5

Basic Principles of JSPs

IN THIS CHAPTER

- What Is a JSP?
- Deploying a JSP in Tomcat
- Elements of a JSP Page
- Currency Converter JSP
- Using JavaBeans in a JSP
- The Currency Converter JSP Using a JavaBean
- The JSP Life Cycle
- JSPs and the Deployment Descriptor File

Chapter 4, "Basic Principles of Servlets," introduced you to simple Web applications using servlets. Although very useful for writing dynamic server-side code, servlets suffer from some disadvantages. In particular, coding complex HTML pages is somewhat tedious and error prone (due to the need to pepper the code with `out.println()` statements), and a servlet developer has to take on the dual roles of developer of application logic and designer of Web pages. JavaServer Pages (JSPs) were designed to address these disadvantages.

In this chapter, you will study the JSP syntax and the translate-compile life cycle used to generate a servlet that services the HTTP request using the information in the JSP Web page.

What Is a JSP?

A JSP is "just a servlet by another name" and is used in the same way to generate dynamic HTML pages. Unlike a servlet, which you deploy as a Java class, a JSP is deployed in a textual form similar to a simple HTML page. When the JSP is first accessed, it is translated into a Java class and compiled. The JSP then services HTTP requests like any other servlet.

A JSP consists of a mix of HTML elements and special JSP elements. You use the JSP elements, among other things, to embed Java code in the HTML. For example, Listing 5.1 shows a simple JSP that prints out the current date. The special JSP scripting element `<%= ... %>` is used to introduce the Java code.

LISTING 5.1 A Simple JSP

```
<HTML>
    <HEAD>
       <TITLE>Date JSP</TITLE>
    </HEAD>
    <BODY>
        <BIG>
            Today's date is <%= new java.util.Date() %>
        </BIG>
    </BODY>
</HTML>
```

Before examining JSP elements in more detail, you will see what Tomcat does when this JSP is deployed.

Deploying a JSP in Tomcat

Deploying a JSP is very simple. There are only a couple of rules:

- JSPs are stored in their textual form in the application directory.
- A JSP must have a `.jsp` suffix.

So, to deploy the JSP in Listing 5.1:

1. Create a new Web application directory in Tomcat's `webapps` directory; call it `basic-jsp`.
2. Create a subdirectory called `WEB-INF` in the `basic-jsp` directory.
3. Copy Listing 5.1 into a file called `date.jsp` in this application directory.
4. Stop and restart Tomcat so it recognizes the new Web application.

Access the JSP using the URL `http://localhost:8080/basic-jsp/date.jsp`. If there are no errors, you will see a screen similar to the one in Figure 5.1.

For this simple example, it is unlikely that you experienced any problems, but a more complex JSP might have generated some errors. The next section on the JSP translate-compile cycle will explain how to handle these errors. If you do not find the answer to your problem there, Chapter 6, "Troubleshooting Servlets and JSPs," covers the subject in far greater detail.

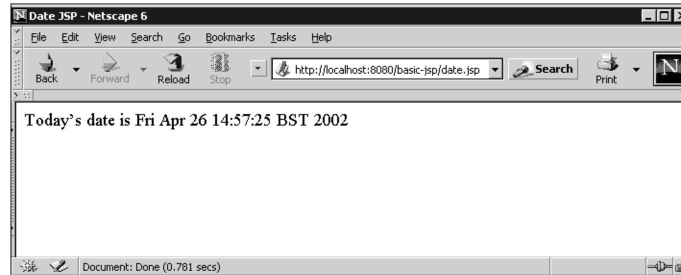

FIGURE 5.1 `date.jsp` displayed using Netscape 6.

JSP Translate—Compile Cycle

Unlike the servlets covered in Chapter 4, JSPs are not compiled before deployment.

NOTE

Actually, it is possible to compile JSPs and deploy them in exactly the same way as servlets, but this is not normally done and will not be covered.

The JSP file is stored in its textual form in the Web application. When a client first requests the page, Tomcat (or more correctly the Jasper JSP container) automatically initiates the translate-compile cycle illustrated in Figure 5.2. This diagram assumes the translation and compilations are both successful. Later in this chapter we discuss how to fix any failures in this cycle.

The JSP is first translated into Java source code, and if there are no translation errors, it is then compiled into a servlet class file.

The translation and compilation obviously causes an initial delay for the first access. If all is well, the JSP will be displayed, but if the translation or compilation fails, the client will be presented with an HTTP 500 "Internal Server Error" accompanied by an error message from Jasper and a Java compiler stack trace. To prevent end users from seeing this, you should always force the translation and compilation of your JSP by making the first page request yourself.

Tomcat implements a mechanism for performing the translate-compile cycle without displaying the JSP. This is achieved by appending the special query string `?jsp_precompile=true` to the URL. This parameter is not passed to the JSP. Instead, it is taken as an instruction for Jasper to translate and compile the JSP. Even if the compilation is successful the JSP is not displayed. This has several advantages:

- In complex applications, you do not need to set up the environment for the JSP.
- Page parameters do not have to be passed to the JSP.
- It simplifies the creation of automatic JSP compilation scripts.

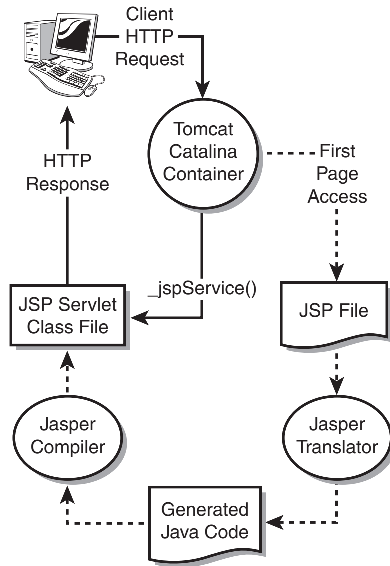

FIGURE 5.2 The Tomcat JSP translate-compile cycle.

If the translate-compile cycle fails, the client will receive the Tomcat error as normal. Figure 5.3 shows the how errors are handled in the translate-compile cycle.

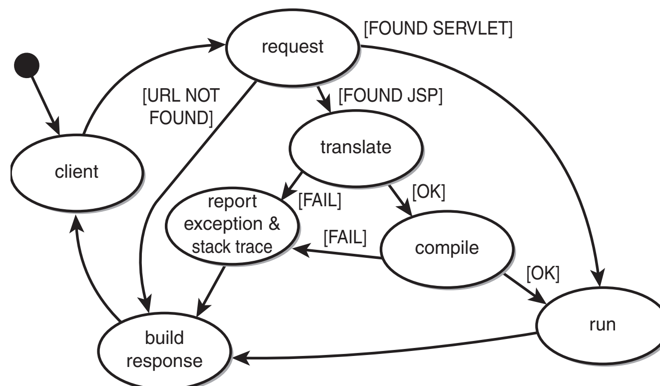

FIGURE 5.3 Error handling during the JSP translate-compile cycle.

Fixing JSP Errors

If the translation fails, Tomcat will report the error in the browser window with a reference to the line in the original JSP where the error occurred. It should then be a simple matter to correct the error and redeploy the JSP.

If there are no translation errors, Tomcat generates a Java source code file for the JSP. The default location for this file is in a Web application directory under `<CATALINA_HOME>/work/<host>/`.

Tomcat then compiles this Java file. If an error occurs during compilation, the error message might be more vague. For example:

```
An error occurred between lines: 7 and 17 in the jsp file: /example.jsp
```

Fortunately, this is always followed by a reference to the translated source code file. Here the error is on line 77 of the translated file:

```
Generated servlet error:
D:\Apache Tomcat 4.0\work\localhost\basic-jsp\example$jsp.java:77:
➥ Class org.apache.jsp. Currency not found.
    Currency dollars = Currency.getInstance(Locale.US);
    ^
1 error, 1 warning
```

From this you will often be able to identify the Java problem, but if not, you can view the generated Java file to get a more complete picture. Listing 5.2 shows the file generated for the simple `date.jsp` in Listing 5.1. As you can see, lines from the original JSP are included as comments.

LISTING 5.2 Generated Servlet Code for `date.jsp`

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;

public class date$jsp extends HttpJspBase {

    static {
    }
    public date$jsp( ) {
    }

    private static boolean _jspx_inited = false;
```

LISTING 5.2 Continued

```
    public final void _jspx_init()
➥throws org.apache.jasper.runtime.JspException {
    }

    public void _jspService(HttpServletRequest request,
➥ HttpServletResponse  response)
        throws java.io.IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String  _value = null;
        try {

            if (_jspx_inited == false) {
                synchronized (this) {
                    if (_jspx_inited == false) {
                        _jspx_init();
                        _jspx_inited = true;
                    }
                }
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=ISO-8859-1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                      "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            // HTML // begin [file="/date.jsp";from=(0,0);to=(6,28)]
                out.write("<HTML>\r\n    <HEAD>\r\n
➥ <TITLE>Date JSP</TITLE>\r\n    </HEAD>\r\n
➥ <BODY>\r\n        <BIG>\r\n            Today's date is ");
```

LISTING 5.2 Continued

```
            // end
            // begin [file="/date.jsp";from=(6,31);to=(6,53)]
                out.print( new java.util.Date() );
            // end
            // HTML // begin [file="/date.jsp";from=(6,55);to=(10,0)]
                out.write("\r\n        </BIG>\r\n    </BODY>\r\n</HTML>\r\n");

            // end

        } catch (Throwable t) {
            if (out != null && out.getBufferSize() != 0)
                out.clearBuffer();
            if (pageContext != null) pageContext.handlePageException(t);
        } finally {
            if (_jspxFactory != null) _
                jspxFactory.releasePageContext(pageContext);
        }
    }
}
```

CAUTION

This generated file will be overwritten when the JSP is next accessed, so fix the error in the original JSP and not in this generated file.

The Compiled JSP Class

The Java code you write in your JSP is placed in a Java class with a name supplied by Tomcat. The majority of the code is placed in a method called `_jspService()`, which has the following signature:

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
                 throws ServletException, java.io.IOException;
```

Tomcat imports the following packages into the translated Java class, so there is no need to include these in your JSP:

```
javax.servlet.*
javax.servlet.http.*
javax.servlet.jsp.*
org.apache.jasper.runtime.*
```

There are also a number of implicitly declared objects that are available for use in JSPs. These are listed in Table 5.1.

TABLE 5.1 JSP Implicit Objects

JSP Name	Java Servlet Object
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
config	javax.servlet.ServletConfig
pageContext	javax.servlet.jsp.PageContext
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext
out	javax.servlet.jsp.JspWriter
page	java.lang.Object (in the Java code on the JSP, this is a synonym for the this object)
exception	java.lang.Throwable (available only if the JSP is designated an error page)

NOTE

We will discuss the use of each of these objects in some detail in Chapter 7, "The Web Application Environment."

Let's now turn our attention to the elements that make up a JSP page.

Elements of a JSP Page

As you saw in Listing 5.1, a JSP element is embedded in otherwise static HTML. Similar to HTML all JSP elements are enclosed in open and close angle brackets (< >). Unlike HTML (but like XML), all JSP elements are case sensitive. JSP elements are distinguished from HTML tags by beginning with either <% or <jsp:.

NOTE

Although the standard JSP syntax is similar to XML, it does not completely conform to the XML syntax rules. The JSP 1.2 specification defines an alternative XML syntax for JSPs and uses the term *JSP document* to refer to a JSP that is a well-formed XML document. A JSP document has a number of advantages, but as yet it is not in common use. This book uses the standard JSP syntax.

There are three basic JSP element types:

- Directives
- Scripting elements
- Action elements

Each of these elements is discussed in turn in the following sections.

JSP Directives

Directives have the following syntax:

```
<%@ directive { attr="value" }* %>
```

They provide information used to control the translation of the JSP into Java code. There are three directives:

- `include`
- `taglib`
- `page`

The `include` Directive

The `include` directive is used, as the name suggests, to bring in static content from another file. The following example shows how to add a copyright notice to the JSP:

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

This directive is acted on at translate time, and the file is subjected to the Tomcat translate-compile cycle.

The `taglib` Directive

The `taglib` directive is used to introduce a tag library to the JSP. Tag libraries are a convenient way of encapsulating code and extending the functionality of JSP tags. Tag libraries are covered in detail in Chapter 10, “Custom Tags and TagLibs,” and Chapter 11, “JSP Expression Language and JSTL”

Page Directives

Page directives are used to define properties for the page together with any files incorporated via the `include` directive. Table 5.2 provides a list of the common page directives (for others see the JSP specification) with examples of how they are used.

TABLE 5.2 Common JSP Page Directives

Directive	Example	Effect
buffer	<%@ page buffer="8kb" %>	Set the output buffer size. A value of none indicates that the output is not buffered. The default is buffered.
autoFlush	<%@ page autoflush="false" %>	Specifies whether the output buffer should be automatically flushed when it is full. The default is true.
contentType	<%@ page contentType="text/ plain;ISO-8859-1"	Defines the MIME type and char acter encoding for the response. MIME type text/html and character encoding ISO-8859-1 are the defaults.
errorPage	<%@ page errorPage="/error/messedup.jsp"	The client will be redirected to the specified URL when an exception occurs that is not caught by the current page.
isErrorPage	<%@ page isErrorPage="true" %>	Indicates whether this page is the target URL for an errorPage directive. If true, an implicit scripting java.lang.Exception object called exception is defined.
import	<%@ page import=" java.math.*"	A comma-separated list of package names to be imported for this JSP. The following are imported by default and do not need to be specified in the JSP: java.lang.* javax.servlet.* javax.servlet.jsp.* javax.servlet.http.* org.apache.jasper.runtime.*
isThreadSafe	<%@ page isThreadSafe="false" %>	If set to true, this page can be run multithreaded. This is the default, so you should ensure access to shared objects is synchronized.
session	<%@ page session="false" %>	Set to true if the page is to participate in an HTTP session . This is the default.

Listing 5.3 shows the `date.jsp` with page directives to import the `java.util` package and construct the response using a Central European character set. (The use of page directives is covered in greater detail in Chapter 7.)

LISTING 5.3 `date.jsp` with Page Directives

```
<%@ page import="java.util.*" contentType="text/html;iso-8859-2" %>
<HTML>
    <HEAD>
       <TITLE>Date JSP</TITLE>
    </HEAD>
    <BODY>
        <BIG>
            Today's date is <%= new Date() %>
        </BIG>
    </BODY>
</HTML>
```

JSP Scripting Elements

Scripting elements come in the three forms described in Table 5.3.

TABLE 5.3 JSP Scripting Elements

Scripting Element	Example	Description
Declarations	`<%! String msg="Hello"; %>`	Declarations of variables or methods, each one separated by semicolons. Variables declared this way are declared in the class itself and not inside the `_jspService()` method.
Scriptlets	`<% int i = 42;` `answer = select[i];%>`	Elements that contain Java code fragments. These are added to the `_jspService()` method.
Expressions	`<%= msg %>`	JSP expressions are statements that are evaluated, the result cast into a `String` and placed on the JSP page.

CAUTION

Declarations and scriptlets contain normal Java code and require semicolons. Expressions are embedded in `out.print()` statements and should not be terminated with a semicolon. Misplacing semicolons is a common source of JSP compilation errors.

We will cover the use of each of these scripting elements later in this chapter.

JSP Action Elements

The final JSP element type is an action element. The JSP specification defines some standard actions, three of which are concerned with support for JavaBeans. You can also create your own custom actions with the use of tag libraries.

Action elements conform to XML/Namespace syntax with a start tag, an optional body, and an end tag. The JSP specification reserves the special `<jsp:` suffix for standard actions.

Action elements associated with JavaBeans are covered later in this chapter, whereas the other remaining action elements are covered in Chapter 7.

You will now rewrite the Currency Converter example as a JSP.

Currency Converter JSP

Listing 5.4 shows the Currency Converter servlet from Chapter 4, rewritten as a JSP.

LISTING 5.4 `currency-converter.jsp`

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

<%! private static final double EXCHANGE_RATE = 0.613; %>

<%
    Currency dollars = Currency.getInstance(Locale.US);
    Currency pounds = Currency.getInstance(Locale.UK);
    String amount = request.getParameter("amount");
    try {
        NumberFormat nf = NumberFormat.getInstance();
        double newValue = nf.parse(amount).doubleValue();
        poundValue *= EXCHANGE_RATE ;
        nf.setMaximumFractionDigits(pounds.getDefaultFractionDigits());
        nf.setMinimumFractionDigits(pounds.getDefaultFractionDigits());
%>

<HTML>
  <HEAD><TITLE>Currency Conversion JSP</TITLE></HEAD>
  <BODY>
    <BIG>
        <%= dollars.getSymbol(Locale.US) %> <%= amount %> =
➥ <%= pounds.getSymbol(Locale.UK) %> <%= nf.format(poundValue) %>
```

LISTING 5.4 Continued

```
    </BIG>
  </BODY>
</HTML>

<%
    }
    catch (ParseException e) {
        out.println ("Bad number format");
    }
%>
```

As you can see, the majority of the code is encapsulated in two scriptlets, and the HTML formatting is both more obvious and more concise. JSP expressions have been used to output the dollar amount and the converted pound value.

Deploy (copy) this JSP to the `webapps/basic-jsp` directory. Either call this JSP directly, providing the amount as a parameter to the URL, like this:

```
http://localhost:8080/basic-jsp/currency-converter.jsp?amount=3
```

Or edit the `currency-form.html` file from Chapter 4 to call the new JSP as shown in Listing 5.5. Copy this form to `webapps/basic-jsp` directory.

LISTING 5.5 Currency Converter HTML Form Accessing a JSP

```
<HTML>
  <HEAD><TITLE>Currency Form</TITLE></HEAD>
  <BODY>
    <FORM METHOD=GET ACTION="currency-converter.jsp">
      <H1>Simple Currency Converter</H1>
      <P>Use this form to convert US Dollars to UK Pounds Sterling</P>
      Type in the amount to be converted in the box
      <INPUT TYPE=TEXT NAME="amount">
      <INPUT TYPE=SUBMIT>
    </FORM>
  </BODY>
</HTML>
```

From the simple currency converter example shown in Listing 5.4 you can start to see how, with JSPs, it is possible to separate the roles of HTML designer from Java programmer. With the use of JavaBeans, this separation can be made greater as discussed in the next section.

Using JavaBeans in a JSP

The use of implicit object declarations, JSP expressions, and scripting elements goes a long way to simplify and encapsulate the Java code, but the amount of code in a JSP can be reduced further using JavaBeans.

JavaBeans are self-contained, reusable software components that must be written to conform to a particular design convention (called an idiom). The convention is

- A JavaBean class must have a no argument constructor (this is what you get by default).
- A JavaBean can provide properties that allow customization of the bean.
- For each property, you should define a *getter* method to retrieve the property and a *setter* method to modify it.
- A getter method cannot have any parameters and must return an object of the type of the property.
- A setter method must take a single parameter of the type of the property and return a void.

Let's look at a JavaBean written for the Currency Converter application.

Currency Converter Using a JavaBean

Adding a JavaBean to the Currency Converter application involves the following steps:

1. Create a `CurrencyConverterBean` class.
2. Add methods to set and get the US dollar amount.
3. Add a method to get the converted UK pound value.

Listing 5.6 shows the code for the `CurrencyConverterBean` class.

NOTE

For the sake of simplicity, the code to output the currency symbols is included in the getter methods. You might want to split this out into discrete methods.

LISTING 5.6 `CurrencyConverterBean`

```
package converters;

import java.util.*;
import java.text.*;

public class CurrencyConverterBean {

    private static final double EXCHANGE_RATE = 0.613;
    private Currency dollars = Currency.getInstance(Locale.US);
    private Currency pounds = Currency.getInstance(Locale.UK);
    private NumberFormat nf = NumberFormat.getInstance();
    private String amount;

public void setAmount (String amount){
        this.amount = amount;
    }

    public String getAmount () {
        return dollars.getSymbol(Locale.US) + amount;
    }

    public String getPoundValue(){
        try {
            double newValue = nf.parse(amount).doubleValue();
            newValue *= EXCHANGE_RATE ;
            nf.setMaximumFractionDigits(pounds.getDefaultFractionDigits());
            nf.setMinimumFractionDigits(pounds.getDefaultFractionDigits());
            return pounds.getSymbol(Locale.UK) + nf.format(newValue);
             }
        catch (ParseException e) {
            return ("Bad number format");
        }
    }
}
```

NOTE

Because the default no argument JavaBean constructor is not used, it has been omitted.

Support for JavaBeans is provided in JSPs with the use of the three standard action elements `<jsp:useBean>`, `<jsp:setProperty>`, and `<jsp:getProperty>`. Each of these will now be covered in turn.

Using a JavaBean

JavaBeans are defined in a JSP using the standard action element `<jsp:useBean>`, as follows:

```
<jsp:useBean id="<bean name>" class="<bean class>" scope="<scope>">
```

An example is

```
<jsp:useBean id="converter" class="converters.CurrencyConverterBean"
➥ scope="page" />
```

This tag creates an instance of the `CurrencyConverterBean` class and associates it with the `id` attribute `converter` for use in the entire JSP page, as defined by the scope attribute.

Other values for the scope attribute are as follows:

`<jsp:useBean>` **Scope Attribute**	**JavaBean Is in Scope For…**
`page`	Only this page
`request`	This page and any page the request is forwarded to
`session`	The duration of the session
`application`	All components in the Web application

NOTE

Because the scope of JavaBeans can extend beyond the page, they are a useful way of passing information between requests.

Getting and Setting Bean Properties

You retrieve JavaBean properties using the standard element `<jsp:getProperty>`, as shown in the following example:

```
<jsp:getProperty name="converter" property="amount" />
```

The `name` attribute corresponds to the `id` attribute defined in the `<jsp:useBean>` element. The value of the property is converted to a string and placed in the output stream.

NOTE

Alternatively, you can access a JavaBean property inside a scriptlet using its getter method. For example:

```
<%= converters.CurrencyConverterBean.getAmount() %>
```

You set a JavaBean Property using the `<jsp:setProperty>` element.

For example, use the following to set the amount in the `CurrencyConverterBean` to the value `3`:

```
<jsp:setProperty name="converter" property="amount" value="3"/>
```

An alternative form of the `<jsp:setProperty>` element can be used to set a JavaBean property to the value of a request parameter. The following sets the property `amount` to the value passed as a parameter also called `amount`:

```
<jsp:setProperty name="date" property="amount" param="amount" />
```

Using this form you can omit the `param` attribute when the request parameter name is the same as the property name. The following also sets the property `amount` to the value passed as the parameter called `amount`:

```
<jsp:setProperty name="date" property="amount" />
```

Initializing JavaBeans

It is usual to set a bean's properties inside the body of the `<jsp:useBean>` element as follows:

```
<jsp:useBean id="date" class="Date" scope="page" >
    <jsp:setProperty name="date" property="timeZone" value="GMT" />
</jsp:useBean>
```

This idiom reinforces the fact that the bean must be initialized before it is used.

The Currency Converter JSP Using a JavaBean

Listing 5.7 shows the new Currency Converter JSP using the JavaBean standard actions. As you can see the JSP is now extremely straightforward. All the Java code has been transferred to the JavaBean. Although this is obviously a very simple example it still demonstrates the principle of role separation very well. The Java programmer writes the JavaBean that encapsulates all the business logic. The HTML designer then uses the JSP actions to display the JavaBean properties at the appropriate place on the JSP page.

LISTING 5.7 `date.jsp` Utilizing a JavaBean

```
<jsp:useBean id="converter"
➥class="converters.CurrencyConverterBean" scope="page" >
    <jsp:setProperty name="converter" property="amount" />
</jsp:useBean>

<HTML>
  <HEAD><TITLE>Currency Conversion JSP</TITLE></HEAD>
  <BODY>
    <BIG>
         <jsp:getProperty name="converter" property="amount" /> =
➥ <jsp:getProperty name="converter" property="poundValue" />
    </BIG>
  </BODY>
</HTML>
```

Before completing this introduction to JSPs, you need to briefly cover two more topics: the JSP life cycle and the use of the deployment descriptor file with JSPs.

The JSP Life Cycle

JSPs have the same `init-service-destroy` life cycle as servlets. The only difference is that you should use the `jspInit()` and `jspDestroy()` methods from the JSP interface instead of `init()` and `destroy()` from the servlet interface.

When the first request is delivered to a JSP page, Tomcat calls the `jspInit()` method, if there is one, to initialize resources. Similarly, Tomcat invokes the `jspDestroy()` method to reclaim these resources when Tomcat is shut down or the JSP is replaced.

TIP

Put the `jspInit()` and `jspDestroy()` methods in a JSP declaration.

JSPs and the Deployment Descriptor File

So far in this chapter, we have not discussed the deployment descriptor because none of the examples have needed one. Because JSPs are just servlets, there are no special deployment descriptor elements for them; the only difference lies in how you map a servlet name to a JSP.

In Chapter 4, you saw that a `<servlet>` element in the `web.xml` file associates a name with a servlet class using the `<servlet-class>` element. A JSP uses the `<jsp-file>` element instead of the `<servlet-class>` element as follows:

```
<servlet-name>Currency Converter</servlet-name>
<jsp-file>/currency-converter.jsp</jsp-file >
```

As with ordinary servlets, you can hide the implementation details of your Web application by using `<servlet-mapping>` elements for your JSPs. You can use the `web.xml` file shown in Listing 5.8 to ensure that the `date.jsp` JSP can be accessed via a wide range of URLs.

LISTING 5.8 Deployment Descriptor for a JSP

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

<web-app>

<display-name>Currency Converter</display-name>
    <description>
        This is a simple web application with an HTML form passing
        a single parameter to a JSP.
    </description>
    <servlet>
        <servlet-name>Currency Converter</servlet-name>
        <jsp-file>/currency-converter.jsp </jsp-file>
    </servlet>

    <servlet-mapping>
        <servlet-name>Currency Converter</servlet-name>
        <url-pattern>/convert</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>Currency Converter</servlet-name>
```

LISTING 5.8 Continued

```
        <url-pattern>*.jsp</url-pattern>
    </servlet-mapping>

 </web-app>
```

If this JSP is deployed in an application called `basic-jsp`, it can be accessed using any of the following URLs:

```
http://localhost:8080/basic-jsp/currency-converter.jsp
http://localhost:8080/basic-jsp/convert
http://localhost:8080/basic-jsp/fred.jsp
```

NOTE

Servlet mapping was discussed in Chapter 4. Return to the appropriate section in Chapter 4 if you need to refresh your memory on how servlet mapping works.

Other `web.xml` elements, such as initialization parameters, can be accessed from within JSP scriptlets either using the normal servlet API objects or one of the implicit JSP variables shown in Table 5.1.

Summary

This chapter has provided an introduction to JSPs. You now know that a JSP is a servlet written using HTML and special JSP elements. JSPs have several advantages over servlets: Not only do JSPs simplify the coding of complex HTML documents, but they also allow the roles of Java programmer and HTML designer to be separated.

Servlets and JSPs are not only written differently, but there are also differences in the way they are deployed. Unlike servlets that you compile and deploy as a Java class, a JSP is deployed in its original textual-document form. When the JSP is accessed for the first time, Tomcat detects this fact and subjects the JSP to a translation and compilation cycle. The time taken for this translate-compile cycle causes a significant delay in the display of the JSP for the first access. Precompiling the JSP as it is deployed can eliminate this delay.

The use of JavaBeans to encapsulate the Java code in an accompanying bean class helps simplify JSPs even further by moving most if not all of the Java code into the JavaBean class.

This concludes the introduction to JSPs. Chapter 6 covers troubleshooting your applications. Chapter 7 tackles more advanced servlet and JSP topics.