Chapter 3

JavaServer Pages

JavaServer Pages (JSP) and Servlets are complementary technologies for producing dynamic Web pages via Java. While Servlets are the foundation for server-side Java, they are not always the most efficient solution with respect to development time. Coding, deploying, and debugging a Servlet can be a tedious task. Fixing a simple grammar or markup mistake requires wading through `print()` and `println()` calls, recompiling the Servlet, and reloading a Web Application. Making a grammar or markup mistake is not hard, and the problem is compounded in complex Servlets. JSP complements Servlets by helping solve this problem and simplifying Servlet development.

This chapter discusses the following topics:

- An explanation of JSP and why you would want to use the technology.
- The JSP life cycle—that is, how a container manages a JSP.
- Examination of the similarities and differences between JSP and Servlets.
- An introduction to JSP syntax and semantics.
- Configuring JSP via the Web Application Deployment Descriptor, web.xml.
- An explanation of the JSP implicit objects and why implicit objects are helpful.
- How to use the alternative JSP XML syntax.

As with Chapter 2, do read this chapter straight through. Chapters 2 and 3 describe the basic functionality on which all of the other chapters depend. Chapter 2 introduced Servlets and demonstrated several practical uses of them. This chapter complements Chapter 2 by providing a similar discussion on JavaServer Pages.

JSP 2.0 Specification

The first JavaServer Pages specification was released in 1999. Originally JSP was modeled after other server-side template technologies to provide a simple method of embedding dynamic code with static markup. When a request is made for the content of a JSP, a container interprets the JSP, executes any embedded code, and sends the results in a response. At the time this type of functionality was nothing terribly new, but it was and still is a helpful enhancement to Servlets.

JSP has been revised several times since the original release, each adding functionality, and is currently in version 2.0. The JSP specifications are developed alongside the Servlet specifications and can be found on Sun Microsystems' JavaServer Pages product information page, `http://java.sun.com/products/jsp`.

The functionality defined by the JSP 2.0 specifications can be broken down as follows:

JSP

The JSP specifications define the basic syntax and semantics of a JavaServer Page. A basic JavaServer Page consists of plain text and markup and can optionally take advantage of embedded scripts and other functionality for creating dynamic content.

JavaBeans

JavaBeans are not defined by the JSP specifications, but JSP does provide support for easily using and manipulating them. Often objects used on the server-side of a Web Application are in the form of what is commonly called a JavaBean.

Custom Tags and JSP Fragments

JSP provides a mechanism for linking what would normally be static markup to custom Java code. This mechanism is arguably one of the strong points of JSP and can be used in place of or to complement embedded scripts of Java code.

Expression Language

JSP includes a mechanism for defining dynamic attributes for custom tags. Any scripting language can be used for this purpose; usually Java is implemented, but the JSP specification defines a custom expression language designed specifically for the task. Often the JSP EL is a much simpler and more flexible solution, especially when combined with JSP design patterns that do not use embedded scripts.

Discussing the basics of JSP is the focus of this chapter. JavaBeans, Custom Tags, and the JSP Expression Language are all fully discussed in later chapters after a proper foundation of JSP is established.

JSP Life Cycle

Much like Servlets, understanding JSP requires understanding the simple life cycle that JSP follows. JSP follows a three-phase life cycle: initialization, service, and destruction, as shown in Figure 3-1. This life cycle should seem familiar and is identical to the one described for Servlets in Chapter 2.

While a JSP does follow the Servlet life cycle, the methods have different names. Initialization corresponds to the `jspInit()` method, service corresponds to the `_jspService()` method, and destruction corresponds to the `jspDestroy()` method. The three phases are all used the same as a Servlet and allow a JSP to load resources, provide service to multiple client requests, and destroy loaded resources when the JSP is taken out of service.

JSP is designed specifically to simplify the task of creating text producing `HttpServlet` objects and does so by eliminating all the redundant parts of coding a Servlet. Unlike with Servlets there is no distinction between a normal JSP and one meant for use with HTTP. All JSP are designed to be used with HTTP and to generate dynamic content for the World Wide Web. The single JSP `_jspService()` method is also responsible for generating responses to all seven of the HTTP methods. For most practical purposes a JSP developer does not

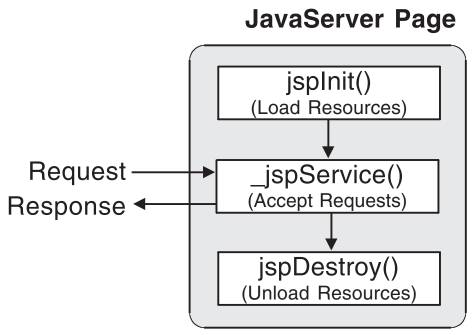


Figure 3-1 JSP Life Cycle

need to know anything about HTTP, nor anything more than basic Java to code a dynamic JSP.

The Difference Between Servlets and JSP

A clear and important distinction to make about JSP is that coding one is nothing like coding a Servlet. From what this chapter has explained, it might appear that JSP is just a simple version of Servlets. In many respects JSP is in fact a simple method of creating a text-producing Servlet; however, do not be fooled into thinking this mindset is always true. As the chapter progresses, it will be clear that JSP and Servlets are two very distinct technologies. Later, after custom tags are introduced, the degree of separation between the two will seem even larger. The use of JSP for easily making a Servlet really only applies in the simplest of cases.

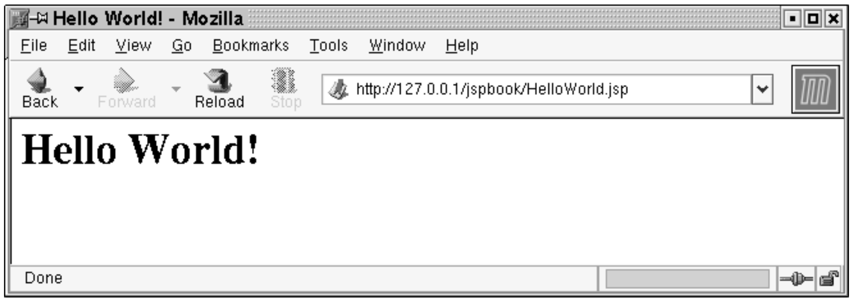
To show how vastly different the code for a JSP can be from a Servlet, Listing 3-1 displays the code for the JSP equivalent of the HelloWorld Servlet (Listing 2-1 that appeared in Chapter 2).

Listing 3-1 HelloWorld.jsp

```
<html>
<head>
<title>Hello World!</title>
</head>
<body>
<h1>Hello World!</h1>
</body>
</html>
```

They are quite different! You'll recall the code for `HelloWorld.java` and notice the two look nothing alike. The code for the JSP is actually identical to the text generated by the HelloWorld Servlet, not the source code. Authoring an HTML-generating JSP is as easy as just authoring the HTML. Compared to using `print()` and `println()` methods in Servlets, the JSP approach is obviously easier. This is why simple JSP are usually considered a quick method of creating text-producing Servlets.

Deploying a JSP is also simpler; a Web Application automatically deploys any JSP to a URL extension that matches the name of the JSP. Test out `HelloWorld.jsp` by saving it in the base directory of the jspbook Web Application then browsing to `http://127.0.0.1/jspbook/HelloWorld.jsp`. Figure 3-2 shows the output of `HelloWorld.jsp` as rendered by a Web browser, identical to the HTML generated by the HelloWorld Servlet, Figure 2-6.


Hello World! - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back
Forward
Reload
Stop
http://127.0.0.1/jspbook/HelloWorld.jsp
Done

Hello World!

Figure 3-2 Browser Rendering of HelloWorld.jsp

From looking at the results of the example, it certainly does appear `HelloWorld.jsp` is a simple form of `HelloWorld.java`. What is not shown, but is very important to understand, is that `HelloWorld.jsp` is also actually compiled into equivalent Servlet code. This is done in what is called the *translation phase* of JSP deployment and is done automatically by a container. JSP translation both is and is not something of critical importance for a JSP developer to be aware of. JSP translation to Servlet source code is important because it explains exactly how a JSP becomes Java code. While it varies slightly from container to container, all containers must implement the same JSP life cycle events. Understanding these life cycle methods helps a JSP developer keep code efficient and thread-safe. However, JSP translation is not of critical importance because it is always done automatically by a container. Understanding what a container will do during the translation phase is good enough to code JSP. Keeping track of the generated Servlet source code is not a task a JSP developer ever has to do.

JSP translated to Servlet code can be found by looking in the right place for a particular container. Tomcat stores this code in the `/work` directory of the Tomcat installation. Generated code is never pretty, nor does it always have the same name. Listing 3-2 was taken from `HelloWorld$jsp.java` in the `/work/localhost/jspbook` directory. It is the Servlet code generated for `HelloWorld.jsp` that Tomcat automatically compiled and deployed.

Listing 3-2 Tomcat-Generated Servlet Code for HelloWorld.jsp

```
package org.apache.jsp;

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


public class HelloWorld$jsp extends HttpJspBase {


    static {
    }
    public HelloWorld$jsp( ) {
    }

    private static boolean _jspx_inited = false;

    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="/HelloWorld.jsp";from=(0,0);to=(8,0)]
                out.write("<html>\r\n<head>\r\n<title>Hello
World!</title>\r\n</head>\r\n<body>\r\n<h1>Hello
World!</h1>\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);
        }
    }
}
```

Do not bother trying to read through and understand the generated code. The important point to understand is that a container handles a JSP as a Servlet but does so behind the scenes. This ties directly back to the greater point that JSP are really just Servlets. The difference between the two technologies is not in the life cycles or how a container manages them at runtime. The difference between Servlets and JSP is the syntax they offer for creating the same functionality. With JSP it is almost always simpler to create text-producing Servlets, but normal Servlets are still best suited for sending raw bytes to a client or when complete control is needed over Java source code.

JSP Syntax and Semantics

JSP is not governed by the syntax and semantics defined by the Java 2 specifications. Translated JSP source code is just Java, but when you author a JSP, you abide instead by the rules laid out in the JSP specification. With each release of the JSP specification, these rules grow, and they cannot be easily summed by a few sentences. The majority of this chapter focuses on explaining the current syntax and semantics of JSP. Much of the functionality found in JSP is taken directly from the underlying Servlet API which was already covered in Chapter 2. Expect to see lots of code examples demonstrating syntax, while repetitious semantics are only skimmed with a reference to the full explanation previously given in Chapter 2.

Elements and Template Data

Everything in a JSP is broken down into two generic categories called *elements* and *template data*. Elements are dynamic portions of a JSP. Template data are the static bits of text between. Template data are easily categorized as they are all the text arbitrarily placed on a JSP and meant to be directly sent to a client. Elements are easily categorized as custom actions, tags, and the content allowed to be in between as defined by the JSP specifications.

The concept of elements and template data is important to understand as it dictates when, where, and what text will do when placed in a JSP. This chapter has yet to introduce any elements, but it has shown a use of template text. The `HelloWorld.jsp` example was entirely template text. The corresponding Servlet generated from `HelloWorld.jsp` treated this text as static content and sent it as the content of a response. While `HelloWorld.jsp` only had one big chunk of template text, more complex JSP follow the same rule. Any chunk of template text is taken and sent directly to a client as it appears on the JSP. Elements on the other hand are not sent directly to a client. An element is interpreted by a JSP container and defines special actions that should be taken when generating a response.

Template text does not change, and this little section defines it in total. Elements are what make JSP dynamic, and elements are further explained throughout the rest of the chapter. Elements can be broken down into three different categories: scripting elements, directives, and actions. The following self-named sections explain these elements.

Two Types of Syntax

JSP containers are required to support two different formats of JSP syntax: normal and XML-compatible. The normal JSP syntax is a syntax designed to be easy to author. The XML-compatible JSP syntax takes the normal JSP syntax and modifies it to be XML-compliant[1]. Both syntaxes provide the same functionality, but the XML-compatible syntax is intended to be more easily used by development tools. In the examples of this book the normal JSP syntax is preferred as it is easily read, understood, and will be familiar if you have used older versions of JSP.

Just because the XML syntax will not be appearing much in this book's examples does not mean it is the lesser of the two syntaxes. The JSP XML syntax introduced in the JSP 1.2 specification, from a developer's perspective, is certainly a hassle to use compared to the regular syntax. This is largely due to the complexity and strict enforcement of syntax the JSP 1.2 XML syntax had. JSP 2.0 remedies the problem by providing a much more flexible XML syntax. Later on in the chapter this new, more flexible XML syntax is further explained.

Scripting Elements

The simplest method of making a JSP dynamic is by directly embedding bits of Java code between blocks of template text by use of scripting elements. In theory JSP does not limit scripting elements to only those containing Java code, but the specification only talks about Java as the scripting language, and every container by default has to support Java. Examples in this book use Java as a scripting language. There are three different types of scripting elements available for use in JSP: *scriptlets*, *expressions*, and *declarations*.

Scriptlets

Scriptlets provide a method for directly inserting bits of Java code between chunks of template text. A scriptlet is defined with a start ,`<%`, an end, `%>`, with code between. Using Java, the script is identical to normal Java code but without needing a class declaration or any methods. Scriptlets are great for providing low-level functionality such as iteration, loops, and conditional statements, but they also provide a method for embedding complex chunks of code within a JSP.

For many reasons complex scriptlets should be avoided. This is mainly due to the fact that the more scriptlets are used the harder it is to understand and maintain a JSP. In this chapter most of the scriptlet examples are purposely kept simple. This aids in directly demonstrating the core functionality of JSP, but it is also done so that examples do not appear to encourage heavy use of scriptlets. As

1. eXtensible Markup Language, http://www.w3.org/XML

an introduction, Listing 3-3 provides a simple JSP that loops to produce multiple lines of text. Looping is accomplished the same as in Java but by placing the equivalent Java code inside scriptlet elements.

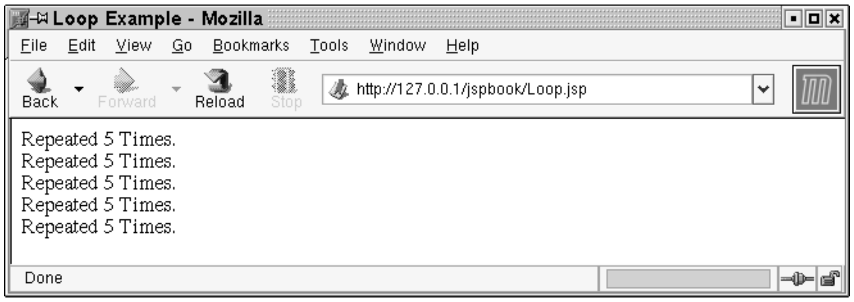
Listing 3-3 Loop.jsp

```
<html>
<head>
<title>Loop Example</title>
</head>
<body>
<% for (int i=0; i<5;i++) { %>
 Repeated 5 Times.<br>
<% } %>
</body>
</html>
```

Save `Loop.jsp` in the base directory of the jspbook Web Application and browse to `http://127.0.0.1/jspbook/Loop.jsp`. The page shows up with the statement, “Repeated 5 Times.”, repeated five times. Figure 3-3 shows what a browser rendering of the output looks like.

It is important to note that the contents of the scriptlet did not get sent to a client. Only the results of the scriptlet did. This is important because it shows that scriptlets are interpreted by a container and that code inside a scriptlet is not by default shared with visitors of the JSP.

A JSP may contain as many scriptlets as are needed, but caution should be taken not to overuse scriptlets. Scriptlets make a JSP very hard to maintain and are not easily documented; for example, tools like `javadoc` do not work with JSP.


Loop Example - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back
Forward
Reload
Stop
http://127.0.0.1/jspbook/Loop.jsp
Done

```
Repeated 5 Times.
Repeated 5 Times.
Repeated 5 Times.
Repeated 5 Times.
Repeated 5 Times.
```

Figure 3-3 Browser Rendering of Loop.jsp

Expressions

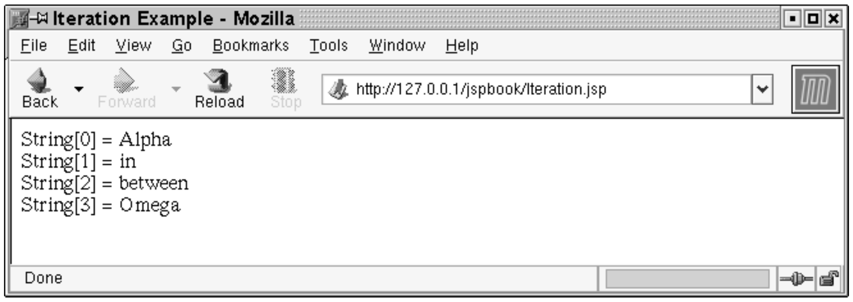
Expressions provide an easy method of sending out dynamic strings to a client. An expression must have a start, `<%=`, end, `%>`, and an expression between. An expression element differs in syntax from a scriptlet by an equal sign that must appear immediately after the start. Expressions always send a string of text to a client, but the object produced as a result of an expression does not have to always end up as an instance of a `String` object. Any object left as the result of an expression automatically has its `toString()` method called to determine the value of the expression. If the result of the expression is a primitive, the primitive's value represented as a string is used.

Combined with scriptlets, expressions are useful for many different purposes. A good example is using a scriptlet that is combined with an expression to provide a method of iterating over a collection of values. The scriptlet provides a loop, while expressions and static content are used to send information in a response. `Iteration.jsp` (Listing 3-4) provides an example of this along with a demonstration of passing a non-`String` object as the result of an expression.

Listing 3-4 Iteration.jsp

```
<html>
<head>
<title>Iteration Example</title>
</head>
<body>
<%
 String[] strings = new String[4];
 strings[0] = "Alpha";
 strings[1] = "in";
 strings[2] = "between";
 strings[3] = "Omega";
 for (int i=0; i<strings.length;i++) { %>
 String[<%= i %>] = <%= strings[i] %><br>
<% } %>
</body>
</html>
```

Save `Iteration.jsp` in the base directory of the jspbook Web Application and browse to `http://127.0.0.1/jspbook/Iteration.jsp`. The page displays a list of an iteration through all of the values of the array. Figure 3-4 shows a browser rendering of the output sent by `Iteration.jsp`.


Iteration Example - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://127.0.0.1/jspbook/Iteration.jsp
Done

```
String[0] = Alpha
String[1] = in
String[2] = between
String[3] = Omega
```

Figure 3-4 Iteration.jsp

Declarations

Declarations are the third and final scripting element available for use in JSP. A declaration is used like a scriptlet to embed code in a JSP, but code embedded by a declaration appears outside of the `_jspService()` method. For this reason code embedded in a declaration can be used to declare new methods and global class variables, but caution should be taken because code in a declaration is not thread-safe, unless made so by the writer of that code.

Listing 3-5 is a JSP designed to keep a page counter of how many times it has been visited. The JSP accomplishes this by declaring a class-wide variable in a declaration, using a scriptlet to increment the variable on page visits, and an expression to show the variable's value.

Listing 3-5 PageCounter.jsp

```
<%! int pageCount = 0;
 void addCount() {
   pageCount++;
 }
%>
<html>
<head>
<title>PageCounter.jsp</title>
</head>
<body>
<% addCount(); %>
This page has been visited <%= pageCount %> times.
</body>
</html>
```

Save `PageCounter.jsp` in the base directory of the jspbook Web Application and browse to `http://127.0.0.1/jspbook/PageCounter.jsp`. Refresh the browser a few times to watch the JSP count how many times it has been visited. Figure 3-5 shows a browser rendering of the output after visiting the JSP 6 times.

Using scriptlets, expressions, and declarations most anything can be created using JSP. An analogy to Servlets would be: *scriplets* are code placed inside a service method, *expressions* are `print()` method calls, and *declarations* are code placed globally for a class to use. After the brief explanation above it should be fairly straightforward to start coding using sciptlets, expressions, and declarations, but it is still helpful to understand what the three different scripting elements translate to in a Java code.

Listing 3-6 is the code Tomcat generated from `PageCounter.jsp`. It contains the translation of all three of the scripting elements. The lines of importance are highlighted with an asterisk.

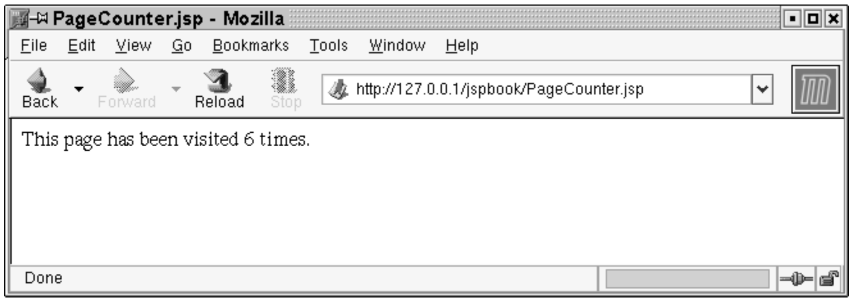
Listing 3-6 PageCounter$jsp.java

```
package org.apache.jsp;

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


public class PageCounter$jsp extends HttpJspBase {

    // begin [file="/PageCounter.jsp";from=(0,3);to=(4,0)]
```


PageCounter.jsp - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://127.0.0.1/jspbook/PageCounter.jsp
Done

This page has been visited 6 times.

Figure 3-5 Browser Rendering of PageCounter.jsp

```
        int pageCount = 0;
        void addCount() {
          pageCount++;
        }
    // end

    static {
    }
    public PageCounter$jsp( ) {
    }

    private static boolean _jspx_inited = false;

    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="/PageCounter.jsp";from=(4,2);to=(10,0)]

out.write("\r\n<html>\r\n<head>\r\n<title>PageCounter.jsp</title>\r\
n</head>\r\n<body>\r\n");

            // end
            // begin
[file="/PageCounter.jsp";from=(10,2);to=(10,15)]
                 addCount();
            // end
            // HTML // begin
[file="/PageCounter.jsp";from=(10,17);to=(11,27)]
                out.write("\r\nThis page has been visited ");

            // end
            // begin
[file="/PageCounter.jsp";from=(11,30);to=(11,41)]
                out.print( pageCount );
            // end
            // HTML // begin
[file="/PageCounter.jsp";from=(11,43);to=(14,0)]
                out.write(" times.\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);
        }
    }
}
```

Thankfully this is the last bit of generated code that appears in this book. Reading through translated JSP is rarely required nor is it usually helpful[2], but it is certainly needed to understand what a container does when a JSP is translated to a Servlet. Taking the highlighted lines from `PageCounter$jsp.java`, each can be linked back to scripting elements from `PageCounter.jsp`.

In `PageCounter.jsp` a declaration is used to define a class-wide variable for counting the number of page visits and a method for incrementing it.

```
<%! int pageCount = 0;
 void addCount() {
   pageCount++;
 }
%>
```

Translated into `PageCounter$jsp.java`, the code of the declaration appears outside of the `_jspService()` method and is class-wide. This allows for the `addCount()` method to be coded as a real method of the Servlet and the `pageCount` variable to be accessible by all calls to the `_jspService()` method. The methods are **not thread-safe**, but for this particular example, it does not matter if the `pageCount` variable changes halfway through generating a response.

```
public class PageCounter$jsp extends HttpJspBase {

    // begin [file="/PageCounter.jsp";from=(0,3);to=(4,0)]
         int pageCount = 0;
         void addCount() {
           pageCount++;
         }
    // end
```

In `PageCounter.jsp` a scriptlet and expression are used to increase and send the value of the `pageCount` variable to client.

```
<% addCount(); %>
This page has been visited <%= pageCount %> times.
```

2. Except in the case of debugging, where it is often useful to see the JSP and the generated Servlet side-by-side.

Translated in PageCounter$jsp.java, the scriptlet is used verbatim, but the expression is turned into a call to the print() method of a PrintWriter object obtained from the corresponding HttpServletResponse. Both scripting elements are located in the _jspService() method, are thread-safe, and local to a specific client request.

```
    public void _jspService(HttpServletRequest request,
HttpServletResponse  response)
        throws java.io.IOException, ServletException {
...
            // begin
[file="/PageCounter.jsp";from=(10,2);to=(10,15)]
                 addCount();
            // end
            // HTML // begin
[file="/PageCounter.jsp";from=(10,17);to=(11,27)]
                out.write("\r\nThis page has been visited ");

            // end
            // begin
[file="/PageCounter.jsp";from=(11,30);to=(11,41)]
                out.print( pageCount );
            // end
...
    }
```

The point of the preceding code is to illustrate exactly where scriptlets, expressions, and declarations appear in Java source code generated from a JSP. It is important to understand that both scriptlets and expressions appear inside the _jspService() method and are by default thread-safe to a particular request. Declarations are not thread-safe. Code inside a declaration appears outside the _jspService() method and is accessible by a requests being processed by the JSP. Declarations, unlike scriptlets, can also declare functions for use by scriptlets.

Good Coding Practice with Scripting Elements

Before going out and recklessly coding JSP with scripting elements, there are a few important points to be made. Most prominent is the issue of thread safety[3].

3. "Thread safety" is not a Servlet-specific or JSP-specific issue. Whenever Java code is using multiple threads, state concurrency issues arise. Thread safety is a common term when describing these issues as it is important to ensure code is *thread-safe*—that is, able to work properly if multiple threads are running it. Chapter 9 provides a complete discussion on thread-safety and state management as the topics apply to Servlets and JSP.

Expressions can be considered harmless. Unless purposely used to cause a conflict, an expression is always going to be thread-safe. Declarations and scriptlets are more problematic. Using a scriptlet is analogous to declaring and using variables locally in the appropriate service method of a Servlet. Variables declared by a scriptlet are initialized, used, and destroyed once per a call to the `_jspService()` method. By default this makes scriptlets thread-safe, but they do not ensure the objects they access are thread-safe. If a scriptlet accesses an object in a scope outside of the `_jspService()` method, then a synchronized block should be used to ensure thread safety. Declarations are not thread-safe! A declaration appears outside the `_jspService()` method. Far too often declarations are completely misunderstood as an enhanced form of a scriptlet. They are not!

A common debate is whether scriptlets and declarations are needed at all with JSP. The power of JSP comes from easily creating text-producing Servlets. A JSP is maintainable if it is primarily markup, which is easily edited by page authors. Many developers take the stance that scripting elements destroy this feature of JSP and should be completely replaced by custom actions and the JSP expression language (new as of JSP 2.0, and covered in a later chapter). This viewpoint is valid, is endorsed by the authors ,and is further covered in later chapters, but custom actions are relatively heavyweight components compared to a simple embedded script. Scriptlets, declarations, and particularly expressions certainly have a place with JSP, but they should not be overused. If a page is littered with countless scripting elements, they are likely to do more harm than good. Always be conscious of how scripting elements are being used and if the code might be better encapsulated in other objects that can be used by a few scripting elements.

Directives

Directives are messages to a JSP container. They do not send output to a client, but are used to define page attributes, which custom tag libraries use and which other pages include. All directives use the following syntax.

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

Directives may optionally have extra whitespace after the `<%@` and before the `%>`. There are three different JSP directives for use on a page[4]: `page`, `taglib`, and `include`.

4. There are other directives that can only be used in tag files: `tag`, `attribute`, and `variable`.

<%@ page %>

The `page` directive provides page-specific information to a JSP container. This information includes settings such as the type of content the JSP is to produce, the default scripting language of the page, and code libraries to import for use. Multiple page directives may be used on a single JSP or pages included via JSP as long as no specific attribute, except `import`, occurs more than once. Attributes for the `page` directive are as follows.

language The `language` attribute defines the scripting language to be used by scriptlets, expressions, and declarations occurring in the JSP. The only defined and required scripting language value for this attribute is `java`. Different containers may provide additional language support; however, it is uncommon to see a JSP use a language other than Java for scripting elements.

In this book it is always assumed that the language appearing in scripting elements is Java (as that is currently the only defined language). Translation of a scripting element using Java code fragments into real Java code is easily done by any developer with previous Java experience. Understanding how and why a scriptlet example works is assumed to be intuitive because it is identical to understanding the Java equivalent.

extends The `extends` attribute value is a fully qualified Java programming language class name that names the superclass of the class to which this JSP is transformed. The `extends` attribute is analogous to the extends keyword used when authoring a Java class. This attribute should be used sparingly as it prevents a container from using a pre-built optimized class.

import The `import` attribute describes the types that are available for use in scripting elements. The value is the same as in an import declaration in the Java programming language, with multiple packages listed with either a fully qualified Java programming language-type names or a package name followed by `.*`, denoting all the public types declared in that package.

The default import list is `java.lang.*`, `javax.servlet.*`, `javax.servlet.jsp.*`, and `javax.servlet.http.*`. These packages can be assumed to be available by default with every JSP. The `import` attribute is currently only defined for use when the value of the language directive is `java`.

session The `session` attribute indicates that the page requires participation in an HTTP session. If `true`, then the implicit scripting variable `session` references the current/new session for the page. If `false`, then the page does not participate in a session and the `session` implicit scripting variable is unavailable, and any reference to it within the body of the JSP page is illegal and results in a fatal translation error. The default value of the `session` attribute is `true`.

buffer The `buffer` attribute specifies the buffering model for the initial `out` implicit scripting variable to handle content output from the page. If the attribute's value is `none`, then there is no buffering and output is written directly through to the appropriate `ServletResponse PrintWriter`. Valid values for the attribute are sizes specified in kilobytes, with the `kb` suffix being mandatory. If a buffer size is specified, then output is buffered with a buffer size of at least the specified, value. Depending upon the value of the `autoFlush` attribute, the contents of this buffer are either automatically flushed or an exception is thrown when overflow would occur. The default value of the `buffer` attribute is 8kb.

autoFlush The `autoFlush` attribute specifies whether buffered output should be flushed automatically when the buffer is filled, or whether an exception should be raised to indicate buffer overflow. A value of `true` indicates automatic buffer flushing and a value of `false` throws an exception. The default value of the `autoFlush` attribute is `true`. It is illegal to set the `autoFlush` attribute to `false` when the value of the buffer attribute is `none`.

isThreadSafe The `isThreadSafe` attribute indicates the level of thread safety implemented in the page. If the value is `false`, then the JSP container shall dispatch multiple outstanding client requests, one at a time, in the order they were received, to the page implementation for processing by having the generated Servlet implement `SingleThreadModel`. If the attribute's value is `true`, then the JSP container may choose to dispatch multiple client requests to the page simultaneously. The default value of the `isThreadSafe` attribute is `true`.

isErrorPage The `isErrorPage` attribute indicates if the current JSP page is intended to be an error page for other JSP. If `true`, then the implicit scripting variable `exception` is defined, and its value is a reference to the offending `Throwable` object. If the `isErrorPage` attribute value is `false`, then the `exception` implicit variable is unavailable, and any reference to it within the body of the JSP page is illegal and will result in a fatal translation error. The default value of the `isErrorPage` attribute is `false`.

errorPage The `errorPage` attribute defines a relative URL to a resource in the Web Application to which any Java programming language `Throwable` object thrown but not caught by the page implementation is forwarded for error processing. The `Throwable` object is transferred to the error page by binding the object reference to request scope with the name `javax.servlet.jsp.jspException`.

If the value of the `autoFlush` attribute is `true`, and if the contents of the initial `JspWriter` have been flushed to the `ServletResponse` output stream, then any subsequent attempt to dispatch an uncaught exception from the offending page to an `errorPage` may fail.

contentType The `contentType` attribute defines the character encoding for the JSP page, and for the response of the JSP page and the MIME type for the response of the JSP page. The default value of the `contentType` attribute is `text/html` with `ISO-8859-1` character encoding for regular JSP syntax and `UTF-8` encoding for JSP in XML syntax.

pageEncoding The `pageEncoding` attribute defines the character encoding for the JSP. The default value of the `pageEncoding` attribute is `ISO-8859-1` for regular JSP and `UTF-8` for JSP in XML syntax.

isScriptingEnabled The `isScriptingEnabled` attribute determines if scripting elements are allowed for use. The default value (true) enables scriptlets, expressions, and declarations. If the attribute's value is set to false, a translation-time error will be raised if the JSP uses any scriptlets, expressions (non-EL), or declarations. This attribute is helpful for creating 'scriptless' JSP and can also be set using the `web.xml scripting-enabled` element.

isELEnabled The `isELEnabled` attribute determines if JSP EL expressions used in the JSP are to be evaluated. The default value of the attribute is `true`, meaning that expressions, `${...}`, are evaluated as dictated by the JSP specification. If the attribute is set to `false`, then expressions are not evaluated but rather treated as static text.

The page directive is by default set to accommodate the most common use of JSP: to make dynamic HTML pages. When creating a simple JSP, it is rarely needed to specify any of the page directive attributes except in cases where extra code libraries are needed for scripting elements or when producing XML content, which is happening more and more.

<%@ include %> and <jsp:include />

The include directive is used to include text and/or code at *translation time* of a JSP. The include directive always follows the same syntax, `<%@ include file="relativeURL" %>`, where the value of `relativeURL` is replaced with the file to be inserted. Files included must be part of a Web Application. Since include directives take place at translation time, they are the equivalent of directly including the source code in the JSP before compilation and do not result in performance loss at runtime.

Server-side includes are a commonly used feature of JSP. Includes allow the same repetitious bit of code to be broken out of multiple pages and have one instance of it included with them all. A good example to use is including a common header and footer with multiple pages of content. Usually this technique is used to keep a site's navigation and copyright information correct and maintainable for all individual pages on the site. As an example take the following two files, `header.jsp` and `footer.jsp`.

The `header.jsp` file (Listing 3-7) includes information that is to appear at the top of a page. It includes site navigation and other miscellaneous information. This header also tracks how many people have visited the site since the last time the Web Application was reloaded by reusing code from `PageCounter.jsp`.

Listing 3-7 header.jsp

```
<%! int pageCount = 0;
 void addCount() {
   pageCount++;
 }
%>
<% addCount(); %>
<html>
<head>
<title>Header/Footer Example</title>
</head>
<body>
<center>
<h2>Servlets and JSP the J2EE Web Tier</h2>
<a href="http://www.jspbook.com">Book Support Site</a> -
<a href="ShowSource">Sites Source code</a><br>
This site has been visited <%= pageCount %> times.
</center>
<br><br>
```

The `footer.jsp` file (Listing 3-8) includes information that is to appear at the very bottom of a page. It includes copyright information, disclaimers, and any other miscellaneous information.

Listing 3-8 footer.jsp

```
<br><br>
<center>
```

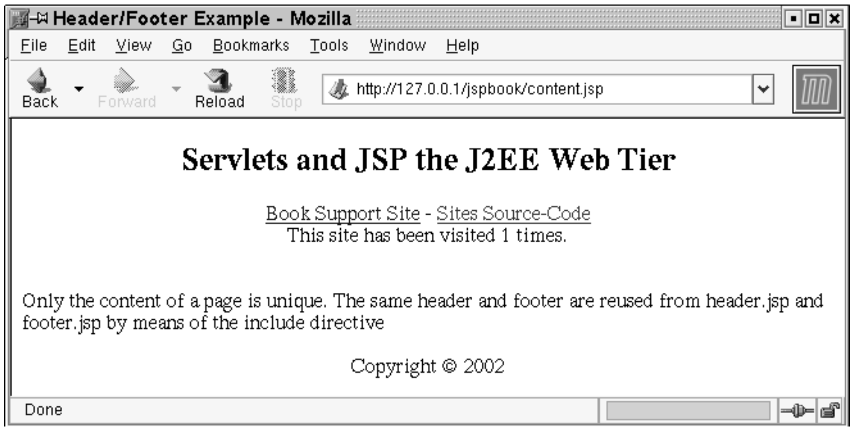
```
Copyright &copy; 2003
</center>
</body>
</html>
```

By themselves `header.jsp` and `footer.jsp` do not do much, but when combined with some content, a full page can be generated. For this example the content does not matter. Arbitrarily make up a JSP, but be sure to include `header.jsp` at the top of the page and `footer.jsp` at the bottom. Listing 3-9 provides an example of such a page.

Listing 3-9 content.jsp

```
<%@ include file="header.jsp" %>
Only the content of a page is unique. The same header and footer
are reused from header.jsp and footer.jsp by means of the include
directive
<%@ include file="footer.jsp" %>
```

Save all three files, `header.jsp`, `footer.jsp`, and `content.jsp`, in the base directory of the jspbook Web Application and browse to `http://127.0.0.1/jspbook/content.jsp`. All three files are mashed together at translation time to produce a Servlet that includes the contents of `header.jsp`, `content.jsp`, and `footer.jsp` in the appropriate order. Figure 3-6 shows a browser rendering of the output.


Header/Footer Example - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://127.0.0.1/jspbook/content.jsp

Servlets and JSP the J2EE Web Tier

Book Support Site - Sites Source-Code
This site has been visited 1 times.

Only the content of a page is unique. The same header and footer are reused from header.jsp and footer.jsp by means of the include directive

Copyright © 2002

Figure 3-6 Browser Rendering of content.jsp

So what is the advantage of this approach over combining `header.jsp`, `content.jsp`, and `footer.jsp` all in one file in the first place? As the files are set up now, many more content pages can be created that reuse the header and footer. The header and footer can also be changed at any given time, and the changes are easily reflected across all of the JSP.

Translation-Time Includes

Translation time occurs when a JSP is being translated into a Servlet. The resulting Servlet does not know or care about what files were used to generate it. As a result the Servlet is unable to tell when a change has occurred in included files. The JSP specifications do not specify a mechanism for solving this problem, but JSP container vendors are free to implement solutions to the problem.

In cases where an entire site relies on translation-time includes for efficiency, a simple solution does exist for having changes in translation-time includes reflected on the entire site. A container relies on having the translated code of a JSP to compile and deploy a corresponding Servlet. When lacking the code, a container must re-translate a JSP and compile and deploy the corresponding Servlet. By forcing a container to re-translate all JSP, it can be ensured that translation-time includes are properly reflected by JSP that use them.

With Tomcat, JSP translated to Servlet source code can be found in the `TOMCAT_HOME/work` directory. Simply delete the contents of this directory to have Tomcat re-translate all JSP.

For non-translation-time includes JSP also defines the include action. Like the include directive, this action is used to include resources with the output of a JSP, but the include action takes place at runtime. While not as efficient, the include action automatically ensures that the most recent output of the included file is used. See the `<jsp:include />` section of this chapter for more information about the include action.

<%@ taglib %>

Custom actions were previously mentioned in this chapter, but are not fully covered until Chapter 7. Custom actions, also called custom tag libraries, allow a JSP developer to link bits of markup to customized Java code. The `taglib` directive informs a container what bits of markup on the page should be considered custom code and what code the markup links to. The `taglib` directive always follows the same syntax, `<%@ taglib uri="uri" prefix="prefixOfTag" %>`, where the `uri` attribute value resolves to a location the container understands and the prefix attribute informs a container what bits of markup are custom actions.

Further explanation and examples of using the `taglib` directive can be found in Chapter 7.

JSP Configuration

Directives are in the simplest sense configuration information for a JSP. The only problem with using directives for configuration is that they must be specified on a per-JSP basis. If you have 20 pages, then you will have to edit at least 20 directives. To simplify the task of doing mass JSP configuration, the `jsp-config` element is available for use in `web.xml`. There are two sub-elements of `jsp-config`: `taglib` and `jsp-property-group`. The taglib element can be used to configure a custom JSP tag library for use with a JSP. The `jsp-property-group` element allows for configuration that is similar to the directives but can be applied to a group of JSP.

For completeness the taglib element will be covered here and referenced in the later pertinent chapter about custom JSP tag libraries. Use of the `taglib` element is straightforward; first specify a taglib-uri child element, which defines the prefix custom tags are to use; next specify a `taglib-location` element, which defines the location of the custom tag library. For example:

```
...
<jsp-config>
  <taglib>
    <taglib-uri>foo</taglib-uri>
    <taglib-location>WEB-INF/foo.tld</taglib-location>
  </taglib>
</jsp-config>
...
```

For more on tag libraries see Chapter 7.

The `jsp-property-group` element is currently of much more relevance because it is an alternative for much of the functionality offered by the JSP directives. Basic use of the `jsp-property-group` element is always the same; first use a child url-pattern element to define the JSP to apply the properties to:

```
<jsp-config>
  <jsp-property-group>
```

```
    <url-pattern>*.jsp</url-pattern>
    ...
  </jsp-property-group>
</jsp-config>
```

In the preceding code the configuration will be applied to all files ending in `.jsp` which would likely be all of the JSP in the Web Application. In general the `url-pattern` element can have any of the values that are valid for use when deploying Servlets or JSP via `web.xml`. By itself the `url-pattern` element does nothing but match a set of properties to a specific set of JSP. The properties themselves must next be specified using more child elements of `jsp-property-group`. The `jsp-property-group` element has the following children elements, which are all fairly self-descriptive.

el-enabled The `el-enabled` element configures if the JSP EL is available for use on the specified JSP. A value of `true` enables EL use. A value of `false` disables it. The functionality is analogous to the page directive's `isELEnabled` attribute.

scripting-enabled The `scripting-enabled` element is analogous to the page directive's `isScriptingEnabled` attribute. A value of `false` will cause a JSP to raise a translation error if any scriptlets, expressions (non-EL), or directives are used. A value of `true` will enable scripting elements for use.

page-encoding The `page-encoding` element is analogous to the page directive's `pageEncoding` attribute. An error will be raised if an encoding is set using the `page-encoding` attribute and a different encoding is specified using the `pageEncoding` attribute of a JSP's page directive.

include-prelude The `include-prelude` element can be used to include the contents of another resource in the Web Application before the contents generated by a JSP. Effectively the `include-prelude` element provides a method of automatically including a header for all JSP that the `jsp-property-group` is configured for.

include-coda The `include-coda` element can be used to include the contents of another resource in the Web Application after the contents generated by a JSP. Effectively the `include-coda` element provides a method that automatically includes a footer for all JSP that the `jsp-property-group` is configured for.

is-xml The `is-xml` element can be used to denote if the JSP are XML documents. JSP authored in XML syntax need not always be explicitly declared as XML; however, declaring so is helpful for both validating the document and taking advantage of JSP-XML features such as `UTF-8` encoding.

Overall, the `jsp-property-group` element should be very intuitive to use. The only new functionality that has been introduced is due to the `include-prelude`, `include-coda`, and `is-xml` elements. Both `include-prelude` and `include-coda` are also very straightforward to use. In use, the two elements replace the need for header and footer includes using the JSP include directive or custom action. The `is-xml` element is new, but do not worry if it is unclear as to what the element is doing. Later on in this chapter JSP in XML syntax is addressed properly.

Application-Wide Headers and Footers

It is very common to build a Web Application that includes the same header and footer on every page. In these cases there are many methods that you can use to go about achieving the functionality, but arguably the easiest is to use the `include-prelude` and `include-coda` elements. Compared to manually including the header and footer on each page, as in Listing 3-9, little is gained, but it is slightly helpful to use the `include-prelude` and `include-coda` elements for two reasons. First, inclusion of the header and footer pages are consolidated to one single point, `web.xml`. The names or locations of the header and footer resources can easily be changed for any given reason. When using the include directive or action, this will not be the case. A change will be a slight bit more of a hassle, but it can still be done. The second benefit to using the `include-prelude` and `include-coda` elements is that pages of a Web Application only have to focus on content, nothing else. Granted, remembering to include a header and footer is not a difficult task, but it is a task all the same.

Using the `include-prelude` and `include-coda` elements for application-wide headers and footers is easy, and always done in a similar fashion as illustrated in Listing 3-10.

Listing 3-10 Application-Wide Header and Footer Files

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <include-prelude>/header.jsp</include-prelude>
    <include-coda>/footer.jsp</include-coda>
  </jsp-property-group>
</jsp-config>
```

As shown, the `jsp-property-group` is configured to apply to all JSP files; however, the configuration could be extended to include other resources if needed. The important point to see is that the configuration is being applied to everything of importance. Next, the `include-prelude` element and `include-coda` element are used to include a header and footer, respectively. The locations given are `/header.jsp` and `/footer.jsp`, but any other resource can be used.

Standard JSP Actions

Actions provide a convenient method of linking dynamic code to simple markup that appears in a JSP. The functionality is identical to the scripting elements but has the advantage of completely abstracting any code that would normally have to be intermixed with a JSP. Actions that are designed to be simple to use and work well help keep a JSP efficient and maintainable.

There are two types of actions available for use with JSP: standard and custom. All actions follow the same syntax, `<prefix:element {attribute="value"}* />`, where the complete action is an XML-compatible tag and includes a given `prefix`, `element`, and a set of `attributes` and `values` that customize the action. Standard actions are completely specified by the JSP specification and are, by default, available for use with any JSP container. Custom actions are a mechanism defined by the JSP specification to enhance JSP by allowing JSP developers to create their own actions. The functionality of custom actions is not defined by the JSP specifications, and custom actions must be installed with a Web Application before being used.

Standard actions are summarized in this section. The standard actions include functionality that is commonly used with JSP and allow for: easily using Java Applets, including files at runtime; manipulating JavaBeans; and forwarding requests between Web Application resources.

<jsp:include/>

JSP complements the include directive by providing a standard action for including resources during runtime. The syntax of the include action is similar to an include directive and is as follows: `<jsp:include page="page" />`, where the page attribute value is the relative location in the Web Application of the page to include.

The include action can be used in a JSP the same as the include directive. Reusing the `header.jsp` and `footer.jsp` files from the include directive example in Listing 3-11 is a JSP that includes a header and footer at runtime.

Listing 3-11 runtimeInclude.jsp

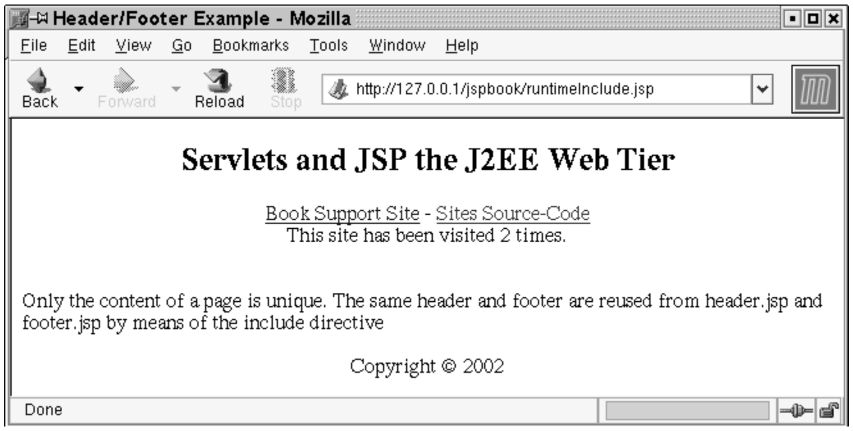
```
<jsp:include page="header.jsp" />
Only the content of a page is unique. The same header and footer are
reused from header.jsp and footer.jsp by means of the include
directive.
<jsp:include page="footer.jsp" />
```

Save the code as `runtimeInclude.jsp` in the base directory of the jspbook Web Application and browse to `http://127.0.0.1/jspbook/runtimeInclude.jsp`. The same page is shown as with `content.jsp`. Figure 3-7 shows a browser rendering of `runtimeInclude.jsp`.

The end result looks the same, but it is important to distinguish the difference between the include directive and the include action. An include that occurs at runtime is always current with the resource it is including. An include done at translation time is only current with the resource as it was at the time of translation. A simple example (Listing 3-12) illustrates the point. Edit `footer.jsp` to include a small change, a disclaimer.

Listing 3-12 Edited footer.jsp

```
<br><br>
<center>
Copyright &copy; 2003<br>
<small>Disclaimer: all information on this page is covered by
this disclaimer.</small>
```


Header/Footer Example - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://127.0.0.1/jspbook/runtimeInclude.jsp
Done

Servlets and JSP the J2EE Web Tier

Book Support Site - Sites Source-Code
This site has been visited 2 times.

Only the content of a page is unique. The same header and footer are reused from header.jsp and footer.jsp by means of the include directive

Copyright © 2002

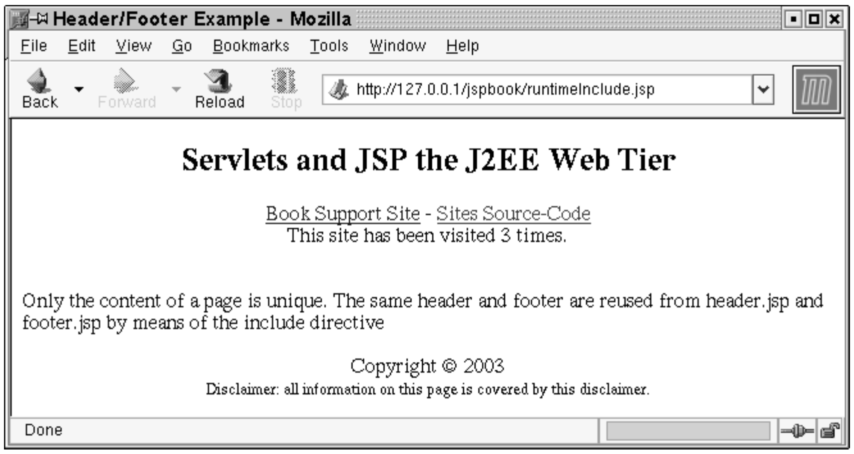
Figure 3-7 Browser Rendering of runtimeInclude.jsp

```
</center>
</body>
</html>
```

Now look at both `content.jsp` and `runtimeInclude.jsp` again to see the change. Figure 3-8 shows a browser rendering of `runtimeInclude.jsp` (`http://127.0.0.1/jspbook/runtimeInclude.jsp`). The page reflects the updates to `footer.jsp` automatically. This holds true for any update done to a resource that is included via the include action. However, `content.jsp` (`http://127.0.0.1/jspbook/content.jsp`) does not appear to reflect the change. It still looks identical to the pages that appeared in both Figure 3-6 and Figure 3-7 previously.

Runtime versus translation-time includes is the reason for the inconsistency. The `content.jsp` file was translated into a Servlet that included the exact contents of both `header.jsp` and `footer.jsp` before the edit was made to `footer.jsp`. To have the changes reflected, `content.jsp` must be re-translated to use the new `footer.jsp`. The `runtimeInclude.jsp` Servlet does not need to be re-translated because it relies on an include `footer.jsp` at runtime. The analogy to the Servlet API is that `runtimeInclude.jsp` uses `RequestDispatcher include()` method calls to access both `header.jsp` and `footer.jsp`, but `content.jsp` does not. The code for `content.jsp` was generated by directly including the code for `header.jsp` and `footer.jsp` before compiling the Servlet.

The difference between the include directive and include action is important to understand because it affects performance and consistency. A JSP that uses


Header/Footer Example - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://127.0.0.1/jspbook/runtimeInclude.jsp
Done

Servlets and JSP the J2EE Web Tier

Book Support Site - Sites Source-Code
This site has been visited 3 times.

Only the content of a page is unique. The same header and footer are reused from header.jsp and footer.jsp by means of the include directive

Copyright © 2003
Disclaimer: all information on this page is covered by this disclaimer.

Figure 3-8 Updated Rendering of runtimeInclude.jsp

include directives has the same performance as if an include was never used, but the drawback is that this JSP will not automatically reflect updates to the included file. A JSP that uses include actions will always reflect the current content of an included page, but it suffers a slight performance loss for doing so.

<jsp:plugin/>, <jsp:fallback/>, <jsp:params/>, and <jsp:param/>

JSP has very strong Java ties. JSP was originally designed as a technology Java developers would easily be able to use. For this reason it is common to see JSP being used in conjunction with the many other Java technologies that currently exist. One of the prominent uses of Java is still Java Applets. A Java Applet is a Java application, run with many restrictions, that executes in a client's Web browser through a Java-supporting plug-in. Java Applets are not heavily tied with the JSP and Servlet specifications and will not be covered in this book.

The JSP specifications define custom actions for easily generating the proper custom code needed to embed a Java Applet in an HTML page. The Applet-related actions are: `plugin`, `fallback`, and `params`. The `plugin` action represents one Applet that should be embedded in an HTML page. The `plugin` action allows for the following attributes:

type The `type` attribute identifies the type of the component: a Bean or an Applet. A *bean* is a component built to match the original intentions of the JavaBean specifications. An *Applet* is the commonly seen client-side Java functionality browsers implement via the Java plug-in.

code The `code` attribute specifies either the name of the class file that contains the Applet's compiled subclass or the path to get the class, including the class file itself. It is interpreted with respect to the `codebase` attribute.

codebase The `codebase` attribute specifies the base URI for the Applet. If this attribute is not specified, then it defaults the same base URI as for the current JSP. Values for this attribute may only refer to subdirectories of the directory containing the current document.

align The `align` attribute determines the location of the Applet relative to where it is being displayed. Valid values are `bottom`, `middle`, and `top`.

archive The `archive` attribute specifies a comma-separated list of URIs for JAR files containing classes and other resources that are to be loaded before the Applet is initialized. The classes are loaded using an instance of an `AppletClass Loader` with the given `codebase`. Relative URIs for archives are interpreted with respect to the Applet's `codebase`. Preloading resources can significantly improve the performance of Applets.

height The `height` attribute defines the height in either pixels or percent that the window displaying the Applet should use. This value can be set at runtime via an expression if needed.

hspace The `hspace` attribute determines the amount of whitespace to be inserted horizontally around the Applet.

jreversion The `jreversion` attribute identifies the spec version number of the JRE the component requires in order to operate; the default is 1.2.

name The `name` attribute specifies a name for the Applet instance, which makes it possible for Applets on the same page to find and communicate with each other.

vspace The `vspace` attribute determines the amount of whitespace to be inserted vertically around the Applet.

width The `width` attribute defines the width in either pixels or percent that the window displaying the Applet should use. This value can be set at runtime via an expression if needed.

The `fallback` action is used to provide notification to a client should the client's browser not be able to use the Java plug-in. The `fallback` action allows for an arbitrary text message to be included as its body, and must be a sub-tag to the `plugin` action. Text included in the `fallback` action is presented to a client should their browser fail to support the Java plug-in.

The `params` action is used to set parameters for the code being embedded by the `plugin` action. The `params` action requires one-to-many sub `param` actions. A `param` action has two attributes, `name` and `value`, that define a name and value, respectively, for a parameter. When using a `params` action with a `plugin` action, the `param` action must be a sub-tag of the `plugin` action.

Combining the `plugin`, `fallback`, `params`, and `param` actions, a fully customized Java Applet can easily be embedded in an HTML page. Using all the actions together is best shown through an example. Listing 3-13 embeds an Applet, `FooApplet.class`, in an HTML page that is generated by a JSP.

Listing 3-13 AppletExample.jsp

```
<html>
  <head>
    <title>Applet Example</title>
  </head>
  <body>
  <jsp:plugin
    type="applet"
    code="FooApplet.class"
    height="100"
    width="100"
    jreversion="1.2">
      <jsp:fallback>
      Applet support not found, can't run example.</jsp:fallback>
  </jsp:plugin>
  </body>
</html>
```

It is important to remember where everything takes place when using Applets and JSP. JSP executes on the server-side; Applets execute on the client-side. The result of `AppletExample.jsp` is going to be a plain HTML document with a reference to `FooApplet.class`. The actual code for `FooApplet.class` is downloaded by a Web browser by issuing a second request to the Web server from which the HTML came. A JSP does not send the Applet code inside the HTML page it generates, nor does the Applet code get executed on the Web server it came from. To further demonstrate this point, Listing 3-14 provides the code for `FooApplet.java`.

Listing 3-14 FooApplet.java

```
import javax.swing.*;
import java.awt.*;

public class FooApplet extends JApplet {
  public void init() {
    JLabel label = new JLabel("I'm an Applet.");
    label.setHorizontalAlignment(JLabel.CENTER);
    getContentPane().add(label, BorderLayout.CENTER);
  }
}
```

Compile the preceding code and place both `FooApplet.class` and `AppletExample.jsp` in the base directory of the jspbook Web Application. (Do not place `FooApplet.class` in the `/WEB-INF/classes/com/jspbook` folder. To use the Applet, a client must be able to download it. Placing the code in the `/WEB-INF` directory prevents any client from doing so.) After placing the two files, browse to `http://127.0.0.1/jspbook/AppletExample.jsp`. If your browser supports the Java plug-in version 1.2, it will load the example Applet. Figure 3-9 shows what the Applet looks like when run by a Web browser.

Should your browser not support the Java plug-in, the `fallback` message is displayed. Depending on the specific browser, it may also try to automatically download and install the Java plug-in.

The `plugin` action is designed to be easy to use, but it brings up several important points. The key point of this section is that the `plugin` action makes it easy to embed an existing Java Applet in an HTML document generated by a JSP. The `plugin` can optionally also include the `fallback`, `params`, and `param` actions as subtags to customize the Applet. Additional points brought up from this functionality were where Applets execute versus JSP and where to place Applet code in a Web Application. The two points are worth further explanation and are discussed in the remainder of this section.

To best clarify what the `plugin` action does, it is helpful to show the HTML source code generated by the action. The source code generated by `AppletExample.jsp` is listed in Listing 3-15 with the lines generated by the `plugin` action highlighted.

Listing 3-15 Output of AppletExample.jsp

```
<html>
  <head>
    <title>Applet Example</title>
  </head>
  <body>
  <object classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
width="100"  height="100"
```

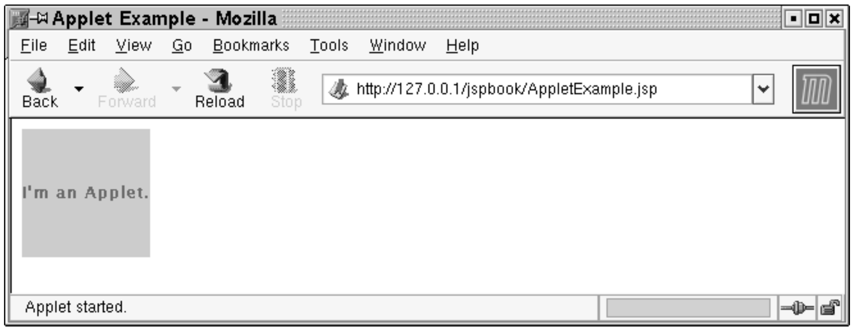

Applet Example - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://127.0.0.1/jspbook/AppletExample.jsp
I'm an Applet.
Applet started.

Figure 3-9 Browser Running FooApplet.class

```
codebase="http://java.sun.com/products/plugin/1.2.2/jinstall-1_2_2-
win.cab#Version=1,2,2,0">
<param name="java_code" value="FooApplet.class">
<param name="type" value="application/x-java-applet">
<COMMENT>
<embed type="application/x-java-applet;"  width="100"  height="100"
pluginspage="http://java.sun.com/products/plugin/"
java_code="FooApplet.class" >
<noembed>

</COMMENT>
Applet support not found, can't run example.
</noembed></embed>
</object>

  </body>
</html>
```

The code is a little cryptic but is certainly not the binary of `FooApplet.class`. What is highlighted is just a perfectly valid use of the HTML `object` element as defined by the HTML 4.0 specification[5]. The HTML `object` element is a more generic form of the now deprecated `applet` element that was designed to allow Applets to be referenced from HTML. The `object` element defines all the necessary information a browser needs to download the `FooApplet.class` file along with loading the appropriate browser plug-in to execute it. After reading the `object` element, the browser generates a completely new HTTP request for the `FooApplet.class`. The URL is a combination of information specified by the object tag, but ends up being `http://127.0.0.1/jspbook/FooApplet.class`.

The URL a Web browser uses to download an Applet is the reason that Applet code should not be placed under the `/WEB-INF/classes` directory. Code in these directories is meant solely for use on the server-side and is not accessible by outside clients. By placing the Applet's code in the base directory alongside `AppletExample.jsp`, a Web browser is free to download and use it.

<jsp:forward/>

JSP provide an equivalent to the `RequestDispather.forward()` method by use of the forward action. The forward action forwards a request to a new resource and clears any content that might have previously been sent to the output buffer by

5. http://www.w3.org/TR/html4/

the current JSP. Should the current JSP not be buffered, or the contents of the buffer already be sent to a client, an `IllegalStateException` is thrown. The `forward` action uses the following syntax: `<jsp:forward url="relativeURL"/>`, where the value of `relativeURL` is the relative location in the current Web Application of the resource to forward the request to. Optionally the forward action may have `param` actions used as subelements to define request parameters. Where applicable, the `param` action values override existing request parameters.

<jsp:forward/> and <jsp:include/> parameters

Both the JSP forward and includes actions can optionally include parameters. The mechanism for doing this is the JSP `param` action. The `param` action may only appear in the body of either the `forward` or `include` actions and is used to define parameters. The syntax of the `param` action is as follows:

```
<jsp:param name="parameter's name" value="parameter's value"/>
```

The parameter is a key/value pair with the `name` attribute specifying the name and `value` attribute specifying the value. The values are made available to the forwarded or included resource via the `HttpServletRequest getParameter()` method, for instance, if the `forward` action was authored as the following:

```
<jsp:forward page="examplePage.jsp">
  <jsp:param name="foo1" value="bar"/>
  <jsp:param name="foo2" value="<%= foo %>"/>
</jsp:forward>
```

The fictitious page `examplePage.jsp` would have two additional request parameters set for it: `foo1` and `foo2`. The value of `foo1` would be 'bar' and the value of `foo2` would be the string representation of whatever the `foo` variable was. In the case where a parameter specified by the `param` action conflicts with an existing parameter, the existing parameter is replaced.

JavaBean Actions

As with Applets, JavaBeans are commonly used Java objects, and JSP provides default support for easily using them. The `<jsp:useBean />`, `<jsp:getProperty />`, and `<jsp:setProperty />` actions all relate to JavaBeans, but will not be fully covered in this chapter. Unlike Applets, JavaBeans are much more commonly used with Servlet and JSP projects. Later chapters rely on JavaBean knowledge, and JavaBeans are explained in depth in this book. Chapter 5 introduces, explains, and shows examples of JavaBeans and the custom JSP actions for using them.

Tag File Actions

A set of JSP standard actions exist for use with custom tags. These actions are `<jsp:attribute/>`, `<jsp:body/>`, `<jsp:doBody/>`, and `<jsp:invoke/>`. Chapter 7 covers these actions in depth.

Whitespace Preservation

Servlets provide direct control over calls to the `PrintWriter` object responsible for sending text in a response. JSP do not. JSP abstracts calls to the `PrintWriter` object and allows for template text to be authored as it should be presented to a client. Whitespace, while usually not important, is preserved as it appears in a JSP. This preservation can be seen by looking at the HTML source code generated by a JSP in any of this chapter's examples.

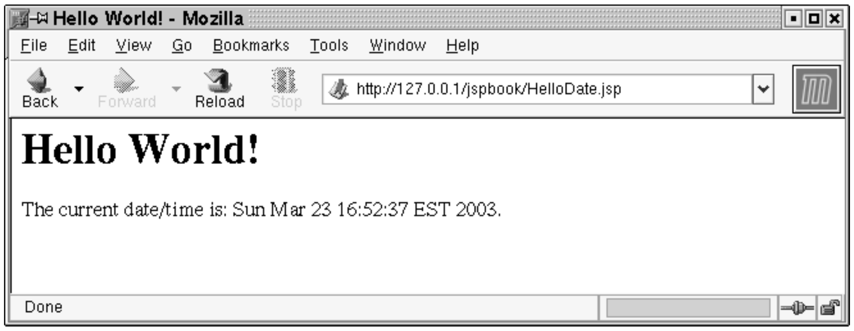
Whitespace preservation is also the reason some seemingly unaccountable formatting is included with JSP output. Take, for example, the following JSP (Listing 3-16) that is a slight modification of `HelloWorld.jsp`.

Listing 3-16 HelloDate.jsp

```
<%@ page import="java.util.Date"%>
<html>
<head>
<title>Hello World!</title>
</head>
<body>
<h1>Hello World!</h1>
The current date/time is: <%= new Date() %>.
</body>
</html>
```

Save `HelloDate.jsp` in the base directory of the jspbook Web Application and browse to `http://127.0.0.1/jspbook/HelloDate.jsp`. The page looks very similar to `HelloWorld.jsp` but now includes a date. In order to import the `java.util.Date` class, the `page` directive is used. Figure 3-10 shows a browser rendering of the results.

The code is a perfectly valid HTML document and looks fine when rendered as HTML. However, you'll notice that the formatting that surrounds the `page` directive was retained. There is an unneeded new line where the `page` directive


Hello World! - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back
Forward
Reload
Stop
http://127.0.0.1/jspbook/HelloDate.jsp
Done

Hello World!

The current date/time is: Sun Mar 23 16:52:37 EST 2003.

Figure 3-10 Browser Rendering of HelloDate.jsp

was used. This can be verified by looking at the HTML source code that was generated as highlighted in Listing 3-17.

Listing 3-17 HelloDate.jsp HTML Source Code Sent to a Browser

```
<html>
<head>
<title>Hello World!</title>
</head>
<body>
<h1>Hello World!</h1>
The current date/time is: Sun Apr 07 19:33:11 EDT 2002.
</body>
</html>
```

It is important to understand that formatting is retained by the JSP. In most cases, especially HTML, extra whitespace formatting does not matter. However, there are situations where whitespace and other extra formatting are of significance, particularly if using XML. In these cases there is an easy fix. Do not use extra formatting around JSP elements. Besides making things a little prettier, there is no need for it. For example, `HelloDate.jsp` can remove the unneeded whitespace if written as shown in Listing 3-18.

Listing 3-18 HelloDate.jsp Removing Unneeded Whitespace

```
<%@ page import="java.util.Date"%><html>
<head>
<title>Hello World!</title>
</head>
```

```
<body>
<h1>Hello World!</h1>
The current date/time is: <%= new Date() %>.
</body>
</html>
```

Attributes

There are two methods for specifying attributes in JSP elements: runtime values and translation time or static values. A *static value* is a hard-coded value that is typed into a JSP before translation time. There have been countless examples of static values; take for instance the `include` action `<jsp:include page="header.jsp"/>`. In this example the `page` attribute has a static value of `header.jsp`. Every time the JSP is visited, the `include` action tries to include this file. However, not all attribute values are required to be static. Some attributes can have a *runtime value*. A runtime value means an expression can be used to dynamically create the value of the attribute. In the preceding example of the *include* action, the following might appear:

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

In this case the value is dynamic and determined at runtime. This would be useful if there was a need to customize which page was included each time the JSP was visited.

Most basic uses of JSP do not rely on runtime values for attributes. Runtime values are much more helpful when used with JSP custom actions and will be further covered in Chapter 7 with custom tags and in Chapter 6 with the JTSL.

Comments

JSP allows for a developer to include server-side comments that are completely ignored when generating a response to send to a client. The functionality is very similar to HTML comments; however, the JSP comments are only available for viewing on the server-side. A JSP comment must have a start, `<%--`, an end, `--%>`, and comment information between. These comments are useful for providing server-side information or for "commenting out" sections of JSP code. Listing 3-19 shows an example of two comments: one to provide some information and another to comment out a bit of code.

Listing 3-19 JSPComment.jsp

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

```
  <title>Server-side JSP Comments</title>
  <body>
<%-- A simple example of a JSP comment --%>
<%--
  Code commented out on <%= new Date() %>.
--%>
  </body>
</html>
```

Save `JSPComment.jsp` in the base directory of the jspbook Web Application and browse to `http://127.0.0.1/jspbook/JSPCcomment.jsp`. A blank page is displayed because all the code on the JSP is commented out. Upon further examination of the source code that was generated by the JSP, it can be shown that none of the commented information was sent in the content of the response (Listing 3-20).

Listing 3-20 Output of JSPComment.jsp

```
<html>
<title>Server-side JSP Comments</title>
<body>


</body>
</html>
```

In addition to server-side JSP comments, more types of comments are available for use. With scriptlets and declarations, both of the Java comments are available for use. A line of embedded code can be commented out using `//`, or a chunk of code may be commented out by use of a block comment with a starting `/*` and end`*/`.

HTML/XML comments `<!-- -->` do not prevent text from being sent by a JSP. HTML/XML comments usually do not get rendered by a Web browser, but the information is passed on to the client-side. By changing `JSPComment.jsp` to use HTML/XML comments, the JSP output clearly illustrates the difference (Listing 3-21).

Listing 3-21 XMLComment.jsp

```
<%@ page import="java.util.Date" %>
<html>
  <title>Server-side JSP Comments</title>
  <body>
```

```
<!-- A simple example of a JSP comment -->
<!--
  Code commented out on <%= new Date() %>.
-->
  </body>
</html>
```

Save `XMLComment.jsp` in the base directory of the jspbook Web Application and browse to `http://127.0.0.1/jspbook/XMLComment.jsp`. A Web browser still displays a blank page, but only because the HTML/XML comments are ignored on the client-side. Listing 3-22 shows the output of `xmlcomment.jsp`.

Listing 3-22 Output of xmlcomment.jsp

```
<html>
<title>Server-side JSP Comments</title>
<body>
<!-- A simple example of a JSP comment -->
<!--
  Code commented out on Sun Mar 10 20:23:01 EST 2002.
-->
</body>
</html>
```

Unlike with `JSPComment.jsp`, `XMLComment.jsp` does send the comments to the client to deal with. Additionally JSP elements included inside the XML/HTML comments are still evaluated on the server-side. This example shows that when a chunk of code is to be commented out, it should be done with a JSP comment. However, should a comment be sent to a client, then the HTML-style comment can be used.

Quoting and Escape Characters

When authoring a JSP, it might be desirable to send text to a client that is equal in part or whole to a JSP element. This results in a conflict with the code's intended purpose and how the container will interpret code. To represent the literal value of JSP elements, in part or whole, escape characters must be used. JSP uses the following escape characters:

- A single-quote literal, `'`, is escaped as `\'`. This is only required should the literal be needed inside a single-quote delimited attribute value.

- A double-quote literal, `"`, is escaped as `\"`. This is only required should the literal be needed inside a single-quote delimited attribute value.
- A back-slash literal, `\`, is escaped as `\\`.
- A `%>` is escaped as `%\>`.
- A `<%` is escaped as `<\%`.

The entities `&apos;` and `&quot;` are available to represent single and double quotes, respectively.

The preceding examples should be fairly straightforward, but the following brief example is given for completeness. The code in Listing 3-23 shows how JSP identifies escape values that the JSP container normally interprets as elements.

Listing 3-23 EscapeCharacters.jsp

```
<% String copy="2000-2003"; %>
<html>
  <title>Server-side JSP Comments</title>
  <body>
  Scriptlets: <\% <i>script</i> %><br>
  Expressions: <\%= <i>script</i> %><br>
  Declarations: <\%! <i>script</i> %><br>
  <center>
  <small>Copyright &copy;
  <%= copy + " Single-Quote/Double-Quote Ltd,  \'/\"" %>
  </small>
  </center>
  </body>
</html>
```

Save `EscapeCharacters.jsp` in the base directory of the jspbook Web Application and browse to `http://127.0.0.1/jspbook/EscapeCharacters.jsp`. A small page appears with a brief explanation of some of the JSP elements. Figure 3-11 shows a browser rendering of the output from the JSP. The literal values are properly shown to a client instead of being misinterpreted by the JSP container.

Implicit Objects

JSP uses scripting elements as an easy method of embedding code within template text, but we have yet to show how to directly manipulate a request, response, session, or any of the other objects used with Servlets in Chapter 2. These objects all still exist with JSP and are available as *implicit objects*. The JSP implicit objects are automatically declared by a JSP container and are always available for use by scripting elements. The following is a list of the JSP implicit objects that are recognizable from Chapter 2.

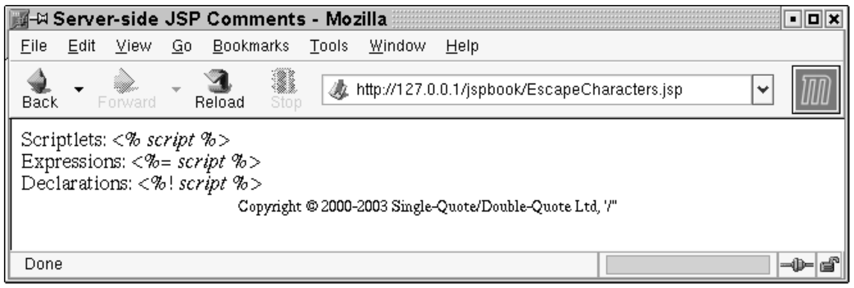
config The `config` implicit object is an instance of a `javax.servlet.ServletConfig` object. Same as with Servlets, JSP can take advantage of initial parameters provided in a Web Application Deployment Descriptor.

request The `request` implicit object is an instance of a `javax.servlet.http.HttpServletRequest` object. The `request` implicit object represents a client's request and is a reference to the `HttpServletRequest` object passed into a `HttpServlet`'s appropriate service method.

response The `response` implicit object is an instance of a `javax.servlet.http.HttpServletRequest` object. The `response` implicit object represents a response to a client's response and is a reference to the `HttpServlet Response` object passed into a `HttpServlet`'s appropriate service method.

session The `session` implicit object is an instance of a `javax.servlet.http.HttpSession` object. By default JSP creates a keep session context with all clients. The session implicit object is a convenience object for use in scripting elements and is the equivalent of calling the `HttpServletRequest getSession()` object.

application The `application` implicit object is an instance of a `javax.servlet.ServletContext` object. The application implicit object represents a Servlet's view of a Web Application and is equivalent to calling the `Servlet Config getServletContext()` method.


Server-side JSP Comments - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://127.0.0.1/jspbook/EscapeCharacters.jsp
Done

Scriptlets: <% *script* %>
Expressions: <%= *script* %>
Declarations: <%! *script* %>
Copyright © 2000-2003 Single-Quote/Double-Quote Ltd, '/"

Figure 3-11 Browser Rendering of EscapeCharacters.jsp

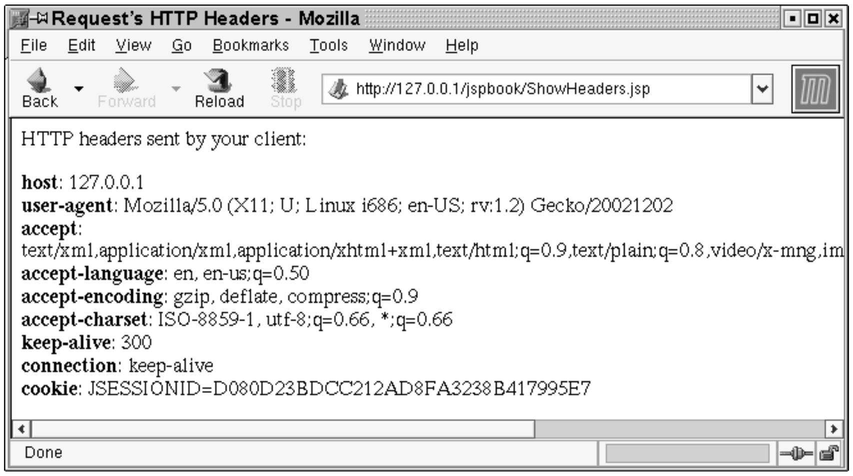
Using any of the JSP implicit objects is as easy as assuming they already exist within a scripting element. With the objects listed previously, all of the Servlet examples in previous chapters can be replicated in JSP. Take, for example, the ShowHeaders Servlet (Listing 2-8 in Chapter 2). The ShowHeaders Servlet displayed a small HTML page listing all the HTTP request headers sent by a client. The Servlet relied on the `HttpServletRequest getHeaderNames()` and `getHeader()` methods. After translating this Servlet into a JSP, the code appears as Listing 3-24.

Listing 3-24 ShowHeaders.jsp

```
<%@ page import="java.util.*"%>
<html>
  <head>
    <title>Request's HTTP Headers</title>
  </head>
  <body>
    <p>HTTP headers sent by your client:</p>
<%
   Enumeration enum = request.getHeaderNames();
   while (enum.hasMoreElements()) {
     String headerName = (String) enum.nextElement();
     String headerValue = request.getHeader(headerName);
%>
    <b><%= headerName %></b>: <%= headerValue %><br>
<% } %>
  </body>
</html>
```

Save the preceding code as `ShowHeaders.jsp` in the base directory of the jspbook Web Application and browse to `http://127.0.0.1/jspbook/ShowHeaders.jsp`. The results are identical to the previous Servlet at `http://127.0.0.1/jspbook/ShowHeaders`. Figure 3-12 shows what the output of `ShowHeaders.jsp` looks like when rendered by a Web browser.

Repeating Servlet code examples and translating them into JSP isn't the goal of this chapter. The preceding example is intended to clearly show how to use the implicit objects and how they can be used to achieve all the functionality of a Servlet. The scriptlets in the preceding JSP show this by using the `request` implicit object as if it had previously been declared by the JSP.


Request's HTTP Headers - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://127.0.0.1/jspbook/ShowHeaders.jsp
Done

HTTP headers sent by your client:

host: 127.0.0.1
user-agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.2) Gecko/20021202
accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-mng,im
accept-language: en, en-us;q=0.50
accept-encoding: gzip, deflate, compress;q=0.9
accept-charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
keep-alive: 300
connection: keep-alive
cookie: JSESSIONID=D080D23BDCC212AD8FA3238B417995E7

Figure 3-12 Browser Rendering of ShowHeaders.jsp

```
<%
   Enumeration enum = request.getHeaderNames();
   while (enum.hasMoreElements()) {
     String headerName = (String) enum.nextElement();
     String headerValue = request.getHeader(headerName);
%>
```

It is important to note that in no place was an object named `request` declared for use by the JSP. It was just used. The JSP container automatically and appropriately declares the implicit objects when translating the JSP into a Servlet.

JSP defines a few more implicit objects to accompany the aforementioned. The ones not listed do not directly map to Servlet equivalents. The additional implicit objects are `pageContext`, `page`, `out`, and `exception`, which are all explained in the following sections.

pageContext

The `pageContext` implicit scripting variable is an instance of a `javax.servlet.jsp.PageContext` object. A `PageContext` object represents the context of a single JavaServer Page including all the other implicit objects, methods for forwarding to and including Web Application resources, and a scope for binding objects to the page. The `PageContext` object is not always helpful when used by itself because the other implicit objects are already available for use. A `PageContext` object is primarily used as a single object that can easily be passed to other objects such as custom actions. This is useful since the page context holds references to the other implicit objects.

Request Delegation

The `pageContext` implicit object provides the equivalent of the `include` and `forward` directives for providing JSP request delegation. Scriptlets can use the following `PageContext` methods to provide JSP request delegation:

```
forward(java.lang.String relativeUrlPath)
```

The `forward()` method is used to redirect, or 'forward', the current `ServletRequest` and `ServletResponse` to another resource in the Web Application. The *relativeUrlPath* value is the relative path to a resource in the Web Application:

```
include(java.lang.String relativeUrlPath)
```

The `include()` method causes the resource specified to be processed as part of the current `ServletRequest` and `ServletResponse` being processed.

The `forward()` and `include()` methods can be used to include or forward a `ServletRequest` and `ServletResponse` to any resource in a Web Application. The resource can be a Servlet, JSP, or a static resource. The functionality is identical to that previously mentioned for Servlet request delegation.

Page Scope

In addition to the request, session, and application scopes, JSP introduces the *page scope*. The `PageContext` object provides the `getAttribute()`, `setAttribute()`, and `removeAttribute()` methods for binding objects to the current page. Objects bound in page scope only exist for the duration of the current page. Page scope objects are not shared across multiple JSP, and page scope is intended only for passing objects between custom actions and scripting elements. When using JSP to JSP communication, the request scope is still the appropriate scope to use.

out

The `out` implicit object is an instance of a `javax.servlet.jsp.JspWriter` object and is used to send content in a response. The `JspWriter` object emulates some of the functionality found in the `java.io.PrintWriter` and `java.io.BufferedWriter` objects to provide a convenient method of writing text in a buffered fashion. The `out` implicit object can be configured on a per JSP basis by the `page` directive.

Buffering

The initial `JspWriter` object is associated with the `PrintWriter` object of the `ServletResponse` in a way that depends on whether the page is or is not buffered. If the page is not buffered, output written to this `JspWriter` object will be written through to the `PrintWriter` directly. But if the page is buffered, the `PrintWriter` object will not be created until the buffer is flushed, meaning operations like `setContentType()` are legal until the buffer gets flushed. Since this flexibility simplifies programming substantially, buffering is the default for JSP pages.

By using buffering, the issue is raised about what happens when the buffer is exceeded. Two possibilities exist:

Flush the Buffer One straightforward option is to simply flush the buffer once it is full. Content that would normally overflow the buffer now would not because the buffer writes extra content to a client. The drawback to this approach is that HTTP headers cannot be changed once content has been sent to a client. Headers always appear at the beginning of a HTTP response so they must be finalized before any content is flushed by the buffer.

Throw an Exception Flushing the buffer is not a good approach when strict control needs to be kept over when content is sent to a client. In cases like this, exceeding the buffer is a fatal error. Doing so causes an exception to be thrown.

Both approaches are valid, and thus both are supported by JSP. The behavior of a page is controlled by the `autoFlush` attribute, which defaults to `true`. In general, JSP that need to be sure correct and complete data has been sent to their client may want to set `autoFlush` to `false`. On the other hand, JSP that do not need strict control can leave the `autoFlush` attribute as `true`, which is commonly the case when sending HTML to a browser. The two types of buffer uses are best suited for different needs and should be considered on a per use basis.

JspWriter and Response Committed Exceptions

A far too common and misunderstood error when using JSP is the `IllegalStateException` exception with "response already committed" given as the exception's message. This error arises after a `JspWriter` has sent some information to a client and a JSP tries to do something assuming no content has been sent. Avoiding this exception is easily done but requires that when a developer programs, he or she is conscious of how the `JspWriter` object works. The following are the two primary culprits of the aforementioned exception.

Manipulating Headers With JSP, manipulation of the HTTP response headers is only allowed before the actual content of the response is sent. When phrased like this, it should seem quite intuitive, but far too often a JSP developer will ask why an `IllegalStateException` is thrown when they are changing header information. An easy fix for this problem is to either increase the buffer size by increasing the value of the `page` directive `buffer` attribute or simply moving problematic code to the top of the JSP. Moving header-changing code before content-generating code usually ensures there are no buffer conflicts when editing HTTP header information.

Forwarding When forwarding between JSP, complete control of the `ServletRequest` and `ServletResponse` objects is given to the forwarded JSP. This allows for the forwarded page to have complete control over generating the appropriate response. Unlike Servlets, JSP automatically calls the `HttpServletRequest getWriter()` method to get a suitable object for writing information to a client. Forwarding between two JSP ensures calling this method twice, which in a normal Servlet would throw an exception. However, JSP bends this rule slightly by taking advantage of the `JspWriter` buffer. Should a JSP forward a request to another JSP after content has been sent to the buffer but before the response has been committed to a client, then everything is fine. The buffered data is simply discarded and the new JSP can freely create a response to a client. Should a JSP commit a response to a client and then forward the request to a different JSP, an exception occurs.

Committing a response and then forwarding a request always throws an `IllegalStateException`. The problem can be solved by either including all information in one JSP, not committing the response, or including the desired JSP rather than forwarding to it. An inclusion reuses the `JspWriter` object of the page doing the include.

Understand and avoid the above two problems. The cryptic error commonly plagues new JSP developers. Solving the problem is easy if the `JspWriter` object and associated buffer are properly understood.

config

The normal JSP deployment scheme automatically done by a container works, but nothing stops a JSP developer from declaring and mapping a JSP via the Web

Application Deployment Descriptor, web.xml. A JSP can be manually deployed in the same fashion as a Servlet by creating a Servlet declaration in web.xml and replacing the servlet-class element with the jsp-page element. After being declared, the JSP can be mapped to a single or set of URLs same as a Servlet.

As an example, if it was necessary to remove the ShowHeaders Servlet and map ShowHeaders.jsp to the /ShowHeaders path in addition to the automatically defined /ShowHeaders.jsp path, the task could be accomplished with the following entries in web.xml.

```
<servlet>
  <servlet-name>ShowHeaders</servlet-name>
  <jsp-file>/ShowHeaders.jsp</jsp-file>
</servlet>
<servlet-mapping>
  <servlet-name>ShowHeaders</servlet-name>
  <url-pattern>/ShowHeaders</url-pattern>
</servlet-mapping>
```

The only change was replacing the previous line, <servlet-class>com.jspbook.ShowHeaders</servlet-class>, with the jsp-file element and the location of the JSP.

Initial Configuration Parameters

Through use of the jsp-file element, a JSP can be mapped using a custom entry in web.xml. All of the child elements of the servlet element are still valid, and initial parameters can be defined. In Chapter 2 the InternationalizedHelloWorld Servlet, Listing 2-3, was used to demonstrate the functionality of initial parameters. Listing 3-25 is a quick rehash of the example, but in JSP form.

Listing 3-25 InternationalizedHelloWorld.jsp

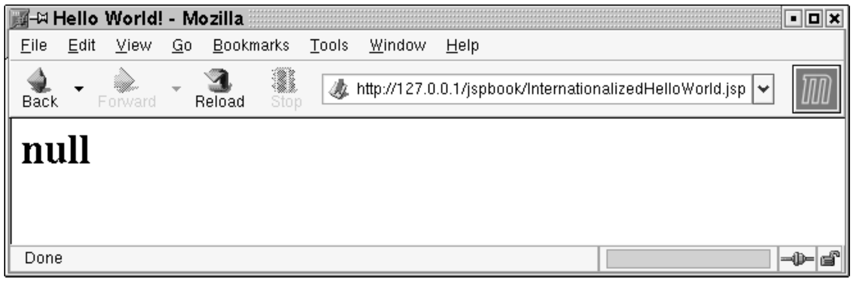
```
<html>
<head>
<title>Hello World!</title>
</head>
<body>
<h1><%=config.getInitParameter("greeting")%></h1>
</body>
</html>
```

The code is nothing spectacular. What is important to notice is the JSP relies on an initial parameter named "greeting". Without the initial parameter, the JSP would not function correctly, but a container will automatically deploy the page anyhow. Save the code as `InternationalizedHelloWorld.jsp` in the base directory of the jspbook Web Application and browse to `http://127.0.0.1/jspbook/InternationalizedHelloWorld.jsp`. A page appears that says "null". Figure 3-13 shows a browser rendering of the output. By default a JSP has no initial parameters, and a JSP container doesn't validate that initial parameters are properly defined before deploying a JSP. The result is a HelloWorld example that says nothing.

To fix the JSP, an entry in `web.xml` needs to be made so the "greeting" initial parameter can be defined. Add the following elements to `web.xml`.

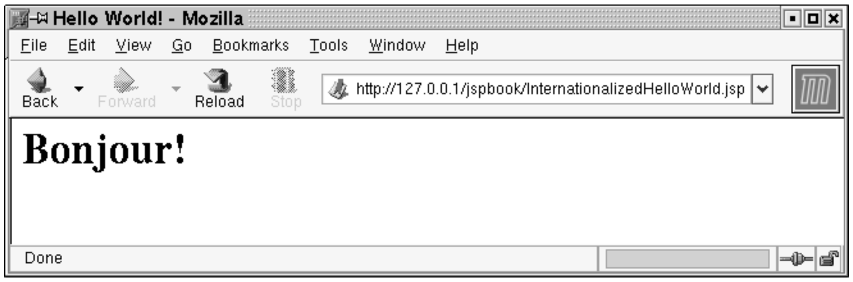
```
<servlet>
  <servlet-name>InternationalizedHelloWorldJSP</servlet-name>
  <jsp-file>/InternationalizedHelloWorld.jsp</jsp-file>
  <init-param>
    <param-name>greeting</param-name>
    <param-value>Bonjour!</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>InternationalizedHelloWorldJSP</servlet-name>
  <url-pattern>/InternationalizedHelloWorld.jsp</url-pattern>
</servlet-mapping>
```

The `servlet` element defines a Servlet deployment for the Servlet generated from `InternationalizedHelloWorld.jsp`, and the `servlet-mapping` element maps the URL pattern `/InternationalizedHelloWorld.jsp` to the JSP. Inside the servlet element, the needed "greeting" initial parameter is given to make the JSP display a "Hello World" message. Reload the jspbook Web Application and


Hello World! - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://127.0.0.1/jspbook/InternationalizedHelloWorld.jsp
Done

null

Figure 3-13 InternationalizedHelloWorld.jsp without Initial Parameters


Hello World! - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back
Forward
Reload
Stop
http://127.0.0.1/jspbook/InternationalizedHelloWorld.jsp
Done

Bonjour!

Figure 3-14 InternationalizedHelloWorld.jsp with Parameters

browse back to `http://127.0.0.1/jspbook/InternationalizedHelloWorld.jsp`. This time the JSP displays the appropriate hello message. Figure 3-14 shows a browser rendering of the output.

page

The `page` implicit object represents the current class implementation of the page being evaluated. If the scripting language of the page is `java`, which by default it is, the page object is equivalent to the `this` keyword of a Java class.

JSP in XML Syntax

JSP comes in two different varieties of syntax. The original, or classic, JSP uses a free-form syntax. With JSP 1.2, another XML-compliant form of JSP syntax, JSP Documents, was introduced. Both syntaxes provide the same functionality and take advantage of all the features of JSP. The reason the second syntax was introduced was to keep JSP current with the widespread adoption of XML. XML-compliant JSP can be created and manipulated using any existing XML tool. Classic JSP requires a specialized parser built to specifically understand the unique syntax of JSP.

Since the introduction of XML-compliant JSP, there have been no significant moves in the JSP community toward supporting the new syntax. The majority of JSP developers, books, and tools still largely use the classic JSP. Reasons for this are partly due to the fact that JSP documents are new, but are largely related to the fact that JSP XML syntax is not easy to use. In some senses the first release of the JSP XML syntax was very half-baked in an odd way. It is too strict. The syntax does not lack compliance to XML rules, nor does it lack functionality. It is just too restrictive and cryptic for the average JSP developer to use.

The best way to illustrate the original flaw in JSP XML syntax is by showing a small example. This page is a simple version of what can be expected to be seen in most JSP. Listing 3-26 is the version of the page in the classic JSP syntax.

Listing 3-26 ClassicJSP.jsp

```
<html>
<head>
<title>A Simple Page in Classic JSP</title>
</head>
<body>
<h1>A Title</h1>
<% String text = "<b>bold text</b>";
   String link = "http://www.jspbook.com";
  if (true) { %>
  Here is bold text: <%= text %><br>
<% } %>
A link to a <a href="<%= link %>">website</a>.
</body>
</html>
```

Everything above should be recognizable. It is a page that uses a few scriptlets and expressions. Listing 3-27 shows the same code in a JSP Document.

Listing 3-27 JSPDocument.jsp

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page">
  <html>
    <head>
      <title>A Simple Page in XML Compatible JSP</title>
    </head>
    <body>
    <h1>A Title</h1>
    <jsp:scriptlet>
    String text = "<b>bold text</b>";
    String link = "http://www.jspbook.com";
      if (true) {
    </jsp:scriptlet>
       Here is bold text:
       <jsp:expression>text</jsp:expression><![CDATA[<br>]]>
    <jsp:scriptlet>}</jsp:scriptlet>
    <![CDATA[A link to a <a href="]]>
    <jsp:expression>link</jsp:expression>
    <![CDATA[">website</a>.]]>
    </body>
  </html>
</jsp:root>
```

The first point to notice is that the page gets bigger. This is always a minor drawback to using any form of XML. A little bit of space inefficiency is paid for compliance to XML document structuring rules. The space itself is not of concern in this case, but what should be a concern is that the JSP can no longer use free-form text. The JSP scripting elements must be expanded into full tags, and noncompliant HTML must be surrounded by a special XML syntax, `<![CDATA[ ]]>`. This is tedious to author and makes a page hard to maintain without a special XML reading and writing tool.

JSP Documents are not all bad. The idea behind them is a good one. XML, when used as intended, can be a very helpful thing. Custom XML documents can be incredibly easy to understand and are easily manipulated by countless XML tools that currently exist. The only drawback is that HTML is not XML. JSP is largely promoted as a tool that makes dynamic HTML generation easy. This is not a restriction of JSP, but it is arguably the most common use of the technology. The question to answer is, "To what extent is XML compatibility needed in your code?" If JSP need to be manipulated easily by other code, then XML is a good choice. The other question to ask is, "Are you only using JSP to simplify creating dynamic HTML?" If so, then it is a better choice to use the original JSP syntax.

In previous versions of JSP, the majority of users were geared toward using JSP for creating dynamic HTML. This is largely due to the fact that HTML has been the dominant technology on the Web, and this explains very much why JSP was originally created to simplify the task of creating it. However, HTML is no longer the most popular technology to use. XML, while not perfect, adequately fills the deficiencies of many technologies, including HTML, and has gained huge momentum, which is shown by industry-wide use. Currently, one of the best approaches to managing content is to either store it or communicate it via XML. Using XML for these purposes allows information to easily be shared and maintained in a meaningful manner. Because of this great flexibility, the trend has been to move away from more limited technologies, such as HTML, and toward XML. JSP reflects these changes as it too has changed to better incorporate XML for use in the J2EE Web Tier.

Understanding JSP documents is important. XML use will only continue to grow in the future, and it is important to understand what flexibility JSP has for interacting with it. Understanding how to author JSP documents is also easy as long as you understand a few simple conversion rules between classic JSP and JSP in XML syntax.

XML Rules

XML rules is quite the pun. XML does rule as a technology for authoring and sharing information on the Internet, but XML does have some important rules one must follow when using it. JSP Documents automatically inherit these rules. It does little good to directly explain JSP in XML syntax if regular XML syntax is not understood. However, this book is not about XML. It is about JSP and Servlets. A full tutorial on XML is not given in this book. Only the basics are explained to help get through the majority of the JSP in XML syntax use cases. If you are planning on extensively using XML with JSP and do not yet know much about XML, this book is not a substitute for an XML guide. To accompany this text, either read through the XML specifications, `http://www.w3.org/ XML`, or pick up a good book on XML.

JSP Documents

Aside from understanding the XML rules needed to author JSP in XML syntax, there are only a few simple conversions between regular JSP and JSP Documents. Not all pieces of regular JSP syntax are in an XML-incompatible form. JSP actions and custom actions are already in XML-compatible syntax. They are used identically in a JSP Document as used in regular JSP. The rest of JSP, namely scripting elements and directives, need to be converted to an XML form.

JSP Document Declaration

The JSP Document must be completely encapsulated by a root XML element, `root`. This element needs to also have the JSP namespace, `jsp`, pre-appended along with a declaration for the namespace. In general, a JSP Document always resembles Listing 3-28.

Listing 3-28 Declaration of a JSP Document

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page">
  // JSP Document contents
</jsp:root>
```

The content encapsulated by the JSP Document is the content of the JSP.

Scripting Elements

All scripting elements must be converted for use in a JSP Document. The scripting element syntax classic JSP uses directly conflicts with XML syntax. Instead of using `<% %>`, `<%= %>`, and `<%! %>`, for scriptlets, expressions, and declarations, use `<jsp:scriptlet></jsp:scriptlet>`, `<jsp:expression></jsp:expression>`, and `<jsp:declaration></jsp:declaration>`, respectively.

This conversion in most cases is quite simple. Refer back to the first classic JSP versus JSP Document example, Listing 3-25 and Listing 3-26. Here is a section of the code from the classic JSP example that uses scriptlets and expressions.

```
<% String text = "<b>bold text</b>";
   String link = "http://www.jspbook.com";
  if (true) { %>
  Here is bold text: <%= text %> <br>
<% } %>

A link to a <a href="<%= link %>">website</a>.
```

Highlighted are the scriptlets and expressions. The conversion to JSP in XML syntax is the following:

```
<jsp:scriptlet>
String text = "<b>bold text</b>";
String link = "http://www.jspbook.com";
if (true) {
</jsp:scriptlet>
  Here is bold text:
  <jsp:expression>text</jsp:expression><![CDATA[<br>]]>
<jsp:scriptlet>}</jsp:scriptlet>
<![CDATA[A link to a <a href="]]>
<jsp:expression>link</jsp:expression>
<![CDATA[">website</a>.]]>
```

`The straight change from classic scripting elements to JSP Document equivalents should be easily seen. Any text search and replace tool can easily accomplish the job. The more difficult part is checking to make sure the conversion results in a valid XML document. In the preceding case, it didn't. The example specifically included one of the most common errors that occurs when using JSP Documents. In a classic JSP, it is perfectly valid to use an expression or scriptlet right in the middle of template text: A link to a <a href="<%= link %>">website</a>.`

In the trivial conversion to a JSP Document, this initially becomes the following:

```
A link to a <a href="
<jsp:expression>link</jsp:expression>
">website</a>.
```

However, the above code is not XML because the document is no longer well formed. The template text was being treated as XML. Embedding an expression tag for an attribute value is not allowed. To solve this problem, the conversion has to also include a specific encapsulation of the template text with XML `CDATA` sections or represent the problematic content with entities.

```
<![CDATA[A link to a <a href="]]>
<jsp:expression>link</jsp:expression>
<![CDATA[">website</a>.]]>
```

`CDATA` sections were used in the preceding snippet. It is a choice of personal preference choosing to use `CDATA` sections or entities when handling offending code. The point is, that converting straight between `<% %>` and `<jsp:scriptlet> </ jsp:scriptlet>` is trivial. What matters most is making sure a well-formed XML document is created. If not, replace offending code with entities or `CDATA` sections.

Directives

Recall that JSP directives always follow the format `<%@directive {attribute="value"}* %>` ,where *directive* is the directive's name and *attributes* is a set of attributes with specified values. Like the scripting elements, this syntax does not comply with XML and needs to be converted. Unlike scripting elements the conversion is always trivial. JSP Documents use directives same as classic JSP but with the following syntax: `<jsp:directive.directive {attribute="value"}*/>`. The conversion is just a straight swap and includes the same directive and attribute values.

For clarity, Listing 3-29 shows a brief example of a JSP in classic syntax, which uses a `page` and include directive.

Listing 3-29 JSPDocumentDirectives.jsp

```
<%@page errorPage="ErrorPage.jsp"%>
<%@include file="header.jsp"%>
```

```
  <h1>A Title</h1>
  <p>Some text.</p>
<%@include file="footer.jsp"%>
```

Converting the preceding code to XML syntax is as easy as doing a direct replacement of the directives. In general, this will always be the case with directives (Listing 3-30).

Listing 3-30 JSPDirectives.jsp

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page">
  <jsp:directive.page errorPage="ErrorPage.jsp"/>
  <jsp:directive.include file="header.jsp"/>
    <h1>A Title</h1>
    <p>Some text.</p>
  <jsp:directive.include file="footer.jsp"/>
</jsp:root>
```

Encapsulating Template Text

One of the unaccountably ridiculous requirements of the original JSP in XML syntax is to require JSP Documents to surround template text with `<jsp:text>` elements. There are no XML requirements mandating this. This requirement was intended to be a feature for aiding JSP parsers but greatly complicates authoring template text in JSP Documents.

Summary

This chapter is an introduction to JavaServer Pages (JSP). JSP is a complementary technology to Servlets that provides an incredibly efficient way of developing a text-producing Servlet. Unlike Servlets, JSP is not authored in a Java 2-compliant syntax, but JSP is translated to and managed by a container same as a Servlet. After authoring a JSP, there is no need to manually deploy the JSP to a URL extension via `web.xml`. A container automatically deploys a JSP, but a `web.xml` entry can still be used to provide initial parameters or arbitrary URL extensions for a JSP.

A JSP is divided into two main parts: template text and dynamic elements. Template text consists of everything that would normally appear in `print()` or `println()` calls of a Servlet. Dynamic elements are special bits of syntax defined by the JSP specifications. A dynamic element is not treated directly as text but is instead evaluated by a container to perform some custom functionality.

JSP elements are broken down into three main categories: scripting elements, directives, and actions. Scripting elements are a method of directly embedding code between template text. Directives are a method of giving a JSP container configuration information at translation time. Actions are used to link XML-compatible tags to custom code that is not included in the JSP. The JSP specifications define a few default actions, but there also exists a method for binding custom code to custom actions. Custom actions are one of the more powerful features of JSP and are left for full coverage in Chapter 7.

There are two different syntax styles available for authoring JSP. The first is the classic JSP syntax and has been available since the original release of JSP. This classic syntax is what the majority of this book uses and is what is commonly considered the easiest syntax to author JSP. The alternative JSP syntax is available for situations where it is helpful to have a JSP be authored as an XML-compliant document. Both syntax styles provide the same functionality. Converting between the two types of JSP syntax is usually a trivial task.