Pro J2EE 1.4: From Professional to Expert

SUE SPIELMAN AND MEERAJ KUNNUMPURATH
WITH
NEIL ELLIS AND JAMES L. WEAVER

Apress®

Pro J2EE 1.4: From Professional to Expert

ISBN (pbk): 1-59059-340-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewers: Thomas Marrs and Matthew Moodie
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, John Franklin, Jason Gilmore, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski
Project Manager: Tracy Brown Collins
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Production Manager: Kari Brooks
Production Editor: Kelly Winquist
Compositor: Diana Van Winkle, Van Winkle Design Group
Proofreader: Nancy Sixsmith
Indexer: Bill Johncocks
Artist: Diana Van Winkle, Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 3

Working with JSP 2.0

The JavaServer Pages (JSP) technology has been a key part of the Java 2 Enterprise Edition (J2EE) platform for quite some time. The goal of the JSP technology is to simplify dealing with presentation and dynamic data on Web pages. JSP technology is based on Servlet technology; in fact, all JSP pages are eventually compiled into servlet code. However, writing a JSP page is far less involved than writing servlet code. In fact, page authors who don't have Java experience often write JSP pages. A JSP page is a markup document that can be in either JSP syntax or Extensible Markup Language (XML) format.

This chapter covers the basic mechanisms used in a JSP page. This includes the following:

- Introducing JSP technology
- Understanding the page life cycle
- Understanding directives, actions, scripting, implicit objects, and scoping
- Examining the new expression language in detail
- Introducing localization and handling content
- Addressing debugging and performance concerns
- Using best practices

Introducing JSP Technology

If the purpose of JSP technology were to be summed up in one sentence, it would be this: JSP provides a flexible mechanism to produce dynamic content.

That one sentence will be the focus of the next two chapters. JSP works with the concepts of template and dynamic data; *template data* is fixed content, such as text or XML data, and *dynamic data* changes according to some request. You can combine dynamic data with template data. This provides a flexible mechanism for creating Web pages.

It's also possible to have functionality that's used throughout a Web application and to encapsulate that functionality for use on JSP pages. JSP offers two mechanisms for encapsulated functionality: JavaBeans and custom actions. JavaBeans are just Java classes that follow the JavaBean specification, and custom actions provide a mechanism for building libraries, listeners, and functions and for performing validations. Custom actions are so powerful and feature rich that Chapter 5 is just about using custom actions in JSP 2.0.

JSP has always been focused on the presentation aspects of Web applications, but Java programmers have gotten their coding hands into the logic processing and have mistakenly included logic in JSP pages. Over time, this has led to some unfortunate, and unreadable, pages. *Scriptlets*, small sections of Java code that are embedded into the markup of a JSP page, is the true culprit here.

However, the JSP 2.0 specification will allow you to say "goodbye" to scriptlet coding in JSP files. While the mechanism remains available for use, using scriptlets in JSP files is just plain poor engineering.

Introducing JSP 2.0

JSP technology has been the presentation layer of the J2EE platform since the platform was introduced in 1999. If you're working with the J2EE platform, then at some point you'll probably need to use JSP technology to build Web applications. Previous versions of the JSP specification introduced such features as custom tag extensions, XML representations of JSP pages, and a validation mechanism.

In JSP 2.0, first you'll notice that the new major revision number is warranted. Since JSP 1.2, there has been some significant functionality added. This includes the following:

- The incorporation of the expression language (EL) introduced in the JSP Standard Tag Library (JSTL)
- JSP fragments that allow for a portion of JSP syntax to be encapsulated into a Java object
- Tag files that are a much simpler way to write custom actions using JSP syntax
- Simple tag handlers that represent a new interface with a much simpler life cycle than previous custom action interfaces and that can be integrated with JSP fragments
- The addition of new standard actions to support JSP fragments and tag files

The pages that follow in this chapter and the next two provide a detailed look at JSP and all of these new features.

Understanding the Page Life Cycle

When a JSP page is created, it's considered to be a Web component. Web components run in their appropriate containers. For example, a JSP file runs within a JSP container. The container provides the services that support the component; in a sense, the container enforces the contract between itself and its component. As you saw in Chapter 2, the container is responsible for translating the JSP from a text-based document into runnable servlet code.

A JSP page and the container actually deal with two phases: *translation time* and *run time*. Translation time is where a number of tasks are performed: The container takes a JSP file, performs various tasks such as validation, generates the appropriate servlet code, and compiles and loads the servlet. This stage is where translation errors occur if you have syntax errors on your JSP page or custom actions that can't be resolved.

The container translates a JSP page the first time the JSP page is requested, which sometimes slows performance the first time you request a JSP page (see the "Performance" section later in this chapter for some specific performance enhancements for JSP pages). After the JSP page has been translated, it's just a matter of executing the servlet code. If the JSP page changes, the next time you request the JSP page, it'll be retranslated so that new servlet code can be generated.

The execution of a JSP page—or, more precisely, the servlet code that represents the page—occurs at run time. When a request for a certain JSP page comes into the container, the implementation class of the JSP page that was created at translation time is instantiated. This is analogous to what you know about running Java classes; the container executes the servlet code and takes the appropriate actions. Understanding the two phases of JSP technology is important because different tasks happen in each phase. Figure 3-1 shows the entire process.

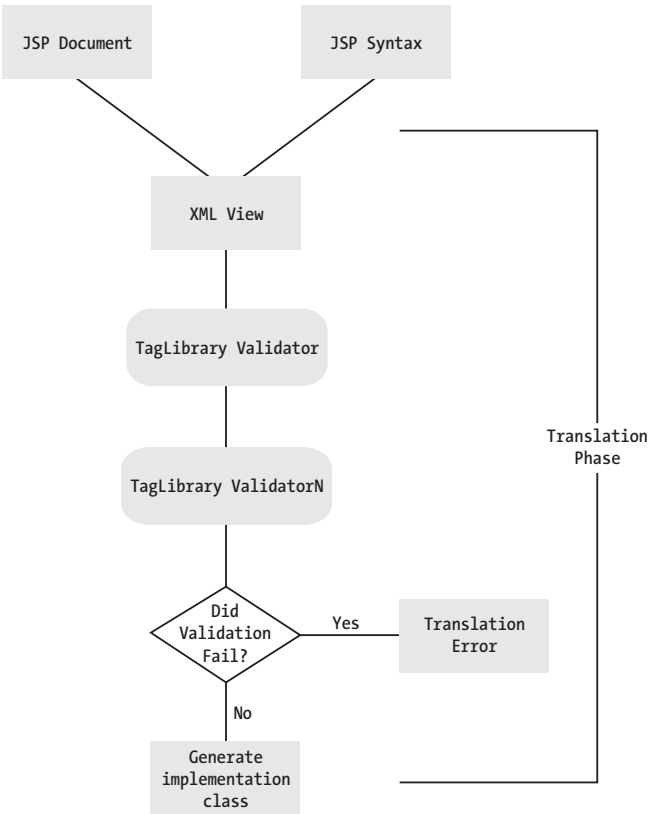

Figure 3-1. JSP phases

When the JSP container generates the servlet class that corresponds to the JSP, there's a well-defined contract that's established between the container and the JSP. The overall job of the JSP container can be summed up as follows: The JSP container delivers requests from a client to a JSP page implementation object, and responses from the JSP page implementation object to the client.

JSP Documents

It's possible to represent JSP content in XML syntax. A compliant XML document that represents a JSP page is called a *JSP document*.

> **NOTE** *A JSP document is a JSP page that's a namespace-aware XML document. It's identified as a JSP document to the JSP container so that it can be translated into an implementation class.*

A JSP document needs to be identified, either implicitly or explicitly, to the JSP container so it can be processed as an XML document. This includes such activities as checking for a well-formed file and, if present, applying any requests (such as entity declarations). JSP documents generate dynamic content using the standard JSP semantics. You'll see how you can use the new `<jsp:element>` and `<jsp:text>` standard actions in JSP documents later in the "<jsp:element>" section.

The main purpose of JSP documents is in the generation of dynamic XML content, but they can also generate any dynamic content. You may want to create a JSP document vs. a JSP page for the following reasons:

- You can pass JSP documents directly to the JSP container. As more and more content is written in XML, XML-based languages such as Extensible HTML (XHTML) and Scalable Vector Graphics (SVG) can exchange documents in applications such as Web services; the generated content may be sent directly to a client, or it may be part of some XML processing pipeline.

- XML-aware tools can manipulate JSP documents.

- You can apply transformation to a textual representation, using languages such as XSL Transformation (XSLT), to generate a JSP document.

- You can use object serialization to generate a JSP document automatically.

You can also author tag files using XML syntax. The rules are similar to that of JSP documents. But we'll make the distinction between the *XML view* of a JSP page and a JSP document. The XML view is an XML document that's derived from the JSP page and produced by the container. The XML view of a JSP page is intended for use in validating the JSP page against some description of the set of valid pages.

Using the `TagLibraryValidator` class associated with a tag library, it's possible to validate a JSP page.

It's important to understand the distinction between the XML view and the JSP documents for this discussion of the new standard actions `<jsp:element>` and `<jsp:text>`. Figure 3-2 shows the relationship between the JSP document, JSP syntax, and XML view of a page.

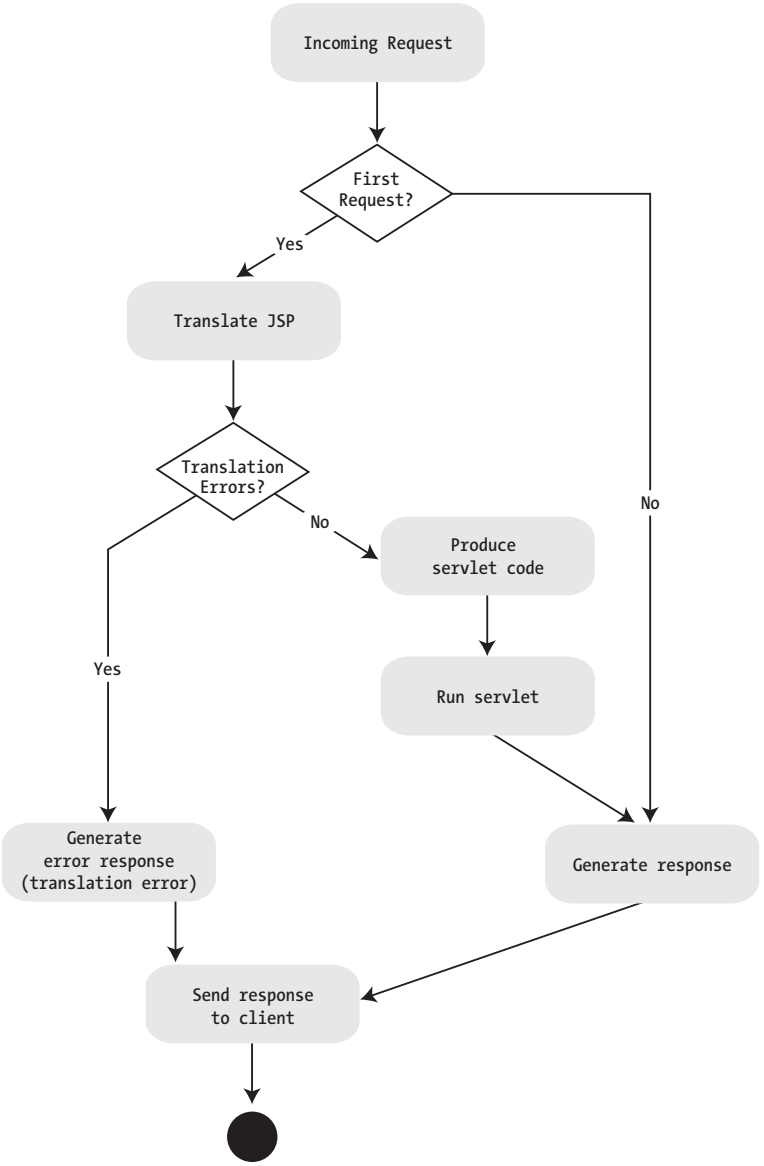

Figure 3-2. Translation phase of documents

JSP Document Syntax

The following describes the syntax elements of a JSP document:

- A JSP document may not have a `<jsp:root>` as its top element. While `<jsp:root>` was mandatory in JSP 1.2, most JSP documents in JSP 2.0 probably won't use it since JSP now has a namespace.

- JSP documents identify standard actions through a well-defined uniform resource indicator (URI) in its namespace; any prefix is valid as long as the correct URI identifying JSP 2.0 standard actions is used. Custom actions are identified using the URI that identifies their tag library; `taglib` directives aren't required and can't appear in a JSP document.

- A JSP document can use XML elements as template data. These template data elements may have qualified names because they're in a namespace, or they can be unqualified. You can use the `<jsp:text>` element to define some template data verbatim.

- A JSP document must be a valid XML document. Therefore, you can't use some JSP features in a JSP document. The elements you can use are as follows:

  - JSP directives and scripting elements in XML syntax

  - EL expressions in the body of elements and in attribute values

  - All JSP standard actions

  - The `<jsp:root>`, `<jsp:text>`, and `<jsp:output>` elements

  - Custom action elements

  - Template data described using `<jsp:text>` elements

  - Template data described through XML fragments

A Simple JSP Document

In the simplest sense, a JSP document looks like an XML document. Using the URI namespace of both JSP and the JSTL, you can define a prefix that will be used for accessing those actions. It's customary to use `fmt` as the prefix for the formatting tag library, but it's possible to specify whatever prefix you want to use in your documents.

The custom action element <fmt:formatDate> is one of JSTL actions in the formatting library. We're showing the action here so that you can get a feel for how you can use JSP and JSTL syntax within a JSP document. We'll cover the JSTL in much greater detail in the next chapter. The JSP document doesn't have an XML declaration; therefore, the encoding will default to UTF-8. However, the output will include an XML declaration because of the defaulting rules and the absence of a <jsp:output> element that would direct the container to do otherwise (see Listing 3-1).

Listing 3-1. JPSX Sample

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

            <jsp:directive.page contentType="text/html"/>
            <head>
                     <title>JSPX Example</title>
            </head>
            <body>
                    <h1>JSPX Example</h1>
                     <hr/>
    <p>This example shows how it is now possible to write JSP files as XML
 documents. Note that you no longer have to use &lt;jsp:root&gt; because it
 is now defined as a namespace. You can see how to use JSP directives,
 JSP actions, and JSTL actions.</p>
      <jsp:useBean id="now" class="java.util.Date"/>
         <fmt:formatDate value="${now}" pattern="MMMM d, yyyy, H:mm:ss"/>
         </body>
</html>
```

HTTP

While you're probably familiar with HTTP, you should make sure you understand the relationship between the Hypertext Transfer Protocol (HTTP) and JSP pages. While it isn't a requirement that JSP be based on HTTP, it's by far the most common protocol in JSP development. This means that the same characteristics of HTTP present themselves as implementation issues for page authors over and over again. For example, HTTP is a stateless protocol, which means that the server doesn't store any information about requests. Each request is a separate entity, and the server doesn't recognize multiple requests from the same client as being related. When using a JSP page, it may be necessary to remember the client's state so that you can run the appropriate business logic and create the appropriate response.

The point to understand is that when working with JSP pages, you'll typically be dealing with the semantics and characteristics of HTTP. The HttpJspPage interface is what's used to interact with this protocol.

HttpJspPage Interface

Aside from creating the appropriate response from a request for a given protocol, a JSP page may also indicate how some events are to be handled. In JSP 2.0, the JSP author can affect only the `init` and `destroy` events. The `javax.servlet.jsp` package contains a number of classes and interfaces that describe and define the contracts between a JSP page implementation class and the runtime environment provided for an instance of the class by the container.

Two interfaces, `JspPage` and `HttpJspPage`, define the main contract between the implementation and the container. Neither of these classes is actually used by the page author, but rather the container uses them to generate the appropriate servlet code for the JSP page. `HttpJspPage` extends `JspPage` and is used by most pages that use HTTP. The `HttpJspPage` interface has three methods (which can all be found in the generated servlet code of a JSP page):

public void jspInit(): This method is invoked when the JSP page is initialized. It allows page authors to provide initialization to the JSP page. This method will redefine the `init()` method defined in the `Servlet` class. When this method is called, all the methods in `Servlet`, including `getServletConfig()`, are available.

public void jspDestroy(): The `jspDestroy()` method is invoked when the JSP page is about to be destroyed. A JSP page can override this method by including a declaration element and use it to clean up any resources that the JSP page may have. This method redefines the `destroy()` method defined in the `Servlet` class.

public void _jspService(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response): The `_jspService()` method corresponds to the body of the JSP page and contains the Java code that represents the JSP syntax from the original page. This method is defined automatically by the JSP container and is never defined by the JSP page author.

In most circumstances, it isn't necessary or required for the page author to provide the `jspInit()` and `jspDestroy()` methods. Page authors never define the `_jspService()` method. The container handles defining and implementing this method.

To summarize, the following three major events take place in the page life cycle:

1. The container invoking the `jspInit()` method initializes the page. The `jspInit()` method may be defined by the page author. This method is called only once to initialize the JSP page on the first request. An example of this is as follows:

```
<%!
public void jspInit(){
    String rootPath =  getServletConfig().getServletContext(). ↵
                                getRealPath("/") ;
//Do other interesting things ...
}
%>
```

2. The `_jspService()` method is generated by the container and called to handle each request. The response is produced from within this method and then returned to the container so that it can be passed back to the client.

3. The `jspDestroy()` method is primarily invoked when the JSP page is destroyed; however, a JSP container implementation can call this method at any time to reclaim resources. The page author can define this method if any cleanup is required.

```
<%!
public void jspDestroy(){
    // Do any cleanup in this method that is called
    // by the Servlet container
}
%>
```

Introducing JSP Syntax and Usage

The entire purpose of a JSP page is to take a request and create a response for it. Simple enough. However, to do so, there's just a little bit more to know. The following sections will introduce the syntax and usage of JSP 2.0 files.

Because JSP pages contain a fair amount of HTML, they're very XML-like. By definition, a JSP page is a well-formed, structured document. During the translation phase, the JSP page is exposed as an XML document, called the *XML view*. When this is done, the JSP container will use the XML view as input to an XML parser that can then take advantage of entity declarations and validation.

In the previous versions of the JSP 1.2 specification, there were some standard actions that were available only in JSP syntax, not in XML. In JSP 2.0, these discrepancies have been addressed so that it's possible to compose a JSP page in XML. The advantage of being able to define a JSP page in XML format is that you're now able to reliably use a JSP page anywhere you can use an XML document, such as in a transformation to support various presentation devices. It also makes it easier to use JSP in many of the common XML editing tools. We use the term *JSP document* because the XML file is using the JSP namespace throughout the XML document.

General Rules of Syntax

The following general rules apply to creating well-formed JSP files:

- JSP tags are case sensitive.

- It's possible to use JSP syntax or the XML version for directives and scripting elements.

- Tags can be based on XML syntax.

- Attribute values are always quoted. Similar to XML, either single or double quotes can be used. If you're using XML entity references, you can also use `&apos` and `&quot`.

- JSP pages are typically named with the `.jsp` extension. However, with JSP 2.0 it's possible to have JSP fragments that may not compile into full JSP pages. `.jspf` or `.jsf` are suggested as extensions for those files. JSP files that are delivered as XML documents usually have the `.jspx` extension.

- The `<jsp-property-group>` element of `web.xml` can indicate that some group of files, perhaps not using any of the previous extensions, contains JSP pages. You can also use this property to indicate which ones are delivered as XML documents.

- Whitespace is preserved within the body text of a JSP document when the document is translated into a servlet.

- Uniform resource locators (URLs) that start with a slash (`/`) are considered context-relative to the Web application, and the `ServletContext` provides the base URL. Page-relative URLs that don't start with `/` are considered relative to the current JSP page. This follows the conventions defined in the Servlet 2.4 specification.

JSP Elements

A JSP page has two parts: elements and template data. The elements are the part that the JSP container understands and translates. Anything else contained on a JSP page that the container doesn't translate is considered template data. The following are three types of element categories:

Directives: These configure how the container, when creating the servlet, generates the Java code. Directives are typically used to set global values within a particular JSP page; therefore, they affect the entire page.

Actions: These can be either standard or custom. Standard actions are those well-known tags that affect the runtime behavior of the JSP page, and therefore the response generation. Standard actions are available by default to all JSP pages without declaring the tag library. Custom actions are those created by the page author or a third party. You'll spend Chapter 5 learning how to create custom actions in JSP 2.0.

Scripting elements: These provide the logic that joins the template data and actions together. In JSP 2.0, you can now use the EL to simplify data access from a variety of sources. You can use EL in JSP standard actions, custom actions, and template data.

You'll now look at each one in more detail.

Directives

Directives pass information to the JSP container; therefore, a directive doesn't produce any output to the current output stream of the JSP page. The directives have the following common syntax:

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

While this syntax is both easy to type and concise, you'll note that it isn't compatible with XML. There are equivalent ways to represent a directive in XML formant, but it's slightly different for each directive. We'll show each one as we discuss it.

You can use the following three directives in JSP 2.0:

- The `page` directive
- The `include` directive
- The `taglib` directive

Page Directive

The `page` directive provides a mechanism to define the attributes for the JSP page. In the following example, you're using the `page` directive to set the content type of the page as well as the language that will be used for scripting:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
```

When using the `language` attribute, it applies to the body of the entire translation unit.

> **NOTE** *The* translation unit *is defined as the JSP page and any files included using the* `include` *directive.*

Table 3-1 defines the page directive attributes.

Table 3-1. The page *Directive*

ATTRIBUTE	DESCRIPTION	POSSIBLE VALUES	DEFAULT VALUE
language	Defines the scripting language to be used in scriptlets, expression scriptlets, and declarations.	scriptingLanguage	java.
extends	Provides a mechanism to define a fully qualified class name of the superclass that the container will extend to generate the translated page. It's important to note that using this attribute can eliminate the benefit and some functionality you gain from the container. Using this attribute will restrict your container.	className	Not used.
import	Same meaning as the import statement in Java classes.	ImportList; comma-separated list	Not used.
session	Specifies whether this page participates in the HTTP session. If set to true, then the implicit object session is available for access. This object is of type javax.servlet.http.HttpSession. If set to false, the object isn't available, and attempted access to it will cause an error.	true|false	true.
buffer	Specifies the buffering model for writing content to the output stream. The value of none indicates to write directly through to the ServletResponse using its PrintWriter. When specifying a buffer size, it must be in kilobytes (KB). Output is buffered with a buffer size that's not smaller than the size specified.	none|sizekb	Container implementation dependant. Usually at least 8KB.
autoFlush	When set to true, the output buffer is automatically flushed when the buffer is full. Set to false, this will raise an exception when the buffer is full.	true|false	true.
isThreadSafe	Tells the container whether the page is thread safe with proper synchronization implemented on the page. A value of true indicates that the container may send multiple requests to the page. If the value is false, requests will be queued and sent one at a time.	true|false	true.
info	Used to set an informational string that's incorporated into the page and accessible using the Servlet.getServletInfo() method.	info_text	Not used.

(continued)

Table 3-1. The page *Directive (continued)*

ATTRIBUTE	DESCRIPTION	POSSIBLE VALUES	DEFAULT VALUE
errorPage	Defines a URL to a resource to which any java.lang.Throwable object thrown but not caught by the current page is forwarded for error processing.	error_url	Not used.
isErrorPage	Indicates if this JSP page is intended to be the URL target of another JSP page's errorPage. If set to true, then the implicit script language variable exception is available and references the Throwable object from the offending JSP page. If this value is false, then the exception variable isn't available and will cause a translation error if it's referenced.	true|false	false.
contentType	Defines the character encoding for the JSP page and the Multipurpose Internet Mail Extensions (MIME) type for the response of the JSP page. Values are of the form MIMETYPE or MIMETYPE;charset=CHARSET with optional whitespace after the semicolon (;).	ctinfo	The default value for MIMETYPE is text/html for JSP pages in standard syntax or text/xml for JSP documents in XML syntax. If neither CHARSET nor pageEncoding is specified, the default value for CHARSET is ISO-8859-1 for JSP pages in standard syntax or UTF-8 for JSP pages in XML syntax.
pageEncoding	Defines the character encoding for the JSP page. Value is of the form CHARSET, which must be the Internet Assigned Numbers Authority (IANA) value for a character encoding.	peinfo	The CHARSET value of contentType is used as default if present, or ISO-8859-1 otherwise.
isELIgnored	Defines whether EL expressions are evaluated for this translation unit. If true, EL expressions (of the form ${...}) are evaluated when they appear in template text or action attributes. If false, the container ignores EL expressions.	true|false	For backward compatibility with older JSP containers, if the web.xml is in Servlet 2.3 format, the default is false. If the web.xml is in Servlet 2.4 format, the default is true.

As you can see, the page directive offers the page author a wide range of attributes to use. If you use an undefined attribute, you'll get a translation error from the container. Listing 3-2 shows a JSP page, called pageDirective.jsp, which uses many of the page directive attributes.

Listing 3-2. pageDirective.jsp

```
<%@ page contentType="text/html; charset=UTF-8"
         session="true" buffer="16kb" autoFlush="true"
         isErrorPage="false" isThreadSafe="true"
         info="Page directive sample usage"
         import="java.util.Date"%>

<html>
  <head>
    <title>Page directive sample usage</title>
  </head>
  <body>
    This is a sample of using an imported class from the page directive:
    <br/>
    <%= "The current date is: " + new Date() %>
  </body>
</html>
```

Include Directive

The include directive tells the container to substitute the defined resource content inline to the JSP page. This can be template text or code. This substitution will occur wherever you specify the include directive. The resource specified in the include directive must be a relative URL that's accessible by the container. The syntax of the include directive is as follows:

```
<%@ include file="relativeURL" %>
```

If the included file changes, it doesn't necessarily trigger a recompilation of the JSP file. While some JSP containers may have their own algorithm for notification, if an included file changes, recompilation isn't required in the JSP 2.0 specification. If the container supports this type of functionality, then when an included file is changed, it'll force a recompile of the JSP page and save you the trouble of making sure an older compiled version isn't lying around.

Any elements present in the included resource will also be processed. So, for example, you can include the previous page directive sample in a JSP page using the following syntax:

```
<%@ include file="pageDirective.jsp" %>
```

The page directive and execution of the Date() will be present and processed in the current JSP page.

It isn't a requirement that the `file` attribute be the URL of a JSP page. It can be any resource file that's relative to the current Web application. Using an `include` directive is a common way to template your JSP files. If you have a simple layout, there may be areas that don't change frequently, such as a header or footer that contains a copyright statement. A convenient way to make sure you have to make updates to only one file is to use the `include` directive. For example:

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

The content that's included using the `include` directive is parsed at translation time of the JSP page. You use the `include` action to include resources at run time.

Taglib Directive

The `taglib` directive allows a JSP page to work with custom tag libraries (*taglibs*). The `taglib` directive in a JSP page declares that the page uses a tag library, uniquely identifies the tag library using a URI, and associates a tag prefix with the actions in the library. In JSP 2.0, it's also possible to have tag files in a directory called `/WEB-INF/tags`. You can specify this directory in the `tagdir` attribute to indicate to the container that it should evaluate any tag files that may be present and possibly generate implicit tag library descriptor (TLD) files for them. Chapter 5 covers using tag libraries, creating your own custom actions, using tag files and JSP fragments, using TLDs, and using the libraries within your JSP pages.

The syntax of the `taglib` directive is as follows:

```
<%@ taglib (uri="tagLibraryURI" | tagdir="tagExtensionDir") prefix="tagPrefix" %>
```

Table 3-2 describes the attributes for the `taglib` directive.

Table 3-2. The `taglib` *Directive*

ATTRIBUTE	DESCRIPTION
`uri`	Either an absolute URI or a relative URI specification that uniquely identifies the TLD associated with this prefix.
`tagdir`	Indicates this prefix is to be used to identify tag file extensions installed in the `/WEB-INF/tags` directory or a subdirectory. If a TLD is present in the specified directory, it's used. Otherwise, an implicit tag library is used. Only one of `uri` or `tagdir` may be specified. Otherwise a translation error will occur.
`prefix`	Defines the prefix string that's used to access tags (or actions) defined in the tag library. Empty prefixes are illegal. The following prefixes are reserved: `jsp`, `jspx`, `java`, `javax`, `servlet`, `sun`, and `sunw`.

If the container can't locate a tag library description that's specified in the `uri` attribute, a translation error will be generated. The `taglib` directive needs to appear prior to any actions from that library being used with the defined prefix.

In the following example, you define a taglib with a prefix called `proj2ee`. Any tag files that are found in the `/WEB-INF/tags` directory will be available to you in your JSP page. You can see that there's an action defined called `simpletag`:

```
<%@ taglib prefix="proj2ee" tagdir="/WEB-INF/tags" %>
<proj2ee:simpletag />
```

In the next example, you're accessing a defined tag library using a URI:

```
<%@ taglib prefix="proj2ee" uri="/apress-taglib" %>
<proj2ee:classic />
```

Standard Actions

Actions may affect the current `out` stream and use, modify, and create objects. Standard actions make it easier for the page author to accomplish common tasks. They affect how a JSP page behaves at run time and may act upon the incoming request. A standard action is one that compliant JSP containers will support vs. a custom action that's created by a page author or other party. JSP 2.0 introduces a number of new standard actions, bringing the total number of standard actions to 15. The following sections walk through each one.

<jsp:useBean>

One of the powerful features of JSP pages is that they have access to the Java programming environment. The standard action `<jsp:useBean>` can make a Java object available for use. It's then possible to get and set the properties on the object (a JavaBean). The bean is defined within a given *scope* and is given an ID with a newly declared scripting variable.

It's now possible to disable scripting on JSP pages using the `isScriptingEnabled page` directive attribute. When a `<jsp:useBean>` element is used in a scriptless page, no Java scripting variables are created; instead, an EL variable is created.

The flexibility of the `<jsp:useBean>` action is in the ways that a bean can be described. It's possible to use a variety of attributes, including the class `name`, class and type, bean name and type, or just a type. The exact semantics depend on the attributes given. The `<jsp:useBean>` will try to find an existing object using the `id` and `scope` attributes. If the object isn't found, it'll attempt to create the object using the other attributes. Table 3-3 describes the attributes.

Table 3-3. `<jsp:useBean>` *Attributes*

ATTRIBUTE	DESCRIPTION	POSSIBLE VALUES	DEFAULT VALUE
`id`	The case-sensitive name used to identify the object instance in the specified scope's namespace. The `id` is also the scripting variable name declared and initialized with that object reference.	The `id` name conforms to the current scripting language variable naming conventions.	None
`scope`	The scope in which the reference to the bean is available.	`page`, `request`, `session`, or `application`	`page`
`class`	Case-sensitive fully qualified class name. If the `class` name or `beanName` aren't specified, then the object must be available in the given scope.	Any valid class name	No default value
`beanName`	The name of a bean, as provided to the `instantiate()` method of the `java.beans.Beans` class. This attribute can take a request-time attribute expression as a value. It's possible to supply a `beanName` and `type`, and not include a `class` attribute.	Valid bean name	No default value
`type`	Optional attribute that defines the type of the scripting variable defined. The Java type casting rules are followed for class and superclasses. If the `type` isn't the object class, superclass, or interface implemented by the class, a `java.lang.ClassCastException` is thrown at request time.	Java class	Value of the `class` attribute

We'll now present a `<jsp:useBean>` example, but it makes sense to first cover the `<jsp:getProperty>` and `<jsp:setProperty>` standard actions because it's common to use all three actions together.

<jsp:getProperty>

This action accesses a defined property of a bean. The bean that you're interested in getting the property from must be declared on the JSP page (for example, using `<jsp:useBean>`) prior to using `<jsp:getProperty>`. This action makes the value of the property available so that it can be written to the current `JspWriter`. For example:

```
<p>The course id is
  <jsp:getProperty name="courseBean" property="courseId" />
</p>
```

The property value is converted to a `String` within the `<jsp:getProperty>` action in one of two ways depending on the type of the property. If the property is an object, then the `toString()` method is called. If the property is a primitive, then the `valueOf()` method of the wrapper class is called.

The syntax of `<jsp:getProperty>` is as follows:

```
<jsp:getProperty name="name" property="propertyName" />
```

Table 3-4 describes the attributes available to `<jsp:getProperty>`.

Table 3-4. `<jsp:getProperty>` *Attributes*

ATTRIBUTE	DESCRIPTION	POSSIBLE VALUES	DEFAULT VALUE
`name`	The name of the object instance from which the property is obtained	Valid JavaBean class	None given
`property`	Name of the property to get	Valid defined property	None given

<jsp:setProperty>

This property sets a bean property value. The bean must have been previously defined using the `<jsp:useBean>` standard action. It's possible to set both simple or indexed properties. You can set a bean's property in the following ways using `<jsp:setProperty>`:

- Using a parameter in the request object
- Using a `String` constant value
- Using a value from a request-time attribute

If `<jsp:setProperty>` is being used to set an indexed property, the value must be an array. As a convenient way to set all of the properties in a bean at once, you can use a wildcard. For example:

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

This would take all of the parameters in a request, match their names and types to the defined properties in the bean, and then assign the values to those properties that match. In the previous sample, the appropriate values would be set into the bean properties contained in the `courseBean`. If a parameter in the request has a value of `""`, the corresponding property isn't modified.

Another approach is to set specific values in a bean. Specifying the name of the bean, the property you're interested in setting, and the name of the parameter to use from the request will accomplish this:

```
<jsp:setProperty name="courseBean" property="courseName" param="title" />
```

Under the covers, the <jsp:setProperty> standard action uses Java introspection to determine the properties, their types, if they're simple or indexed, and the getter/setter methods.

The syntax of <jsp:setProperty> is as follows:

```
<jsp:setProperty name="beanName" property_expression />
```

Here the property_expression takes the form of one of the following:

- property="*"
- property="propertyName"
- property="propertyName" param="parameterName"
- property="propertyName" value="propertyValue"

Table 3-5 describes the available attributes.

Table 3-5. <jsp:setProperty> *Attributes*

ATTRIBUTE	DESCRIPTION	POSSIBLE VALUES	DEFAULT VALUE
name	The name of the bean instance.	This value must match the id attribute value used in <jsp:useBean> prior to using a <jsp:setProperty>.	None given
property	Name of the bean property that's being set.	Valid defined property in the bean specified by the name attribute.	None given
param	The name of the request parameter whose value you want to give to a bean property. If you omit param, the request parameter name is assumed to be the same as the bean property name. If the param isn't set in the request object, or if it has the value of "", the <jsp:setProperty> element has no effect.	The name of the request parameter usually comes from a Web form.	None given
value	The value to assign to the given property.	This attribute can accept a request-time attribute expression as a value. An action may not have both param and value attributes.	None given

<jsp:useBean>, <jsp:setProperty>, and <jsp:getProperty> Interaction

You'll now look at a sample interaction between `<jsp:useBean>` and `<jsp:setProperty>`. You define a bean called `courseBean` that holds information about a selected course. You'll notice that you have a setter/getter for only the `courseName` property, not the `courseId` property. `courseName` is the only property you want to have set from the JSP page. The `courseId` will be set by a private method called `setCourseId()`, but will be available for getting.

The important point here is that you can determine what you want to make public to your JSP pages and what you want to handle in your bean code. The `getCourse-Description()` method returns a value to a `<jsp:getProperty>` even though there isn't a property called `courseDescription`. So all in all, this little bean shows a lot of variations of how `<jsp:getProperty>` and `<jsp:setProperty>` interact with a bean. Listing 3-3 shows the code for `courseBean`.

Listing 3-3. `courseBean`

```
package com.apress.proj2ee.chapter03.beans;

public final class CourseBean {
  private String courseName;
  private int courseId;

  public CourseBean(){}

  public String getCourseName() {
    return courseName;
  }

  public void setCourseName(String courseName) {
    this.courseName = courseName;
    setCourseId(courseName);
  }

  public String getCourseDescription(){
     switch (courseId) {
     case 1:
       return("This is an introduction course to J2EE 1.4");
     case 2:
       return("This is an intermediate course on J2EE 1.4");
     case 3:
       return("You'd better be a Java pro for this course");
     default:
       return("Sorry, but I'm not sure what course you are interested in.");
    }
  }
```

```
  public String getCourseId(){
    return(Integer.toString(courseId));
  }

  private void setCourseId(String courseName){
    if (courseName.equalsIgnoreCase("Intro")) {
      courseId=1;
    } else if (courseName.equalsIgnoreCase("Intermediate")) {
      courseId=2;
    } else if (courseName.equalsIgnoreCase("Pro")) {
      courseId=3;
    } else {
      courseId=0;
    }
  }
}
```

You can make this bean (and its functionality) available to your JSP page by using the `<jsp:useBean>` action. You'll notice that you have an embedded `<jsp:setProperty>`. This is because you can instantiate the bean and then have any appropriate values set from the incoming request. The following code demonstrates this (all the code for this page is in `bean.jsp`):

```
<jsp:useBean id="courseBean" scope="page"
             class="com.apress.proj2ee.beans.CourseBean" >
  <jsp:setProperty name="courseBean" property="*" />
</jsp:useBean>
```

Once the bean has been declared on the page, you can access any of the properties you need to get at by using `<jsp:getProperty>`:

```
<h1>Your course selection information is: </h1>
<p>The name of the course is
  <jsp:getProperty name="courseBean" property="courseName" />
</p>
<p>The course id is
  <jsp:getProperty name="courseBean" property="courseId" />
</p>
<p>The description of this course is:
  <jsp:getProperty name="courseBean" property="courseDescription" />
</p>
```

You saw earlier that values for bean properties can come from the request, usually through a form. The parameters of the form need to match the name of the property for the values to be set correctly using this:

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

In this sample, the JSP page that has the form submit is called `selectcourse.jsp`:

```
<html>
  <head>
    <title>Working with JavaBeans in a JSP Page</title>
  </head>
  <body>
    <form method="post" action="bean.jsp" >
      <p>Select the course you are interested in:
        <select name="courseName">
          <option value="Intro">Introduction to J2EE 1.4
          <option value="Intermediate">Intermediate J2EE 1.4
          <option value="Pro">Advanced J2EE 1.4
        </select>
      </p>
      <p><input type="submit" value="Select a course">
    </form>
  </body>
</html>
```

<jsp:param>

You use the `<jsp:param>` action to pass parameter values to other actions. This is done using a name/value pair. Actions that work with the `<jsp:param>` action include `<jsp:include>`, `<jsp:forward>`, and `<jsp:plugin>`. If `<jsp:param>` is used within any other action besides the ones just mentioned, a translation error will occur.

When being used with either `<jsp:include>` or `<jsp:forward>`, the page will see the original request object, with the original parameters as well as the new parameters, but the new values take precedence over any existing values. For example, if the request has a parameter `courseName=Intro`, and a parameter `courseName=Pro` is specified for `<jsp:forward>`, the forwarded request shall have `courseName=Pro,Intro`. Note that the new parameter has precedence. The scope of the new parameters is only that of the `<jsp:include>` or `<jsp:forward>` call.

The syntax of `<jsp:param>` is as follows:

```
<jsp:param name="parameterName" value="parameterValue" />
```

Table 3-6 describes the attribute descriptions.

Table 3-6. `<jsp:param>` *Attributes*

ATTRIBUTE	DESCRIPTION	POSSIBLE VALUES	DEFAULT VALUE
`name`	Name of the parameter	Anything	None given
`value`	Value of the parameter	Any value including a request-time expression	None given

<jsp:include>

This element provides a mechanism for including both static and dynamic resources on a JSP page. The resource must be in the same context as the current page and must be written to the current `JspWriter` (or `out` variable). Since the included page has only access to the output stream, it can't set anything in the response; this includes headers or cookies. If an included page tries to use methods such as `setCookie()`, it will be ignored. This is a similar constraint as imposed by the `include()` method of the servlet `RequestDispatcher` class.

When using the `<jsp:include>` action, it may be necessary to use the `<jsp:param>` action as this is how parameter values are passed to the included page. There's also an attribute called `flush` that determines whether the page buffer is flushed prior to the inclusion. Once the buffer has been flushed, it's no longer possible to forward to another resource in the Web application, so use this with caution. This includes using the error page.

The syntax of `<jsp:include>` without using a parameter is as follows:

```
<jsp:include page="url" flush="true|false"/>
```

If parameters are specified, then each parameter value is defined in one or more `<jsp:param>` actions like so:

```
<jsp:include page="url" flush="true|false">
  { <jsp:param .... /> }*
</jsp:include>
```

Table 3-7 describes the attribute descriptions for this action.

Table 3-7. `<jsp:include>` *Attributes*

ATTRIBUTE	DESCRIPTION	POSSIBLE VALUES	DEFAULT VALUE
`page`	Defines the resource that will be included. The URL is interpreted relative to the current JSP page.	Accepts a request-time attribute value (which must evaluate to a `String` hat's a relative URL specification).	None given
`flush`	Optional Boolean attribute. If `true`, the buffer is flushed prior to the inclusion.	`true|false`	`false`

Since we're talking about including content, you may recall that we've already talked about the `include` directive. Just to refresh your memory, the `include` directive looks like this:

```
<%@ include file="filename" %>
```

So what's the difference between the two? There are a few actually, so we'll go through them now.

You use the `include` directive at compilation time to include static content. The container parses the content; typically a resource included with the `include` directive will be included on multiple JSP pages. This may be something such as a header or footer file. However, if the resource changes, then the including page needs to be recompiled. This is typically a function of how smart your JSP container is and whether it knows time stamps have changed and files need to be recompiled.

When using the `<jsp:include>` action, you can include both static and dynamic content at request time. The content isn't parsed, but it's included inline to the JSP page. The resulting JSP page is then translated.

To summarize when you may want to use the `include` directive vs. `<jsp:include>`, consider the following:

- Use the `include` directive when you have resources you don't think are going to change often.

- Use `<jsp:include>` when the resources require dynamic data or change frequently.

<jsp:expression>, <jsp:scriptlet> , and <jsp:declaration>

You can use three scripting elements in JSP documents. They are `<jsp:expression>`, `<jsp:scriptlet>`, and `<jsp:declaration>`. When used in a JSP document, these elements are valid only in the XML syntaxes. You use the `jsp:declaration` element to declare scripting language constructs that are available to all other scripting elements. There are no attributes associated with the `jsp:declaration` element. The body is the declaration itself. Declarations don't produce any output into the current `out` stream.

Declarations are initialized when the JSP page is initialized and are made available to other declarations, scriptlets, and expressions. An example of a declaration is as follows:

```
<jsp:declaration> Int i </jsp:declaration>
```

The `< jsp:scriptlet>` element describes actions to be performed in response to some request. These actions are defined as *scriptlets*, which are program fragments. A `<jsp:scriptlet>` element has no attributes and its body is the scriptlet itself:

```
<jsp:scriptlet> if (Calendar.getInstance().get(Calendar.AM_PM)
                            == Calendar.AM) {</jsp:scriptlet>
Good Morning
<jsp:scriptlet> } else {</jsp:scriptlet>
Good Afternoon
<jsp:scriptlet> } </jsp:scriptlet>
```

The <jsp:expression> element describes complete expressions in the scripting language that get evaluated at response time. A <jsp:expression> element has no attributes, and its body is the expression. For example:

```
<jsp:expression> application.getAttribute("port") </jsp:expression>
```

<jsp:forward>

A <jsp:forward> allows the runtime dispatch of the current request to a static resource, a JSP page, or a servlet. The URL that's defined must be in the same context as the current page. A <jsp:forward> effectively stops the execution of the current page, making this a server-side redirect. Therefore, the response buffer is cleared, and any changes that are needed are made to the request parameters. The parameters follow the same logic covered in the "<jsp:param>" section.

If anything was written to the output stream that wasn't buffered prior to the <jsp:forward>, a java.lang.IllegalStateException will be thrown. This is similar to the forward() method call of a servlet RequestDispatcher.

<jsp:forward> has a single attribute, page, which is a relative URL. This could be a static value, or it could be computed at request time, as in these two examples:

```
<jsp:forward page="/common/termsofuse.jsp" />
<jsp:forward page="<%= nextPage %>" />
```

<jsp:plugin>

If you have Java applets that are part of your application, using the <jsp:plugin> action provides the support for including a Java applet in a JSP page. While it's possible to have the appropriate HTML code embedded in your JSP page, using <jsp:plugin> allows the functionality to be browser neutral. In other words, you don't have to worry about browser differences. The action will generate the appropriate <object> and <embed> tags with the appropriate attributes based on the configuration defined in the <jsp:plugin> attributes. The JSP container will use the user-agent string to determine what version of the browser is being used.

Also, the following two optional support tags work with <jsp:plugin>:

- <jsp:params> passes any additional parameters to the applet or bean.
- <jsp:fallback> specifies any content that should be displayed in the browser if the plug-in couldn't be started; this could be because the generated <object> and <embed> tags aren't supported or because of some other runtime issue. If the plug-in can start but the applet or JavaBeans component can't be found or started, a plug-in specific message will be presented to the user, most likely a pop-up window, giving a ClassNotFoundException.

If either one of the previously mentioned support tags are used in any other context, other than as a child of <jsp:plugin>, a translation error will occur.

As you may imagine if you've worked with applets before, a whole slew of attributes are available in this action to handle to configuration settings. The syntax of the <jsp:plugin> action is as follows:

```
<jsp:plugin type="bean|applet"
            code="objectCode"
            codebase="objectCodebase"
          { align="alignment"}
          { archive="archiveList"}
          { height="height"}
          { hspace="hspace"}
          { jreversion="jreversion"}
          { name="componentName"}
          { vspace="vspace"}
          {title="title"}
          { width="width"}
          { nspluginurl="url"}
          { iepluginurl="url"}
          { mayscript="true|false" }>
 { <jsp:params>
   { <jsp:param name="paramName" value="paramValue" /> }+
   </jsp:params> }
 { <jsp:fallback> arbitrary_text </jsp:fallback> }
</jsp:plugin>
```

Table 3-8 describes the attributes.

Table 3-8. <jsp:plugin> *Attributes*

ATTRIBUTE	DESCRIPTION	REQUIRED
type	Identifies component type, either Java bean or applet.	Yes
code	Same as the HTML syntax defined in the HTML specification.	Yes
codebase	Same as the HTML syntax.	Yes
align	Same as the HTML syntax.	No
archive	Same as the HTML syntax.	No
height	Same as the HTML syntax, possible to use a runtime expression value.	No
hspace	Same as the HTML syntax.	No
jversion	Identifies the version number of the JRE the component requires to operate. Default to "1.2".	No
name	Same as the HTML syntax.	No
vspace	Same as the HTML syntax.	No
title	Same as the HTML syntax.	No
width	Same as the HTML syntax.	No
nspluginurl	URL where JRE plug-in can be downloaded for Netscape Navigator. The default is implementation defined.	No
iepluginurl	URL where JRE plug-in can be downloaded for IE. The default is implementation defined.	No
mayscript	As defined by HTML specification.	No

Because of the security concern of having a plug-in execute without a user being aware of it, some browsers don't allow an object's height or width to be zero. You'll need to check your browser to see if this is a restriction. If so, even though these attributes aren't required, you'll need to include them to generate code that will work correctly.

<jsp:attribute>

The `<jsp:attribute>` action is new in JSP 2.0 and fits in with the new JSP fragment mechanism. Prior to JSP 2.0, you could pass input to tag handlers in two ways, either through attribute values or through the element body. Attribute values were always evaluated once (if they were specified as an expression), and the result was passed to the tag handler. The body could contain scripting elements and action elements and be evaluated zero or more times on demand by the tag handler.

Using `<jsp:attribute>`, based on the configuration of the action being invoked, the body of the element specifies a value that's evaluated once, or it specifies a JSP fragment, which is in a form that makes it possible for a tag handler to evaluate it as many times as needed.

The `<jsp:attribute>` action allows the page author to define the value of an action's attribute in the body of an XML element. The `<jsp:attribute>` must appear only as a subelement of a standard or custom action. If you try to use it otherwise, a translation error will occur. When being used for custom action invocations, `<jsp:attribute>` can be used for both classic and simple tag handlers.

To get an idea of what we're talking about, you'll now see a simple use of `<jsp:attribute>`. In the following sample, you have a custom action called `text`. This action is using `<jsp:attribute>` to define an attribute called `headingType` with the value of `H1`:

```
<apress:text>
  <jsp:attribute name="headingType">
    H1
  </jsp:attribute>
</apress:text>
```

You can use `<jsp:attribute>` in a number of ways. Each option has different expected behavior. These are the various ways to use `<jsp:attribute>`:

For custom action attributes of type `javax.servlet.jsp.tagext.JspFragment`, the container will create a `JspFragment` out of the body of the `<jsp:attribute>` action and pass it to the tag handler. This applies for both classic and simple tag handlers. A translation error will result if the body of the `<jsp:attribute>` action isn't scriptless in this case.

If the custom action using `<jsp:attribute>` accepts dynamic attributes (which we'll discuss in detail in Chapter 5) and the name of the attribute isn't one explicitly indicated for the tag, then the container will evaluate the body of `<jsp:attribute>` and assign the computed value to the attribute using the dynamic attribute algorithm. Since the type of the attribute is unknown and the body of `<jsp:attribute>` evaluates to a `String`, the container will pass in an instance of `String` to the action.

For standard or custom action attributes that accept a request-time expression value, the container will evaluate the body of the <jsp:attribute> action and use the result of this evaluation as the value of the attribute. The body of the attribute action can be any JSP content in this case. If the type of the attribute isn't String, the standard type conversion rules will be applied.

For standard or custom action attributes that *do not* accept a request-time expression value, the container must use the body of the <jsp:attribute> action as the value of the attribute. A translation error will result if the body of the <jsp:attribute> action contains anything but template text.

If the enclosing action is <jsp:element>, the values of the name attribute and the body of the of the attribute will be used to construct the element dynamically.

If the body of the <jsp:attribute> action is empty, it's the equivalent of specifying "" as the value of the attribute. Note that after being trimmed, nonempty bodies can also result in a value of "". The <jsp:attribute> action accepts a name attribute and a trim attribute.

Table 3-9 describes the attributes.

Table 3-9. <jsp:attribute> *Attributes*

ATTRIBUTE	DESCRIPTION	REQUIRED	DEFAULTS
name	Associates the action with one of the attributes the tag handler is declared to accept.	Yes	None given
trim	Determines whether the whitespace appearing at the beginning and at the end of the element body should be discarded. Whitespace includes spaces, carriage returns, line feeds, and tabs.	No	true

The container will trim at translation time only, not at run time. This means that if you have a custom action that produces whitespace, it won't be affected by the trim attribute, regardless of what the value of trim is.

<jsp:body>

Most page authors are probably used to the body of a standard or custom action invocation being defined implicitly as the body of the XML element of the invocation. You can also define the body of a standard or custom action explicitly using the <jsp:body> standard action. Using <jsp:body> is required if one or more <jsp:attribute> elements appear in the body of the tag. If one or more <jsp:attribute> elements appear in the body of a tag invocation but no <jsp:body> element appears or an empty <jsp:body> element appears, it's the equivalent of the tag having an empty body.

It's also legal to use the <jsp:body> standard action to supply bodies to standard actions for any standard action that accepts a body (except for <jsp:body> and <jsp:attribute>).

For example, in the following code, you have a custom action that includes defining an attribute called nonfragment. You then use the <jsp:body> to indicate the body of the custom action:

```
<proj2ee:simpletag x="10" y="20" z="test">
  <jsp:attribute name="nonfragment">
    Nonfragment Template Text
  </jsp:attribute>
  <jsp:body>
    In body: <br/>
  </jsp:body>
</proj2ee:simpletag>
```

The body standard action accepts no attributes.

<jsp:invoke>

While <jsp:invoke> is defined as a standard action, it's specifically for use in tag files. We'll talk about tag files extensively in Chapter 5, so we won't go into all the details of a tag file here. However, the main point is that a tag file is actually a JSP file used to describe custom actions and uses JSP syntax. If <jsp:invoke> is used in a JSP file that isn't a tag file, a translation error will occur. <jsp:invoke> takes the name of an attribute that's a fragment and invokes the fragment, sending the output of the result to the JspWriter or to a page scope variable that can be used later in the page. If the fragment identified by the given name is null, <jsp:invoke> will behave as though a fragment was passed in that produces no output.

You'll now look at some different ways to use <jsp:invoke> to get a better idea of how it works. The most basic command is to invoke a JSP fragment with the given name specified in the fragment attribute, with no parameters. The fragment will be invoked using the JspFragment.invoke() method. Remember that each JSP fragment is represented as a JspFragment object in the container. Passing in null for the writer parameter means that the results will be sent to the JspWriter of the JspContext associated with the JspFragment. The following is an example of this type of invocation:

```
<jsp:invoke fragment="frag"/>
```

It's also possible to give a fragment access to variables that it may need for its evaluation. JSP fragments have access to the same page scope variables as the page or tag file in which they were defined, as well as to variables in the request, session, and application scopes. Tag files have access to a local page scope, separate from the page scope of the calling page.

When a tag file invokes a fragment that appears in the calling page, the JSP container provides a way to synchronize variables between the local page scope in the tag file and the page scope of the calling page. We'll talk about the way synchronization is handled for variables in tag files in detail in Chapter 5. For now, for each variable that's to be synchronized, the tag file author must declare the variable with a scope of either AT_BEGIN or NESTED. The container will then generate code to synchronize the page scope values for the variable in the tag file with the page scope equivalent in the calling page or tag file.

The following is an example of a tag file providing a fragment access to a variable:

```
<%@ variable name-given="var1" scope="AT_BEGIN" %>
...
<c:set var="var1" value="1"/>
<jsp:invoke fragment="anotherFragment"/>
```

The `<jsp:invoke>` action also provides a way to store output of a fragment. Using the `var` or `varReader` attribute, it's possible to send output to a scoped attribute. To accomplish this, the `JspFragment.invoke()` method uses a custom `java.io.Writer` that's passed in instead of using `null`.

If `var` is specified, a `java.lang.String` object is made available in a scoped attribute with the name specified by `var`. If `varReader` is specified, a `java.io.Reader` object is made available in a scoped attribute with the name specified by `varReader`. Choosing whether you need a `String` or a `Reader` object will depend on what you plan on doing with the output of the fragment.

A `Reader` object can be passed to another custom action for further processing and allows for the action to perform a reset on the `Reader` by calling the `reset()` method. This can be useful if you want the result of the invoked fragment to be read again without reexecuting the fragment. The optional `scope` attribute indicates the scope of the variable specified in `var` or `varReader`. The default is page scope. The following is an example of using `var` or `varReader` and the `scope` attribute:

```
<jsp:invoke fragment="frag2" var="resultString" scope="session"/>
<jsp:invoke fragment="frag3" varReader="resultReader" scope="page"/>
```

Table 3-10 describes the attributes available for the `<jsp:invoke>` action.

<jsp:doBody>

Like `<jsp:invoke>`, the `<jsp:doBody>` action can be used only in tag files. A translation error will occur if the action is used in a JSP page. The `<jsp:doBody>` standard action behaves exactly like `<jsp:invoke>`, except that it operates on the body of the tag instead of on a specific fragment passed as an attribute.

`<jsp:doBody>` invokes the body of the tag, sending the output of the result to a `JspWriter` or to a defined scoped attribute. There's no need for the `fragment` attribute to specify the name for this standard action because it always acts on the tag body. The `var`, `varReader`, and `scope` attributes are all supported with the same semantics as defined for the `<jsp:invoke>` action. Fragments are given access to variables in the same way for `<jsp:doBody>` as they are for `<jsp:invoke>`. If no body was passed to the tag, `<jsp:doBody>` will behave as though a body was passed in that produces no output. The body of a tag is passed to the simple tag handler as a `JspFragment` object. A translation error shall result if the `<jsp:doBody>` action contains a nonempty body. This is an example of using `<jsp:doBody>`:

```
<jsp:body>
bdy of tag that defines an AT_BEGIN scripting variable ${var1}.
</jsp:body>
```

Table 3-11 describes the attributes for `<jsp:doBody>`.

Table 3-10. `<jsp:invoke>` *Attributes*

ATTRIBUTE	DESCRIPTION	REQUIRED	DEFAULTS
`fragment`	The name used to identify the fragment during this tag invocation.	Yes	None given
`var`	The name of a scoped attribute to store the result of the fragment invocation. The result will be stored as a `java.lang.String` object. A translation error will occur if both `var` and `varReader` are specified. If neither `var` nor `varReader` are specified, the result of the fragment goes directly to the `JspWriter`.	No	None given
`varReader`	The name of a scoped attribute to store the result of the fragment invocation. The result will be stored as a `java.io.Reader` object. See the previous `var` description restrictions.	No	None given
`scope`	A valid JSP scope in which to store the resulting variable. A translation error will result if this attribute appears without specifying either the `var` or the `varReader` attribute. A scope of `session` should be used with caution since not all calling pages may be participating in a session. A container must throw an `IllegalStateException` at run time if scope is `session` and the calling page doesn't participate in a session.	No	`page`

Table 3-11. `<jsp:doBody>` *Attributes*

ATTRIBUTE	DESCRIPTION	REQUIRED	DEFAULTS
`var`	The name of a scoped attribute to store the result of the fragment invocation. The result will be stored as a `java.lang.String` object. A translation error will occur if both `var` and `varReader` are specified. If neither `var` nor `varReader` are specified, the result of the fragment goes directly to the `JspWriter`.	No	None given
`varReader`	The name of a scoped attribute to store the result of the fragment invocation. The result will be stored as a `java.io.Reader` object. See the previous `var` description for restrictions.	No	None given
`scope`	A valid JSP scope in which to store the resulting variable. A translation error will result if this attribute appears without specifying either the `var` or `varReader` attribute. A scope of `session` should be used with caution since not all calling pages may be participating in a session. A container must throw an `IllegalStateException` at run time if scope is `session` and the calling page doesn't participate in a session.	No	`page`

<jsp:element>

You use the <jsp:element> action to dynamically define the value of the tag of an XML element. If you want to have a standard, flexible way to do so, <jsp:element> is your action. You can use this action in JSP pages, tag files, and JSP documents. This action has an optional body that can use the <jsp:attribute> and <jsp:body> actions to generate any element. The only mandatory attribute is name, which is a String. The value of the name attribute determines the tag of the XML element that's generated. Custom actions can generate any content, both structured and unstructured.

The syntax of the <jsp:element> action may have a body. Two forms of this action are valid, depending on whether the element has any attributes. In the first form, no attributes are present:

```
<jsp:element name="name">
  optional body
</jsp:element>
```

In the second form, zero or more attributes are requested, using <jsp:attribute> and <jsp:body>, as appropriate:

```
<jsp:element name="name">
  optional_subelement*
</jsp:element>

optional_sublement ::=
<jsp:body>
<jsp:attribute>
```

Table 3-12 describes the only mandatory attribute.

Table 3-12. <jsp:element> *Attribute*

ATTRIBUTE	DESCRIPTION	REQUIRED	DEFAULTS
name	The value of name is that of the element generated. The name can be a QName. There are no constraints on this value, and it's accepted as is.	Yes	None given

The following sample shows how to use <jsp:element> with the optional <jsp:attribute> and <jsp:body> elements. You're accessing some dynamic content from the HTTP request using the param variable:

```
<jsp:element name="course" >
<jsp:attribute name="title">${param.title}</jsp:attribute>
<jsp:body>Testing jsp element body using an EL variable
                  ${param.testValue}</jsp:body>
</jsp:element>
```

The <jsp:text> action generates template text in either a JSP page or a JSP document. <jsp:text> is similar to the XSLT xsl:text element in that it will preserve whitespace as opposed to using an HTML <p>. While this is usually fixed content, it can also generate dynamic content. This is where you need to be careful. The following example is a fragment that could be in either a JSP page or a JSP document:

```
<jsp:text>
  This is some template text without using any expressions
</jsp:text>
```

The next sample shows how an expression may be used within the <jsp:text>:

```
<jsp:text>
  This is content that will display the value of the incoming userId:
  ${request.userId}
</jsp:text>
```

When using this action, some of the functionality may be slightly different depending on whether you're using it in a JSP page or a JSP document. When being used in a JSP document, the content needs to conform to being a well-formed XML document. So if you're using expressions in your content, you need to make sure you use the correct entity reference. For example, using the following:

```
<jsp:text>
  This is illegal in a JSP document: ${request.lastLoginAttempt < 5}
</jsp:text>
```

would cause a translation error, but the following example wouldn't because of the entity reference:

```
<jsp:text>
  This is legal in a JSP document: ${request.lastLoginAttempt lt 5}
</jsp:text>
```

When a <jsp:text> is encountered, the content in the body of the element is passed through to the current out variable. The <jsp:text> action may seem familiar to those who work with XSLT, as it's similar to the <xslt:text> element.

There are no attributes defined for <jsp:text>, but the action can have an optional body, as you've seen in the examples. This action can appear anywhere that template data can appear and may have EL expressions within the body.

<jsp:output>

You can use the <jsp:output> element only in JSP documents and in tag files that are in XML syntax. It's used to modify the XML declaration property of the output of a JSP document or tag file. In JSP 2.0, the XML declaration is the only property that can be modified.

Most of the time you won't need to use <jsp:output> because the default value of the property is defined. Using the omit-xml-declaration attribute, it's possible to prevent the declaration from being generated. The valid values are yes, no, true, and false and will indicate whether an XML declaration is to be inserted at the beginning of the output. The generated XML declaration is of the following form:

```
<? xml version="1.0" encoding="encodingValue" ?>
```

The encodingValue is the character encoding as set by the JSP page.
Table 3-13 describes the attribute available with <jsp:output>.

Table 3-13. <jsp:output> *Attributes*

ATTRIBUTE	DESCRIPTION	REQUIRED	DEFAULTS
omit-xml-declaration	Indicates whether to omit the generation of an XML declaration. Acceptable values are true, yes, false, and no.	No	The default value for a JSP document that has a <jsp:root> element is yes. The default value for JSP documents without a <jsp:root> element is no. The default value for a tag file in XML syntax is always yes.
doctype-root-element	Must be specified if and only if doctype-system is specified, or a translation error must occur. Indicates the name that's to be output in the generated DOCTYPE declaration.	No	—
doctype-system	Specifies that a DOCTYPE declaration is to be generated and gives the value for the system literal.	No	—
doctype-public	Must not be specified unless doctype-system is specified. Gives the value for the public ID for the generated DOCTYPE	No	<!DOCTYPE nameOfRootElement SYSTEM "doctypeSystem">

<jsp:root>

The <jsp:root> element can appear only as the root element in a JSP document or in a tag file in XML syntax. If it's used in a JSP page, a translation error will occur. The <jsp:root> element isn't required by JSP documents and tag files. <jsp:root> has two purposes:

- The first is to indicate that the JSP file is in XML syntax, without having to use configuration group elements or by using the `.jspx` extension.

- The other use of the `<jsp:root>` element is to accommodate the generation of content that isn't a single XML document: either a sequence of XML documents or some non-XML content.

You can use a `<jsp:root>` element to provide zero or more `xmlns` attributes that correspond to namespaces for the standard actions, for custom actions, or for generated template text. Unlike in JSP 1.2, not all tag libraries used within the JSP document need to be introduced on the root; tag libraries can be incorporated as needed inside the document using additional `xmlns` attributes:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:c="http://java.sun.com/jstl/core_rt"
          version="2.0">
```

The `<jsp:root>` element has one mandatory attribute, the version of the JSP specification that the page is using. When `<jsp:root>` is used, the container will, by default, not insert an XML declaration. It's possible to change the default by using the `<jsp:output>` element.

Table 3-14 describes the attribute description for `<jsp:root>`.

Table 3-14. `<jsp:root>` *Attribute*

ATTRIBUTE	DESCRIPTION	REQUIRED	DEFAULTS
`version`	The version of the JSP specification used in this page. Valid values are `1.2` and `2.0`. It's a translation error if the container doesn't support the required version.	Yes	None defined

Available Object Scope

One of the most powerful features of the JSP architecture is that a JSP page can access, create, and modify server-side objects. These objects can then be made visible to actions, EL expressions, and scripting elements. When an object is created, it defines or defaults to a given scope. Some objects are created implicitly by the container, and others can be created explicitly through actions, EL, or scripting.

The scope describes what entities can access the object. For example, if an object is defined to have page scope, then it's visible only for the duration of the current request on that page. Actions can access objects using a name in the `pageContext` object. An object exposed through a scripting variable has a scope within the page. Scripting elements can access some objects directly via a scripting variable. Some implicit objects are visible via scripting variables and EL expressions in any JSP page.

The following are the available object scopes.

Page Scope

Objects with page scope are accessible only within the page where they're created. The object is valid only during the processing of the response, and once the response is sent back to the client or the request is forwarded, the reference is no longer valid. References to objects with page scope are stored in the `pageContext` object and can be accessed using the `getAttribute()` method of the `pageContext` object. This is the default scope given when objects are created with the `<jsp:useBean>` standard action.

Request Scope

Objects with request scope are accessible from pages processing the same request where they were created. Once the request is processed, all references to the object are released. As long as the `HttpRequest` object still exists, even if the request is forwarded to another page, the object is still available. References to objects with request scope are stored in the request object and made available using the `setAttribute()` method of the request implicit object, as shown in the following sample:

```
request.setAttribute(USER_NAME, userId);
```

Session Scope

Objects with session scope are accessible from pages processing requests that are in the same session as the one in which they were created. For an object to be defined in session scope, the page itself must be session aware as defined by its `page` directive. An object is no longer valid once the session becomes invalidated. References to objects with session scope are stored in the session object, as follows:

```
session.setAttribute(USER_NAME, userId);
```

Application Scope

Objects with application scope are accessible from JSP pages that reside in the same application. This creates a global object that's available to pages that are session aware as well as those that aren't. References to objects with application scope are stored in the application object associated with page activation. The application object is an instance of `javax.servlet.ServletContext` that's obtained from the servlet configuration object. When the `ServletContext` is released, application scope variables are no longer accessible.

Note that application scope variables use a single namespace. Therefore, you need to take care that your code is thread safe when accessing objects in application scope. Application scope variables are typically created and populated at application startup and then used in a read-only mode for the remainder of the application.

Implicit Objects

A number of interfaces are provided in the Servlet application programming interface (API). Since a JSP page eventually becomes a servlet, it makes sense that you should have access to some of these objects through the JSP page. Certain classes encapsulate the functionality in which you're interested. Among the classes are `HttpServletRequest`, `HttpServletResponse`, and `HttpSession`. There's no need for the page author to do any extra coding to access these objects. They're made available as standard variables to scripting languages, as well as the expression language, and are automatically available.

Implicit objects are available through the `pageContext` object. The `pageContext` is an object that provides a context to store references to objects used by the page, encapsulates implementation-dependant features, and provides convenience methods. A JSP page implementation class can use a `pageContext` to run unmodified in any compliant JSP container while taking advantage of implementation-specific improvements such as a high-performance `JspWriter`.

Table 3-15 defines the available implicit objects.

Table 3-15. Implicit Objects

VARIABLE NAME	TYPE	SEMANTICS	SCOPE
`request`	Protocol-dependant subtype of `javax.servlet.ServletRequest` (for example, `javax.servlet.http.HttpServletRequest`	The request triggering the service invocation.	`request`
`response`	Protocol-dependant subtype of `javax.servlet.ServletResponse` (for example, `javax.servlet.http.HttpServletResponse`)	The response to the request.	`page`
`pageContext`	`javax.servlet.jsp.PageContext`	The page context for this JSP page.	`page`
`session`	`javax.servlet.http.HttpSession`	The session object created for the requesting client (if any). This variable is valid only for HTTP-based protocols.	`session`
`application`	`javax.servlet.ServletContext`	The servlet context obtained from the servlet configuration object using the `getServletConfig().getContext()` method.	`application`
`out`	`javax.servlet.jsp.JspWriter`	An object that writes into the output stream.	`page`
`config`	`javax.servlet.ServletConfig`	The `ServletConfig` for this JSP page.	`page`
`page`	`java.lang.Object`	The instance of this page's implementation class processing the current request.	`page`

In addition to all of the implicit objects mentioned in Table 3-15, the variable described in Table 3-16 is available to JSP error pages.

Table 3-16. The `exception` *Variable*

VARIABLE NAME	TYPE	SEMANTICS	SCOPE
`exception`	`java.lang.Throwable`	The uncaught `Throwable` that resulted in the error page being invoked	`page`

Scripting Elements

You use scripting elements to insert code, usually Java code, into a JSP page. Scripting elements are commonly used to manipulate objects and perform some computation on an object or runtime value. Scripting elements have been the cause of many a mangled design; Java code is usually needed because the separation of presentation and business logic isn't defined well enough in the application. Therefore, you end up with a big mess of code interlaced with presentation when, in fact, this shouldn't be the case. JSP 2.0 adds the much cleaner and readable EL expressions as an alternative to scripting elements. We'll talk about the EL in detail in "The JSP 2.0 Expression Language."

The following scripting element sections are intentionally short because scripting elements should go the way of the dinosaurs in JSP pages. A good design and skillful page author shouldn't be using scripting elements in code any longer. It's possible to disable scripting elements through the use of the `<scripting-invalid>` element in the `web.xml` deployment descriptor. That way, if any scriptlets are found in a JSP page at translation time, they will cause a translation error. This section is only here because if you're looking at JSP pages that already contain scripting elements, at least you'll understand what you're seeing.

There are three scripting element types: declarations, scriptlets, and expressions. The syntax starts with a `<%` as follows:

```
<%! this is a declaration %>
<% this is a scriptlet %>
<%= this is an expression %>
```

Declarations

Declarations declare variables and methods in the scripting language used in a JSP page. A declaration must be a complete declarative statement according to the syntax of the scripting language being used. It's used to define variables and methods of instance scope. Variables defined in declarations may be accessed from multiple threads, so they must be thread safe. When using a declaration, no output is sent to the current `out` variable. Declarations are initialized when the JSP page is initialized and are made available to other declarations, scriptlets, and expressions.

Scriptlets

A *scriptlet* is a chunk of Java code that's executed at run time. Scripting can do just about anything you can do in Java and therefore can affect the current `out` variable. It's typical that scriptlets are intermingled with template text. This causes a JSP page to be quite difficult to read since each section of scriptlet has to be contained between the `<% scriptlet code %>` syntax. Scriptlets don't have the same thread-safety issue as declarations because the variables and code are local to the `_jspService()` method.

Expressions

An *expression* is a way to send a value of a Java expression to the `out` variable. The expression is evaluated at request-processing time, and the result is converted to a `String`. If the result of the expression is an object, the `toString()` method is called. If the value can't be coerced into a `String` at translation time, then a `ClassCastException` will be raised.

Comments

Before we move on to the JSP 2.0 expression language, we'll talk about how to comment your JSP pages. Two types of comments are available in JSP page.

The first is for the page author to comment upon what the JSP page is doing. In this case, you don't want those comments showing up in the output to the client. A JSP comment is of the following form:

```
<%-- this is a JSP comment that would not show up in the client output --%>
```

The second way provides a mechanism for comments that are intended to be included in the output sent to the client. To generate comments that appear in the response output stream to the requesting client, you use the HTML and XML comment syntax as follows:

```
<!-- comment that the user will see  -->
```

The JSP container treats these comments as uninterpreted template text. If you desire, the generated comment can also have dynamic data by using this expression syntax:

```
<!-- comments <%= some expression %> more comments ... -->
```

The body of the content is completely ignored by the container. Comments are useful for documentation but also are used to comment out some portions of a JSP page. Note that JSP comments don't nest. An alternative way to place a comment in JSP is to use the comment mechanism of the scripting language. For example:

```
<% /** this is another comment ... **/ %>
```

The JSP 2.0 Expression Language

The EL is a powerful feature added into JSP 2.0. Initially it was introduced in the JSTL 1.0 specification, but the expert groups of JSTL 1.0 and JSP 2.0 have been working closely together from the start on the EL features. The goal has always been to incorporate the EL into the JSP specification. More than likely, various J2EE expert groups will use the EL. The upcoming technology of JavaServer Faces is expected to use the same EL. The language semantics are exposed through an API described in the `javax.servlet.jsp.el` package.

The EL gives the page author a much simpler syntax for doing data manipulation. The EL is invoked by using the construct `${expr}`, where `expr` represents some EL expression. It's possible to use the EL in template text as well as attribute values for both standard and custom actions in JSP 2.0. This is slightly different from what was introduced in the JSTL, as only attribute values were supported. The following example shows the EL being used in an attribute:

```
<c:if test="${course.studentsEnrolled > course.maxStudents}">
  The course <c:out value="${course.title}"/> is currently full.
</c:if>
```

Using the `<c:if>` conditional tag (which we'll talk about in Chapter 4; you can use the EL in the `test` attribute to determine if you can still sign up for a course. If the course is full, you can access the `course` object by using the EL and assigning that to the `value` attribute. Anyone who has worked with JSP pages before can certainly appreciate the ease of use and coding simplification possible with the EL.

When using an identifier (such as `course`, for example) with the EL, it's the same as if you had used `PageContext.findAttribute(identifier)`. The identifier itself can reside in any of the known JSP scopes. If the identifier isn't found in any scope, then a `null` value is returned.

The EL was intentionally kept simple. It provides the following:

- Access to variables in the available namespace (which is the `pageContext`)
- Nested properties and accessors to collections
- Relational, logical, and arithmetic operators
- Extensible functions mapping into static methods in Java classes
- A set of implicit objects

The EL makes JSP pages much cleaner and easier to read and (we hope) scriptlet free. The EL provides the encouragement to lose any of those nasty bad scriptlet habits you may have picked up along the way. It also eliminates the need to know Java to write expressions within your pages. By setting the `isScriptingEnabled` attribute used in the

page directive to false as follows, you'll prevent a JSP page from compiling if it contains any scriptlet code. In this case, JSP pages can use EL expressions but are prevented from using Java scriptlets, Java expressions, or Java declaration elements:

```
<%@ page contentType="text/html; charset=UTF-8" isELIgnored="false" %>
```

It's also possible to disable scripting through the <scripting-enabled> JSP configuration element. This provides a way to define a group of pages all at once instead of using the page directive in every page. The web.xml file snippet containing the JSP configuration element is as follows:

```
<jsp-property-group>
  <url-pattern>*.jsp</url-pattern>
  <el-ignored>false</el-ignored>
</jsp-property-group>
```

For the EL evaluations to be carried out by the container, the Web application needs to be packaged with a Servlet 2.4 deployment descriptor. This is defined in the web.xml as follows:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
         version="2.4">
  ...
</web-app>
```

If your pages are packaged with a Servlet 2.3 deployment descriptor, the JSP 2.0 container will not perform any EL evaluation. A Servlet 2.3 deployment descriptor is as follows:

```
<!DOCTYPE webapp
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
         "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
  ...
</web-app>
```

NOTE *If you're expecting EL evaluations to take place and they're being written to the output stream as template text, chances are you need to change your deployment descriptor to be in Servlet 2.4 format.*

You'll now look further at each area of the EL.

Accessing Variables

When a page author accesses data in a JSP, they work with objects and the values they hold. For the most part these objects have been JavaBeans or collections. The EL has two ways to access data structures. The operators are `.` (also called the *dot* operator) and `[]`. Using these operators makes it possible to easily access encapsulated data from objects. You've already seen this access in the previous example by accessing `course.studentsEnrolled`. Using the dot operator is a shortcut for accessing an object's property.

Take a brief look at the previous sample:

```
The course ${course.title} is currently full.
```

You access the value of `title`, which is a property of the `course` object. You use the `[]` `operator` for accessing collections, which includes lists, maps, and arrays. So, for example, you can access the following `Map` object like so:

```
Course Description: <c:out value="${courseDesc[course.courseId]}"/>
```

Being able to index by a value or by a `String` is convenient, depending on what type of collection is being referenced.

Using Accessors and Nested Properties

The EL follows ECMAScript, better known as *JavaScript*, in its use of the `.` and `[]` operators. You can access properties of both ordered and unordered collections using the `[]` syntax. However, you can use numbers only with ordered collections. When working with an unordered collection, you use a string. For instance, the following expression:

```
${course["title"]}
```

is equivalent to this:

```
${course.title}
```

A situation that might require the bracket notion is where a special character that's legal in a property name such as a period (`.`) or hyphen (`-`) is used in the variable name. In this case, you can't use the dot notation and must use `[]`. For example, if the property was instead called `course-title`, you'd access it with `[]` notation using quotes as follows:

```
${course["course-title"]}
```

Relational, Logical, and Arithmetic Operators

The EL provides support for any type of relational, logical, or arithmetic operation you need to perform. All results are Boolean values.

The relational operators include: ==, !=, <, >, <=, >=, eq, ne, lt, gt, le, and ge. The last six operators are made available to avoid having to use entity references in XML syntax. Entity references are sometimes required because if you place a character such as < inside an XML element, the parser will throw an error because it thinks this is the start of a new element. All illegal XML characters have to be replaced by entity references. Table 3-17 shows an example of each of the logical, relational, and arithmetic operators, using either the operator or the entity reference.

Logical operators include those you'd expect: AND, NOT, and OR. They're used to perform Boolean operations of expressions. You can use parentheses within the expression to order the evaluation. For example, the following:

```
${ (7==7 and 6 == 6) && 5 < 6}
```

would evaluate to ${true && true}; therefore, the resulting expression would be true. The && needs to be expressed using the &amp entity reference for each & in an XML document. Personally, we just prefer to use and since we're always up for the least amount of typing.

You can also use arithmetic and prefix operators with the EL. Arithmetic operators consisting of addition (+), subtraction (-), multiplication (*), division (/ or div), and remainder/modulo (% or mod). The empty prefix operator is also provided for testing whether a value is null or empty. For example:

```
<c:if test="${empty course}">
  A course must be selected.
</c:if>
```

Table 3-17 shows how to use each of the operators.

Table 3-17. Operators

OPERATOR	DESCRIPTION	EXAMPLE	EL RESULT
== (eq)	Equals	${7 == 7}	True
!= (ne)	Not equals	${7 ne 7}	False
< (lt)	Less than	${6 lt 7}	True
> (gt)	Greater than	${6 > 7}	False
<= (le)	Less than or equal to	${6 <= 7}	True
>= (ge)	Greater than or equal to	${6 ge 7}	False
and (&&)	Logical AND operation	${7 == 7 and 6 == 6}	True
not (!)	Logical NOT operation	${not 6 == 6}	False
or (||)	Logical OR operation	${7== 7 || 6 == 6}	True

Functions

The EL has the notion of *qualified functions*. This notion reuses the notion of qualification from XML namespaces (and attributes), XSL functions, and JSP custom actions. Functions are mapped to `public static` methods in Java classes that are specified in the TLD. Each tag library may include zero or more static functions that are listed in the TLD. The name given to the function in the TLD is what's exposed to the EL.

A `public static` method in the specified `public` class will implement the function. The function name must be unique in the tag library. If the function isn't declared correctly, or there are multiple functions with the same name, a translation-time error will be generated. You'll now take a look at a sample.

Define a class called `Functions.java` that can contain a `public static` method called `currentDate()`:

```
package com.apress.proj2ee.chapter03;

import java.util.Date;

/**
 * Defines the functions used in the Pro J2EE 1.4 sample tag library.
 *
 * <p>Each function is defined as a static method.</p>
 */
public class Functions {
  public static Date currentDate() {
    return new Date();
  }
}
```

You then add the function definition to the TLD file like so:

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/ ↵
j2ee web-jsptaglibrary_2_0.xsd"
        version="2.0" >
  ...
  <function>
    <description>Get the current date</description>
    <name>currentDate</name>
    <function-class>com.apress.proj2ee.chapter03.Functions</function-class>
    <function-signature>java.util.Date currentDate()</function-signature>
  </function>
  ...
</taglib>
```

You can then access this function on a JSP page as shown in the following example:

```
<%@ taglib prefix="proj2ee" uri="/apress-taglib"%>

<html>
  <head>
    <title>Pro J2EE 1.4 - Using Functions</title>
  </head>
  <body>
    Evaluate the current date by calling the currentDate function:
     ${proj2ee:currentDate()}
  </body>
</html>
```

The value is returned by the method evaluation, in this case, the current date. If the Java method is declared to return `void`, a `null` is returned. If an exception is thrown during the method evaluation, the exception must be wrapped in an `ELException`, and the `ELException` must be thrown. The EL, for any exception that may occur during parsing or expression evaluation, uses an `ELException` exception.

Implicit Objects Available in the EL

Quite a few implicit objects are exposed through the EL. These objects allow for access to any variables that are held in the particular JSP scopes. Objects include `pageScope`, `requestScope`, `sessionScope`, and `applicationScope`. All of these *x*`Scope` objects are `Maps` that map the respective scope attribute names to their values. Using the implicit objects `param` and `paramValues`, it's also possible to access HTTP request parameters. This holds true for request header information as well as for using the implicit objects `header` and `headerValues`.

The `param` and `header` objects are `Maps` that map the parameter or header name to a `String`. This is similar to doing a `ServletRequest.getParameter(String name)` or `ServletRequest.getHeader(String name)`. The `paramValues` and `headerValues` are `Maps` that map parameter and header names to a `String[]` of all values for that parameter or header. Again, this is as if you had made `ServletRequest.getParameterValues(String name)` or `HttpServletRequest.getHeaders(String)` calls.

`initParam` gives access to context initialization parameters, and `cookie` exposes cookies received in the request. The implicit object `pageContext` gives access to all properties associated with the `PageContext` of a JSP page such as the `HttpServletRequest`, `ServletContext`, and `HttpSession` objects and their properties.

You'll now look at a couple of samples of how to use the implicit objects:

- `${pageContext.request.servletPath}` will return the servlet path obtained from the `HttpServletRequest`.

- `${sessionScope.loginId}` will return the session-scoped attribute named `loginId` or `null` if the attribute isn't found.

- `${param.courseId}` will return the `String` value of the `courseId` parameter or `null` if it's not found.

- `${paramValues.courseId}` will return the `String[]` containing all values of the `courseId` parameter or `null` if it's not found.

Table 3-18 summarizes the available implicit objects.

Table 3-18. Implicit Objects

OBJECT NAME	DESCRIPTION
`pageContext`	The `PageContext` object.
`pageScope`	A `Map` that maps page-scoped attribute names to their values.
`requestScope`	A `Map` that maps request-scoped attribute names to their values.
`sessionScope`	A `Map` that maps session-scoped attribute names to their values.
`applicationScope`	A `Map` that maps application-scoped attribute names to their values.
`param`	A `Map` that maps parameter names to a single `String` parameter value (obtained by calling `ServletRequest.getParameter(String name)`).
`paramValues`	A `Map` that maps parameter names to a `String[]` of all values for that parameter (obtained by calling `ServletRequest.getParameterValues(String name)`).
`header`	A `Map` that maps header names to a single `String` header value (obtained by calling `ServletRequest.getHeader(String name)`).
`headerValues`	A `Map` that maps header names to a `String[]` of all values for that header (obtained by calling `HttpServletRequest.getHeaders(String name)`).
`cookie`	A `Map` that maps cookie names to a single `Cookie` object. Cookies are retrieved according to the semantics of `HttpServletRequest.getCookies()`. If the same name is shared by multiple cookies, an implementation must use the first one encountered in the array of `Cookie` objects returned by the `getCookies()` method. Note that the ordering of cookies is currently unspecified in the Servlet specification.
`initParam`	A `Map` that maps context initialization parameter names to their `String` parameter value (obtained by calling `ServletContext.getInitParameter(String name)`).

Automatic Type Conversion

A convenient feature of the EL is its automatic type conversion. A full set of coercion between various object and primitive types is supported; *coercion* means that the page author isn't responsible for converting parameters into the appropriate objects or primitives.

JSP 2.0 defines appropriate conversions and default values that are used by the EL. For example, a `String` parameter from a request will be coerced to the appropriate object or primitive. If you're dealing with item (or object) `A`, the coercion rules apply for each given type. These rules are available in JSP 2.0 specification in the various sections of JSP.2.8. The container implementation does these coercions for you, but it's always a good idea to understand how, and in what order, the rules are being applied.

Default Values

Another valuable feature of the EL is that it supports default values for expressions. This is handy for allowing JSP pages to handle simple errors gracefully instead of throwing a `NullPointerException`. Using default values allows the page author to handle the logic flow better and therefore allows the user to have a better experience. In most cases, you can avoid common error situations. It becomes possible to allow the users to continue without having an error page displayed. The default values are type correct according to what the EL indicates, and they allow the JSP page to recover from what would have been an error.

We'll now provide an example. The expression `${course.title}` evaluates to `null` if there's no title associated with the `course` object. By evaluating to `null`, the specified default value is used without creating a worry about a `NullPointerException` being thrown by the JSP page. The full expression would look like this:

```
<c:set var="title" value="${course.title}" default="No Title Available"/>
Course Title: <c:out value="${title}"/>
```

When doing iterations, if a value isn't found, it'll default to zero. So, for example, when using the `<c:forEach>` tag, you can specify a `begin`, `end`, and `step` attribute. If the value of `begin` uses a parameter that isn't defined, it'll default to zero. So, in the following sample, if the `start` parameter wasn't defined, then you'd start iterating at zero and end after ten iterations:

```
<c:forEach items="${courseCatalog}"
           begin="${param.start}"
           end="{$param.start + 10}">
  ...
</c:forEach>
```

Handling Errors

An error in a JSP page may occur at either translation or request time. The first time a JSP page is accessed in a client request (unless it has been precompiled) the JSP container will translate and compile the target JSP page. If an error occurs during this translation, you'll see an error that's probably familiar to many page authors: error status code 500 Server Error. Usually a stack trace and some error message describes what might be causing the problem. A welcome addition to the JSP 2.0 specification is the mandatory use of the `jsp:id` attribute. This helps identify exactly where an error might have occurred on the page instead of you having to look at the generated Java code and figure out the line number that might correspond to the problem area.

During the processing of client requests, errors can occur in either the body of the JSP page implementation class or in some other code, such as a JavaBean class. Runtime errors that occur follow the Java exception mechanism. Exceptions can be caught and handled in the body of the JSP page if the page author chooses. The `<c:catch>` action discussed in Chapter 4 is just for this purpose.

If an exception isn't caught, the request will be forwarded, along with the exception, to the error page URL. The error page is specified by using the `errorPage` attribute of the `page` directive and specifying a URL. The `errorPage` itself is identified by using the `isErrorPage` attribute of the `page` directive. The exception is accessible by using the implicit object `exception`. In addition to `exception`, an `errorPage` also has the following request attributes available:

- `javax.servlet.error.status_code`
- `javax.servlet.error.request_uri`
- `javax.servlet.error.servlet_name`

By using the `PageContext.getErrorData()` method, it's possible to get an instance of the `javax.servlet.jsp.Error-Data` class. This provides a simple way to access the attributes mentioned previously through the EL. For example, an error page can access the status code using this syntax:

```
${pageContext.errorData.statusCode}
```

The following code shows an error page:

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

<h3>The following error occurred in your application:</h3>
<br/>
<hr/>
<p><font color="red" size="4">
  Exception = ${pageContext.exception.message}"</p>
<p>
  The request URI is: ${pageContext.errorData.requestURI}"
</p>
<p>
  The status code is: ${pageContext.errorData.statusCode}"</p>
<p>
  The servlet name is : ${pageContext.errorData.servletName}"
</font>
<br/>
</p>
```

This is a simple JSP that forces an exception to be thrown. The `errorPage` attribute of the `page` directive defines the URI of the error page:

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

<%
throw new Exception("Force an exception to be thrown");
%>
```

Internationalization of JSP Pages

If you've been building Web applications, chances are by now you've probably been asked to internationalize your application. If not, then surely in the near future you'll be asked to do so. JSP pages on their own don't provide the platform to internationalize an application. It's a combination of J2SE classes, Servlet APIs, and actions provided in tag libraries such as the JSTL that have an entire collection of locale and formatting actions. We'll discuss the international features of the JSTL in Chapter 4. Here we'll talk about what you can address in the JSP specification, which basically is the character encodings.

What Is Character Encoding?

Character encoding is the organization of numeric codes that represent all the meaningful characters of a script system. Each character is stored as a number. When a user enters characters, the user's key presses are converted to character numeric codes; when the characters are displayed on the screen, the character codes are converted to the glyphs of whatever font is being used.

> **NOTE** Character encoding *is matching the binary representation of a character with the printed character based on a table.*

Java has made it much easier to deal with character encoding because, internally, all characters are represented in Unicode. Unicode provides support for every language there is. As of J2SE 1.4, the Unicode 3.0 character set is supported. The problem lies in the character encodings used across the Web. A number of Unicode transformations exist, including UTF-8, UTF-16BE, UTF-16LE, and the familiar ISO-8859, which is better known as Latin-1, an extension of ASCII containing many European characters. If you need to look up a support character encoding, you can find it at `http://www.iana.org/assignments/character-sets/`.

> **NOTE** *The page character encoding is the character encoding in which the JSP page or tag file itself is encoded.*

You can specify page encoding in a couple of different ways:

- Using the JSP configuration element `<page-encoding>` whose URL pattern matches the page.
- With the `pageEncoding` attribute of the `page` directive of the page.

- Using the `charset` value of the `contentType` attribute of the `page` directive. This determines the page character encoding if neither the `<page-encoding>` element nor the `pageEncoding` attribute is provided.

- If none of the previous list is provided, ISO-8859-1 is used as the default character encoding.

For tag files in standard syntax, the page character encoding is determined from the `pageEncoding` attribute of the `<tag>` directive of the tag file or is ISO-8859-1 if the `pageEncoding` attribute isn't specified.

The JSP page also has to worry about the response character encoding if that response is in the form of text. This is managed primarily by the `javax.servlet.ServletResponse` object's `characterEncoding` property. The JSP container determines an initial response character encoding along with the initial content type for a JSP page and calls `ServletResponse.setContentType()` with this information before processing the page.

If you want to explicitly set the initial content type and initial response character encoding, you can use the `contentType` attribute of the page directive. The initial response content type is set to the `TYPE` value of the `contentType` attribute of the `page` directive. If the page doesn't provide this attribute, the initial content type is `text/html` for JSP pages in standard syntax and `text/xml` for JSP documents in XML syntax.

Localization vs. Internationalization

Localization is the process of translating the text that's displayed to the user into their native language. This is slightly different from *internationalization*, which is the act of making sure your application can support multiple locales. A locale represents local customary formatting such as time and date displays, decimal representations, and monetary displays.

When an application is internationalized, it'll display the correct formatting for whichever locale is being used by the client. When an application is localized, it'll display all of the text in that local language. Localization is typically handled through J2SE mechanisms such as property or resource files. It's possible to have all of the internationalization issues as well as property file message lookups based on the appropriate locale handled by custom actions such as those provided in the JSTL. You'll explore those in more detail in Chapter 4.

Debugging

JSP has never been the easiest technology to debug. Primarily this is the case because, as you now know, JSP pages are translated into servlet source code and then compiled. So they become `.java` files. The problem is that what you see in your JSP page and the error that gets generated from the `.java` code rarely (if ever) correspond line for line. This makes for some frustrating guessing at the line number given, the actual JSP source code, and the sometimes cryptic error message that's displayed in your browser.

It gets even worse when we're talking about runtime errors that might include an exception that's thrown from some compiled servlet code or JavaBean. Most of us have

resorted to printing a slew of statements to the output to see what's going on and then carefully removing (or commenting) them from the JSP code. Fun, huh? Well, here are a couple of tips that can assist you in not turning your hair gray when having to debug JSP pages.

If you're debugging a translation-time problem, locate the generated `.java` file that's created by the container. In Tomcat, these files are located in the `work` directory. Depending on how your Tomcat installation is set up, you'll need to navigate the directory structure until you reach the directory for your Web application. It'll be called the same name as your `.war` file. There you'll find all of the `.java` and `.class` files of the JSP pages that have been accessed. Usually you can open the `.java` file in an editor and go to the line number that's displayed in the stack trace in the browser.

In JSP 2.0, the mandatory support of the `jsp:id` attribute makes it much easier to identify where a translation error is coming from without actually going into the `.java` source. If you're working with tag library validator (TLV) files, then you can add a nicely formatted message that indicates the `jsp:id`. The `jsp:id` can be printed along with the error message, and you can look directly at your JSP source to identify the problem. Chapter 5 contains a complete TLV example.

When dealing with runtime debugging, it gets a little trickier. Your best bet is to investigate the various JSP integrated development environment (IDE) debugging tools available. Rather than endorse one product over another, we'll let you evaluate the various features of each one, and the price tag, to see what best fits your needs. Typically you can have a full debugger with call stacks, breakpoints, and watchpoints. Some are free, such as Eclipse (`http://www.eclipse.org/`), and others cost something. However, if you're going to be doing any heavy-duty JSP work, it probably makes sense to see what IDE best suits your needs for debugging. One way or another, you'll probably need one.

Performance

There's a never-ending quest on the part of Web developers to get Web applications to load and execute faster. Performance, in general, is affected by so many variables that sometimes it becomes extremely difficult to qualify what exactly can be used to acquire "faster" performance or, for that matter, what *performance* really means. Many forests have been toppled for the books available on this very subject.

While many issues concerning the Web container affect the performance of Web applications, each vendor's documentation is the best place to find what parameters to use to fine-tune the performance of the container. And it's up to you to make sure you spend some quality time with that documentation. It could very well make the difference between the so-so performance of your application and having it scream.

That said, we'll present a couple of generic items for optimizing JSP performance: buffering and precompiling JSP files.

Buffering

The JSP container buffers data as it's sent from the server to the client. It's up to the page author how this is accomplished so that you can take the needs of your applica-

tion into consideration. You can use the following two attributes of the page directive to determine how buffering is done for the JSP output:

- The buffer attribute defines the buffer size.
- The autoFlush attribute determines how the container should handle flushing the buffer.

They're both shown in the following page directive:

```
<%@ page contentType="text/html; charset=UTF-8"
         session="true" buffer="16kb" autoFlush="true" %>
```

The out variable writes all content and is an instance of a PrintWriter. The initial JspWriter object is associated with the PrintWriter object of the ServletResponse in a way that depends on whether the page is or isn't buffered. If the page isn't 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 and operations such as setContentType() are legal.

Since this flexibility simplifies programming substantially, buffering is the default for JSP pages. If you're managing your buffers, you have to pay a bit more attention to the state of the buffer. If the buffer size you define using the buffer attribute is exceeded, you can flush the buffer or have an exception raised.

Both approaches are valid, and thus both are supported in the JSP technology. The behavior of a page is controlled by the autoFlush attribute, which defaults to true. In general, JSP pages that need to be sure that correct and complete data has been sent to their client may want to set autoFlush to false. On the other hand, JSP pages that send data that's meaningful even when partially constructed may want to set autoFlush to true, such as when the data is sent for immediate display through a browser. Each application will require that you consider their specific needs to determine what's the correct buffering approach.

Headers aren't sent to the client until the first flush method is invoked. Therefore, it's possible to call methods that modify the response header, such as setContentType(), sendRedirect(), or error methods up until the flush method is executed and the headers are sent. After that point, these methods become invalid, as per the Servlet specification, and you'll get a runtime error.

Precompiling

The major performance hit taken by a JSP page happens the first time it's accessed by the container for a client request. At this point, the container will translate the JSP page into its servlet .java file, and then if there are no translation errors, compile that file into a .class file. While this happens the first time the page is accessed, it can still cause performance issues on large sites. One way to avoid this is to precompile the JSP pages and include them in a .war file. The servlet class files of the corresponding JSP are then made available to the container as JSP files by defining them in the web.xml file:

```
<webapp>
  ...
  <servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>hello_jsp.class</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>/hello.jsp</url-pattern>
  </servlet-mapping>
  ...
</webapp>
```

New for JSP 2.0 is a precompilation protocol that's used with an HTTP request. Also, the new version introduces some basic reserved parameter names. The precompilation protocol is related to—but not the same as—the notion of compiling a JSP page into a servlet class. All request parameter names that start with the prefix `jsp_` are reserved by the JSP 2.0 specification and shouldn't be used by an application as a parameter. This may make for some unhappy campers if you have an application that by some unfortunate chance uses `jsp_` for the prefix of some of its HTTP parameters.

A request to a JSP page that has a request parameter with the name `jsp_precompile` is a *precompilation request*. The `jsp_precompile` parameter may have no value or may have the value `true` or `false`. In all cases, the request will not be delivered to the JSP page but will suggest to the container to precompile the JSP page into its implementation class. This will be a container optimization as to whether it pays attention to the request.

Using JSP Best Practices

JSP pages have been around long enough now that most of use have a handle on what we should be doing with them and what we shouldn't. However, sometimes just knowing isn't enough to stop bad habits. The following is a list of best practices that you can keep in mind when working with JSP pages. At least if you go against any of them, you'll have that annoying voice in the back of your head saying that you know better! If you follow most (if not all) of the following best practices, you'll find you have much better JSP pages:

- Eliminate using scriptlets.
- Use the Model-View-Controller (MVC) pattern and frameworks.
- Place business logic in JavaBeans.
- Use existing custom tag libraries.
- Use custom actions.

- Use templates as much as possible.
- Use the correct inclusion mechanism.
- Use common configuration settings when possible.
- Be user-friendly, and use the JSP exception mechanism.
- Be page author–friendly, and use JSP comments.
- Take advantage of validation.
- Make your pages readable.

We'll cover each of these best practices in the following sections.

Eliminate Using Scriptlets

Admit it: At one point or another you've written Java code into your JSP pages. Well, put it behind you now, and move forward. While the temptation might still occur, avoid writing scriptlet code into your JSP pages. Although it might seem like a good idea at the time (kind of like that tattoo you got that Saturday at 3 a.m.), it'll end up being much more work than just doing it correctly from the start. You'll find that as the application grows more complex and more developers become involved, it's much harder to maintain—not to mention that it's a pain to read. No matter how much you try to "pretty up" the format, you'll still have your co-workers cursing you out one day.

Another advantage of not having Java code embedded in the JSP page is that finally it'll become possible for Joe Page Author, who doesn't know a lick of Java, to maintain the presentation page. This, after all, was the intent of JSP in the first place. This will let the Java developers focus on business logic as they implement the behavior behind the custom tags and allow the presentation page authors to use the custom tags just as they use ordinary HTML tags.

Use the MVC Pattern and Frameworks

MVC enables the development of applications that are easier to create, test, maintain, and enhance. Applications with the tiers properly separated are more reusable because the Java components aren't tied to a Web browser and can be used by other parts of the application. This is even more important if you have an application that will be deployed to different devices, not just to Web browsers. If this is the case, then you'll be able to reuse all of your model code and will just have to deal with the appropriate view target.

Quite a few frameworks implement MVC and allow you to plug and play the portions of your application that provide your value-add. Probably the most popular is the Jakarta Struts Framework. You can find out more about Struts at `http://jakarta.apache.org/struts/index.html`. Struts works well with JSP, Velocity Templates, XSLT, and other

presentation frameworks. Other popular frameworks that incorporate best practices that might be worth your while investigating are J2EE Blueprints, JavaServer Faces, and Apache Turbine.

Place Business Logic in JavaBeans

A best practice for programming in general is eliminating redundancy. The more places you have something repeated, the better the chance of having a bug in it. Java code included directly inside a JSP page isn't as readily accessible to other JSP pages. By putting any business logic within a JavaBean, common behavior not only can be used by other JSP pages but also by other portions of the application. For all intents and purposes, a JavaBean is just a Java class that follows a couple of special rules. They're also easily accessible from any JSP page.

Use Existing Custom Tag Libraries

With the introduction of the JSTL, it just doesn't make sense to reinvent the same date format custom tag over and over again. Extremely valuable actions are available within the JSTL that all JSP page authors should use. The JSTL isn't the only tag library available, but it probably will be the only tag library that's optimized by container vendors—well, except for any tag libraries that they provide exclusively for their container.

If you're not concerned with vendor portability, then there probably is a whole set of custom tag libraries that come with your container. There are also custom tag libraries available with the Struts Framework, as well as on the Jakarta Taglibs project located at `http://jakarta.apache.org/taglibs/index.html`. Before deciding to write your own custom actions, check out what's already available. It'll save you time coding and debugging, and it'll allow you to spend more time focusing on your business application.

Use Custom Actions

JSP 2.0 has introduced a number of new features such as tag files and simple tag handlers that make it easy and compelling to create your own custom actions. If you have logic that's specific to your application, sometimes it's necessary to write your own custom actions.

Keep in mind that when writing custom actions they're reusable within JSP files, but not necessary outside of the JSP pages. This is where you need to determine if it makes sense to create a custom action or a JavaBean that can potentially be reused in different areas of your model. A compromise is to try and extract common behaviors or business logic. Utilize JavaBeans or EJBs that perform those common behaviors and call them from the custom action. That way you can take advantage of the best of both worlds.

Template As Much As Possible

Large-scale Web sites frequently have a common look and feel to them. This is a perfect situation to consolidate common layout features and then create a template mechanism for their use. This allows for a common file to control the layout of the application. When it's time to make changes to the layout, you'll have to modify only the one control file. The rest of the pages will automatically reflect the layout change. This is a much quicker way to make changes in your Web application and to enhance the maintainability of your code. A number of template custom tag libraries are specifically for this purpose and are included with such frameworks as Struts. If you're interested in a more advanced and powerful templating framework available in open-source projects, take a further look into Tiles at `http://www.lifl.fr/~dumoulin/tiles/`.

Another variation on this theme is the use of stylesheets. Stylesheets allow for a single file to dictate the appearance of your pages. For example, Cascading Style Sheets (CSS) are commonly used to control such display characteristics as fonts, font sizes, and table layouts. As with templates, stylesheets allow for a one-stop shop for making display control changes. These changes will then be picked up by all of the JSP pages at once and increase the maintainability of your code.

Use the Correct Inclusion Mechanism

The process of templating your layout is usually a good time to identify code that might be common between templates. It's best to eliminate code and layout duplication. If you come across JSP syntax or template text that's used in a number of places, refactor the common portion out of multiple pages and into a single file. Again, having one place to make changes is always better than having to do it in multiple places.

You can use two JSP *include* mechanisms. When refactoring common code, make sure you're using the correct include mechanism to bring that common piece of functionality onto the page. The general rule is that if the content changes frequently, use the `include` action. If the file is primarily static (HTML or template text), use the `include` directive.

Use Common Configuration Settings When Possible

Try to make it a general rule to combine your common configuration settings into one JSP file and then include that file in each of your JSP pages. This is useful for a number of reasons. It makes sure that the `page` directives are set correctly for common items such as content type and for disabling scripting. It also provides a single point of contact for all of the custom tag library prefixes.

There's nothing more annoying (OK, there are probably a few things more annoying, but this is right up there) than having different prefixes defined within a JSP page for the same custom tag library. Usually this is the case because there was more than one page author working on different pages. It's much more consistent to just define all custom tag libraries in one file and then have everyone access the same prefixes throughout the application.

If it's necessary on a particular page to redefine a configuration setting, it's not the end of the world. This should be the exception, though, not the rule.

Be User-Friendly, and Use the JSP Exception Mechanism

How many developers know users who become overwhelmingly happy when they have an exception stack displayed in their browser when they were expecting to see the quarterly profit results from accounting? Not many, probably. Exception stacks provide helpful and useful information for developers but rarely are considered as helpful to users.

With the introduction of the `<c:catch>` action in the JSTL it's possible to catch exceptions that might not be devastating to the user. The page author can determine what the best or appropriate action to take is. Sometimes the user is none the wiser that an exception occurred. This mechanism isn't meant to replace the JSP error page mechanism but instead provide a way for the page author to shield the user from exceptions that aren't really all that bad.

When something really bad happens, you should use the JSP error page mechanism. When using an error page, it's possible to give a more user-friendly message and point the user in a direction that will allow them to continue with their work or contact the appropriate person. You can use error pages to capture the same stack information that would've been displayed to the user in such a way that it's sent directly to the developer or stored in some persistent format. You can use the error page to log any information that would be useful for the developer without the unsightly exception stack being displayed to the user.

Be Page Author–Friendly, and Use JSP Comments

While your stroke of genius might be obvious to you, it's rarely obvious to others. What better way for your co-workers to appreciate your skill than to explain to them exactly what you're doing in your code? To some, adding comments to files seems to be akin to having your teeth pulled. However, it doesn't take that much extra time. If we can do the extra typing, so can you.

Sometimes it helps to have a comment template defined so that at least pages will have a consistent description. It's equally important to comment within JSP code so that it's easy to follow. Since there are multiple ways to comment a JSP page, you should consider which type of comment to use in the page. HTML comments, which are in the form `<!-- comments ... -->`, will be viewable in the compiled JSP page's HTML source code. This means that any user can go to a View ➤ Source from their favorite browser and view your comments, which may not be what you want.

Using JSP comments in the form `<%-- JSP comment --%>` means they can't be viewed as part of the page's source through the browser. Java comments can also occur in JSP pages inside Java scriptlet sections; however, you know you'll never need to use Java comments because you won't be defining scriptlet sections in your pages any longer, correct? If you do, however, they won't be viewable in the browser. Most of the time, you'll want to use JSP comments.

Take Advantage of Validation

In JSP 2.0 it has become much easier to create JSP documents that use well-formed XML syntax. By having your JSP pages in XML format, it makes it easier to perform XML validation on your pages. You can use XML tools to validate the JSP against a specified document type definition (DTD). In addition, using TLV classes is important. By taking advantage of this JSP facility, it makes it much easier for page authors to create pages using custom actions knowing that they're using the actions correctly. Using a TLV prevents the misuse of an action in the translation stage, instead of at run time.

Make Your Pages Readable

We debated whether to include this best practice, thinking that readers might say, "Do you really need to tell me that?" We concluded that the answer is "yes," primarily because of the hundreds (maybe thousands) of JSP pages we've seen that make us think, "Obviously people don't think this is important." We'll be the first to admit that part of this problem might be because of the overuse of scriptlet code in JSP pages. The constant <%%> all over the place gets to be quite annoying. However, even if you take that out of the equation, there are still a couple of general rules that, when followed, can make for an easier read.

For one, make sure you include line spacing between logical sections. Comments preceding the JSP code can also help explain what the reader is about to see. Make sure you close your tags, primarily HTML tags. Raise your hand if you've ever forgotten the `</table>` and had to track it down. It also helps to use some standard indenting so that it's easier to read the structure. By using some common sense or establishing style guidelines for your project, you can save yourself and your fellow developers wasted time trying to read through files that are "glommed" together.

Summary

This chapter has covered a lot of ground. The release of the JSP 2.0 specification has incorporated many useful and exciting features for page authors. This chapter introduced some of these features, including JSP fragments and tag files. We talked about how JSP fits into the MVC model. We then went into details about understanding the page life cycle. You spent much of this chapter getting up to speed on the various JSP directives and actions. You should understand all of the available objects that are at your disposal in the JSP environment and how to use them within JSP pages.

The introduction of the EL is a major improvement toward simplifying JSP pages. We went into detail on the available features in the EL as well as how to use it in your pages. We then covered international issues, debugging, and performance concerns when dealing with JSP. Last, the chapter ended with some best practices so that you can write some excellent JSP pages.