characters `&lt;`. Repeat the exercise for the later **for** loop (around line 24), which also contains a "<" sign.
18. With luck, when you now save and refresh, the page will work correctly.
19. In a future exercise, we'll take the XML JSP document source from this exercise and improve on it so that most of the language scripting is removed.

CERTIFICATION OBJECTIVE

Expression Language (Exam Objectives 7.1, 7.2, and 7.3)

Given a scenario, write EL code that accesses the following implicit variables including pageScope, requestScope, sessionScope, and applicationScope, param and paramValues, header and headerValues, cookie, initParam and pageContext.

Given a scenario, write EL code that uses the following operators: property access (the . operator), collection access (the [] operator).

Given a scenario, write EL code that uses the following operators: arithmetic operators, relational operators, and logical operators.

Expression Language (EL) is all about the EL-imination of Java syntax from your pages. Here's how the JSP specification eloquently states the goal of EL: "The EL can be used to easily access data from the JSP pages. The EL simplifies writing *script-less* JSP pages that do not use Java scriptlets or Java expressions and thus have a more controlled interaction with the rest of the Web Application" (JavaServer Pages 2.0 Specification, page xix).

As the name implies, Expression Language provides an alternative to the expression aspect of Java language scripting—`<jsp:expression>...</jsp:expression>` or `<%...%>` . EL by itself is not a replacement for scriptlets. For that, you'll need to wait for the JSP Standard Tag Language (JSTL) in Chapter 8.

The goal of EL is simplicity. Although EL sacrifices some of the sophistication possible in a Java language expression, it is easier to use: The syntax is succinct and robust. Apart (obviously) from the syntax, EL is different from Java in other ways. Some rules are the same, and some are different. Because EL is a new and popular addition to JSP technology, you can be sure that many questions in the exam will test you on these rules.

exam watch

The exam creators have a nasty habit of mixing up EL and scriptlet syntax within the same question. This book organizes its content by exam objective, but don't expect the questions (here or in the real exam) to take a purist approach! After all, they reflect real life—and you may well have to deal with scriptlets and EL in the space of a few consecutive lines of JSP source.

Expression Language Overview

Expression Language began life as part of the JSP Standard Tag Library (JSTL), which we meet in Chapter 8. EL is now incorporated as part of the JSP 2.0 specification and is entirely independent of JSTL. However, it's only with JSTL that it fully comes into its own. EL can supply only the equivalent of the "right-hand side of the equal sign" in a typical computing statement. For example, EL lacks any looping constructs. And although there are conditional operators in EL, you can't take any action on them: There's not even an "if . . . then . . ." mechanism. JSTL supplies the missing pieces, so you will still find EL used most often in conjunction with JSTL.

That's not to say that you can't use EL independently. And especially when your goal is the SCWCD, there's plenty to learn about it. So in this section we'll concentrate exactly on that—EL capabilities. Some of the time we'll use EL in conjunction with Java language scripting elements, such as scriptlets. There's nothing technically wrong with that, but it's not considered best practice—after all, EL is meant to encourage Java-free JavaServer Pages! However, until we do learn about JSTL, scriptlets remain the easiest way to create expressions with data to display.

on the job

Expression Language can be enabled or disabled in three different ways. We encountered one of these ways in Chapter 6, when we looked at the page directive settings. The page directive attribute* isELEnabled *can turn on EL for a single page—or not.

There are a couple of alternative ways of controlling EL enablement that aren't explicit exam objectives. One is with the* `<jsp-property-group>` *element, which has a subelement* `<el-enabled>`*. We met* `<jsp-property-group>` *in the context of identifying JSP documents (subelement of* `<jsp-config>`*).

Finally, EL is enabled at an application level by having a deployment descriptor at servlet level 2.4. A previous deployment descriptor level indicates that EL should be switched off.

Expression Language Basics

As we saw briefly in Chapter 6, an expression begins with `${` and ends with `}`. The part between the curly braces must be a valid EL expression. The string in the JavaServer Page source code is subject to translation, like anything else in the page. Translation checks syntax validity but won't (for reasons that we'll come to) check that the variables you use actually exist (remember that translation incorporates compilation). At run time, the string representing the expression is sent to a method called `resolveVariable()`, in an object supplied by your JSP container provider of type VariableResolver. This returns an object, which is sent to the JSP output stream—typically via an `out.print()` statement in your generated servlet source. Mostly, the expression will resolve one way or another. Even if your variables don't exist, sensible defaults are provided, which mostly prevent the expression ending in a run-time error.

EL is equally valid in standard JSP syntax or JSP document (XML) syntax. So the following are equivalent. First a JSP in normal syntax:

```
<html>
<head><title>As a normal JSP</title></head>
<body>
<% request.setAttribute("whichever", "EL in either syntax"); %>
<p>${whichever}</p>
</body></html>
```

And now the same as a JSP document:

```
<html xmlns:jsp="http://java.sun.com/JSP/Page">
<head><title>As a JSP document</title></head>
<jsp:output omit-xml-declaration="true" />
```

TABLE 7-3 The Five Kinds of EL Literal

Type	Example	Comments
Boolean	`${true}`	Valid values are true and false—just like Java Boolean literals.
Integer	`${18782}`	Underpinned by a java.lang.Long value. Don't append "l" or "L" to the literal value as would happen for a Java long literal.
floating point	`${1.618034}` or `${2.998e+9}`	Underpinned by a java.lang.Double value.
Strings	`${"Galleon"}` or `${'Coracle'}`	Characters surrounded by double or single quotes.
Null	${null}	Equivalent to the Java **null** literal. Doesn't output anything.

```
<jsp:directive.page contentType="text/html" />
<body>
<jsp:scriptlet>
  request.setAttribute("whichever", "EL in either syntax");
</jsp:scriptlet>
<p>${whichever}</p>
</body></html>
```

In both cases, the EL syntax `${whichever}` picks up and displays the value of the whichever attribute set up in the scriplet: "EL in either syntax."

However, you do have to keep Java code (such as scriptlets) free of EL (after all, EL is not valid *Java* syntax). So the following will not work:

```
<% request.setAttribute( "anAttribute", ${aValueFromEL} );
```

EL Literals

EL has a smaller range of literals than Java. The ones it does use are similar. Table 7-3 shows the different values. Because you don't declare variables or assign to variables in EL, there are no explicit keywords for types; nonetheless, there are five that are defined.

EL Operators

Operators in EL come in four categories:

- arithmetic
- relational

- logical
- empty

EL operators (like EL literals) offer a subset of what's available in the Java language, and again you have to beware of some differences in behavior between EL and Java.

Arithmetic Operators

There are five arithmetic operators—for addition (+), subtraction (−), multiplication (*), division (/), and modulo (%). As you can see, the operator symbols are identical to Java. However, there are alternative forms for the division and modulo operators—`div` and `mod`, respectively. Let's consider each of the operators in turn.

Addition Addition is expressed like this: `${a + b}`. Addition works much as you would expect. If *a* represents an Integer object of value 2, and *b* an Integer object of value 3, then the result is 5. The inputs don't have to be numeric objects—string values for *a* and *b* of "2" and "3" would work as well. If either of attributes a and b doesn't exist, and is null, that's not a problem—they are treated as zero values. A zero-length string—""—is likewise treated as zero. As in much of EL, there's quite a bit of work behind the scenes to ensure that a result is obtained somehow, as long as the inputs to the calculation are remotely sensible. However, the following calculation won't work: `${"Not a Number" + 3.0}`. You will get a javax.servlet.jsp.el.ELException, complaining that "Not a Number" cannot be converted to a java.lang.Double value. This example also goes to show that the addition operator in EL—unlike Java—is not overloaded to handle string concatenation. There's no operator overloading or string concatenation in EL.

Subtraction Subtraction is expressed as you would expect: `${a – b}`. The same comments made about the addition operator apply to subtraction as well.

Multiplication Multiplication is expressed `${a * b}`. No surprises there.

Division Division is expressed `${a / b}` or `${a div b}`. Even if the inputs are both integers, double division is performed—not whole-number division ignoring the remainder. There is no direct EL equivalent for Java's integer division behavior. Being as EL division is always double division, it behaves like Java floating decimal division, so divide by zero is not an error but results in an answer of "Infinity." EL also shares an irritating feature of double division on binary-oriented computers, and the result may be imprecise. This isn't unique to EL; it's equally true of floating-point division in regular Java syntax, and indeed of many other programming languages on most computing platforms. For example, it might surprise you to learn that `${9.21 / 3}` doesn't give the neat result of 3.07, but rather 3.0700000000000003.

Modulo Modulo is expressed `${a % b} or ${a mod b}`. This time, integers are respected as integers, but a double for either input causes the calculation to be worked as a double. Again, the caveat about imprecise double arithmetic applies (try, for example, `${9.1 mod 3}`).

exam watch

Just as Java arithmetic has "promotion" rules for the inputs to a calculation, so does EL. In Java language, for example, in the calculation 9.0 + 3, "3"—an integer literal—is promoted to a double before the calculation takes place, and the result is a double. This is because the other operand (9.0) is a double literal. This is the ceiling for Java arithmetic—after all, there's nowhere to go beyond a double in Java primitive terms. In EL, promotion applies on a grander scale—there isn't a double (or java.lang.Double) ceiling. In some cases, the operands might be of type java.math.BigInteger or java.math.BigDecimal. There's only an outside chance you will have to face questions involving the promotional rules with BigInteger and BigDecimal, but you might want to check out the arithmetic promotion (or "coercion") rules laid out in the JSP 2.0 Specification, section 2.3.5.

Relational Operators

EL has a full complement of relational operators, which have conventional and alternate forms, as shown in Table 7-4. Alternative forms exist to make writing JSP documents that much easier. You'll remember from earlier in the chapter that the

TABLE 7-4

EL Relational Operators

Greater than	>	gt
Less than	<	lt
Equals	==	eq
Greater than or equals	>=	ge
Less than or equals	<=	le
Not equals	!=	ne

< and > signs are bad news for well-formed XML, except when used to mark the beginning and end of tags. To avoid having to use escape sequences such as `&gt;=` every time you want to express "greater than or equals" in an expression, use *ge* instead. It's a good habit to get into, for the alternative form works just as well in conventional syntax and is much more readable in JSP document syntax.

The result of a relational operation is boolean *true* or *false*. So the following not very useful expression will cause "true" to be written to output: `${9 ge 3}`. You are not restricted to numeric inputs. Most usefully, you can do lexical string comparison, so `${"zebra" eq "antelope"}` will return "false." Under the covers, the String `equals()` method is invoked rather than a straight comparison of objects. In general, EL relational evaluation will invoke useful comparison methods on objects (such as `equals()` and `compareTo()`) when they are appropriate and available.

Logical Operators

EL has a more limited set of logical operators than the Java language. As for relational operators, there is a symbolic and alternative form. These are both shown in Table 7-5. These operators allow you to join conditional tests together to return a composite boolean result. For example, `${9 > 3 && "z" gt "a"}` would return true. Like Java, EL will evaluate only the left-hand side of an expression involving `&&` and ||, if that is sufficient to intuit the overall result:

- If the left-hand side of an expression involving && is false, the whole expression must be false.
- If the left-hand side of an expression involving || is true, the whole expression must be true.

In either case, the right-hand side remains unevaluated. For all the examples we've seen, this doesn't matter. However, it might matter more to you after you see EL functions in Chapter 8.

TABLE 7-5

EL Logical Operators

	Symbolic	Alternative
Logical "and"	&&	and
Logical "or"	||	or
Logical "not"	!	not

The *empty* Operator

EL's *empty* operator can be invoked like this: `${empty obj}`. This expression will evaluate to *true* if `obj` represents something **null**—as would happen if the `obj` attribute didn't exist. However, the *empty* operator generalizes the concept of emptiness beyond a crude **null** test. There are other circumstances where `${empty obj}` results in **true**, which is any of the following:

- `obj` is an empty string ("").
- `obj` is an empty array.
- `obj` is an empty Map or an empty Collection (which covers every collection class in the java.util package—all of them inherit Map or Collection somewhere along the line).

Under any other circumstance, `${empty obj}` will return **false**.

exam watch

For all the operators—arithmetic, relational, logical—the precedence rules work in the same way as for the equivalent operators in the Java language. Parentheses can (and should) be used to clarify potential misinterpretations of code. However, you should have some awareness of precedence for the exam. Expressed crudely, the order is as follows:

not, empty;
multiplication, division, modulo;
addition, subtraction;
relational;
and, or.

For more precision, refer to the JSP Specification, section 2.3.10.

EL Property Access

Having dealt with the basics of EL—syntax, literals, operators—we can move on to some of its more exciting aspects. As you would have suspected from the preceding discussion, EL can access objects. Mostly, EL is used to access attributes that have been set up in some scope: page, request, session, or application. In this respect, EL is like the standard action `<jsp:getProperty>`, although its syntax works rather differently—and that's what we need to explore next.

The . and [] Operators

Any attribute in any scope can be displayed with EL. The scope doesn't even need to be specified. Suppose that there is an attribute called "title," holding a String object with some text; then `${title}` is all that is required to display that text. What happens, though, if the attribute isn't as simple as a String? What if you are holding a complex object as an attribute? For such an object to be useful to EL, it has to be a JavaBean—at least in the sense of having "getter" methods. Let's suppose that the object in question is the Dog JavaBean we met at the beginning of the chapter. This had five methods: `getName()` (returning a String), `getWeight()` (returning a float), `isInsured()` (boolean), `getSex()` (char), and `getBarkVolume` (another String). According to bean law, this exposes five properties derived from the "getter" method names: name, weight, insured, sex, and barkVolume. The properties share the type of their corresponding "getter" method.

Now let's suppose that a Dog object exists as a session attribute, with a name of "currentDog." To display a property of the dog, you use the attribute name and the property name. The simplest approach is to separate the two with a dot. So

```
${currentDog.name}
```

would display the name of the dog, and

```
${currentDog.insured}
```

would display "true" or "false" according to whether the dog was insured or not. Type conversion from boolean to String is managed somewhere before the result reaches page output, as is true for all other primitive types or objects returned by an expression.

This is the best way to use EL for property access. However, there is an alternative syntax, although it's really better kept for a different purpose we'll come to in a moment. These variants will also display the dog's name:

- `${currentDog["name"]}`
- `${currentDog['name']}`

You're not limited to one level, either. Let's suppose our Dog class had an additional method, `getFather()`, which returned another Dog object—representing the father of the current dog. This would expose another property on the current dog, called "father." The father dog—being a Dog object—has all the same properties as the current dog. So if you now wanted to display the name of the current dog's father, you could do so this way:

```
${currentDog.father.name}
```

The alternative syntax would look like this:

- `${currentDog["father"]["name"]}`
- `${currentDog['father']['name']}`

You can even mix and match double quotes and single quotes, as long as you are consistent within any particular pair of square brackets, so although it's inconsistent, `${currentDog['father']["name"]}` would also work. In essence, anything that EL can interpret as a String can go between the square brackets.

Arrays, Lists, and Maps

EL capabilities go further. Suppose that I have an array defined as a page attribute through the following scriptlet:

```
<% String[] dayArray = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
pageContext.setAttribute("days", dayArray); %>
```

An expression later in the JSP page source can access the days of the week using syntax that is practically identical to array syntax. For example, `${days[0]}` will send "Mon" to page output, while `${days[6]}` will send "Sun." Within the expression's curly braces is the name of the attribute (days) followed by square brackets. Within the square brackets you can place any integer—either a literal or some attribute that can be sensibly converted to an integer. So if you wrote this code farther down the page, it would output "Wed Thu":

```
<% pageContext.setAttribute("two", new Integer(2));
pageContext.setAttribute("three", "3"); %>
${days[two]}
${days[three]}
```

The first page attribute, called "two," is set to an Integer with a value of 2. So `${days[two]}` gets the third value in the array—"Wed." You can see, though, from the second attribute, that you don't have to stick with explicit numeric types as with java.lang.Integer. The second page attribute, called "three," still works when loaded with a String. Provided that a method like `Integer.parseInt()` can extract an **int** value from the String, everything will work.

exam Watch

If you write an expression such as `${days[7]}`, you might expect an ArrayIndexOutOfBoundsException or an ELException arising from this as an underlying cause. Not so. EL silently suppresses this problem—you just get blank output. Even if you use an attribute name that doesn't exist—`${days[notAnAttributeName]}`—nothing goes wrong; you just get blank output. However, let's consider what happens if the attribute supplied is a valid attribute but can't be converted to an integer value. Given this page attribute, `<% pageContext.setAttribute("four", "the_Word_Four"); %>` the expression ${days[four]} would end in a run-time error (ELException).

Having worked through Arrays, you'll be delighted to know that Lists work in the same way. Any collection class that implements java.util.List can have its members accessed with identical syntax. Under the covers, the `List.get(int index)` method is executed.

Finally, there is the case of classes that implement java.util.Map. You'll recall from the SCJP (if nowhere else) that Maps hold a collection of key-value pairs. Each key must be unique, and is normally a String value (but can be any Object). Let's consider a variation on the days of the week example, which uses a Map:

```
<jsp:directive.page import="java.util.*" />
<% Map longDays = new HashMap();
longDays.put("MON", "Monday");
longDays.put("TUE", "Tuesday");
longDays.put("WED", "Wednesday");
longDays.put("THU", "Thursday");
longDays.put("FRI", "Friday");
longDays.put("SAT", "Saturday");
longDays.put("SUN", "Sunday");
pageContext.setAttribute("longDays", longDays);
pageContext.setAttribute("wed", "WED");
%>
<br /> ${longDays[wed]}
<br /> ${longDays["THU"]}
<br /> ${longDays.FRI}
```

The output from this code is

```
Wednesday
Thursday
Friday
```

What happens is this: The code loads a HashMap object with the full names of the seven days of the week, keyed by abbreviated capital codes ("MON," "TUE," etc.). The HashMap is loaded into a page attribute called longDays. Another page attribute is set up, called "wed" and with a value of "WED"—which matches one of the keys in the HashMap. In general terms, expressions accessing a Map work on this principle: `${nameOfMap[keyValue]}`. From the two expressions in the code, you can see that it doesn't matter if the key value is a literal ("`THU`") or derived from an attribute (`wed`).

What about the third expression, though: `${longDays.FRI}`? That appears to use the JavaBean syntax we used earlier—even though there is obviously no "`getFRI()`" method to fall back on within the Map. Yet it still works. If you use a Map's key value as if it were a property name on a bean, you will still find the corresponding value.

EL Implicit Objects

To add to EL's versatility still further, it has its own set of implicit objects. It's similar to the idea of implicit objects that you can use in general JSP page source, but it's important—especially for the exam!—that you learn the distinctions between the two sets. The full list is shown in Table 7-6. Note that all the EL implicit objects, with the exception of *pageContext,* are of type java.util.Map, so they obey the Map rules we just explored.

Let's briefly explore these implicit objects in turn.

pageScope, requestScope, sessionScope, and applicationScope These implicit objects are used to access attributes in a given scope. Of course, you can just name an attribute in expression language without any qualification, like this: `${myAttribute}`. Under the covers, `PageContext.find("myAttribute")` is used to search all scopes through page, request, session, and application, stopping when it finds an attribute of the right name. But if you want to target only a session scope attribute, then `${sessionScope.myAttribute}` will do the trick. All the alternative Map syntaxes will work as well—`${applicationScope ["myAttribute"]}` to find the attribute in application scope, for example.

param, paramValues These are used to recover parameter values—singly or in bulk. Let's suppose that your HTTP header request contains the following query string: `?myParm=firstValue&myParm=secondValue`. Let's also suppose this is a GET request, so there are no additional parameter values for `myParm` hidden in a POSTed request body. The result of `${param.myParm}` is "firstValue." The result of `${paramValues.myParm[1]}` is "secondValue." To put it another way, the implicit

TABLE 7-6 EL Implicit Objects

Variable Name	Description	Closest JSP Scripting "Equivalent"
pageContext	Represents the JSP `PageContext` object	Accessing properties of the `pageContext` implicit object
pageScope	A Map of page scope attributes	`pageContext.getAttribute()`
requestScope	A Map of request scope attributes	`request.getAttribute()`
sessionScope	A Map of session scope attributes	`session.getAttribute()`
applicationScope	A Map of application scope attributes	`application.getAttribute()`
param	A Map of ServletRequest parameter names and first values	`request.getParameter()`
paramValues	A Map of ServletRequest parameter names and all values	`request.getParameterValues()`
header	A Map of HttpServletRequest header names and first values	`request.getHeader()`
headerValues	A Map of HttpServletRequest header names and all values	`request.getHeaders()`
cookie	A map of HttpServletRequest cookie names and cookie objects	`request.getCookies()` and iterating through the returned Cookie array for a cookie of a given name
initParam	A Map of Servlet**Context** parameter names and values	`config.getServletContext() .getInitParameter()`

object param can be used solely to access the first value of a parameter (so in this example, it is impossible to use param to retrieve "secondValue"). However, using array syntax as shown, you can use the implicit object paramValues to access any of the available values for a given parameter.

header, headerValues These are used in a very similar way to param and paramValues, but they are targeted to recover request headers. The request header "Accept" is a good one to experiment with. This specifies the MIME types that a client is willing to receive back in the response, and it often consists of multiple values. The syntax is identical as for param and paramValues, so `${header.accept}` returns the first value of the accept header, and `${headerValues.accept[2]}` returns the third value. ("Accept" is one of the headers set up by the browser you met in Chapter 1, Exercise 1. Try pointing this browser to your own JSP, which contains the EL header and headerValues syntax described here.)

initParam This is used to access ServletContext initialization parameters, whose values are available across the entire web application. Don't be fooled into thinking that Servlet initialization parameters are returned! The syntax is exactly as for param, so `${initParam.myParm}` is used to return the value of an initialization parameter named "myParm."

cookie This is used to access a named Cookie in the HttpRequestHeader. A good example is the session cookie. Here are some variant approaches, all of which will display the value of the cookie:

```
${cookie.JSESSIONID.value}
${cookie["JSESSIONID"].value}
${cookie["JSESSIONID"]["value"]}
```

on the job

The implicit objects* cookie, header*, and* headerValues *are available only in JSP containers supporting the HTTP protocol, which, of course, most will do. These implicit objects relate to HTTP-only concepts.

exam watch

An implicit object name always takes precedence. Suppose that I set up a page attribute called "header"; then* `${header}` *would still refer to the implicit object* header*, not my page attribute (or attribute in any other scope, come to that).

pageContext This can be used to access properties of the PageContext object associated with the JSP page. Properties, as always, mean anything available from a "get" method that has no parameters. So request, session, and servletContext (not application!) are all available as properties by virtue of `getRequest()`, `getSession()`, and `getServletContext()` methods. If these objects have properties of their own, they can be used in expressions. So `${pageContext.request.method}`, for example, will display the HTTP method (GET, POST, etc.) associated with the request.

EXERCISE 7-4

ON THE CD

An EL Calculator

In this exercise, you'll write a single JSP document that acts as a simple calculator. You'll be able to type in two figures, select an operation (add, subtract, multiply, divide, modulo), and display the result. You'll use EL both to perform the calculations and to display the result.

There's a double purpose to this exercise. Writing the calculator will help your fluency with EL. *Using* the calculator will help you see how EL handles calculations. Try all kinds of inputs, with and without decimal points—some of the results may surprise you! You'll be much better prepared for anything the exam can throw at you in terms of EL arithmetic.

Create the usual web application directory structure under a directory called ex0704, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch07/ex0704.war—check there if you get stuck.

Create the JSP Document

1. Create a file called calculator.jspx directly in context directory ex0704.
2. Include a `<jsp:output>` element to omit the XML declaration that otherwise gets inserted into page output for JSP documents.
3. Include a `<jsp:directive.page>` element to set the content type to "text/html."
4. Write HTML elements to make a valid HTML document—`<html>`, `<head>`, `<body>`, etc. Make sure `<html>` is the root element in your document.
5. Include a namespace reference in the opening `<html>` tag to qualify the `jsp`: elements (`xmlns:jsp="http://java.sun.com/JSP/Page"`).
6. Place an HTML form in the document. This should have an input text field named `arg1`, a select field named `operation`, an input field named `arg2`, and a submit button. Give the select field five options to match the five arithmetic operations: add, subtract, multiply, divide, and modulo.
7. Beneath the form, place some template text saying "Did you miss out one of the numbers?" Next to this, write an EL expression that will output "true" if either of the input parameters `arg1` or `arg2` is empty.
8. Beneath this text, you're going to place a mixture of template text, scriptlets, and expressions that restate the calculation and show the result. For example, "Result: 1 plus 2 = 3." This is easier said than done; some hints follow.
9. "Result:" is just template text, and the value of the first input to the calculation (`arg1`) can be derived in EL using the *param* implicit object.
10. To display the operation (add, subtract, multiply, etc.), you could again use the param implicit object with the `operation` parameter. Alternatively, use a `<jsp:scriptlet>` to obtain the request parameter value for `operation`, and condition your text accordingly. (This is what the solution code does—the

option values are abbreviations such as "Add," "Sub," and "Mlt." There is logic to test the value and display a suitable word or phrase instead, such as "plus," "minus," and "multiplied by.")

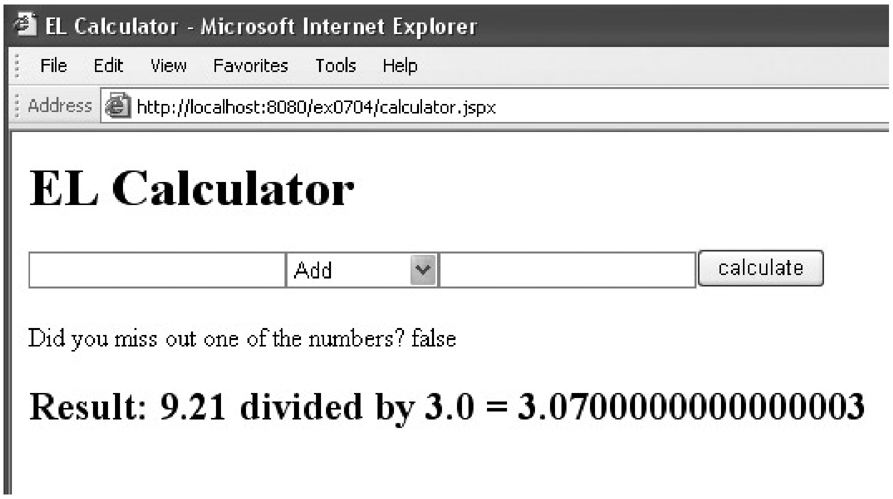
11. The value of the second input to the calculation (`arg2`) can be derived in EL by using the param implicit object again. The equal sign (=) is template text.
12. For the calculation itself, you'll need to write five EL expressions involving the `arg1` and `arg2` parameters. You'll need a scriptlet (or rather, a series of scriptlets using `if...` and `else if {...}` to test the `operation` parameter such that only one of the EL expressions is executed).

Deploy and Run the Application

13. Create a WAR file that contains the contents of ex0704, and deploy this to your web server. Start the web server if it has not started already.
14. Use your browser to request calculator.jspx, with a URL such as

```
http://localhost:8080/ex0704/calculator.jspx
```

15. Test it out with all five arithmetic operations, using a mixture of integers and doubles (i.e., anything with a decimal point) as inputs.
16. The following illustration shows the solution page in action.


EL Calculator - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost:8080/ex0704/calculator.jspx
Add
calculate

EL Calculator

Did you miss out one of the numbers? false

Result: 9.21 divided by 3.0 = 3.0700000000000003

CERTIFICATION SUMMARY

In this chapter you began by learning about standard actions. You saw that these observe strict XML element syntax, comprising opening tags, closing tags, and attributes. The first three standard actions you encountered were `<jsp:useBean>`, `<jsp:setProperty>`, and `<jsp:getProperty>`. You found that `<jsp:useBean>` can be used to make an object available in a page, either by obtaining an object from an existing attribute or by creating a new object and attaching it to an attribute. You also saw that the object targeted by `<jsp:useBean>` must adhere to JavaBean standards—at least in having a no-argument constructor, and (for it to be at all useful) having "get" and "set" methods. You found that `<jsp:useBean>` has an *id* attribute, whose value is shared with the name for the page, request, session, or application scope attribute it is bound to. You learned that the *class* attribute specifies the actual class of the object created, but that a *type* attribute can be used to specify a different type of reference variable (implemented interface or superclass) for the bean object. You saw that a *scope* attribute can be used to associate the object with any one of the four different scopes (`scope="page|request|session|application"`).

You then learned that `<jsp:setProperty>` can be used to set the value for one of the properties on the bean object declared with `<jsp:useBean>`. You saw that a property is named after a "get" and "set" method on the bean, such that `getLoan()`exposes a property called "loan" and that `setBankBalance()` exposes "bankBalance." You saw that you use the *property* attribute to name one of the properties and use the *name* attribute to tie in to an existing bean (as named in the *id* attribute of `<jsp:useBean>`). You learned that the value can be set using the *value* attribute (where the value of the *value* attribute can be a literal or a run-time expression), or by using the *param* attribute (associating the value with a request parameter). You then saw that `<jsp:getProperty>` can be used—like an expression—to send the value of a property to page output. You saw that this also has attributes of *name* (the name of the bean: matching the *id* value on `<jsp:useBean>`) and *property* (the name of the property whose value to display).

In the next section of the chapter, you met three more standard actions—`<jsp:forward>`, `<jsp:include>`, and `<jsp:param>`. You learned that `<jsp:forward>` and `<jsp:include>` do much the same job as the `forward()` and `include()` methods on a RequestDispatcher object. You saw that `<jsp:forward>` and `<jsp:include>` each have a *page* attribute, and this is used to specify a page within the web application using an absolute or relative URL (beginning with a slash or not).

You learned that `<jsp:include>` also has a *flush* attribute to control whether or not existing response output should be sent to a client before including a file, but that `<jsp:forward>` doesn't—because if output has been written to the client already, it's illegal to forward to another resource. You learned the differences between the `<jsp:include>` standard action and the `<%@ include file="..."> page` directive. You saw that the `include` standard action doesn't include a target file until request time; hence, the value for the file can be a run-time expression. You learned that, by contrast, the `include` directive incorporates the contents of a file at translation time.

You moved on to `<jsp:param>`, and saw how this can be used to graft on parameters available to the forwarded-to or included resource. You also learned that these parameters are no longer available on return to the forwarding or including resource.

The next main topic covered was that of JSP documents. You learned that JSP documents are JSP page source written in well-formed XML and by default are used to produce XML as output. You learned the basics of well-formed XML documents: how each one must have a root element, how opening tags must match to closing tags, and how every element must nest inside another without overlapping (aside from the root element). You saw that JSP traditional scripting element syntax has to be replaced with equivalent XML elements—so `<jsp:scriptlet>...</jsp:scriptlet>` for `<%...%>`, `<jsp:expression>...</jsp:expression>` for `<%=...%>`, and `<jsp:declaration>...</jsp:declaration>` for `<%!...%>`. You found that the Java language contents of scriptlets, expressions, and declarations can remain unaltered—except that some symbols (in particular < and >) need character sequences called entities to keep the XML syntax well-formed (so `&lt;` for < and `&gt;` for >).

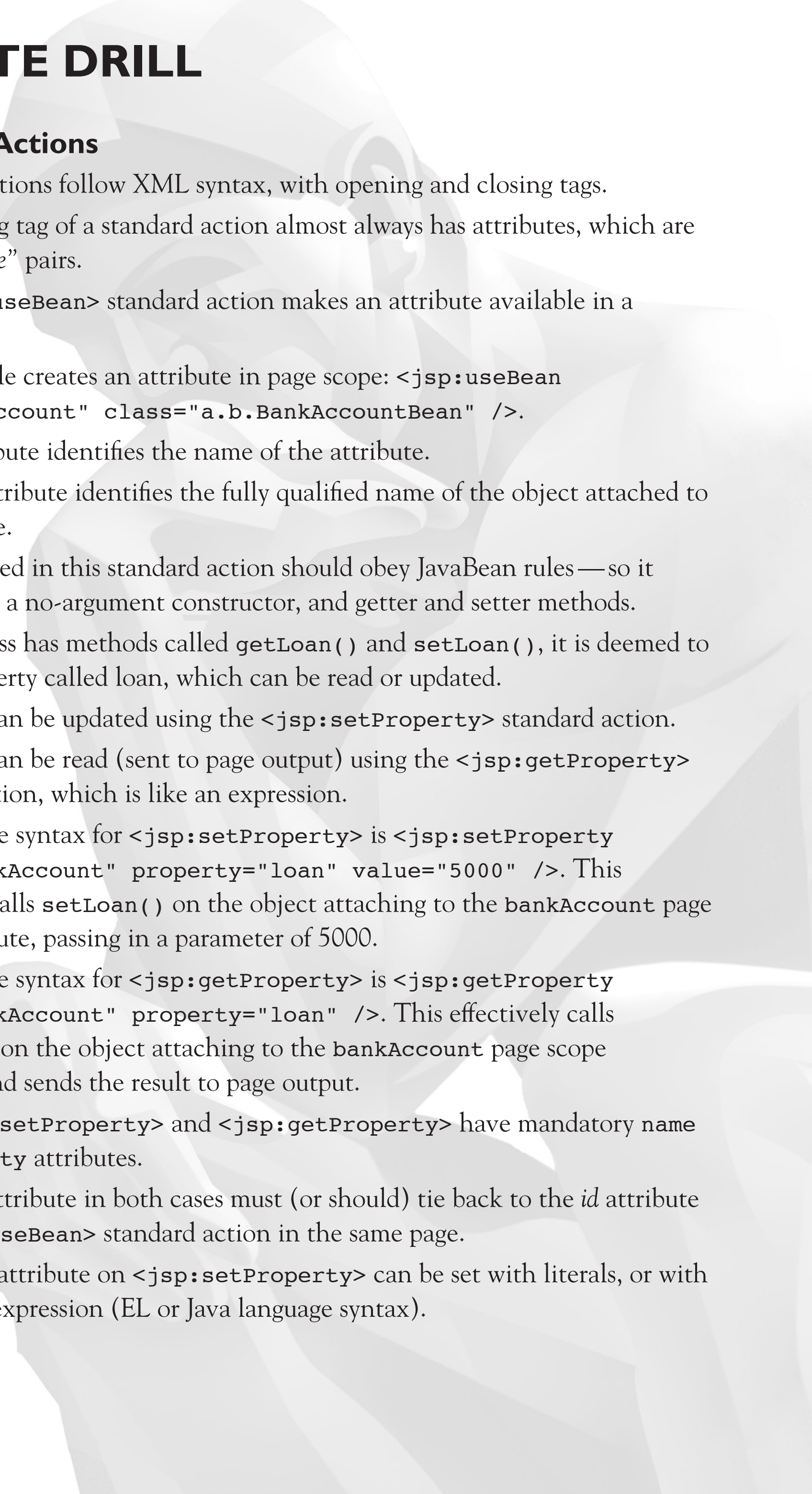
You also saw that directive syntax changes as well, so `<jsp:directive.page .../>` for `<%@ page ...%>` and `<jsp:directive.include .../>` for `<%@ include ...%>`. However, you found that the attributes for `page` and `include` directives remain unaltered between the two styles. You learned that comment syntax changes subtly, so `<!-- ... -->` becomes the equivalent of `<%-- ... --%>` for commenting out lines in page source.

You learned that there are three different ways for the JSP container to recognize JSP documents. Two of the methods rely on a web.xml file at servlet specification level 2.4: (1) having a .jspx extension on the file, and (2) setting the `<is-xml>` element to "true" in the deployment descriptor for a given `<url-pattern>`—this within `<jsp-property-group>` in element `<jsp-config>`. The third method, you saw, works whatever level of deployment descriptor you have, and that is to use `<jsp:root>` as the root element in your document.

In the final section of the chapter, you learned about Expression Language, abbreviated EL. You saw how EL is a Java-language free equivalent for expressions. You saw that EL has access to any attribute in any scope and that the simplest way to display an attribute value in your page output is like this: `${attributeName}`. You learned that if that attribute is a bean, you can access properties on the bean using dot syntax (`${attributeName.propertyName}`) or square bracket syntax (`${attributeName["propertyName"]}`). You saw how the square bracket syntax is useful for accessing Array or List elements, by providing an integer in the square brackets. You also learned that an attribute that is a java.util.Map can have its key values treated like bean properties, using the dot or square bracket syntax.

You saw that you have a range of five literals available to you in EL—integers, floating point numbers, booleans (true or false, like Java), strings (enclosed with single or double quotes), and a **null** literal. You were introduced to the five arithmetic operators in EL: +, −, *, / or div, and % or mod, and a range of relational and logical operators. You saw that there are character equivalents for relational operators so that you can more easily write XML-friendly syntax. You also met EL's *empty* operator and found that this qualifies empty strings and empty arrays and collections as returning **true** for an empty test as well as, of course, a **null** value.

Finally, you met the range of EL's implicit objects. You found that you can access attributes in a particular scope using *pageScope*, *requestScope*, *sessionScope*, or *applicationScope*. You saw that you can use *param* or *paramValues* to access request parameters, and *header* or *headerValues* to access request headers. You learned that *initParam* is used to access ServletContext parameters and that *cookie* is used to access a named cookie on a request.



✓ TWO-MINUTE DRILL

JSP Standard Actions

- ❏ Standard actions follow XML syntax, with opening and closing tags.
- ❏ The opening tag of a standard action almost always has attributes, which are *name*=“*value*” pairs.
- ❏ The `<jsp:useBean>` standard action makes an attribute available in a JSP page.
- ❏ This example creates an attribute in page scope: `<jsp:useBean id="bankAccount" class="a.b.BankAccountBean" />`.
- ❏ The *id* attribute identifies the name of the attribute.
- ❏ The *class* attribute identifies the fully qualified name of the object attached to the attribute.
- ❏ The class used in this standard action should obey JavaBean rules—so it should have a no-argument constructor, and getter and setter methods.
- ❏ If such a class has methods called `getLoan()` and `setLoan()`, it is deemed to have a property called loan, which can be read or updated.
- ❏ Properties can be updated using the `<jsp:setProperty>` standard action.
- ❏ Properties can be read (sent to page output) using the `<jsp:getProperty>` standard action, which is like an expression.
- ❏ One possible syntax for `<jsp:setProperty>` is `<jsp:setProperty name="bankAccount" property="loan" value="5000" />`. This effectively calls `setLoan()` on the object attaching to the `bankAccount` page scope attribute, passing in a parameter of 5000.
- ❏ One possible syntax for `<jsp:getProperty>` is `<jsp:getProperty name="bankAccount" property="loan" />`. This effectively calls `getLoan()` on the object attaching to the `bankAccount` page scope attribute, and sends the result to page output.
- ❏ Both `<jsp:setProperty>` and `<jsp:getProperty>` have mandatory `name` and `property` attributes.
- ❏ The `name` attribute in both cases must (or should) tie back to the *id* attribute in a `<jsp:useBean>` standard action in the same page.
- ❏ The `value` attribute on `<jsp:setProperty>` can be set with literals, or with a run-time expression (EL or Java language syntax).

- Another possible syntax for `<jsp:setProperty>` is `<jsp:setProperty name="bankAccount" property="loan" param="loanField" />`. This effectively calls `setLoan()` on the object attaching to the `bankAccount` page scope attribute, passing in the value for the request parameter `loanField` as a parameter.
- `<jsp:setProperty name="bankAccount" property="*" />` has the effect of calling set methods on all the properties of the `bankAccount` bean whose names match the names of request parameters.
- `<jsp:useBean>` has an optional attribute called *scope*, with valid values of "page", "request", "session", or "application". The bean is created or retrieved from the given scope (the default—if *scope* is absent—is "`page`").
- `<jsp:useBean>` has another optional attribute called *type*. This must be a superclass of, or interface implemented by, the *class* attribute value.
- The *type* attribute enables the use of a different type of reference variable from the underlying class of the object that holds the attribute value.

Dispatching Mechanisms

- The `<jsp:include>` standard action acts very much like the `RequestDispatcher.include()` method.
- The `<jsp:forward>` standard action is likewise like the `RequestDispatcher.forward()` method.
- Both `<jsp:forward>` and `<jsp:include>` have one mandatory attribute: `page`.
- Example `forward:<jsp:forward page="/anotherPage.jsp" />`.
- The *page* attribute references the file to include or forward to.
- The value for the *page* attribute can begin with a forward slash. The JSP container then treats the web application context directory as the root.
- No forward slash for the *page* attribute denotes a relative URL. The JSP container looks for the forwarded-to or included file relative to the location of the forwarding or including page.
- A `<jsp:forward>` is illegal (IllegalStateException) if any of the response has already been sent to the client.
- `<jsp:include>` also has an optional attribute, *flush*. This determines whether any existing page output should be sent to the client before including the file.

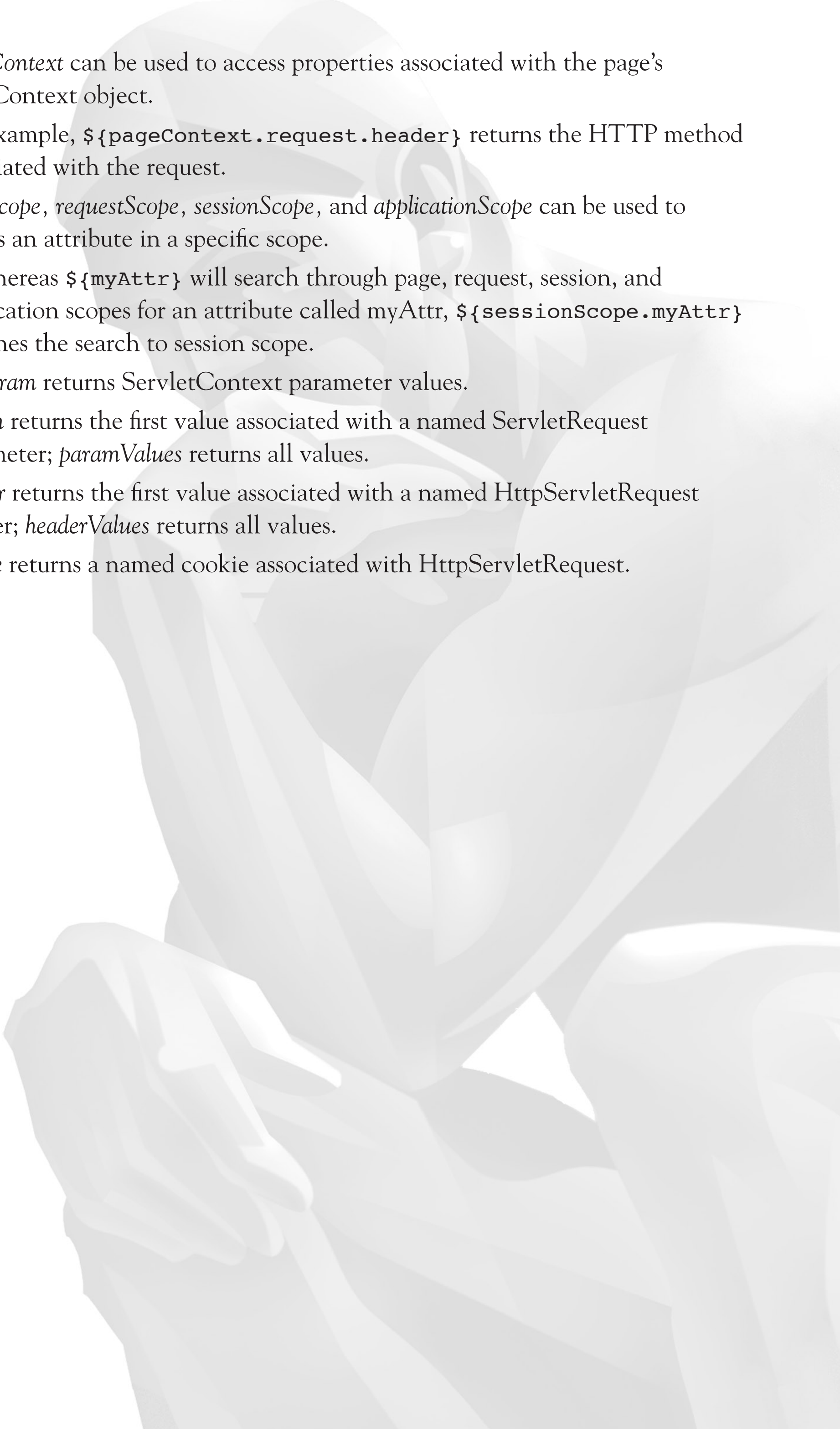
- ❑ `<jsp:param>` can be included in the body of `<jsp:forward>` or `<jsp:include>`.
- ❑ `<jsp:param>` has *name* and *value* attributes: `<jsp:param name="parmName" value="parmValue" />`.
- ❑ `<jsp:param>` adds in request parameters that are available to the forwarded-to or included resource but disappear on return to the forwarding or including pages.
- ❑ In the case of more than one value being present for a given parameter name, `<jsp:param>` request parameter values are loaded at the front.

JSPs in XML

- ❑ JSP documents are JSP source files written entirely in XML syntax.
- ❑ JSP documents typically have a .jspx extension.
- ❑ JSP documents can also be identified by setting `<is-xml>true</is-xml>` in the deployment descriptor for a given `<url-pattern>`. Both these elements are subelements of `<jsp-property-group>`, which is a subelement of `<jsp-config>`.
- ❑ Otherwise, a JSP document must use `<jsp:root>` as its root element.
- ❑ XML syntax demands a single root element in a file (as `<web-app>` is for the deployment descriptor).
- ❑ XML syntax demands that each element is properly nested. It is illegal for the closing tag from one element to come between the opening and closing tag of another element.
- ❑ JSP document syntax provides replacements for all the <%-type scripting element syntax.
- ❑ `<jsp:scriptlet>...</jsp:scriptlet>` replaces `<%...%>`.
- ❑ `<jsp:expression>...</jsp:expression>` replaces `<%=...%>`.
- ❑ `<jsp:declaration>...</jsp:declaration>` replaces `<%!...%>`.
- ❑ Otherwise, Java language syntax remains unchanged, but XML-unfriendly characters (such as < and >) need to be replaced with entities (such as `&lt;` and `&gt;`).
- ❑ `<jsp:directive.page .../>` replaces `<%@ page ...%>`.
- ❑ `<jsp:directive.include .../>` replaces `<%@ include ...%>`.
- ❑ `<!-- ... pp>` replaces `<%-- ... --%>`.

Expression Language

- ❑ Expression Language (EL) replaces Java-language syntax expressions.
- ❑ The base syntax is `${expression}`.
- ❑ The result from an EL expression is sent to page output.
- ❑ Any attribute in any scope can be accessed in an expression.
- ❑ An EL expression *cannot* access local variables in `_jspService()` directly.
- ❑ There are five literal types in EL: boolean, integer, floating decimal, string, and **null**.
- ❑ An EL boolean has values of true and false, like Java.
- ❑ An EL integer is any number without a decimal point, while floating decimals have a decimal point.
- ❑ An EL string is denoted by double or single quotes around the literal ("myString" or 'myString').
- ❑ There are five arithmetic operators in EL: +, −, *, / (or div), and % (or mod).
- ❑ There are six relational operators in EL: < (or lt), > (or gt), <= (or le), >= (or ge), != (or ne), and == (eq).
- ❑ There are three logical operators in EL: && (or and), || (or or), and ! (or not).
- ❑ There is also an *empty* operator in EL, which returns true for null, empty strings, empty arrays, and empty collections.
- ❑ EL can access properties on beans with the dot operator.
- ❑ `${bankAccount.balance}` returns a property called balance for an attribute bean in some scope called bankAccount.
- ❑ EL can access items in arrays or java.util.List objects with square bracket syntax.
- ❑ `${daysOfWeek[6]}` accesses the seventh element in an array or java.util.List object associated with an attribute called daysOfWeek.
- ❑ Dot or square bracket syntax can be used to return keyed items in a java.util .Map object.
- ❑ Assuming that "capital" is the name of an attribute holding a java.util.Map, `${capital.Poland}` or `${capital["Poland"]}` would return the value associated with the key "Poland."
- ❑ EL has 11 implicit objects: *pageContext*, *pageScope*, *requestScope*, *sessionScope*, *applicationScope*, *initParam*, *param*, *paramValues*, *header*, *headerValues*, and *cookie*.



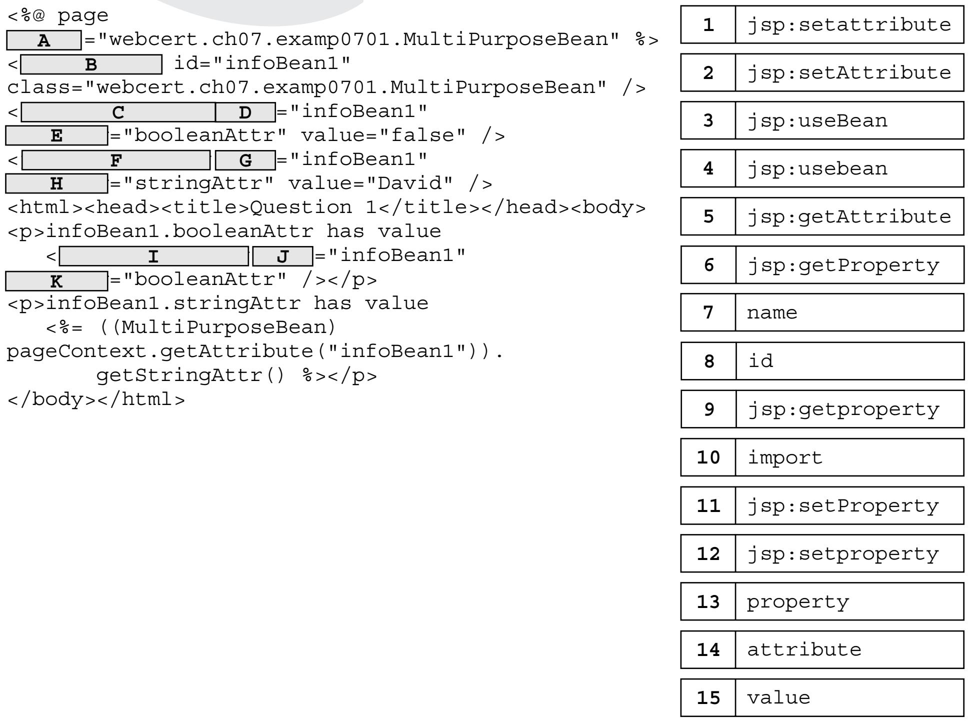
- ❑ *pageContext* can be used to access properties associated with the page's PageContext object.
- ❑ For example, `${pageContext.request.header}` returns the HTTP method associated with the request.
- ❑ *pageScope*, *requestScope*, *sessionScope*, and *applicationScope* can be used to access an attribute in a specific scope.
- ❑ So whereas `${myAttr}` will search through page, request, session, and application scopes for an attribute called myAttr, `${sessionScope.myAttr}` confines the search to session scope.
- ❑ *initParam* returns ServletContext parameter values.
- ❑ *param* returns the first value associated with a named ServletRequest parameter; *paramValues* returns all values.
- ❑ *header* returns the first value associated with a named HttpServletRequest header; *headerValues* returns all values.
- ❑ *cookie* returns a named cookie associated with HttpServletRequest.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there might be more than one correct answer. Choose all the correct answers for each question.

JSP Standard Actions

1. (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.



```
<%@ page
[   A   ]="webcert.ch07.examp0701.MultiPurposeBean" %>
<[     B     ] id="infoBean1"
class="webcert.ch07.examp0701.MultiPurposeBean" />
<[         C         ][ D  ]="infoBean1"
[   E   ]="booleanAttr" value="false" />
<[         F         ][ G  ]="infoBean1"
[   H   ]="stringAttr" value="David" />
<html><head><title>Question 1</title></head><body>
<p>infoBean1.booleanAttr has value
   <[         I         ][ J  ]="infoBean1"
[   K   ]="booleanAttr" /></p>
<p>infoBean1.stringAttr has value
   <%= ((MultiPurposeBean)
pageContext.getAttribute("infoBean1")).
       getStringAttr() %></p>
</body></html>
```

1	jsp:setattribute
2	jsp:setAttribute
3	jsp:useBean
4	jsp:usebean
5	jsp:getAttribute
6	jsp:getProperty
7	name
8	id
9	jsp:getproperty
10	import
11	jsp:setProperty
12	jsp:setproperty
13	property
14	attribute
15	value

2. Which of the following are potentially legal lines of JSP source? (Choose two.)

A.

```
<jsp:useBean id="beanName1" class="a.b.MyBean" type="a.b.MyInterface" />
```

B.

```
<% String className = "a.b.MyBean"; %>
<jsp:useBean id="beanName2" class="<%=className%>" />
```

C.

```
<% String beanName = "beanName3"; %>
<jsp:useBean id="<%=beanName3%>" class="a.b.MyBean" />
```

D.

```
<% String myValue = "myValue"; %>
<jsp:setProperty name="beanName1" property="soleProp" value="<%=myValue%>" />
```

E.

```
<% String propName = "soleProp"; %>
<jsp:getProperty name="beanName1" property="<%=propName%>" />
```

3. Which of the following are false statements about `<jsp:useBean>` standard action attributes? (Choose three.)

   A. If present, the *class* attribute must match the object type of your bean.

   B. If the *type* attribute is used, the *class* attribute must be present.

   C. The reference variable used for a bean doesn't always have the same type as the bean object it refers to.

   D. If both are used, *class* and *type* attributes must have different values.

   E. If both are used, *class* and *type* attributes must have the same value.

   F. If both are used, *class* and *type* attributes can have the same value.

4. Given a NameBean with a "name" property and an AddressBean with an "address" property, what happens when the following JSP is requested with the following URL? (Choose one.)

   Calling URL:

```
http://localhost:8080/examp0701/Question4.jsp?name=David%20Bridgewater&address=
Leeds%20UK
```

   JSP page source:

```
<jsp:useBean id="name" class="webcert.ch07.examp0701.NameBean" />
<jsp:useBean id="address" class="webcert.ch07.examp0701.AddressBean" />
<jsp:setProperty name="name" property="name" />
<jsp:setProperty name="address" param="*" />
<jsp:getProperty name="name" property="name" />
<jsp:getProperty name="address" property="address" />
```

A. A translation time error occurs.
B. A request time error occurs.
C. “null null” is displayed.
D. “David Bridgewater null” is displayed.
E. “null Leeds UK” is displayed.
F. “David Bridgewater Leeds UK” is displayed.

5. Which of the following techniques would correctly put a bean into application scope? (You can assume that any necessary page directives are present and correct elsewhere in the JSP page.) (Choose four.)

A.
```
<jsp:useBean id="app1" class="webcert.ch07.examp0701.AddressBean"
scope="application" />
```

B.
```
<% AddressBean ab2 = new AddressBean();
application.setAttribute("app2", ab2); %>
```

C.
```
<% AddressBean ab3 = new AddressBean();
pageContext.setAttribute("app3", ab3, PageContext.APPLICATION_SCOPE); %>
```

D.
```
<% AddressBean ab4 = new AddressBean();
ServletContext context = getServletContext();
context.setAttribute("app4", ab4); %>
```

E.
```
<% AddressBean ab5 = new AddressBean();
pageContext.setAttribute("app5", ab5); %>
```

F.
```
<jsp:useBean name="app6" class="webcert.ch07.examp0701.AddressBean"
scope="application" />
```

Dispatching Mechanisms

6. Consider the source for the following two JSP pages, a.jsp and b.jsp. What is the outcome of requesting each in turn? You can assume that “c.jsp” is available in the same web application directory as a.jsp and b.jsp. (Choose two.)

Source for a.jsp:

```
<%@page buffer="none" autoFlush="true"%>
<jsp:forward page="c.jsp"/>
```

Source for b.jsp:

```
<%@page buffer="none" autoFlush="true"%><jsp:forward page="c.jsp"/>
```

A. Neither JSP page translates.

B. a.jsp translates; b.jsp does not.

C. b.jsp translates; a.jsp does not.

D. Both JSP pages translate.

E. Neither JSP page runs successfully.

F. a.jsp runs successfully; b.jsp does not.

G. b.jsp runs successfully, a.jsp does not.

H. Both a.jsp and b.jsp run successfully.

7. What is the outcome of making the HTTP GET request shown to params.jsp (source follows)? (Choose one.)

   The HTTP request is in this form:

```
http://localhost:8080/examp0702/params.jsp?X=1&Y=2&Z=3
```

   Source of params.jsp:

```
<jsp:include page="included.jsp">
  <jsp:param name="X" value="4" />
  <jsp:param name="X" value="5" />
  <jsp:param name="Y" value="6" />
</jsp:include>
${param.X}
<%=request.getParameter("Y")%>
```

   Source of included.jsp:

```
${param.X}
${param.Y}
<% String[] x = request.getParameterValues("X");
for (int i = 0; i < x.length; i++) {
  out.write(x[i]);
}
%>
```

A. 1 2 45 4 6
B. 1 2 145 4 6
C. 4 6 451 4 6
D. 1 2 145 1 2
E. 4 6 451 1 2
F. 4 6 45 1 2
G. None of the above

8. Which of the following are helpful statements about the `include` standard action and the `include` directive? (Choose three.)

A. The `include` directive is useful for the inclusion of pages that change frequently.
B. The `include` standard action is useful when soft-coding the page to include.
C. Given the same page to include, the `include` directive may be more efficient than the `include` standard action at request time.
D. The body of the `include` standard action can influence existing request parameters.
E. Given the same page to include, the `include` directive may be more efficient than the include standard action at translation time.
F. An `include` directive can be processed or not according to JSTL, EL, or scriptlet page logic.

9. What will be the result of requesting the JSP page represented by the following source? Assume that "forwardedTo.jsp" is an empty file. (Choose one.)

```
<%@ page import="java.util.*,java.text.*" %>
<%! private String returnTimeStamp(PageContext pageContext) {
  DateFormat df = DateFormat.getDateTimeInstance();
  String s = df.format(new Date());
  pageContext.setAttribute("timestamp", s);
  return s;
} %>
<jsp:forward page="forwardedTo.jsp" />
<%=returnTimeStamp(pageContext)%>
<%System.out.println(pageContext.getAttribute("timestamp"));%>
```

A. Translation error.
B. Run-time error.
C. A formatted date appears in the page output.
D. A formatted date appears in the server console.

E. A formatted date appears both in the page output and in the server console.

F. None of the above.

10. What is the outcome of making the HTTP GET request shown to params.jsp (source follows)? (Choose one.)

The HTTP request is in this form:

```
http://localhost:8080/examp0702/params.jsp?X=1&Y=2&Z=3
```

Source of params.jsp:

```
<jsp:forward page="included.jsp">
  <jsp:param name="X" value="4" />
  <jsp:param name="X" value="5" />
  <jsp:param name="Y" value="6" />
<jsp:forward/>
${param.X}
<%=request.getParameter("Y")%>
```

Source of included.jsp:

```
${param.X}
${param.Y}
<% String[] x = request.getParameterValues("X");
for (int i = 0; i < x.length; i++) {
   out.write(x[i]);
}
%>
```

A. 1 2 145

B. 4 6 451

C. 1 2 145 1 2

D. 4 6 451 1 2

E. 4 6 451 4 6

F. None of the above

JSPs in XML

11. What is the outcome of accessing the following page, defined as a JSP document in a web application? The line numbers are for reference only and should not be considered part of the JSP page source. (Choose one.)

```
01 <html xmlns:jsp="http://java.sun.com/JSP/Page">
02 <jsp:directive.page contentType="text/html" />
03 <jsp:declaration>
04   public int squared(int value) {
05     return value * value;
06   }
07 </jsp:declaration>
08 <jsp:scriptlet>
09   int value = Integer.parseInt
10   (request.getParameter("number"));
11   int squared = squared(value);
12   out.write(value + " squared is " + squared);
13   if (squared < 100) {
14     out.write("; try a bigger number.");
15   }
16 </jsp:scriptlet>
17 </html>
```

A. Translation error at line 1

B. Translation error at line 2

C. Translation error at line 4

D. Translation error at line 12

E. Translation error at line 13

F. Run-time error

G. No errors, with page displaying successfully

12. Which of the following JSP documents will produce output? You can assume that a.b.StringBean exists and has a valid property called "string." (Choose two.)

A.

```
<jsp:useBean id="string" class="a.b.StringBean">
<jsp:setProperty name="string" property="string" value="Question12" />
  <jsp:getProperty name="string" property="string" />
</jsp:useBean>
```

B.

```
<jsp:useBean xmlns:jsp="http://java.sun.com/JSP/Page"
id="string" class="a.b.StringBean">
  <jsp:setProperty name="string" property="string" value="Question12" />
  <jsp:getProperty name="string" property="string" />
</jsp:useBean>
```

C.

```
<jsp:useBean xmlns:jsp="http://java.sun.com/JSP/Page"
id="string" class="a.b.StringBean">
  <jsp:setProperty name="string" property="string" value="Question12" />
  <data><jsp:getProperty name="string" property="string" /></data>
</jsp:useBean>
```

D.

```
<jsp:useBean xmlns:jsp="http://java.sun.com/JSP/Page"
id="string" class="a.b.StringBean">
  <jsp:setProperty name="string" property="string" value="Question12" />
</jsp:useBean>
<data><jsp:getProperty name="string" property="string" /></data>
```

E.

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
<jsp:useBean id="string" class="a.b.StringBean">
  <jsp:setProperty name="string" property="string" value="Question12" />
</jsp:useBean>
<data><jsp:getProperty name="string" property="string" /></data>
</jsp:root>
```

13. Which of the following techniques will cause JSP page source to be treated as a JSP document by the JSP container? (Choose two.)

   A. Setting the `<is-xml>` subelement of `<jsp-config>` to a value of true
   B. Using a .jspx extension with a version 2.4 deployment descriptor
   C. Using a .xml extension with a version 2.4 deployment descriptor
   D. Using `<jsp:root>` as the root element of your source
   E. Using a deployment descriptor at level 2.4
   F. Writing your page source in XML syntax

14. Of the five JSP page source extracts below, there are two pairs. Each member of the pair gives rise to identical output. Which is the odd one out? (Choose one.)

   A.

```
<% int i, j, k;
  i = 1; j = 2; k = 3; %>
<%= i + j / k %>
```

B.

```
<jsp:scriptlet>int i, j, k;
  i = 1; j = 2; k = 3;</jsp:scriptlet>
<jsp:expression>(i + j) / k</jsp:expression>
```

C.

```
<% int i, j, k;
  i = 1; j = 2; k = 3; %>
<%= (i + j) / k + ".0" %>
```

D.

```
<% pageContext.setAttribute("i", new Integer(1));
pageContext.setAttribute("j", new Integer(2));
pageContext.setAttribute("k", new Integer(3));
%>${pageScope.i + pageScope.j / pageScope.k}
```

E.

```
<% pageContext.setAttribute("i", new Integer(1));
pageContext.setAttribute("j", new Integer(2));
pageContext.setAttribute("k", new Integer(3));
%>${(pageScope.i + pageScope.j) / pageScope.k}
```

15. Which of the following tags will successfully complete the following JSP page extract, at the points marked `<jsp:???>` and `</jsp:???>` ? (Choose one.)

```
<html xmlns:jsp="http://java.sun.com/JSP/Page" >
<jsp:directive.page contentType="text/html" />
<head><title>Question 15</title></head>
<jsp:???><![CDATA[<img src="]]></jsp:???>
<jsp:expression>session.getAttribute("theImage")</jsp:expression>
<jsp:???><![CDATA[" />]]></jsp:???>
</html>
```

A. `<jsp:param>` and `</jsp:param>`
B. `<jsp:element>` and `</jsp:element>`
C. `<jsp:img>` and `</jsp:img>`
D. `<jsp:output>` and `</jsp:output>`
E. `<jsp:text>` and `</jsp:text>`

Expression Language

16. What is the consequence of accessing the following JSP page with the URL shown? (Choose one.)

URL for accessing Question16.jsp:

```
http://localhost:8080/examp0704/Question16.jsp?A=1&A=2
```

JSP page source:

```
<!-- Source for Question16.jsp -->
<p>Parameter A has values
<% for (int j = 0; j < request.getParameterValues("A").length; j++) { %>
${paramValues.A[j]},
<% } %> </p>
```

A. Translation error
B. Run-time error
C. Output of: "Parameter A has values 1,2,"
D. Output of: "Parameter A has values 1, ,"
E. Output of: "Parameter A has values 0,0,"
F. Output of: "Parameter A has values , ,"

17. Which of the following are implicit variables in EL? (Choose two.)

A. `session`
B. `param`
C. `paramValues`
D. `initParams`
E. `request`
F. `page`
G. `contextScope`

18. Which of the following EL expressions will return a `<servlet-name>` associated with the JSP executing the expression? (Choose one.)

A.

```
${pageContext.config.getServletName}
```

B.

```
${pageContext.config.servletName}
```

C.

```
${pageContext.servletConfig.servletName}
```

D.

```
${pageContext.servletConfig.getServletName}
```

E.

```
${pageContext.getServletConfig().getServletName()}
```

19. What expression is required at the point marked ??? in the following JSP page to output the number 46? (Choose two.)

```
<html xmlns:jsp="http://java.sun.com/JSP/Page" >
<head><title>Question 19</title></head>
<jsp:output omit-xml-declaration="true" />
<jsp:directive.page contentType="text/html" />
<body>
<jsp:scriptlet>
  request.setAttribute("a", new Integer(2));
  session.setAttribute("b", new Integer(3));
  application.setAttribute("c", new Integer(4));
  request.setAttribute("d", new Integer(5));
</jsp:scriptlet>
???
</body></html>
```

A.

```
${pageContext.c * pageContext.d * pageContext.a
+ pageContext.a * pageContext.b}
```

B.

```
${applicationScope.c * requestScope.d * requestScope.a
+ requestScope.a * sessionScope.b}
```

C.

```
${(applicationScope.c * requestScope.d * requestScope.a)
+ (requestScope.a * sessionScope.b)}
```

D.

```
${(pageContext.c * pageContext.d * pageContext.a)
+ (pageContext.a * pageContext.b)}
```

E.

```
${(application.c * request.d * request.a)
+ (request.a * session.b)}
```

F.

```
${application.c * request.d * request.a
+ request.a * session.b}
```

20. (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 [    A    ]="http://java.sun.com/JSP/Page" >
<head><title>Question 20: Drag and
Drop</title></head>
<jsp:output omit-xml-declaration="true" />
<jsp:[      B      ] contentType="text/html" />
<body>
<jsp:[    C    ]>
  private Integer generateLuckyNumber() {
    Double d = new Double (Math.random() * 100);
    Integer i = new Integer (d.intValue());
    return i;
  }
</jsp:[    D    ]>
<jsp:[    E    ]>pageContext.setAttribute("luckyNo",
generateLuckyNumber());
</jsp:[    F    ]>
<p>Your session cookie has the name
${[  G  ].JSESSIONID.name} (which should come as no
surprise)<br />
and the value ${[  H  ].JSESSIONID.value} (which is
harder to predict).</p>
<br />
<p>Your lucky number for the day is
${[    I    ][[    J    ]]}</p>
</body></html>
```

1	declaration
2	Declaration
3	scriptlet
4	cookies
5	page.directive
6	directive.page
7	Scriptlet
8	Cookie
9	xmlns:jsp
10	xlmns;jsp
11	cookie
12	luckyNo
13	"luckyNo"
14	expression
15	Expression
16	pageContext
17	pageScope
18	page

LAB QUESTION

For this question, you're going to take the solution code from a previous exercise and apply many of the techniques you learned in this chapter. The previous exercise to use is Exercise 6-3 (from Chapter 6), packaged on the CD in /sourcecode/chapter06/ex0603.war. Establish a new context directory called lab07, and unpackage the WAR file into this.

The solution code from Exercise 6-3 displays a short list of countries and capitals. There are four JSPs involved in the solution: master.jsp, which includes header.jsp, setup.jsp, and footer.jsp. Your mission is to turn master.jsp and setup.jsp into JSP documents (i.e., in XML syntax), so rename these to master .jspx and setup.jspx. When including files into master.jspx,

- Include header.jsp and footer.jsp via an `include` directive.
- Include setup.jspx via an `include` standard action.

Other than this, use expression language and standard actions wherever possible. You'll need to revisit header.jsp and setup.jsp so that when these are incorporated into master.jspx, they don't damage the XML syntax. You'll quickly discover any problems as you deploy the JSPs and attempt to access master.jspx.

SELF TEST ANSWERS

JSP Standard Actions

1. ☑ **A** matches with **10** (must be an `import` attribute for the page directive); **B** matches with **3** (only a `<jsp:useBean>` has *id* and *class* attributes); **C** and **F** match with **11** (has to be `<jsp:setProperty>` because of the *value* attribute); **D**, **G**, and **J** match with **7** (must be the *name* attribute in all cases); **E**, **H**, and **K** match with **13** (must be the *property* attribute in all cases); **I** matches with **6** (must be `<jsp:getProperty>` to display the property, which is the clear intention of the code here).
   ☒ No other combinations will work.

2. ☑ **A** and **D**. **A** is correct; it's normal `<jsp:useBean>` syntax. Of course, a.b.MyBean must implement a.b.MyInterface for the action to translate. **D** is also correct. It's a valid `<jsp:setProperty>` element. There's one slightly unusual aspect: The value attribute's value setting comes from a run-time expression. But this is one of the few cases when it's legal to embed a run-time expression inside an action attribute.
   ☒ **B** and **C** are incorrect because you can't use a run-time expression for either the *class* attribute or *id* attribute values of `<jsp:useBean>`. Both must be known at translation time (these values are, effectively, hard-coded in the generated servlet). Because there is compile-time checking done on class existence and validity, the *class* attribute could never be soft-coded. **E** is incorrect for similar reasons: The property name of a `<jsp:getProperty>` element must be known when the servlet is translated, for the generated servlet must select the right method to turn the property into a String for display.

3. ☑ **B**, **D**, and **E** are correct answers, for all are false statements. **B** is a false statement (so a correct answer) because *type* can be used without *class*—in which case, your bean must exist already, or you will get a run-time error. **D** and **E** are false statements (and so correct answers) because it's neither true that *class* and *type* must be different nor that they must be the same.
   ☒ **A** is a true statement (so an incorrect answer), for the *class* attribute does indeed define the object type of the bean. **C** is a true statement (so an incorrect answer) as the type of the reference variable used for the bean can be different from the object type of the bean itself (if *class* and *type* are set differently). **F** is a true statement (so an incorrect answer)—it's pretty pointless setting the *class* and *type* attributes to the same value, but it's still legal. You might as well omit the *type* attribute under these circumstances, though.

4. ☑ **A** is the correct answer: A translation-time error occurs. The second `<jsp:setProperty>` element should have the attribute setting of *property*="*" for the page to translate and compile; *param*="*" is incorrect syntax.

☒ **B** is incorrect because the incorrect syntax of the standard action prevents translation. **C**, **D**, **E**, and **F** are incorrect because there is no output (**F** would be the correct answer if the syntax error were corrected).

5. ☑ **A**, **B**, **C**, and **D** are correct. **A** is the `<jsp:useBean>` standard action used exactly as it should be to create a bean in application scope. **B** sets up a bean in a scriptlet and uses the *application* implicit object to set the bean up as an attribute. **C** also uses a scriptlet but uses the three-parameter version of `pageContext.setAttribute` to provide the name, the bean, and the scope of the attribute. **D** again uses a scriptlet—there's more manual work this time, getting hold of the servlet context with the `getServletContext()` method instead of using the *application* implicit object—but the net result is still as intended.
☒ **E** is incorrect because using the two-parameter version of `PageContext.setAttribute()` results in a bean being placed in page scope, not application scope. **F** is incorrect because *name* is used instead of *id* (*name* is a valid attribute of `<jsp:getProperty>` and `<jsp:setProperty>` but not of `<jsp:useBean>`).

Dispatching Mechanisms

6. ☑ **D** and **G**. Both pages translate successfully (so answer **D** is correct). However, b.jsp runs successfully (forwarding to c.jsp, whose output is displayed), whereas a.jsp terminates with an IllegalStateException when run (hence answer **G** is also correct). Why should this be? The only material difference between the sources for a.jsp and b.jsp is the carriage return separating the page directive from the `<jsp:forward>` standard action. This is present in a.jsp, but not in b.jsp. To understand why this should make a difference, you need to note that the page directive effectively does away with the normal output buffer (by setting ***buffer***=*"none"* and ***autoFlush***=*"true"*). This means that any output at all – *even an innocent carriage return in the template text* – is instantly committed to the response output. Once anything has been committed to the response output, a forward call is illegal.
☒ **A**, **B**, **C**, **E**, **F**, and **H** are incorrect, according to the reasoning in the correct answer.

7. ☑ **E** is correct. Consider first of all that parameter Z is not displayed in either JSP, so it is a red herring. On arrival at params.jsp, the request has parameter X with a value of 1, and Y with a value of 2. Now X is supplemented with two additional values. These are placed in order of their appearance in `<jsp:param>` standard actions, but at the "front" of the parameter's value list. So X's values are 4, 5, and 1—in that order. Y is supplemented with one additional value, making its values 6 and 2—again, in that order. Because the first instruction is to include the page included.jsp, we must go there first. The *param* EL implicit object is used to display

the value of X. *param* retrieves the first available value, so we have our first output, 4. This technique is repeated for Y, so the next output is 6. Now a scriptlet is used to iterate through all of the values of parameter X. The method `request.getParameterValues()` will respect the correct order, so the next output is 451 (the three values of X in succession). Now we return back to params.jsp. Any parameter values added within the body of the `<jsp:include>` action are lost. So X has only a single value of 1, and Y a single value of 2. These are displayed—first 1 (from X with the same EL technique we saw before), then 2 (from Y, retrieved with a Java language expression using the `ServletRequest.getParameter()` method).
☒ **A**, **B**, **C**, **D**, **F**, and **G** are incorrect, according to the reasoning in the correct answer.

8. ☑ **B**, **C**, and **D** are the correct answers. **B** is correct: You can supply an expression for the value of the page attribute of the include standard action, and thereby soft-code your choice of page. **C** is correct, though it's not absolutely clear-cut (hence "*may* be more efficient"). The `include` directive probably involves harder work at translation time. But unlike the standard action, everything that's needed is then there in the same servlet. The `<jsp:include>` standard action will involve a request-time trip to the included file, translating this if not translated already, and returning the response for inclusion in the including servlet. **D** is correct: By including a `<jsp:param>` standard action in the body of a `<jsp:include>`, you can augment the values of existing request parameters.
☒ **A** is incorrect—it's the `<jsp:include>` standard action that is best for including pages that change frequently, not the `include` directive. You're guaranteed with the `<jsp:include>` standard action that the latest version of the included page will be processed; the `include` directive doesn't have the same guarantee in the JSP spec. Even if your JSP container provides that guarantee, there will be more to translate (both including and included pages have to be amalgamated and translated with the `include` directive; whereas only the included page has to be revamped when using the `<jsp:include>` standard action). **E** is incorrect. It's not obvious whether the include directive or the `<jsp:include>` standard action will win out at translation time. Let's say page P includes page Q. With the `include` directive, Q must be merged into P, and then there is one big servlet to translate and compile. With the `<jsp:include>` standard action, P and Q stay separate and are translated and compiled into two separate servlets. Who can say which will be processed more quickly? So I deem this to be an unhelpful statement. **F** is out and out incorrect—an `include` directive can't be influenced by page logic (unlike a `<jsp:include>` standard action). It will be processed at translation time come what may (unless it's commented out!).

9. ☑ **F** is the correct answer. A blank page is output, and nothing is output to the server console. The crucial thing to recognize is that a `<jsp:forward>` standard action effectively causes the rest of the page logic to be bypassed (do not pass go; do not collect $200/£200 . . .).

☒ **A** is incorrect—there's nothing to cause a page error (including forwarding to a .jsp file containing nothing at all—that's still legal). **B** is incorrect—there's nothing to cause a run-time error either. **C** is incorrect and shows a misunderstanding of forwarding: The use of forward negates any page output from the forwarding JSP. **D** is incorrect, though you could be forgiven for thinking otherwise, for a regular servlet behaves differently. The `System.out.println()` statement is an innocent bystander that has nothing to do with JSP page output, after all. However, it is bypassed as explained in the correct answer. **E** is incorrect for a combination of the reasons given for **C** and **D**.

10. ☑ **F** is the correct answer. In fact, this page will not translate because of the malformed end tag for the forward standard action: It should be `</jsp:forward>`. This question uses a number of evil psychological techniques that are not unknown on the real exam. First, it looks almost identical to a previous question (question 7), so you tend to assume the same kind of approach will pay off—and waste time trying to work out what the code is actually doing. Secondly, there is a decoy red herring. Because the forwarded-to page is called included.jsp, a casual glance makes you think this is a question about including. Then you spot this obvious mistake and think "Aha! This is really a question about forwarding!"—when really it's a syntax question all along. If there's a moral (and I'm not sure there is), then I suppose it's to be on the alert for syntax errors first and foremost. You have to become the translation phase!
☒ **A**, **B**, **C**, **D**, and **E** are all incorrect because the syntax error mentioned prevents translation, so there's no output at all. For what it's worth, if the syntax error were corrected, **B** would be the correct answer.

JSPs in XML

11. ☑ **E** is the correct answer. There will be a translation error at line 13. The "<" sign is illegal XML syntax within the `<jsp:scriptlet>` tag (or indeed, in any tag), for to the parser it looks like another tag beginning before the present one has ended. You have to "escape" the sign in some way—for example, by writing `&lt;`.

☒ **A** is incorrect because the `<html>` tag is correctly formed, including the XML namespace component (remember—this can appear in any tag you want). **B** is incorrect because the page directive is correctly formed, with a legal MIME value. **C** is incorrect because a Java method signature within a `<jsp:declaration>` is normal practice. **D** is incorrect because access to implicit variables like `out` is still perfectly OK, and there's nothing wrong with the Java syntax (and nothing to offend XML syntax). **F** and **G** are incorrect—since there is a translation error, the page won't produce a run-time error, let alone display successfully.

12. ☑ **C** and **E** are the correct answers. Both produce output: **C** produces valid XML output with an XML declaration, and **E** produces valid non-XML output without an XML declaration.

☒ **A** is incorrect because when using XML syntax, you have to supply a namespace for any tags that you use (unlike JSP Syntax, where standard actions are found even without a `taglib` directive). **B** is incorrect because although output is produced, it's identified as XML output (has an XML declaration)—but doesn't constitute valid XML (because the output, "Question12," isn't surrounded by any tags. That's solved by the inclusion of the <data> tag in correct answer **C**. **D** doesn't work because it's invalid JSP source in XML terms: There are two top-level tags. You can have only one top-level tag in an XML document, including JSP page source—so the page doesn't translate (that's a problem solved by the `<jsp:root>` element in correct answer **E**).

13. ☑ **B** and **D**. **B** is correct—provided that web.xml is at version 2.4, a file with a .jspx extension will be recognized as a JSP document. **D** is also correct—`<jsp:root>` at the root element of your source constitutes a JSP document, even at prior levels of web.xml.
☒ **A** is incorrect, though very nearly correct: `<is-xml>` is a correct element, but it's a subelement of `<jsp-property-group>`, which is in turn a subelement of `<jsp-config>`. **C** is incorrect—although you can use XML files directly as JSP page source, an .xml suffix is insufficient to identify them as XML JSP page source to the container. **E** is incorrect—just using a level 2.4 deployment descriptor won't do anything by itself toward interpretation of your JSP page sources as JSP documents. And finally, **F** is incorrect—you can write a syntactically correct JSP page in XML syntax and still have it treated as JSP syntax (rather than as a JSP document). The mere presence of XML in the source is not enough.

14. ☑ **D** is the correct answer. This is not an easy question! The output is 1.6666666666666665 from this piece of code, and none of the other JSP fragments give that result. Owing to precedence rules, the division in the expression (2/3) is done first. EL division is double-based (not integer-based), hence the imprecise double answer. This is added to 1, to give the answer shown.
☒ **A**, **B**, **C**, and **E** are incorrect answers, for all pair up: **A** and **B** produce identical output, and so do **C** and **E**. **A** and **B** both output 1, for different reasons. In **A**, precedence dictates that the division is done first. This is Java language division, so the result of 2/3 is 0. Adding this to 1 therefore gives 1. The parentheses force a different calculation in **B**: (1 + 2)/3. The result of this is also 1. **C** and **E** both output 1.0. **C** does the same calculation—(1 + 2)/3. This is then concatenated with ".0" to give an end result of 1.0. In **E** the EL expression performs effectively the same calculation, derived from the Integer page attributes: (1 + 2)/3. However, for division, EL arithmetic coerces the operands to Doubles. The result is a Double literal (in Java terms), expressed with a .0 on the end even though this isn't required.

15. ☑ **E** is the correct answer. The HTML `<img>` element is broken up to accommodate the expression that soft-codes the images to display. The broken pieces are incorrect XML, so they have to be treated as character data and accordingly have to be wrapped up in XML's

intimidating CDATA syntax. However, that leaves the two pieces of character data unenclosed by tags, making the JSP page source illegal XML. Under these circumstances, you use the `<jsp:text>` element, which satisfies this requirement and does nothing else. A bland but useful tag.
☒ **A** is incorrect—`<jsp:param>` is a standard action for enclosing in `<jsp:forward>` and `<jsp:include>`. **B** is incorrect—there is a tag called `<jsp:element>`, but it's for soft-coding XML elements to include in the output, and won't do here. **C** is incorrect, for `<jsp:img>` is made up. Finally, **D** is incorrect—`<jsp:output>` is a tag that exists, but for adjusting the output type of the document produced by the JSP, so it has no role to play in solving the problem posed here.

Expression Language

16. ☑ **F** is the correct answer. The local counter variable, *j,* is not visible inside the code that evaluates the expression. The expression evaluation code will try to find an attribute called "j" using `PageContext.findAttribute("j")`. Because *j* won't be there, the expression returns nothing at all, so only the template text appears.
☒ **A**, **B**, **C**, **D**, and **E** are incorrect according to the reasoning for the correct answer.

17. ☑ **B** and **C** are the correct answers—*param* is used to get hold of the first value for a named request parameter, and *paramValues* to get hold of all the values for a named request parameter.
☒ **A**, **E**, and **F** are incorrect—although *page, request,* and *session* are implicit variables in Java language scriptlets and expressions, they don't work directly inside expressions. Instead, use `pageContext.page`, `pageContext.request`, and `pageContext.session`. **D** is incorrect, but not by much: use *initParam* (no "s") to get hold of a named context initialization parameter. And **G** is incorrect—there's an *applicationScope* implicit variable, but not a *contextScope*.

18. ☑ **C** is the correct answer. The implicit variable *pageContext* represents a PageContext object. This has a `getServletConfig()` method to return the ServletConfig object associated with the PageContext. ServletConfig, in turn, has a `getServletName()` method to return the servlet name. To turn this into EL syntax, you take each method name and strip off the "get" and the terminating parentheses. What you're left with is a bean property name—as long as you turn the first capital letter now into lowercase. Put the dot operators between each, and you get `${pageContext.servletconfig.servletName}`, which works.
☒ **A** is incorrect—there is no config property for PageContext (it's `servletConfig`), and a method name (`getServletName`) even without the parentheses won't work. **B** is incorrect, though closer: `config` just has to change to `servletConfig`. **D** has one of the faults of **A**, whereas **E** is wrong altogether because it uses Java language syntax. By using correct method names, though, **E** does furnish a clue toward the correct property names when using EL syntax.

19. ☑ **B** and **C** are the correct answers. The implicit variables ending in "scope" are the ones to go for when trying to retrieve attributes. In this example, it doesn't matter if the parentheses are used to group the arguments in the expression (as in **C**)—precedence takes care of the result (as in **B**).
☒ **A**, **D**, **E**, and **F** are incorrect. **A** and **D** are incorrect because although *pageContext* is an EL implicit variable, you can't use it like this to access attributes in any scope. **E** and **F** are incorrect because they use implicit variable names that are legal in Java language scripting, but not in EL.

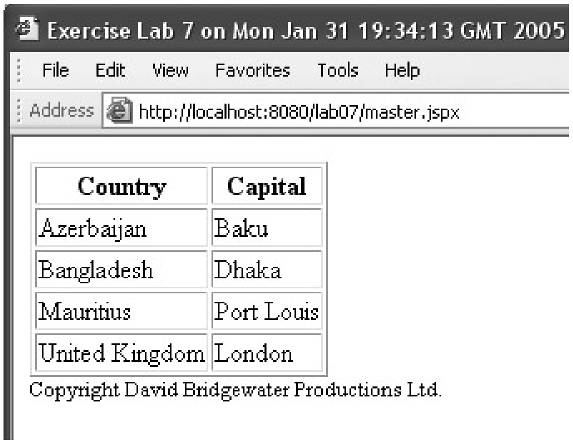
20. ☑ **A** matches with **9** (the `xmlns:jsp` announces a namespace declaration); **B** matches with **6** (the correct answer is `directive.page`—not `page.directive`!); **C** and **D** match with **1** (must be the beginning and end of a `<jsp:declaration>`, for the content is an entire Java method); **E** and **F** match with **3** (beginning and end of a `<jsp:scriptlet>`; **G** and **H** match with **11** (`cookie` is the implicit variable name; there are some misleading alternatives); **I** matches with **17** (`pageScope` is clearly where the "luckyNo" attribute is stored); and **J** matches with **13** ("luckyNo"—needs to be a String to return the named attribute value with square bracket syntax—dot syntax would probably be better here, as in `${pageScope.luckyNo}`).
☒ No other combinations will work.

LAB ANSWER

Deploy the WAR file from the CD called lab07.war, in the /sourcecode/chapter07 directory. This contains a sample solution. Once the WAR file is deployed, you can call the top-level JSP document, master.jspx, using a URL such as

```
http://localhost:8080/lab07/master.jspx
```

The resulting solution should look as it did in Exercise 6-3, and is shown in the following illustration.


Exercise Lab 7 on Mon Jan 31 19:34:13 GMT 2005
File Edit View Favorites Tools Help
Address http://localhost:8080/lab07/master.jspx

Country	Capital
Azerbaijan	Baku
Bangladesh	Dhaka
Mauritius	Port Louis
United Kingdom	London

Copyright David Bridgewater Productions Ltd.