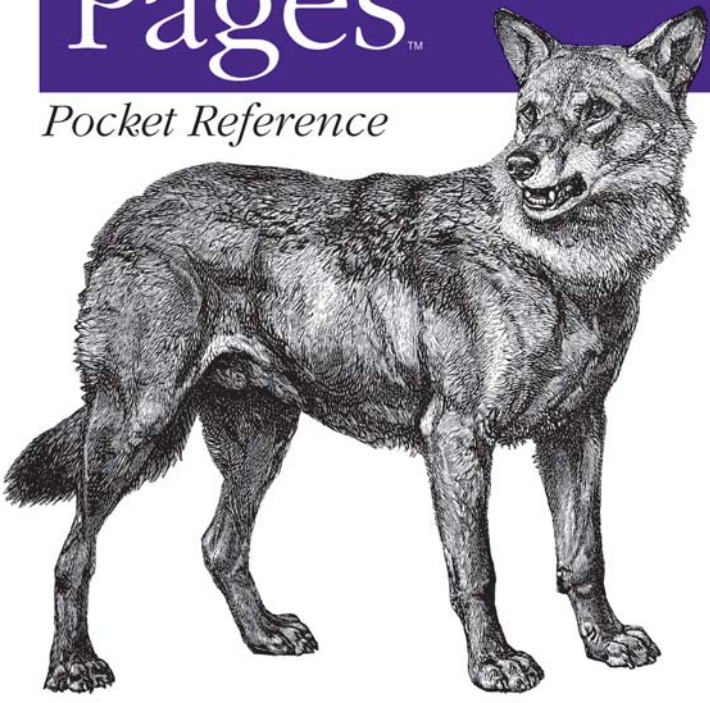
Server-Side Java Development

JavaServer Pages™

Pocket Reference



O'REILLY®

Hans Bergsten

Java/Web Programming

O'REILLY®

JavaServer Pages Pocket Reference

JavaServer Pages (JSP) is harmonizing how web designers and programmers create dynamic web pages. The reason for this is simple: JSP capitalizes on the power of Java servlets to create effective, reusable web applications. JSP allows you to develop robust, powerful web content, and the best part is that you're not required to be a hard-core Java programmer.

JavaServer Pages Pocket Reference is the perfect companion volume to O'Reilly's best-selling *JavaServer Pages*, also by Hans Bergsten. This book provides detailed coverage of JSP syntax and processing, directive elements, standard action elements, scripting elements, implicit objects, custom actions, Tag Library Descriptors, and WAR files.

Hans Bergsten is the founder of Gefion Software, and has been an active participant in the working groups for both the Java servlet and JSP specifications. He also contributes to the Apache Tomcat reference implementation as a member of the Apache Jakarta Project Management Committee.

Visit O'Reilly on the Web at *www.oreilly.com*

ISBN 0-596-00231-9

US $9.95
CAN $14.95

90000

9 780596 002312

6 36920 00231 4

JavaServer Pages™

Pocket Reference

Hans Bergsten

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

JavaServer Pages™ Pocket Reference
by Hans Bergsten

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Editor: Robert Eckstein
Production Editor: Rachel Wheeler
Cover Designer: Pam Spremulli
Interior Designer: Melanie Wang

Printing History:

July 2001: First Edition

0-596-00231-9
[C] [10/01]

Contents

JavaServer Pages Pocket Reference

The JavaServer Pages™ (JSP) specification is built on top of the Java™ servlet specification and is intended to provide for better separation of the presentation (e.g., HTML markup) and business logic (e.g., database operations) parts of web applications. JSP is supported by all major web and application servers. A partial listing of JSP-compliant products is available at Sun Microsystems' JSP web page:

http://java.sun.com/products/jsp/

A JSP page is a web page that contains both static content, such as HTML, and JSP elements for generating the parts that differ with each request, as shown in Figure 1. The default filename extension for a JSP page is *.jsp*.

Everything in the page that's not a JSP element is called *template text*. Template text can be in any format, including HTML, WML, XML, and even plain text. Since HTML is by far the most common web page language in use today, most of the descriptions and examples in this text are HTML-based. You should be aware, though, that JSP has no dependency on HTML. Template text is not interpreted at all; it's passed straight through to the browser. JSP is therefore well-suited to serve any markup language.

When a JSP page request is processed, the static template text and the dynamic content generated by the JSP elements are merged, and the result is sent as the response to the client.

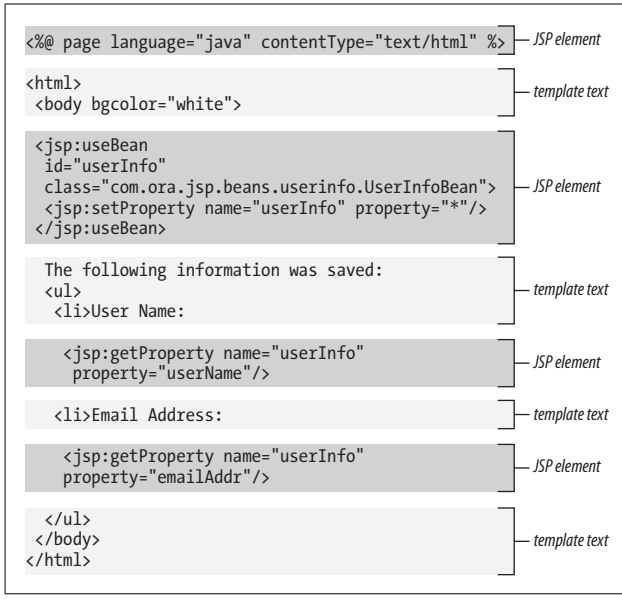


Figure 1. Template text and JSP elements

JSP Processing

Before a JSP page is sent to a browser, the server must process all the JSP elements it contains. This processing is performed by a *web container*, which can be either a native part of a web server or a separate product attached to the web server. The web container turns the JSP page into a Java servlet, then executes the servlet.

Converting the JSP page into a servlet (known as the *JSP page implementation class*) and compiling the servlet take place in the *translation phase*. The web container initiates the translation phase for a JSP page automatically when the first request for the page is received. The translation phase takes a bit of time, of course, so users may notice a slight delay the first time they request a JSP page. The translation phase can also

be initiated explicitly, to avoid hitting the first user with the delay. This is referred to as *precompilation*.

The web container is also responsible for invoking the JSP page implementation class to process each request and generate responses. This is called the *request processing phase*. The two phases are illustrated in Figure 2.

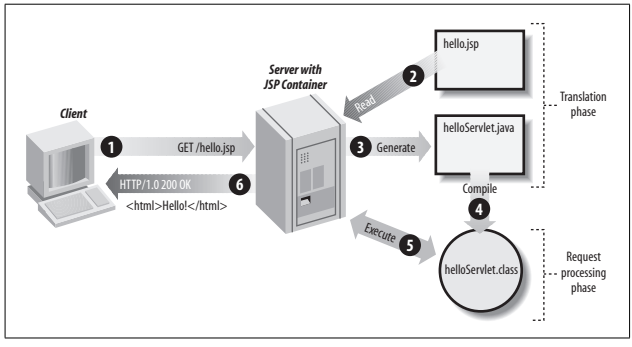


Figure 2. JSP page translation and processing phases

As long as the JSP page remains unchanged, the translation phase is skipped. When the page is modified, it goes through the translation phase again.

Let's look at a simple example. In the tradition of programming books, we start with an application that writes "Hello World" (with a twist—it also shows the current time on the server):

```
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <h1>Hello World</h1>
    It's <%= new java.util.Date().toString() %> and all
    is well.
  </body>
</html>
```

This JSP page produces the result shown in Figure 3.

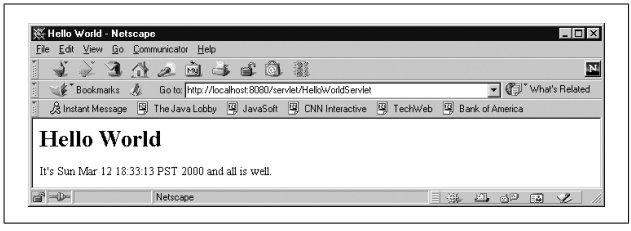


Figure 3. The output from the Hello World page

This is as simple as it gets. The code represented by the JSP element (which we have highlighted in bold in the code) is executed, and the result is combined with the regular HTML in the page. In this case the JSP element is a scripting element with Java code for writing the current date and time.

There are three types of JSP elements: directives, actions, and scripting elements. The following sections describe the elements of each type.

Directive Elements

Directive elements specify information about the page itself; information that doesn't differ between requests for the page. Examples are the scripting language used in the page, whether or not session tracking is required, and the name of the page that will be used to report any errors.

The general directive syntax is:

```
<%@ directiveName attr1="value1" attr2="value2" %>
```

You can use single quotes instead of double quotes around the attribute values. The directive name and all attribute names are case-sensitive.

Include Directive

The `include` directive includes a file, merging its content with the including page before the combined result is converted to

a JSP page implementation class. It supports the attribute described in Table 1.

Table 1. Attributes for the include directive

Name	Default	Description
`file`	No default	A page- or context-relative URI path for the file to include.

A single page can contain multiple `include` directives. Together, the including page and all included pages form a *JSP translation unit*.

Example:

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

Page Directive

The page directive defines page-dependent attributes, such as scripting language, error page, and buffering requirements. It supports the attributes described in Table 2.

Table 2. Attributes for the page directive

Name	Default	Description
`autoFlush`	`true`	Set to `true` if the page buffer should be flushed automatically when it's full or to `false` if an exception should be thrown when it's full.
`buffer`	`8kb`	Specifies the buffer size for the page. The value must be expressed as the size in kilobytes followed by `kb`, or be the keyword `none` to disable buffering.
`contentType`	`text/html`	The MIME type for the response generated by the page, and optionally the charset for the source page (e.g., `text/html;charset=Shift_JIS`).
`errorPage`	No default	A page- or context-relative URI path to which the JSP page will forward users if an exception is thrown by code in the page.

Table 2. Attributes for the page directive (continued)

Name	Default	Description
`extends`	No default	The fully qualified name of a Java class that the generated JSP page implementation class extends. The class must implement the `JspPage` or `HttpJspPage` interface in the `javax.servlet.jsp` package. Note that the recommendation is to *not* use this attribute. Specifying your own superclass restricts the web container's ability to provide a specialized, high-performance superclass.
`import`	No default	A Java import declaration; i.e., a comma-separated list of fully qualified class names or package names followed by `.*` (for all public classes in the package).
`info`	No default	Text that a web container may use to describe the page in its administration user interface.
`isErrorPage`	`false`	Set to `true` for a page that is used as an error page, to make the implicit exception variable available to scripting elements. Use `false` for regular JSP pages.
`isThreadSafe`	`true`	Set to `true` if the container is allowed to run multiple threads through the page (i.e., let the page serve parallel requests). If set to `false`, the container serializes all requests for the page. It may also use a pool of JSP page implementation class instances to serve more than one request at a time. The recommendation is to always use `true` and to handle multithread issues by avoiding JSP declarations and ensuring that all objects used by the page are thread-safe.
`language`	`java`	The scripting language used in the page.
`session`	`true`	Set to `true` if the page should participate in a user session. If set to `false`, the implicit `session` variable is not available to scripting elements in the page.

A JSP translation unit (the source file and any files included via the `include` directive) can contain more than one page

directive as long as each attribute, with the exception of the `import` attribute, occurs no more than once. If multiple `import` attribute values are used, they are combined into one list of `import` definitions.

Example:

```
<%@ page language="java"
  contentType="text/html;charset=Shift_JIS"%>
<%@ page import="java.util.*, java.text.*" %>
<%@ page import="java.sql.Date" %>
```

Taglib Directive

The `taglib` directive declares a tag library, containing custom actions, that is used in the page. It supports the attributes described in Table 3.

Table 3. Attributes for the taglib directive

Name	Default	Description
`prefix`	No default	Mandatory. The prefix to use in the action element names for all actions in the library.
`uri`	No default	Mandatory. Either a symbolic name for the tag library defined in the application's *web.xml* file, or a page- or context-relative URI path for the library's TLD file or JAR file.

Example:

```
<%@ taglib uri="/orataglib" prefix="ora" %>
```

Standard Action Elements

Actions are executed when a client requests a JSP page. They are inserted in a page using XML element syntax and perform such functions as input validation, database access, or passing control to another page. The JSP specification defines a few standard action elements, described in this section, and includes a framework for developing custom action elements.

An action element consists of a start tag (optionally with attributes), a body, and an end tag. Other elements can be nested in the body. Here's an example:

```
<jsp:forward page="nextPage.jsp">
  <jsp:param name="aParam" value="aValue" />
</jsp:forward>
```

If the action element doesn't have a body, you can use a shorthand notation in which the start tag ends with `/>` instead of `>`, as shown by the `<jsp:param>` action in this example. The action element name and attribute names are case-sensitive.

Action elements, or *tags*, are grouped into *tag libraries*. The action name is composed of two parts, a library prefix and the name of the action within the library, separated by a colon (e.g., `jsp:useBean`). All actions in the JSP standard library use the prefix `jsp`, while custom actions can use any prefix except `jsp`, `jspx`, `java`, `javax`, `servlet`, `sun`, or `sunw`, as specified per page by the `taglib` directive.

Some action attributes accept a *request-time attribute value*, using the JSP expression syntax:

```
<% String headerPage = currentTemplateDir +
  "/header.jsp"; %>
<jsp:include page="<%= headerPage %>" flush="true" />
```

Here the `page` attribute value is assigned to the value held by the scripting variable `headerPage` at request time. You can use any valid Java expression that evaluates to the type of the attribute.

The attribute descriptions for each action in this section define whether a request-time attribute value is accepted or not.

<jsp:fallback>

You can use the `<jsp:fallback>` action only in the body of a `<jsp:plugin>` action. Its body specifies the template text to use for browsers that do not support the HTML `<embed>` or `<object>` elements. This action supports no attributes.

Example:

```
<jsp:plugin type="applet" code="Clock2.class"
  codebase="applet"
  jreversion="1.2" width="160" height="150" >
  <jsp:fallback>
    Plug-in tag OBJECT or EMBED not supported by browser.
  </jsp:fallback>
</jsp:plugin>
```

<jsp:forward>

The `<jsp:forward>` action passes the request-processing control to another JSP page or servlet in the same web application. The execution of the current page is terminated, giving the target resource full control over the request.

When the `<jsp:forward>` action is executed, the buffer is cleared of any response content. If the response has already been committed (i.e., partly sent to the browser), the forwarding fails with an `IllegalStateException`.

The action adjusts the URI path information available through the implicit `request` object to reflect the URI path information for the target resource. All other request information is left untouched, so the target resource has access to all the original parameters and headers passed with the request. Additional parameters can be passed to the target resource through `<jsp:param>` elements in the `<jsp:forward>` element's body.

The `<jsp:forward>` action supports the attribute described in Table 4.

Table 4. Attributes for <jsp:forward>

Name	Java type	Request-time value accepted	Description
page	`String`	yes	Mandatory. A page- or context-relative URI path to which the resource will forward users.

Example:

```
<jsp:forward page="list.jsp" />
```

<jsp:getProperty>

The `<jsp:getProperty>` action adds the value of a bean property, converted to a `String`, to the response generated by the page. It supports the attributes described in Table 5.

Table 5. Attributes for <jsp:getProperty>

Name	Java type	Request-time value accepted	Description
`name`	`String`	no	Mandatory. The name assigned to a bean in one of the JSP scopes.
`property`	`String`	no	Mandatory. The name of the bean's property to include in the page.

Example:

```
<jsp:getProperty name="clock" property="hours" />
```

<jsp:include>

The `<jsp:include>` action includes the response from another JSP page, servlet, or static file in the same web application. The execution of the current page continues after including the response generated by the target resource.

When the `<jsp:include>` action is executed, the buffer is flushed of any response content. Although the `flush` attribute can control this behavior, the only valid value in JSP 1.1 is `true`. This limitation will likely be lifted in a future version of JSP.

Even in the target resource, the URI path information available through the implicit `request` object reflects the URI path information for the source JSP page. All other request information is also left untouched, so the target resource has access to all the original parameters and headers passed with

the request. Additional parameters can be passed to the target resource through <jsp:param> elements in the <jsp:include> element's body.

The <jsp:include> action supports the attributes described in Table 6.

Table 6. Attributes for <jsp:include>

Name	Java type	Request-time value accepted	Description
page	String	yes	Mandatory. A page- or context-relative URI path for the resource to include.
flush	String	no	Mandatory in JSP 1.1, with true as the only accepted value.

Example:

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

<jsp:param>

You can use the <jsp:param> action in the body of a <jsp:forward> or <jsp:include> action to specify additional request parameters for the target resource, as well as in the body of a <jsp:params> action to specify applet parameters. It supports the attributes described in Table 7.

Table 7. Attributes for <jsp:param>

Name	Java type	Request-time value accepted	Description
name	String	no	Mandatory. The parameter name.
value	String	yes	Mandatory. The parameter value.

Example:

```
<jsp:include page="navigation.jsp">
  <jsp:param name="bgColor" value="<%= currentBGColor %>"
    />
</jsp:include>
```

<jsp:params>

You can use the <jsp:params> action only in the body of a <jsp:plugin> action, to enclose a set of <jsp:param> actions that specify applet parameters. This action supports no attributes.

Example:

```
<jsp:plugin type="applet" code="Clock2.class"
  codebase="applet"
  jreversion="1.2" width="160" height="150" >
  <jsp:params>
    <jsp:param name="bgcolor" value="ccddff" />
  </jsp:params>
</jsp:plugin>
```

<jsp:plugin>

The <jsp:plugin> action generates HTML <embed> or <object> elements (depending on the browser type) that result in the download of the Java Plug-in software (if required) and subsequent execution of the specified Java applet or JavaBeans™ component. The body of the action can contain a <jsp:params> element to specify applet parameters and a <jsp:fallback> element to specify the text that will be shown in browsers that do not support the <embed> or <object> HTML elements. For more information about the Java Plug-in, see *http://java.sun.com/products/plugin/*.

The <jsp:plugin> action supports the attributes described in Table 8.

Table 8. Attributes for <jsp:plugin>

Name	Java type	Request-time value accepted	Description
align	String	no	Optional. The alignment of the applet area, one of bottom, middle, or top.

Table 8. Attributes for <jsp:plugin> (continued)

Name	Java type	Request-time value accepted	Description
archive	String	no	Optional. A comma-separated list of URIs for archives containing classes and other resources that will be "preloaded." The classes are loaded using an instance of an AppletClassLoader with the given codebase. Relative URIs for archives are interpreted with respect to the applet's codebase.
code	String	no	Mandatory. The fully qualified class name for the object.
codebase	String	no	Mandatory. The relative URL for the directory that contains the class file. According to the HTML 4.0 specification, the directory must be a subdirectory of the directory containing the page.
height	String	no	Optional. The height of the applet area, in pixels or percentage.
hspace	String	no	Optional. The amount of whitespace to be inserted to the left and right of the applet area, in pixels.
iepluginurl	String	no	Optional. The URL for the location of the Internet Explorer Java Plug-in. The default is implementation-dependent.
jreversion	String	no	Optional. The specification version number of the JRE the component requires in order to operate. The default is 1.1.

Table 8. Attributes for <jsp:plugin> (continued)

Name	Java type	Request-time value accepted	Description
name	String	no	Optional. The applet name, used by other applets on the same page that need to communicate with it.
nspluginurl	String	no	Optional. The URL for the location of the Netscape Java Plug-in. The default is implementation-dependent.
title	String	no	Optional. The text to be rendered in some way by the browser for the applet (e.g., as a "tool tip").
type	String	no	Mandatory. The type of object to embed, one of `applet` or `bean`.
vspace	String	no	Optional. The amount of whitespace to be inserted above and below the applet area, in pixels.
width	String	no	Optional. The width of the applet area, in pixels or percentage.

Example:

```
<jsp:plugin type="applet" code="Clock2.class"
  codebase="applet"
  jreversion="1.2" width="160" height="150" >
  <jsp:params>
    <jsp:param name="bgcolor" value="ccddff" />
  </jsp:params>
  <jsp:fallback>
    Plug-in tag OBJECT or EMBED not supported by
    browser.
  </jsp:fallback>
</jsp:plugin>
```

<jsp:setProperty>

The `<jsp:setProperty>` action sets the value of one or more bean properties. It supports the attributes described in Table 9.

Table 9. Attributes for <jsp:setProperty>

Name	Java type	Request-time value accepted	Description
`name`	`String`	no	Mandatory. The name assigned to a bean in one of the JSP scopes.
`property`	`String`	no	Mandatory. The name of the bean property to set, or an asterisk (*) to set all properties with names matching the request parameters.
`param`	`String`	no	Optional. The name of a request parameter that holds the value to use for the specified property. If omitted, the parameter name and property name must be the same.
`value`	See below	yes	Optional. An explicit value to assign to the property. This attribute cannot be combined with the `param` attribute.

The property type can be any valid Java type, including primitive types and arrays (i.e., an indexed property). If the `value` attribute specifies a runtime attribute value, the type of the expression must match the property's type.

If the value is a string, either in the form of a request parameter value or explicitly specified by the `value` attribute, it is converted to the property's type as described in Table 10.

Table 10. Conversion of string value to property type

Property type	Conversion method
`boolean` or `Boolean`	`Boolean.valueOf(String)`
`byte` or `Byte`	`Byte.valueOf(String)`

Table 10. Conversion of string value to property type (continued)

Property type	Conversion method
`char` or `Character`	`String.charAt(int)`
`double` or `Double`	`Double.valueOf(String)`
`float` or `Float`	`Float.valueOf(String)`
`int` or `Integer`	`Integer.valueOf(String)`
`long` or `Long`	`Long.valueOf(String)`

Example:

```
<jsp:setProperty name="user" property="*" />
<jsp:setProperty name="user" property="modDate"
  value="<%= new java.util.Date() %>" />
```

<jsp:useBean>

The `<jsp:useBean>` action associates a Java bean with a name in one of the JSP scopes and makes it available as a scripting variable. An attempt is first made to find a bean with the specified name in the specified scope. If it's not found, a new instance of the specified class is created.

The `<jsp:useBean>` action supports the attributes described in Table 11.

Table 11. Attributes for <jsp:useBean>

Name	Java type	Request-time value accepted	Description
`beanName`	`String`	yes	Optional. The name of the bean, as expected by the `instantiate()` method of the `Beans` class in the `java.beans` package.
`class`	`String`	no	Optional. The fully qualified class name for the bean.
`id`	`String`	no	Mandatory. The name to assign to the bean in the specified scope and the name of the scripting variable.

Table 11. Attributes for <jsp:useBean> (continued)

Name	Java type	Request-time value accepted	Description
`scope`	`String`	no	Optional. The scope for the bean: one of `page`, `request`, `session`, or `application`. The default is `page`.
`type`	`String`	no	Optional. The fully qualified type name for the bean (i.e., a superclass or an interface implemented by the bean's class).

Of the optional attributes, at least one of `class` or `type` must be specified. If both are specified, `class` must be assignable to `type`. The `beanName` attribute must be combined with the `type` attribute and is not valid with the `class` attribute.

The action is processed in these steps:

1. Attempt to locate an object based on the `id` and `scope` attribute values.
2. Define a scripting language variable with the given `id` of the specified `type` or `class`.
3. If the object is found, initialize the variable's value with a reference to the located object, cast to the specified `type`. This completes the processing of the action. If the action element has a nonempty body, it is ignored.
4. If the object is not found in the specified scope and neither `class` nor `beanName` is specified, an `InstantiationException` is thrown. This completes the processing of the action.
5. If the object is not found in the specified scope and the `class` attribute specifies a nonabstract class with a public no-args constructor, a new instance of the class is created and associated with the scripting variable and the specified name in the specified scope. After this, step 7 is performed.

If the object is not found and the specified class doesn't fulfill the requirements, an `InstantiationException` is thrown. This completes the processing of the action.

6. If the object is not found in the specified scope and the `beanName` attribute is specified, the `instantiate()` method of the `java.beans.Beans` class is invoked with the `ClassLoader` of the JSP implementation class instance and the `beanName` as arguments. If the method succeeds, the new object reference is associated with the scripting variable and the specified name in the specified scope. After this, step 7 is performed.
7. If the action element has a nonempty body, the body is processed. The scripting variable is initialized and available within the scope of the body. The text of the body is treated as elsewhere: if there is template text, it is passed through to the response; scriptlets and action tags are evaluated.

   A nonempty body is commonly used to complete initialization of the created instance. In such a case, the body typically contains `<jsp:setProperty>` actions and scriptlets. This completes the processing of the action.

Example:

```
<jsp:useBean id="clock" class="java.util.Date" />
```

Comments

You can use JSP comments in JSP pages to describe what a scripting element or action is doing:

```
<%-- This is a comment --%>
```

All text between the start and stop tags is ignored by the web container and not included in the response. The comment text can be anything except the character sequence representing the closing tag: `--%>`.

Besides describing what's going on in the JSP page, comments can be used to "comment out" portions of the JSP page (for instance, during testing):

```
<jsp:useBean id="user" class="com.mycompany.UserBean" />
<%--
<jsp:setProperty name="user" property="*" />
<jsp:setProperty name="user" property="modDate"
  value="<%= new java.util.Date() %>" />
<% boolean isValid = user.isValid(); %>
--%>
```

The action and scripting elements within the comment are not executed.

Escape Characters

Since certain character sequences represent start and stop tags, you sometimes need to escape a character so the container doesn't interpret it as part of a special character sequence.

In a scripting element, if you need to use the characters %> literally, you must escape the greater-than character with a backslash:

```
<% String msg = "Literal %\> must be escaped"; %>
```

To avoid the character sequence <% in template text being interpreted as the start of a scripting element, you must escape the percent sign:

```
This is template text, and <\% is not a start of a
scriptlet.
```

In an attribute value, you must use the following escapes:

```
attr='a value with an escaped \' single quote'
attr="a value with an escaped \" double quote"
attr="a value with an escaped \\ backslash"
attr="a value with an escaped %\> scripting end tag"
attr="a value with an escaped <\% scripting start tag"
```

Scripting Elements

Scripting elements let you add small pieces of code to a JSP page, such as an `if` statement to generate different HTML depending on some condition. Like actions, they are executed when the page is requested. You should use scripting elements with extreme care; if you embed too much code in your JSP pages you will end up with an application that's very hard to maintain. In addition, simple code syntax errors in scripting elements often lead to error messages that are much harder to interpret than error messages for syntax errors in action elements.

Scriptlets

A scriptlet is a block of code enclosed between a scriptlet-start identifier, `<%`, and an end identifier, `%>`:

```
<%@ page language="java" contentType="text/html" %>
<html>
<body bgcolor="white">
<jsp:useBean id="clock" class="java.util.Date" />

<% if (clock.getHours() < 12) { %>
  Good morning!
<% } else if (clock.getHours() < 17) { %>
  Good day!
<% } else { %>
  Good evening!
<% } %>

</body>
</html>
```

Here, a `clock` bean is first created by the `<jsp:useBean>` action and assigned to a variable with the same name. It is then used in four scriptlets, together forming a complete Java `if` statement with template text in the `if` and `else` blocks:

```
<% if (clock.getHours() < 12) { %>
```

An `if` statement, testing if it's before noon, with a block start brace

`<% } else if (clock.getHours() < 17) { %>`
: The `if` block end brace and an `else-if` statement, testing if it's before 5 P.M., with its block start brace

`<% } else { %>`
: The `else-if` block end brace and a final `else` block start brace, handling the case in which it's after 5 P.M.

`<% } %>`
: The `else` block end brace

The web container combines the code segment in the four scriptlets with code for writing the template text to the response body. The end result is that when the first `if` statement is `true`, "Good morning!" is displayed, and when the second `if` statement is `true`, "Good day!" is displayed. If neither `if` statement is `true`, the final `else` block is used, displaying "Good evening!"

The tricky part when using scriptlets is making sure to get all the start and end braces in place. If you miss just one of the braces, the code the web container generates is not syntactically correct. And, unfortunately, the error message you get is not always easy to interpret.

Expressions

An expression starts with `<%=` and ends with `%>`. Note that the only syntax difference compared to a scriptlet is the equals sign (`=`) in the start identifier. An example is:

```
<%= userInfo.getUserName() %>
```

The result of the expression is written to the response body. Note that unlike statements in a scriptlet, the code in an expression must not end with a semicolon. This is because the web container combines the expression code with code for writing the result to the response body. If the expression ends with a semicolon, the combined code will not be syntactically correct.

In the previous examples using JSP action elements, the attributes were set to literal string values. But in many cases, the value of an attribute is not known when you write the JSP page; the value must instead be calculated when the JSP page is requested. As we mentioned before, for situations like this you can use a JSP expression as a request-time attribute value. Here is an example of how you can use this method to set an attribute of a fictitious log entry bean:

```
<jsp:useBean id="logEntry" class="com.foo.LogEntryBean" />
<jsp:setProperty name="logEntry" property="entryTime"
  value="<%= new java.util.Date() %>" />
...
```

This bean has a property named `entryTime` that holds a timestamp for a log entry, while other properties hold the information to be logged. To set the timestamp to the time when the JSP page is requested, a `<jsp:setProperty>` action with a request-time attribute value is used. The attribute value here is represented by a JSP expression that creates a new `java.util.Date` object (representing the current date and time). The request-time attribute is evaluated when the page is requested, and the corresponding attribute is set to the result of the expression. Any property you set this way must have a Java type matching the result of the expression. In this case, the `entryTime` property must be of type `java.util.Date`.

Declarations

A JSP declaration element starts with `<%!` and ends with `%>`. Note the exclamation point (`!`) in the start identifier; that's what makes it a declaration as opposed to a scriptlet.

This declaration element declares an instance variable named `globalCounter`, shared by all requests for the page:

```
<%@ page language="java" contentType="text/html" %>
<%!
  int globalCounter = 0;
%>
```

Note that a variable declared with a JSP declaration element is shared by all requests for the page. This can cause so-called multithreading problems if more than one request for the page is processed at the same time. For instance, one request may overwrite the value of the variable set by another request. In most cases, you should declare scripting variables using a JSP scriptlet instead:

```
<%
  int requestLocalCounter = 0;
%>
```

A variable declared in a scriptlet is not shared. It holds a unique value for each request.

You can also use a JSP declaration element to declare a method that can then be used in scriptlets in the same page:

```
<%@ page language="java" contentType="text/html" %>
<html>
<body bgcolor="white">

  <%!
    String randomColor() {
      java.util.Random random = new java.util.Random();
      int red = (int) (random.nextFloat() * 255);
      int green = (int) (random.nextFloat() * 255);
      int blue = (int) (random.nextFloat() * 255);
      return "#" +
        Integer.toString(red, 16) +
        Integer.toString(green, 16) +
        Integer.toString(blue, 16);
    }
  %>

  <h1>Random Color</h1>

    <table bgcolor="<%= randomColor() %>" >
      <tr><td width="100" height="100"> </td></tr>
    </table>

</body>
</html>
```

Implicit Objects

When you use scripting elements in a JSP page, you always have access to a number of objects (listed in Table 12) that the web container makes available. These objects are instances of classes defined by the servlet and JSP specifications. Each class is described in detail in this section, following the table.

Table 12. Implicit JSP objects

Variable name	Java type
application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
exception	java.lang.Throwable
out	javax.servlet.jsp.JspWriter
page	java.lang.Object
pageContext	javax.servlet.jsp.PageContext
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
session	javax.servlet.http.HttpSession

application

Variable name: application
Interface name: javax.servlet.ServletContext
Extends: None
Implemented by: Internal container-dependent class
JSP Page type: Available in both regular JSP pages and error pages

Description

The ServletContext provides resources shared within a web application. It holds attribute values representing the JSP application scope. An attribute value can be an instance of any valid Java class. The ServletContext also defines a set of methods that a JSP

page or a servlet uses to communicate with its container; for example, to get the MIME type of a file, dispatch requests, or write to a log file. The web container is responsible for providing an implementation of the `ServletContext` interface.

Each `ServletContext` is assigned a specific URI path prefix within a web server. For example, a context could be responsible for all resources under *http://www.mycorp.com/catalog*. All requests that start with the */catalog* request path, which is known as the *context path*, are routed to this servlet context.

Only one instance of a `ServletContext` may be available to the servlets and JSP pages in a web application. If the web application indicates that it is distributable, there must be only one instance of the `ServletContext` object in use per application in each Java Virtual Machine.

Methods

`public Object getAttribute(String name)`
: Returns the servlet context attribute with the specified name, or `null` if there is no attribute by that name. Context attributes can be set by a servlet or a JSP page, representing the JSP application scope. A container can also use attributes to provide information that is not already available through methods in this interface.

`public java.util.Enumeration getAttributeNames()`
: Returns an `Enumeration` of `String` objects containing the attribute names available within this servlet context.

`public ServletContext getContext(String uripath)`
: Returns a `ServletContext` object that corresponds to a specified URI in the web container. This method allows servlets and JSP pages to gain access to contexts other than their own. The URI path must be absolute (beginning with "/") and is interpreted based on the containers' document root. In a security-conscious environment, the container may return `null` for a given URI.

`public String getInitParameter(String name)`
: Returns a `String` containing the value of the named context-wide initialization parameter, or `null` if the parameter does not exist. Context initialization parameters can be defined in the web application deployment descriptor.

`public java.util.Enumeration getInitParameterNames()`
: Returns the names of the context's initialization parameters as an `Enumeration` of `String` objects, or an empty `Enumeration` if the context has no initialization parameters.

`public int getMajorVersion()`
: Returns the major version of the Java Servlet API the web container supports. A container that complies with the Servlet 2.3 API returns 2.

`public String getMimeType(String filename)`
: Returns the MIME type of the specified file, or `null` if the MIME type is not known. The MIME type is determined by the configuration of the web container and may be specified in a web application deployment descriptor.

`public int getMinorVersion()`
: Returns the minor version of the Java Servlet API the web container supports. A container that complies with the Servlet 2.3 API returns 3.

`public RequestDispatcher getNamedDispatcher(String name)`
: Returns a `RequestDispatcher` object that acts as a wrapper for the named servlet or JSP page. Names can be defined for servlets and JSP pages in the web application deployment descriptor.

`public String getRealPath(String path)`
: Returns a `String` containing the filesystem path for specified context-relative path. This method returns `null` if the web container cannot translate the path to a filesystem path for any reason (such as when the content is being made available from a WAR archive).

`public RequestDispatcher getRequestDispatcher(String path)`
: Returns a `RequestDispatcher` object that acts as a wrapper for the resource located at the specified context-relative path. The resource can be dynamic (servlet or JSP) or static (e.g., a regular HTML file).

`public java.net.URL getResource(String path)`
`  throws MalformedURLException`
: Returns a `URL` to the resource that is mapped to the specified context-relative path. This method allows the web container

to make a resource available to servlets and JSP pages from sources other than a local filesystem, such as a database or a WAR file.

The `URL` provides access to the resource content direct, so be aware that requesting a JSP page returns a `URL` for the JSP source code as opposed to the processed result. Use a `RequestDispatcher` instead to include the results of an execution.

This method returns `null` if no resource is mapped to the pathname.

`public java.io.InputStream getResourceAsStream(String path)`
Returns the resource mapped to the specified context-relative path as an `InputStream` object. See the `getResource()` method for details.

`public String getServerInfo()`
Returns the name and version of the servlet container on which the servlet or JSP page is running as a `String` with the format *servername/versionnumber* (for example, `Tomcat/3.2`). A container may include other optional information, such as the Java version and operating system information, within parentheses.

`public void log(String message)`
Writes the specified message to a web container log file. The name and type of the log file are container-dependent.

`public void log(String message, Throwable cause)`
Writes the specified message and a stack trace for the specified `Throwable` to the servlet log file. The name and type of the log file are container-dependent.

`public void removeAttribute(String name)`
Removes the attribute with the specified name from the servlet context.

`public void setAttribute(String name, Object attribute)`
Binds an object to the specified attribute name in this servlet context. If the specified name is already used for an attribute, this method removes the old attribute and binds the name to the new attribute.

The following methods are deprecated:

`public Servlet getServlet(String name) throws ServletException`
: This method was originally defined to retrieve a servlet from a `ServletContext`. As of the Servlet 2.1 API, this method always returns `null` and remains only to preserve binary compatibility. This method will be permanently removed in a future version of the Java Servlet API.

`public Enumeration getServlets()`
: This method was originally defined to return an `Enumeration` of all the servlets known to this servlet context. As of the Servlet 2.1 API, this method always returns an empty `Enumeration` and remains only to preserve binary compatibility. This method will be permanently removed in a future version of the Java Servlet API.

`public Enumeration getServletNames()`
: This method was originally defined to return an `Enumeration` of all the servlet names known to this context. As of Servlet 2.1, this method always returns an empty `Enumeration` and remains only to preserve binary compatibility. This method will be permanently removed in a future version of the Java Servlet API.

`public void log(Exception exception, String message)`
: This method was originally defined to write an exception's stack trace and an explanatory error message to the web container log file. As of the Servlet 2.1 API, the recommendation is to use `log(String, Throwable)` instead.

config

Variable name: `config`

Interface name: `javax.servlet.ServletConfig`

Extends: None

Implemented by: Internal container-dependent class

JSP page type: Available in both regular JSP pages and error pages

Description

A `ServletConfig` instance is used by a web container to pass information to a servlet or JSP page during initialization. The configuration information contains initialization parameters (defined in the web application deployment descriptor) and the `ServletContext` object representing the web application to which the servlet or JSP page belongs.

Methods

`public String getInitParameter(String name)`
: Returns a `String` containing the value of the specified servlet or JSP page initialization parameter, or `null` if the parameter does not exist.

`public java.util.Enumeration getInitParameterNames()`
: Returns the names of the servlet's or JSP page's initialization parameters as an `Enumeration` of `String` objects, or an empty `Enumeration` if the servlet has no initialization parameters.

`public ServletContext getServletContext()`
: Returns a reference to the `ServletContext` to which the servlet or JSP page belongs.

`public String getServletName()`
: Returns the name of this servlet instance or JSP page. The name may be assigned in the web application deployment descriptor. For an unregistered (and thus unnamed) servlet instance or JSP page, the servlet's class name is returned.

exception

Variable name:	`exception`
Class name:	`java.lang.Throwable`
Extends:	None
Implements:	`java.io.Serializable`
Implemented by:	Part of the standard Java library
JSP page type:	Available only in a page marked as an error page using the `page` directive `isErrorPage` attribute

Description

The `exception` variable is assigned to the subclass of `Throwable` that caused the error page to be invoked. The `Throwable` class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or of one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java `throw` statement. See the Java documentation at *http://java.sun.com/docs/index.html* for a description of the `Throwable` class.

out

Variable name: `out`

Class name: `javax.servlet.jsp.JspWriter`

Extends: `java.io.Writer`

Implements: None

Implemented by: A concrete subclass of this abstract class is provided as an internal container-dependent class.

JSP page type: Available in both regular JSP pages and error pages

Description

The `out` variable is assigned to a concrete subclass of the `JspWriter` abstract class by the web container. `JspWriter` emulates some of the functionality found in the `java.io.BufferedWriter` and `java.io.PrintWriter` classes. It differs, however, in that it throws a `java.io.IOException` from the print methods (the `PrintWriter` does not).

If the `page` directive attribute `autoflush` is set to `true`, all the I/O operations on this class automatically flush the contents of the buffer when it's full. If `autoflush` is set to `false`, all the I/O operations on this class throw an `IOException` when the buffer is full.

Constructor

`protected JspWriter(int bufferSize, boolean autoFlush)`
: Creates an instance with at least the specified buffer size and autoflush behavior.

Methods

`public abstract void clear() throws java.io.IOException`
: Clears the contents of the buffer. If the buffer has already been flushed, throws an `IOException` to signal the fact that some data has already been irrevocably written to the client response stream.

`public abstract void clearBuffer() throws java.io.IOException`
: Clears the current contents of the buffer. Unlike `clear()`, this method does not throw an `IOException` if the buffer has already been flushed. It just clears the current content of the buffer and returns.

`public abstract void close() throws java.io.IOException`
: Closes the `JspWriter` after flushing it. Calls to `flush()` or `write()` after a call to `close()` cause an `IOException` to be thrown. If `close()` is called on a previously closed `JspWriter`, it is ignored.

`public abstract void flush() throws java.io.IOException`
: Flushes the current contents of the buffer to the underlying writer, then flushes the underlying writer. This means the buffered content is delivered to the client immediately.

`public int getBufferSize()`
: Returns the size of the buffer in bytes, or `0` if it is not buffered.

`public abstract int getRemaining()`
: Returns the number of unused bytes in the buffer.

`public boolean isAutoFlush()`
: Returns `true` if this `JspWriter` is set to autoflush the buffer, `false` otherwise.

page

Variable name: `page`
Class name: `Object`
Extends: None
Implements: None
Implemented by: Part of the standard Java library
JSP page type: Available in both regular JSP pages and error pages

Description

The page variable is assigned to the instance of the JSP implementation class, declared as an `Object`. This variable is rarely, if ever, used. See the Java documentation at *http://java.sun.com/docs/index.html* for a description of the `Object` class.

pageContext

Variable name: `pageContext`

Class name: `javax.servlet.jsp.PageContext`

Extends: None

Implements: None

Implemented by: A concrete subclass of this abstract class is provided as an internal container-dependent class.

JSP page type: Available in both regular JSP pages and error pages

Description

A `PageContext` instance provides access to all the JSP scopes and several page attributes, and offers a layer above the container-implementation details to enable a container to generate portable JSP implementation classes. The JSP page scope is represented by `PageContext` attributes. A unique instance of this object is created by the web container and assigned to the `pageContext` variable for each request.

Constants

```
public static final int PAGE_SCOPE = 1;
public static final int REQUEST_SCOPE = 2;
public static final int SESSION_SCOPE = 3;
public static final int APPLICATION_SCOPE = 4;
```

Constructor

`public PageContext()`

Creates an instance of the `PageContext` class. Typically, the `JspFactory` class creates and initializes the instance.

Methods

`public abstract Object findAttribute(String name)`
: Searches for the named attribute in the page, request, session (if valid), and application scope(s) in order and returns the associated value. If the attribute is not found, returns `null`.

`public abstract void forward(String relativeUrlPath) throws ServletException, java.io.IOException`
: Forwards the current request to another active component in the application, such as a servlet or JSP page. If the specified URI starts with a slash, it's interpreted as a context-relative path; otherwise, it's interpreted as a page-relative path.

  The response must not be modified after calling this method, since the response is committed before this method returns.

`public abstract Object getAttribute(String name)`
: Returns the `Object` associated with the specified attribute name in the page scope, or `null` if the attribute is not found.

`public abstract Object getAttribute(String name, int scope)`
: Returns the `Object` associated with the specified attribute name in the specified scope, or `null` if the attribute is not found. The `scope` argument must be one of the `int` values specified by the `PageContext` static scope variables.

`public abstract java.util.Enumeration getAttributeNamesInScope(int scope)`
: Returns an `Enumeration` of `String` objects containing all the attribute names for the specified scope. The `scope` argument must be one of the `int` values specified by the `PageContext` static scope variables.

`public abstract int getAttributesScope(String name)`
: Returns one of the `int` values specified by the `PageContext` static scope variables for the scope of the object associated with the specified attribute name, or `0` if the attribute is not found.

`public abstract Exception getException()`
: Returns the `Exception` that caused the current page to be invoked if its `page` directive `isErrorPage` attribute is set to `true`.

`public abstract JspWriter getOut()`
: Returns the current `JspWriter` for the page. When this method is called by a tag handler that implements `BodyTag` or is nested in the body of another action element, the returned object may be an instance of the `BodyContent` subclass.

`public abstract Object getPage()`
: Returns the `Object` that represents the JSP page implementation class instance with which this `PageContext` is associated.

`public abstract ServletRequest getRequest()`
: Returns the current `ServletRequest`.

`public abstract ServletResponse getResponse()`
: Returns the current `ServletResponse`.

`public abstract ServletConfig getServletConfig()`
: Returns the `ServletConfig` for this JSP page implementation class instance.

`public abstract ServletContext getServletContext()`
: Returns the `ServletContext` for this JSP page implementation class instance.

`public abstract HttpSession getSession()`
: Returns the current `HttpSession`, or `null` if the `page` directive `session` attribute is set to `false`.

`public abstract void handlePageException(Exception e) throws ServletException, java.io.IOException`
: This method is intended to be called by the JSP page implementation class only to process unhandled exceptions, either by forwarding the request exception to the error page specified by the `page` directive `errorPage` attribute or by performing an implementation-dependent action (if no error page is specified).

`public abstract void include(String relativeUrlPath) throws ServletException, java.io.IOException`
: Causes the specified resource to be processed as part of the current request. The current `JspWriter` is flushed before invoking the target resource, and the output of the target resource's processing of the request is written directly to the current `ServletResponse` object's writer. If the specified URI

starts with a slash, it's interpreted as a context-relative path; otherwise, it's interpreted as a page-relative path.

```
public abstract void initialize(Servlet servlet,
  ServletRequest request, ServletResponse response,
  String errorPageURL, boolean needsSession,
  int bufferSize, boolean autoFlush)
  throws java.io.IOException, IllegalStateException,
  IllegalArgumentException
```

This method is called to initialize a `PageContext` object so that it may be used by a JSP implementation class to service an incoming request. This method is typically called from the `JspFactory.getPageContext()` method.

```
public JspWriter popBody()
```

This method is intended to be called by the JSP page implementation class only to reassign the previous `JspWriter`, saved by the matching `pushBody()` method, as the current `JspWriter`.

```
public BodyContent pushBody()
```

This method is intended to be called by the JSP page implementation class only to get a new `BodyContent` object and save the current `JspWriter` on the `PageContext` object's internal stack.

```
public abstract void release()
```

Resets the internal state of a `PageContext`, releasing all internal references and preparing the `PageContext` for potential reuse by a later invocation of `initialize()`. This method is typically called from the `JspFactory.releasePageContext()` method.

```
public abstract void removeAttribute(String name)
```

Removes the object reference associated with the specified attribute name in the page scope.

```
public abstract void removeAttribute(String name, int scope)
```

Removes the object reference associated with the specified attribute name in the specified scope. The `scope` argument must be one of the `int` values specified by the `PageContext` static scope variables.

```
public abstract void setAttribute(String name,
  Object attribute)
```

Saves the specified attribute name and object in the page scope.

`public abstract void setAttribute(String name, Object o, int scope)`
: Saves the specified attribute name and object in the specified scope. The `scope` argument must be one of the `int` values specified by the `PageContext` static scope variables.

request

Variable name: `request`

Interface name: `javax.servlet.http.HttpServletRequest`

Extends: `javax.servlet.ServletRequest`

Implemented by: Internal container-dependent class

JSP page type: Available in both regular JSP pages and error pages

Description

The `request` variable is assigned a reference to an internal container-dependent class that implements a protocol-dependent interface that extends the `javax.servlet.ServletRequest` interface. Since HTTP is the only protocol supported by JSP 1.1, the class always implements the `javax.servlet.http.HttpServletRequest` interface. The method descriptions in this section include the methods from both interfaces.

Methods

`public Object getAttribute(String name)`
: Returns the value of the named attribute as an `Object`, or `null` if no attribute of the given name exists.

`public java.util.Enumeration getAttributeNames()`
: Returns an `Enumeration` containing the names of the attributes available to this request. The `Enumeration` is empty if the request doesn't have any attributes.

`public String getAuthType()`
: Returns the name of the authentication scheme used to protect the servlet (for example, BASIC or SSL), or `null` if the servlet is not protected.

`public String getCharacterEncoding()`
: Returns the name of the character encoding method used in the body of this request, or `null` if the request does not specify a character encoding method.

`public int getContentLength()`
: Returns the length, in bytes, of the request body (if it is made available by the input stream), or `-1` if the length is not known.

`public String getContentType()`
: Returns the MIME type of the body of the request, or `null` if the type is not known.

`public String getContextPath()`
: Returns the portion of the request URI that indicates the context of the request.

`public Cookie[] getCookies()`
: Returns an array containing all the `Cookie` objects the client sent with this request, or `null` if the request contains no cookies.

`public long getDateHeader(String name)`
: Returns the value of the specified request header as a `long` value that represents a date value, or `-1` if the header is not included in the request.

`public String getHeader(String name)`
: Returns the value of the specified request header as a `String`, or `null` if the header is not included with the request.

`public java.util.Enumeration getHeaderNames()`
: Returns all the header names this request contains as an `Enumeration` of `String` objects. The `Enumeration` is empty if the request doesn't have any headers.

`public java.util.Enumeration getHeaders(String name)`
: Returns all the values of the specified request header as an `Enumeration` of `String` objects. The `Enumeration` is empty if the request doesn't contain the specified header.

`public ServletInputStream getInputStream()`
`  throws java.io.IOException`
: Retrieves the body of the request as binary data using a `ServletInputStream`.

`public int getIntHeader(String name)`
: Returns the value of the specified request header as an `int`, or `-1` if the header is not included in the request.

`public java.util.Locale getLocale()`
: Returns the preferred `Locale` in which the client will accept content, based on the `Accept-Language` header.

`public java.util.Enumeration getLocales()`
: Returns an `Enumeration` of `Locale` objects indicating, in decreasing order and starting with the preferred locale, the locales that are acceptable to the client based on the `Accept-Language` header.

`public String getMethod()`
: Returns the name of the HTTP method with which this request was made; for example, `GET`, `POST`, or `PUT`.

`public String getParameter(String name)`
: Returns the value of a request parameter as a `String`, or `null` if the parameter does not exist.

`public String getParameterNames()`
: Returns an `Enumeration` of `String` objects containing the names of the parameters in this request.

`public String[] getParameterValues()`
: Returns an array of `String` objects containing all of the given request parameter's values, or `null` if the parameter does not exist.

`public String getPathInfo()`
: Returns any extra path information associated with the URI the client sent when it made this request, or `null` if there is no extra path information. For a JSP page, this method always returns `null`.

`public String getPathTranslated()`
: Returns the result of `getPathInfo()` translated into the corresponding filesystem path. Returns `null` if `getPathInfo()` returns `null`.

`public String getProtocol()`
: Returns the name and version of the protocol the request uses in the form *protocol/majorVersion.minorVersion*; for example, `HTTP/1.1`.

`public String getQueryString()`
: Returns the query string that is contained in the request URI after the path.

`public java.io.BufferedReader getReader() throws java.io.IOException`
: Retrieves the body of the request as character data using a `BufferedReader`.

`public String getRemoteAddr()`
: Returns the Internet Protocol (IP) address of the client that sent the request.

`public String getRemoteHost()`
: Returns the fully qualified name of the client host that sent the request or, if the hostname cannot be determined, the IP address of the client.

`public String getRemoteUser()`
: Returns the login ID of the user making this request if the user has been authenticated, or `null` if the user has not been authenticated.

`public RequestDispatcher getRequestDispatcher(String path)`
: Returns a `RequestDispatcher` object that acts as a wrapper for the resource located at the given path.

`public String getRequestedSessionId()`
: Returns the session ID specified by the client.

`public String getRequestURI()`
: Returns the part of this request's URI from the protocol name up to the query string in the first line of the HTTP request.

`public String getScheme()`
: Returns the name of the scheme (protocol) used to make this request; for example, `http`, `https`, or `ftp`.

`public String getServerName()`
: Returns the hostname of the server that received the request.

`public int getServerPort()`
: Returns the port number on which the request was received.

`public String getServletPath()`
: Returns the part of this request's URI that calls the servlet. For a JSP page, this is the page's complete context-relative path.

`public HttpSession getSession()`
: Returns the current `HttpSession` associated with this request. If the request does not have a session, a new `HttpSession` object is created, associated with the request, and returned.

`public HttpSession getSession(boolean create)`
: Returns the current `HttpSession` associated with this request. If there is no current session and `create` is `true`, a new `HttpSession` object is created, associated with the request, and returned. If `create` is `false` and the request is not associated with a session, this method returns `null`.

`public java.security.Principal getUserPrincipal()`
: Returns a `Principal` object containing the name of the current authenticated user.

`public boolean isRequestedSessionIdFromCookie()`
: Checks if the requested session ID came in as a cookie.

`public boolean isRequestedSessionIdFromURL()`
: Checks if the requested session ID came in as part of the request URL.

`public boolean isRequestedSessionIdValid()`
: Checks if the requested session ID is still valid.

`public boolean isSecure()`
: Returns a `boolean` indicating whether this request was made using a secure channel, such as HTTPS, or not.

`public boolean isUserInRole(String role)`
: Returns a `boolean` indicating whether the authenticated user is included in the specified logical role or not.

`public void removeAttribute(String name)`
: Removes the specified attribute from the request.

`public Object setAttribute(String name, Object attribute)`
: Stores the specified attribute in the request.

The following methods are deprecated:

`public String getRealPath()`
: As of the Servlet 2.1 API, use `ServletContext.getRealPath(String)` instead.

`public boolean isRequestSessionIdFromUrl()`
: As of the Servlet 2.1 API, use `isRequestedSessionIdFromURL()` instead.

response

Variable name: response

Interface name: javax.servlet.http.HttpServletResponse

Extends: javax.servlet.ServletResponse

Implemented by: Internal container-dependent class

JSP page type: Available in both regular JSP pages and error pages

Description

The response variable is assigned a reference to an internal container-dependent class that implements a protocol-dependent interface that extends the javax.servlet.ServletResponse interface. Since HTTP is the only protocol supported by JSP 1.1, the class always implements the javax.servlet.http.HttpServletResponse interface. The method descriptions in this section include the methods from both interfaces.

Constants

```
public static final int SC_CONTINUE = 100;
public static final int SC_SWITCHING_PROTOCOLS = 101;
public static final int SC_OK = 200;
public static final int SC_CREATED = 201;
public static final int SC_ACCEPTED = 202;
public static final int SC_NON_AUTHORITATIVE_INFORMATION =
203;
public static final int SC_NO_CONTENT = 204;
public static final int SC_RESET_CONTENT = 205;
public static final int SC_PARTIAL_CONTENT = 206;
public static final int SC_MULTIPLE_CHOICES = 300;
public static final int SC_MOVED_PERMANENTLY = 301;
public static final int SC_MOVED_TEMPORARILY = 302;
public static final int SC_SEE_OTHER = 303;
public static final int SC_NOT_MODIFIED = 304;
public static final int SC_USE_PROXY = 305;
public static final int SC_TEMPORARY_REDIRECT = 307;
public static final int SC_BAD_REQUEST = 400;
public static final int SC_UNAUTHORIZED = 401;
public static final int SC_PAYMENT_REQUIRED = 402;
public static final int SC_FORBIDDEN = 403;
public static final int SC_NOT_FOUND = 404;
public static final int SC_METHOD_NOT_ALLOWED = 405;
```

```
public static final int SC_NOT_ACCEPTABLE = 406;
public static final int SC_PROXY_AUTHENTICATION_REQUIRED =
407;
public static final int SC_REQUEST_TIMEOUT = 408;
public static final int SC_CONFLICT = 409;
public static final int SC_GONE = 410;
public static final int SC_LENGTH_REQUIRED = 411;
public static final int SC_PRECONDITION_FAILED = 412;
public static final int SC_REQUEST_ENTITY_TOO_LARGE = 413;
public static final int SC_REQUEST_URI_TOO_LONG = 414;
public static final int SC_UNSUPPORTED_MEDIA_TYPE = 415;
public static final int SC_REQUESTED_RANGE_NOT_SATISFIABLE
= 416;
public static final int SC_EXPECTATION_FAILED = 417;
public static final int SC_INTERNAL_SERVER_ERROR = 500;
public static final int SC_NOT_IMPLEMENTED = 501;
public static final int SC_BAD_GATEWAY = 502;
public static final int SC_SERVICE_UNAVAILABLE = 503;
public static final int SC_GATEWAY_TIMEOUT = 504;
public static final int SC_HTTP_VERSION_NOT_SUPPORTED =
505;
```

Methods

`public void addCookie(Cookie cookie)`
: Adds the specified cookie to the response.

`public void addDateHeader(String headername, long date)`
: Adds a response header with the given name and date value. The date is specified in terms of milliseconds since the epoch (January 1, 1970, 00:00:00 GMT).

`public void addHeader(String headername, String value)`
: Adds a response header with the specified name and value.

`public void addIntHeader(String headername, int value)`
: Adds a response header with the given name and integer value.

`public boolean containsHeader(String name)`
: Returns a `boolean` indicating whether the named response header has already been set.

`public String encodeRedirectURL(String url)`
: Encodes the specified URL for use in the `sendRedirect()` method by including the session ID in it. If encoding (URL rewriting) is not needed, it returns the URL unchanged.

`public String encodeURL(String url)`
: Encodes the specified URL for use in a reference element (e.g., `<a>`) by including the session ID in it. If encoding (URL rewriting) is not needed, it returns the URL unchanged.

`public void flushBuffer() throws IOException`
: Forces any content in the response body buffer to be written to the client.

`public int getBufferSize()`
: Returns the actual buffer size (in bytes) used for the response, or `0` if no buffering is used.

`public String getCharacterEncoding()`
: Returns the name of the charset used for the MIME body sent in this response.

`public Locale getLocale()`
: Returns the locale assigned to the response. This is either a `Locale` object for the server's default locale or the `Locale` set with `setLocale()`.

`public ServletOutputStream getOutputStream()`
`  throws IOException`
: Returns a `ServletOutputStream` suitable for writing binary data in the response. This method should not be used in a JSP page, since JSP pages are intended for text data.

`public PrintWriter getWriter throws IOException`
: Returns a `PrintWriter` object that can send character text to the client. This method should not be used in a JSP page, since it may interfere with the container's writer mechanism. Use the `PageContext` method instead to get the current `JspWriter`.

`public boolean isCommitted()`
: Returns a `boolean` indicating if the response has been committed.

`public void reset()`
: Clears any data that exists in the buffer as well as the status code and headers. If the response has been committed, this method throws an `IllegalStateException`.

`public void sendError(int status) throws IOException`
: Sends an error response to the client using the specified status. If the response has already been committed, this

method throws an `IllegalStateException`. After you use this method, you should consider the response committed and should not write to it.

`public void sendError(int status, String message) throws IOException`
: Sends an error response to the client using the specified status code and descriptive message. If the response has already been committed, this method throws an `IllegalStateException`. After you use this method, you should consider the response committed and should not write to it.

`public void sendRedirect(String location) throws IOException`
: Sends a temporary redirect response to the client using the specified redirect location URL. This method can accept relative URLs; the servlet container will convert the relative URL to an absolute URL before sending the response to the client. If the response is already committed, this method throws an `IllegalStateException`. After you use this method, you should consider the response committed and should not write to it.

`public void setBufferSize(int size)`
: Sets the preferred buffer size (in bytes) for the body of the response. The servlet container uses a buffer at least as large as the size requested. The actual buffer size used can be found with the `getBufferSize()` method.

`public void setContentLength(int length)`
: Sets the length (in bytes) of the content body in the response. In HTTP servlets, this method sets the HTTP `Content-Length` header. This method should not be used in a JSP page, since it may interfere with the container's writer mechanism.

`public void setContentType(String type)`
: Sets the content type of the response being sent to the client.

`public void setDateHeader(String headername, long date)`
: Sets a response header with the given name and date value. The date is specified in terms of milliseconds since the epoch (January 1, 1970, 00:00:00 GMT). If the header is already set, the new value overwrites the previous one.

`public void setHeader(String headername, String value)`
: Sets a response header with the given name and value. If the header is already set, the new value overwrites the previous one.

`public void setIntHeader(String headername, int value)`
: Sets a response header with the given name and integer value. If the header is already set, the new value overwrites the previous one.

`public void setLocale(Locale locale)`
: Sets the locale of the response, setting the headers (including the `Content-Type` header's charset) as appropriate.

`public void setStatus(int statuscode)`
: Sets the status code for this response. Unlike the `sendError()` method, this method only sets the status code; it doesn't add a body and it does not commit the response.

The following methods are deprecated:

`public String encodeRedirectUrl(String url)`
: As of the Servlet 2.1 API, use `encodeRedirectURL(String url)` instead.

`public String encodeUrl(String url)`
: As of the Servlet 2.1 API, use `encodeURL(String url)` instead.

`public void setStatus(int statuscode, String message)`
: As of the Servlet 2.1 API, use `setStatus(int)` to set a status code and `sendError(int, String)` to send an error with a description. This method was deprecated because of the ambiguous meaning of the message parameter.

session

Variable name: `session`

Interface name: `javax.servlet.http.HttpSession`

Extends: None

Implemented by: Internal container-dependent class

JSP page type: Available in both regular JSP pages and error pages, unless the `page` directive `session` attribute is set to `false`

Description

The session variable is assigned a reference to the HttpSession object that represents the current client session. Information stored as HttpSession attributes corresponds to objects in the JSP session scope.

By default, the session persists for the time period specified in the web application deployment descriptor, across more than one page request from the user. The container can maintain a session in many ways, such as using cookies or rewriting URLs.

Methods

`public Object getAttribute(String name)`
: Returns the Object associated with the specified name in this session, or null if the object is not found.

`public java.util.Enumeration getAttributeNames()`
: Returns an Enumeration of String objects containing the names of all the objects in this session.

`public long getCreationTime()`
: Returns the time when this session was created, measured in milliseconds since the epoch (January 1, 1970, 00:00:00 GMT).

`public String getId()`
: Returns a String containing the unique identifier assigned to this session.

`public long getLastAccessedTime()`
: Returns the last time the client sent a request associated with this session as the number of milliseconds since the epoch (January 1, 1970, 00:00:00 GMT).

`public int getMaxInactiveInterval()`
: Returns the maximum time interval, in seconds, that the servlet container will keep this session active between client accesses.

`public void invalidate()`
: Invalidates this session and unbinds any objects bound to it, calling the valueUnbound() methods of all objects in the session implementing the HttpSessionBindingListener interface.

`public boolean isNew()`
: Returns true if a request for this session has not yet been received from the client.

`public void removeAttribute(String name)`
: Removes the object bound with the specified name from this session.

`public void setAttribute(String name, Object attribute)`
: Associates the specified object with this session using the name specified.

`public void setMaxInactiveInterval(int interval)`
: Specifies the time, in seconds, that can elapse between client requests before the servlet container will invalidate this session.

The following methods are deprecated:

`public HttpSessionContext getSessionContext()`
: As of the Servlet 2.1 API, this method is deprecated and has no replacement.

`public Object getValue(String name)`
: As of the Servlet 2.2 API, this method is replaced by `getAttribute(String)`.

`public String[] getValueNames()`
: As of the Servlet 2.2 API, this method is replaced by `getAttributeNames()`.

`public void putValue(String name, Object value)`
: As of the Servlet 2.2 API, this method is replaced by `setAttribute(String, Object)`.

`public void removeValue(String name)`
: As of the Servlet 2.2 API, this method is replaced by `setAttribute(String, Object)`.

Custom Actions

Custom action elements can be developed by programmers to extend the JSP language; for instance, for application-specific presentation, localization, validation, or any other task not provided by the standard JSP action elements.

The general syntax for using a custom action element in a JSP page is the same as that for JSP standard actions: a start tag (optionally with attributes), a body, and an end tag.

Other elements and template text can be nested in the body. Here's an example:

```
<prefix:actionName attr1="value1" attr2="value2">
  The body
</prefix:actionName>
```

If the action element doesn't have a body, the following shorthand notation can be used instead of the start tag and end tag:

```
<prefix:actionName attr1="value1" attr2="value2" />
```

Before you can use a custom action in a JSP page, you must declare the tag library containing the custom action by using the `taglib` directive, identifying the library and assigning the prefix to be used for all custom action elements in the page.

Developing Custom Actions

A custom action—actually, a *tag handler* class for a custom action—is basically a bean, with property setter methods corresponding to the custom action element's attributes. In addition, the tag handler class must implement one of two Java interfaces defined by the JSP specification.

All the interfaces and classes you need to implement a tag handler are defined in the `javax.servlet.jsp.tagext` package. The two primary interfaces are named `Tag` and `BodyTag`. The `Tag` interface defines the methods you need to implement for any action. The `BodyTag` interface extends the `Tag` interface and adds methods used to access the body of an action element. To make it easier to develop a tag handler, two support classes are defined by the API: `TagSupport` and `BodyTagSupport` (shown in Figure 4). These classes provide default implementations for the methods in the corresponding interfaces.

The specification defines interfaces as well as the support classes that implement them to cover all the bases. If you already have a class with functionality that you want to access as a custom action, you can specify that your class implements the appropriate interface and add the few methods

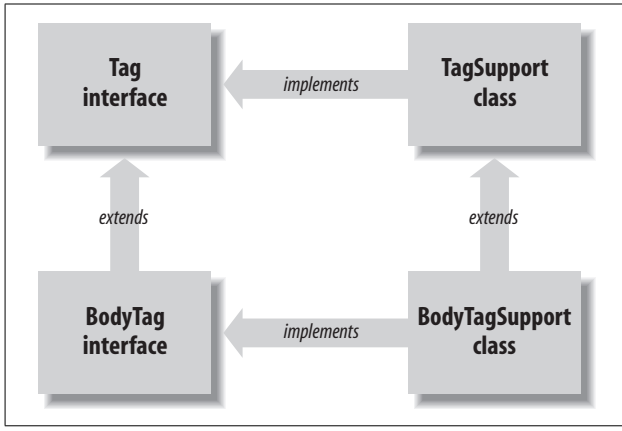


Figure 4. The primary tag-extension interfaces and support classes

defined by that interface. In practice, though, I recommend that you implement your tag handlers as extensions of the support classes. This way you get most of the methods for free, and you can still reuse your existing classes by calling them from the tag handler.

A *tag library* is a collection of custom actions. Besides the tag handler class files, a tag library must contain a *Tag Library Descriptor* (TLD) file. This is an XML file that maps all the custom action names to the corresponding tag handler classes and describes all the attributes supported by each custom action. The class files and the TLD can be packaged in a JAR file to make the tag library easy to install.

Before we get into the intricate details, let's take a brief look at what it takes to develop, deploy, and use a custom action. First you must implement a tag handler class, like the following:

```
package com.mycompany;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
```

```
public class HelloTag extends TagSupport {
    private String name = "World";

    public void setName(String name) {
        this.name = name;
    }

    public int doEndTag() {
        try {
            pageContext.getOut().println("Hello " +
              name);
        }
        catch (IOException e) {} // Ignore it
        return EVAL_PAGE;
    }
}
```

The tag handler class contains a setter method for an attribute named `name`. The `doEndTag()` method (defined by the `Tag` interface) simply writes "Hello " plus the `name` attribute value to the response. Compile the class and place the resulting class file in the *WEB-INF/classes* directory for the application.

Next, create the TLD file. The following is a minimal TLD file for a library containing just the one custom action element in this example:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>test</shortname>

  <tag>
    <name>hello</name>
    <tagclass>com.mycompany.HelloTag</tagclass>
    <bodycontent>empty</bodycontent>
    <attribute>
      <name>name</name>
    </attribute>
  </tag>
</tablib>
```

The TLD maps the custom action name `hello` to the tag handler class `com.mycompany.HelloTag` and defines the `name` attribute. Place the TLD file in the application's *WEB-INF/tlds* directory, using a filename such as *mylib.tld*.

Now you're ready to use the custom action in a JSP page, like this:

```
<%@ taglib uri="/WEB-INF/mylib.tld" prefix="test" %>
<html>
  <body bgcolor="white">
    <test:hello name="Hans" />
  </body>
</html>
```

The `taglib` directive associates the TLD with the element name prefix used for the custom action in this page: `test`. When the page is requested, the web container uses the TLD to figure out which class to execute for the custom action. It then calls all the appropriate methods, resulting in the text "Hello Hans" being added to the response.

Custom Actions That Do Not Process Their Bodies

A tag handler is the object that's invoked by the web container when a custom action element is found in a JSP page. In order for it to do anything interesting, it needs access to all the information about the request and the page, as well as the action element's attribute values (if any). At a minimum, the tag handler must implement the `Tag` interface, which contains methods for giving it access to the request and page information, as well as the methods called by the container when the start tag and end tag are encountered. For the attribute values, the web container treats the tag handler as a bean and calls property setter methods corresponding to the action element attributes, as shown in Figure 5.

Typically, the tag handler class extends the `TagSupport` class (which provides default implementations for all `Tag` methods) and overrides only one of the methods.

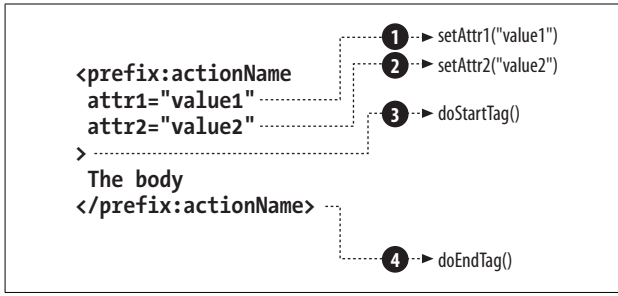


Figure 5. Tag interface methods and property setter methods

Note that while an action element supported by a tag handler that implements the `Tag` interface may have a body, this tag handler will have more limited control over the body content than a tag handler that implements the `BodyTag` interface.

Tag Interface

Interface name: `javax.servlet.jsp.tagext.Tag`

Extends: None

Implemented by: Custom action tag handler classes and `javax.servlet.jsp.tagext.TagSupport`

Description

The `Tag` interface should be implemented by tag handler classes that do not need access to the body contents of the corresponding custom action element and do not need to iterate over the body of a custom action element.

Methods

`public int doEndTag() throws JspException`

Performs actions when the end tag is encountered. If this method returns `SKIP_PAGE`, execution of the rest of the page is aborted and the `_jspService()` method of JSP page implementation class returns. If `EVAL_PAGE` is returned, the code following the custom action in the `_jspService()` method is executed.

`public int doStartTag() throws JspException`
: Performs actions when the start tag is encountered. This method is called by the web container after all property setter methods have been called. The return value controls how the action's body, if any, is handled. If it returns `EVAL_BODY_INCLUDE`, the web container evaluates the body and processes possible JSP elements. The result of the evaluation is added to the response. If `SKIP_BODY` is returned, the body is ignored.

  A tag handler class that implements the `BodyTag` interface (extending the `Tag` interface) can return `EVAL_BODY_TAG` instead of `EVAL_BODY_INCLUDE`. The web container then creates a `BodyContent` instance and makes it available to the tag handler for special processing.

`public Tag getParent()`
: Returns the tag handler's parent (the `Tag` instance for the enclosing action element, if any) or `null` if the tag handler doesn't have a parent.

`public void release()`
: Removes the references to all objects held by this instance.

`public void setPageContext(PageContext pc)`
: Saves a reference to the current `PageContext`.

`public void setParent(Tag t)`
: Saves a reference to the tag handler's parent (the `Tag` instance for the enclosing action element).

TagSupport Class

Class name: `javax.servlet.jsp.tagext.TagSupport`

Extends: None

Implements: `Tag`, `java.io.Serializable`

Implemented by: Internal container-dependent class. Most containers use the reference implementation of the class (developed in the Apache Jakarta project).

Description

`TagSupport` is a support class that provides default implementations for all `Tag` interface methods. It's intended to be used as a

superclass for tag handlers that do not need access to the body contents of the corresponding custom action elements.

Constructor

`public TagSupport()`
: Creates a new instance with the specified name and value.

Methods

`public int doEndTag() throws JspException`
: Returns `EVAL_PAGE`.

`public int doStartTag() throws JspException`
: Returns `SKIP_BODY`.

`public static final Tag findAncestorWithClass(Tag from, Class class)`
: Returns the instance of the specified class, found by testing for a match of each parent in a tag handler nesting structure (corresponding to nested action elements) starting with the specified `Tag` instance, or `null` if not found.

`public String getId()`
: Returns the `id` attribute value, or `null` if not set.

`public Tag getParent()`
: Returns the parent of this `Tag` instance (representing the action element that contains the action element corresponding to this `Tag` instance), or `null` if the instance has no parent (i.e., is at the top level in the JSP page).

`public Object getValue(String k)`
: Returns the value for the specified attribute that has been set with the `setValue()` method, or `null` if not found.

`public java.util.Enumeration getValues()`
: Returns an `Enumeration` of all attribute names for values set with the `setValue()` method.

`public void release()`
: Removes the references to all objects held by this instance.

`public void removeValue(String k)`
: Removes a value set with the `setValue()` method.

`public void setPageContext(PageContext pageContext)`
Saves a reference to the current `PageContext`.

`public void setId(String id)`
Sets the `id` attribute value.

`public void setParent(Tag t)`
Saves a reference to the parent for this instance.

`public void setValue(String k, Object o)`
Saves the specified attribute with the specified value. Subclasses can use this method to save attribute values as an alternative to instance variables.

Example

An example of a custom action that can be implemented as a simple tag handler (that is, just implementing the `Tag` interface) is an action that adds a cookie to the HTTP response. Let's call this action `<ora:addCookie>`. The tag handler class is called `com.ora.jsp.tags.generic.AddCookieTag` and extends the `TagSupport` class to inherit most of the `Tag` interface method implementations:

```
package com.ora.jsp.tags.generic;

import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import com.ora.jsp.util.*;

public class AddCookieTag extends TagSupport {
```

The `<ora:addCookie>` action has two mandatory attributes, `name` and `value`, and one optional attribute, `maxAge`. Each attribute is represented by an instance variable and a standard property setter method:

```
    private String name;
    private String value;
    private String maxAgeString;

    public void setName(String name) {
        this.name = name;
    }

    public void setValue(String value) {
```

```
        this.value = value;
    }

    public void setMaxAge(String maxAgeString) {
        this.maxAgeString = maxAgeString;
    }
```

All setter methods set the corresponding instance variables.

The purpose of the custom action is to create a new `javax.servlet.Cookie` object with the `name`, `value`, and `maxAge` values specified by the attributes and add the cookie to the response. The tag handler class overrides the `doEndTag()` method to carry out this work:

```
    public int doEndTag() throws JspException {
        int maxAge = -1;
        if (maxAgeString != null) {
            try {
                maxAge = Integer.valueOf(maxAgeString).
                  intValue();
            }
            catch (NumberFormatException e) {
               throw new JspException("Invalid maxAge: " +
                    e.getMessage());
            }
        }
        sendCookie(name, value, maxAge,
           (HttpServletResponse) pageContext.getResponse());
        return EVAL_PAGE;
    }

    private void sendCookie(String name, String value,
      int maxAge,
        HttpServletResponse res) {
        Cookie cookie = new Cookie(name, value);
        cookie.setMaxAge(maxAge);
        res.addCookie(cookie);
    }
```

The `maxAge` attribute is optional, so before the corresponding `String` value is converted to an `int`, a test is performed to see if it's set. Similar tests are not necessary for the `name` and `value` variables because the web container verifies that all mandatory attributes are set in the custom action. If a mandatory attribute is not set, the web container refuses to process the page—so you can always be sure that a variable corresponding to a mandatory attribute has

a value. Whether an attribute is mandatory is specified in the TLD for the library.

The tag handler class should also implement the `release()` method, to release all references to objects it has acquired:

```
public void release() {
    name = null;
    value = null;
    maxAgeString = null;
    super.release();
}
```

The `release()` method is called when the tag handler is no longer needed. The `AddCookieTag` class sets all its properties to `null` and calls `super.release()` to let the `TagSupport` class do the same. This makes all property objects available for garbage collection.

A `TagSupport` method that's not needed for this example but can be handy in other situations is the `findAncestorWithClass()` method. It can be used by a tag handler for a nested action element to find its parent. The nested tag handler can then call methods implemented by the parent tag handler class to get from or provide information to the parent. For example, it can provide the `<jsp:param>` elements nested within the body of `<jsp:forward>` and `<jsp:include>` standard JSP action elements. An equivalent custom action for a nested parameter element could be implemented with a tag handler that uses the `findAncestorWithClass()` method like this:

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class ParamTag extends TagSupport {
    private String name;
    private String value;

    public void setName(String name) {
        this.name = name;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public int doEndTag() throws JspException {
        Tag parent = findAncestorWithClass(this,
```

```
            ParamParent.class);
        if (parent == null) {
            throw new JspException("The param action is not " +
                 "enclosed by a supported action type");
        }
        ParamParent paramParent = (ParamParent) parent;
        paramParent.setParam(name, URLEncoder.
          encode(value));
        return EVAL_PAGE;
    }
}
```

Custom Actions That Process Their Bodies

As you can see, it's easy to develop a tag handler that doesn't need to do anything with the action element's body. For a tag handler that does need to process the body, just a few more methods are needed. They are defined by the `BodyTag` interface, an interface that extends the `Tag` interface.

You can use the action element's body in many ways. One use is for input values spanning multiple lines. Say that you develop a custom action that executes a SQL statement specified by the page author. SQL statements are often large, so it's better to let the page author write the statement in the action body instead of forcing it to fit on one line, which is a requirement for an attribute value. You can also use the action element's body in an action that processes the body content in a particular way before it's added to the response (for instance, an action that processes its XML body using an XSL stylesheet specified as an attribute).

As in the `Tag` interface, there's a `BodyTagSupport` class that implements all the methods of the `BodyTag` interface plus a few utility methods.

A tag handler that implements the `BodyTag` interface is at first handled the same way as a tag handler implementing the `Tag` interface: the container calls all property setter methods and the `doStartTag()` method. But then things diverge, as illustrated in Figure 6.

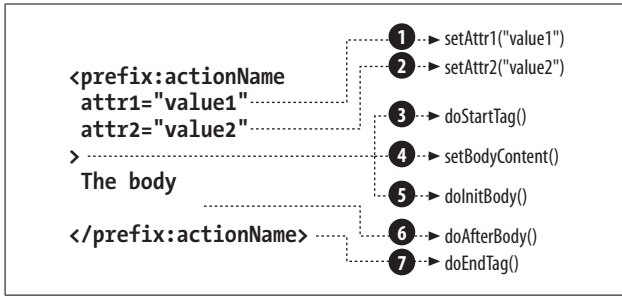


Figure 6. BodyTag interface methods

The additional methods, `setBodyContent()`, `doInitBody()`, and `doAfterBody()`, give the tag handler access to the content of the element's body and an opportunity to process it, as described in the next sections.

BodyTag Interface

Interface name: `javax.servlet.jsp.tagext.BodyTag`

Extends: `javax.servlet.jsp.tagext.Tag`

Implemented by: Custom action tag handler classes and `javax.servlet.jsp.tagext.BodyTagSupport`

Description

The `BodyTag` interface must be implemented by tag handler classes that need access to the body contents of the corresponding custom action element; for instance, in order to perform a transformation of the contents before they are included in the response. This interface must also be implemented by tag handlers that need to iterate over the body of a custom action element.

Methods

`public int doAfterBody() throws JspException`

Performs actions after the body has been evaluated. This method is invoked after every body evaluation. If this method returns `EVAL_BODY_TAG` the body is evaluated again,

typically after changing the values of variables used in it. If it returns `SKIP_BODY`, the processing continues with a call to `doEndTag()`.

This method is not invoked if the element body is empty or if `doStartTag()` returns `SKIP_BODY`.

`public void doInitBody() throws JspException`
Prepares for evaluation of the body. This method is invoked by the page implementation once per action invocation, after a new `BodyContent` has been obtained and set on the tag handler via the `setBodyContent()` method and before the evaluation of the element's body.

This method is not invoked if the element body is empty or if `doStartTag()` returns `SKIP_BODY`.

`public void setBodyContent(BodyContent b)`
Sets the `BodyContent` created for this tag handler. This method is not invoked if the element body is empty or if `doStartTag()` returns `SKIP_BODY`.

BodyTagSupport Class

Class name: `javax.servlet.jsp.tagext.BodyTagSupport`

Extends: `javax.servlet.jsp.tagext.TagSupport`

Implements: `BodyTag`

Implemented by: Internal container-dependent class. Most containers use the reference implementation of the class (developed in the Apache Jakarta project).

Description

`BodyTagSupport` is a support class that provides default implementations of all `BodyTag` interface methods. It's intended to be used as a superclass for tag handlers that need access to the body contents of the corresponding custom action elements.

Constructor

`public BodyTagSupport()`
Creates a new `BodyTagSupport` instance.

Methods

`public int doAfterBody() throws JspException`
: Returns `SKIP_BODY`.

`public int doEndTag() throws JspException`
: Returns `EVAL_PAGE`.

`public void doInitBody()`
: Does nothing in the `BodyTagSupport` class.

`public BodyContent getBodyContent()`
: Returns the `BodyContent` object assigned to this instance.

`public JspWriter getPreviousOut()`
: Returns the enclosing writer of the `BodyContent` object assigned to this instance.

`public void release()`
: Removes the references to all objects held by this instance.

`public void setBodyContent(BodyContent b)`
: Saves a reference to the assigned `BodyContent` object as an instance variable.

BodyContent Class

Class name: `javax.servlet.jsp.tagext.BodyContent`

Extends: `javax.servlet.jsp.JspWriter`

Implements: None

Implemented by: Internal container-dependent class

Description

The container creates an instance of the `BodyContent` class to hold the result of evaluating the element's body content if the corresponding tag handler implements the `BodyTag` interface. The container makes the `BodyContent` instance available to the tag handler by calling the `setBodyContent()` method, so the tag handler can process the body content.

Constructor

`protected BodyContent(JspWriter e)`
: Creates a new instance with the specified `JspWriter` as the enclosing writer.

Methods

`public void clearBody()`
: Removes all buffered content for this instance.

`public void flush() throws java.io.IOException`
: Overwrites the behavior inherited from `JspWriter` to always throw an `IOException`, since it's invalid to flush a `BodyContent` instance.

`public JspWriter getEnclosingWriter()`
: Returns the enclosing `JspWriter`; in other words, either the top-level `JspWriter` or the `JspWriter` (`BodyContent` subclass) of the parent tag handler.

`public abstract java.io.Reader getReader()`
: Returns the value of this `BodyContent` object as a `Reader` with the content produced by evaluating the element's body.

`public abstract String getString()`
: Returns the value of this `BodyContent` object as a `String` with the content produced by evaluating the element's body.

`public abstract void writeOut(java.io.Writer out) throws java.io.IOException`
: Writes the content of this `BodyContent` object into a `Writer`.

Example

Let's look at a tag handler class that extends the `BodyTagSupport` class. The `EncodeHTMLTag` class is the tag handler class for a custom action called `<ora:encodeHTML>`. This action reads its body; replaces all characters with a special meaning in HTML, such as single quotes, double quotes, less-than symbols, greater-than symbols, and ampersands, with their corresponding HTML character entities (i.e., `&#39;`, `&#34;`, `&lt;`, `&gt;`, and `&amp;`); and inserts the result in the response body. The following example shows how the action can be used in a JSP page:

```
<%@ page language="java" %>
<%@ taglib uri="/orataglib" prefix="ora" %>
<html>
  <head>
    <title>Encoded HTML Example</title>
  </head>
  <body>
```

```
    <h1>Encoded HTML Example</h1>
    The following text is encoded by the
    &lt;ora:encodeHTML&gt; custom action:
    <pre>
      <ora:encodeHTML>
        HTML 3.2 Documents start with a <!DOCTYPE>
        declaration followed by an HTML element containing
        a HEAD and then a BODY element:

        <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
        <HTML>
        <HEAD>
        <TITLE>A study of population dynamics</TITLE>
        ... other head elements
        </HEAD>
        <BODY>
        ... document body
        </BODY>
        </HTML>
      </ora:encodeHTML>
    </pre>
  </body>
</html>
```

Note that the body of the <ora:encodeHTML> action in the JSP page example contains HTML elements. If the special characters aren't converted to HTML character entities, the browser interprets the HTML and shows the result of that interpretation instead of the elements themselves. Thanks to the conversion performed by the custom action, however, the page is processed correctly (as shown in Figure 7).

Besides static text, the action body can contain any JSP element. A more realistic example of the use of this action is to insert text from a database into a JSP page, without having to worry about how special characters in the text are interpreted by the browser. The tag handler class is very trivial, as shown here:

```
package com.ora.jsp.tags.generic;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import com.ora.jsp.util.*;

public class EncodeHTMLTag extends BodyTagSupport {
```

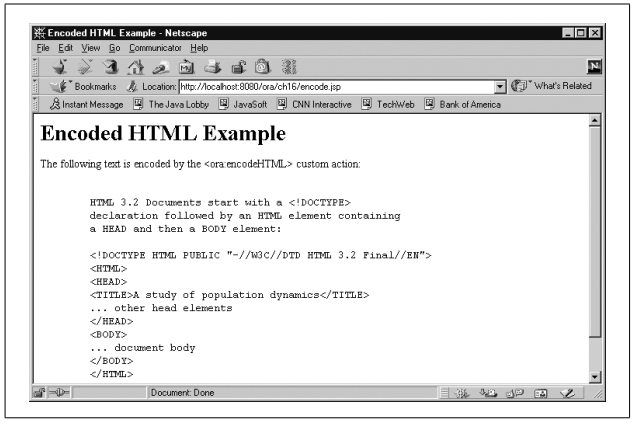


Figure 7. HTML processed by the <ora:encodeHTML> action

```
public int doAfterBody() throws JspException {
    BodyContent bc = getBodyContent();
    JspWriter out = getPreviousOut();
    try {
        out.write(toHTMLString(bc.getString()));
    }
    catch (IOException e) {} // Ignore
    return SKIP_BODY;
}

private String toHTMLString(String in) {
    StringBuffer out = new StringBuffer();
    for (int i = 0; in != null && i < in.length();
      i++) {
        char c = in.charAt(i);
        if (c == '\'') {
            out.append("&#39;");
        }
        else if (c == '\"') {
            out.append("&#34;");
        }
        else if (c == '<') {
            out.append("&lt;");
        }
        else if (c == '>') {
            out.append("&gt;");
```

```
                }
                else if (c == '&') {
                    out.append("&amp;");
                }
                else {
                    out.append(c);
                }
            }
            return out.toString();
        }
    }
```

The action doesn't have any attributes, so the tag handler doesn't need any instance variables and property access methods. The tag handler can reuse all the `BodyTag` methods implemented by the `BodyTagSupport` class except the `doAfterBody()` method.

Two utility methods provided by the `BodyTagSupport` class are used in the `doAfterBody()` method. The `getBodyContent()` method returns a reference to the `BodyContent` object that contains the result of processing the action's body. The `getPreviousOut()` method returns the `BodyContent` of the enclosing action, if any, or the main `JspWriter` for the page if the action is at the top level.

You may wonder why the method is called `getPreviousOut()` and not `getOut()`. The name is intended to emphasize the fact that you want to use the object assigned as the output to the *enclosing* element in a hierarchy of nested action elements. Say you have the following action elements in a page:

```
<xmp:foo>
  <xmp:bar>
    Some template text
  </xmp:bar>
</xmp:foo>
```

The web container first creates a `JspWriter` and assigns it to the `out` variable for the page. When it encounters the `<xmp:foo>` action, it creates a `BodyContent` object and temporarily assigns it to the `out` variable. It then creates another `BodyContent` for the `<xmp:bar>` action and, again, assigns it to `out`. The web container keeps track of this hierarchy of output objects. Template text and output produced by the standard JSP elements end up in the current output object. Each element can access its own `BodyContent` object by calling the `getBodyContent()` method, then read the content. For the `<xmp:bar>` element, the content is the

template text. After processing the content, it can write it to the <xmp:foo> body by getting the BodyContent for this element through the getPreviousOut() method. Finally, the <xmp:foo> element can process the content provided by the <xmp:bar> element and add it to the top-level output object: the JspWriter object it gets by calling the getPreviousOut() method.

The tag handler in this example converts all special characters it finds in its BodyContent object with the toHTMLString() method. Using the getString() method, it gets the content of the BodyContent object and uses it as the argument to the toHTMLString() method. The result is written to the JspWriter obtained by calling getPreviousOut().

The doAfterBody() method in this example returns SKIP_BODY, telling the container to continue by calling doEndTag(). For a tag handler that implements an iterating custom action, doAfterBody() can instead return EVAL_BODY_TAG. The container then evaluates the element's body again, writing the result to the BodyContent for the element, and calls doAfterBody(). The process is repeated until doAfterBody() returns SKIP_BODY.

Actions Creating Objects

Actions can cooperate through objects available in the standard JSP scopes (page, request, session, and application). One example of this type of cooperation is illustrated by the three standard JSP actions: <jsp:useBean>, <jsp:setProperty>, and <jsp:getProperty>. The <jsp:useBean> action creates a new object and makes it available in one of the JSP scopes. The other two actions can then access the properties of the object by searching for it in the scopes. Besides making the object available in one of the scopes, the <jsp:useBean> action also makes it available as a scripting variable, so it can be accessed by scripting elements in the page.

The JSP 1.1 specification states that an attribute named id must be used to name a variable created by an action. The value of the id attribute must be unique within the page. Since it's used as a scripting variable name, it must also follow the variable-name rules for the scripting language. For Java, this

means it must start with a letter followed by a combination of letters and digits and must not contain special characters, such as a dot or a plus sign. An attribute used in another action to refer to the variable can be named anything, but the convention established by the standard actions is to call it `name`.

To create a scripting variable, a custom action must cooperate with the web container. To understand how this works, recall that the JSP page is turned into a servlet by the web container. First, the container needs to generate code that declares the scripting variable in the generated servlet and assigns the variable a value. To do this, it must know the variable name and its Java type. You must provide this information to the container through a `TagExtraInfo` subclass for the custom action. The container calls the `getVariableInfo()` method in the `TagExtraInfo` subclass defined for the custom action when it converts the JSP page to a servlet. This method returns an array of `VariableInfo` instances, providing the required information for the variables created by the custom action. Second, the tag handler class for the custom action must place the object in one of the JSP scopes, using the `PageContext setAttribute()` method. The generated code then uses the `PageContext findAttribute()` method to get the object and assign it to the scripting variable.

TagExtraInfo Class

Class name: `javax.servlet.jsp.tagext.TagExtraInfo`

Extends: None

Implements: None

Implemented by: Internal container-dependent class. Most containers use the reference implementation of the class (developed in the Apache Jakarta project).

Description

For custom actions that create scripting variables or require additional translation time for validation of the tag attributes, a subclass of the `TagExtraInfo` class must be developed and declared

in the TLD. The web container creates an instance of the `TagExtraInfo` subclass during the translation phase.

Constructor

`public TagExtraInfo()`
: Creates a new `TagExtraInfo` instance.

Methods

`public TagInfo getTagInfo()`
: Returns the `TagInfo` instance for the custom action associated with this `TagExtraInfo` instance. The `TagInfo` instance is set by the `setTagInfo()` method (called by the web container).

`public VariableInfo[] getVariableInfo(TagData data)`
: Returns a `VariableInfo[]` array containing information about scripting variables created by the tag handler class associated with this `TagExtraInfo` instance. The default implementation returns an empty array. A subclass must override this method if the corresponding tag handler creates scripting variables.

`public boolean isValid(TagData data)`
: Returns `true` if the set of attribute values specified for the custom action associated with this `TagExtraInfo` instance is valid and `false` otherwise. The default implementation returns `true`. A subclass can override this method if the validation performed by the web container based on the TLD information is not enough.

`public void setTagInfo(TagInfo tagInfo)`
: Sets the `TagInfo` object for this instance. This method is called by the web container before any of the other methods are called.

VariableInfo Class

Class name: `javax.servlet.jsp.tagext.VariableInfo`

Extends: None

Implements: None

Implemented by: Internal container-dependent class. Most containers use the reference implementation of the class (developed in the Apache Jakarta project).

Description

`VariableInfo` instances are created by `TagExtraInfo` subclasses to describe each scripting variable that the corresponding tag handler class creates.

Constructor

`public VariableInfo(String varName, String className, boolean declare, int scope)`
Creates a new instance with the specified values.

Methods

`public String getClassName()`
Returns the scripting variable's Java type.

`public boolean getDeclare()`
Returns `true` if the web container creates a declaration statement for the scripting variable; otherwise, returns `false` (used if the variable has already been declared by another tag handler and is only updated by the tag handler corresponding to the `TagExtraInfo` subclass creating this `VariableInfo` instance).

`public int getScope()`
Returns one of `AT_BEGIN` (makes the scripting variable available from the start tag to the end of the JSP page), `AT_END` (makes the variable available from after the end tag to the end of the JSP page), or `NESTED` (makes the variable available only between the start and stop tags).

`public String getVarName()`
Returns the variable name.

Example

Here's an example of a `TagExtraInfo` subclass for a custom action that creates a variable with the name specified by the `id` attribute and the Java type specified by the `className` attribute:

```
package com.ora.jsp.tags.generic;
import javax.servlet.jsp.tagext.*;
public class UsePropertyTagExtraInfo
  extends TagExtraInfo {
  public VariableInfo[] getVariableInfo(TagData data) {
    return new VariableInfo[] {
```

```
        new VariableInfo(
          data.getAttributeString("id"),
          data.getAttributeString("className"),
          true,
          VariableInfo.AT_END)
      };
    }
  }
```

The web container calls `getVariableInfo()` during the translation phase. It returns an array of `VariableInfo` objects, one per variable introduced by the tag handler.

The `VariableInfo` class is a simple bean with four properties, initialized to the values passed as arguments to the constructor: `varName`, `className`, `declare`, and `scope`. `varName` is simply the name of the scripting variable, and `className` is the name of its class.

The `declare` property is a `boolean`, where `true` means that a brand new variable is created by the action (i.e., a declaration of the variable must be added to the generated servlet). A value of `false` means that the variable has already been created by another action, or another occurrence of the same action, so the generated code already contains the declaration. All the container needs to do in this case is assign a new value to the variable.

The `scope` property has nothing to do with the JSP scopes we have seen so far (page, request, session, and application). Instead, it defines where the new variable is available to JSP scripting elements. A value of `AT_BEGIN` means that it is available from the action's start tag to after the action's end tag. `AT_END` means it is not available until after the action's end tag. A variable with scope `NESTED` is available only in the action's body, between the start and end tags. The scope therefore controls where the variable-declaration and value-assignment code is generated, and the tag handler class must make sure the variable is available in one of the JSP scopes at the appropriate time; e.g., in the `doStartTag()` method for the `AT_BEGIN` and `NESTED` scopes and the `doEndTag()` method for the `AT_END` scope. For a `BodyTag` that iterates over the body, the value can also be updated in the `doAfterBody()` method to provide a new value for each iteration.

Attribute Validation

In the previous example, the `UsePropertyTagExtraInfo` class sets the `varName` and `className` properties of the `VariableInfo` bean to the values of the `id` and `className` attributes specified by the page author in the JSP page. This is done using another simple class named `TagData`, passed as the argument to the `getVariableInfo()` method. The `TagData` instance is created by the web container to provide the `TagExtraInfo` subclass with information about all the action attributes specified by the page author in the JSP page.

A `TagData` instance is also passed as an argument to the `TagExtraInfo` `isValid()` method. This method is called by the web container during the translation phase to allow you to implement validation rules for the custom action's attributes. The container can perform simple validation based on the information available in the TLD about which attributes are required. But a custom action may have optional attributes that are mutually exclusive or that depend on each other. That's when you have to implement the `isValid()` method in a `TagExtraInfo` subclass and provide your own validation code.

The `TagData` class has two methods of interest. The `getAttributeString()` method simply returns the specified attribute as a `String`. But some attributes' values may be specified by a JSP expression—a so-called request-time attribute—instead of a string literal. Since such a value is not known during the translation phase, the `TagData` class provides the `getAttribute()` method to indicate whether an attribute value is a literal string, a request-time attribute, or not set at all. The `getAttribute()` method returns an `Object`. If the attribute is specified as a request-time value, the special `REQUEST_TIME_VALUE` object is returned. Otherwise a `String` is returned, or `null` if the attribute is not set.

TagData Class

Class name: `javax.servlet.jsp.tagext.TagData`

Extends: None

Implements: `Cloneable`

Implemented by: Internal container-dependent class. Most containers use the reference implementation of the class (developed in the Apache Jakarta project).

Description

`TagData` instances are created by the web container during the translation phase. They provide information about the attribute values specified for a custom action to the `TagExtraInfo` subclass for the corresponding tag handler, if any.

Constructors

`public TagData(Object[][] atts)`
: Creates a new instance with the attribute name/value pairs specified by the `Object[][]`. Element 0 of each `Object[]` contains the name; element 1 contains the value or `REQUEST_TIME_VALUE` (if the attribute value is defined as a request-time value, or JSP expression).

`public TagData(java.util.Hashtable attrs)`
: Creates a new instance with the attribute name/value pairs specified by the `Hashtable`.

Methods

`public Object getAttribute(String attName)`
: Returns the specified attribute value as a `String` or as the `REQUEST_TIME_VALUE` object (if the attribute value is defined as a request-time value, or JSP expression).

`public String getAttributeString(String attName)`
: Returns the specified attribute value as a `String`. A `ClassCastException` is thrown if the attribute value is defined as a request-time value (a JSP expression).

`public String getId()`
: Returns the attribute named `id` as a `String`, or `null` if it is not found.

`public void setAttribute(String attName, Object value)`
: Sets the specified attribute to the specified value.

Example

After the web container has checked everything it can on its own based on attribute information in the TLD, it looks for a `TagExtraInfo` subclass, defined by the `<teiclass>` element, for the custom action. If one is defined, it puts all the attribute information in an instance of the `TagData` class and calls the `TagExtraInfo` `isValid()` method:

```
public boolean isValid(TagData data) {
    // Mutually exclusive attributes
    if (data.getAttribute("attr1") != null &&
        data.getAttribute("attr2" != null) {
        return false;
    }

    // Dependent optional attributes
    if (data.getAttribute("attr3") != null &&
        data.getAttribute("attr4" == null) {
        return false;
    }
    return true;
}
```

A `TagExtraInfo` subclass can use the `TagData` instance to verify that all attribute dependencies are okay, as in this example. Unfortunately, in JSP 1.1 there's no way to generate an appropriate error message; the method can only return `false` to indicate that something is not quite right. This will hopefully be rectified in a future version of JSP.

Creating a Tag Library Descriptor

When the web container converts custom action elements into code that creates and calls the correct tag handler, it needs information about which tag handler implements

which custom action element. It gets this information from the Tag Library Descriptor (TLD).

The TLD is an XML file that contains information about all the custom actions in one library. A JSP page that uses custom actions must identify the corresponding TLD and the namespace prefix used for the actions in the page with the `taglib` directive, described in more detail in the next section:

```
<%@ taglib uri="/WEB-INF/tlds/orataglib_1_0.tld"
  prefix="ora" %>
...
<ora:redirect page="main.jsp" />
```

The JSP page then uses the TLD to find the information it needs when it encounters a custom action element with a matching prefix.

Here's an example of part of a TLD:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>ora</shortname>
  <uri>
    /orataglib
  </uri>
  <info>
    A tab library for the examples in the O'Reilly JSP
    book
  </info>

  <tag>
    <name>redirect</name>
    <tagclass>com.ora.jsp.tags.generic.RedirectTag
      </tagclass>
    <bodycontent>JSP</bodycontent>
    <info>
      Encodes the url attribute and possible param tags
      in the body and sets redirect headers.
    </info>
```

```
      <attribute>
        <name>page</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
      </attribute>
    </tag>
    ...
  </taglib>
```

At the top of the TLD file are a standard XML declaration and a `DOCTYPE` declaration specifying the Document Type Definition (DTD) for this file. A DTD defines the rules for how elements in an XML file must be used, such as the order of the elements, which elements are mandatory and which are optional, if an element can be included multiple times, etc. If you're not familiar with XML, don't worry about this. Just remember that you need to copy the first two elements in this example faithfully into your own TLD files. The elements must follow the same order as in this example. Whether an element is mandatory or optional is spelled out in the following element descriptions.

After the two declarations, the first element in the TLD file must be the `<taglib>` element. This is the main element for the TLD, enclosing all the more specific elements that describe the library. Within the body of the `<taglib>` element you can specify elements that describe the library as such, as well as each individual tag handler. Let's start with the five elements that describe the library itself:

`<tlibversion>`
: This mandatory element is used to specify the tag library version. The version should be specified as a series of numbers separated by dots. In other words, you should use the normal conventions for software version numbers (e.g., 1.1, 2.0.3).

`<jspversion>`
: This optional element specifies the version of the JSP specification on which the library depends. The default value is 1.1.

`<shortname>`
: This element is intended to be used by page-authoring tools. It's a mandatory element that should contain the default prefix for the action elements. In the previous example the value is `ora`, meaning that an authoring tool by default generates custom action elements using the `ora` prefix; for instance, `<ora:redirect page="main.jsp">`. If the tool generates the `taglib` directive in the JSP page, authoring tools can also use this element value as the value of the `prefix` attribute. The element value must not include whitespace characters or other special characters, or start with a digit or underscore.

`<uri>`
: This element is also intended to benefit authoring tools. The value can be used as the default value for the `uri` attribute in a `taglib` directive. This element is optional, and it follows the same character rules as the `<shortname>` element.

`<info>`
: This optional element provides a short description of the library; for instance, text a tool may display to help users decide if this is the library they need.

Besides the general elements, the TLD must include at least one `<tag>` element. The `<tag>` element contains other elements that describe different aspects of the custom action:

`<name>`
: This mandatory element contains the unique name for the corresponding custom action element.

`<tagclass>`
: This mandatory element contains the fully qualified class name for the tag handler class.

`<teiclass>`
: This optional element is used to specify the fully qualified class name for the `TagExtraInfo` subclass, if the

action introduces variables or needs to do additional syntax validation (as described in the next section).

`<bodycontent>`
: This optional element can contain one of three values. A value of `empty` means that the action body must be empty. If the body can contain JSP elements, such as standard or custom actions or scripting elements, use the `JSP` value. All JSP elements in the body are processed, and the result is handled as specified by the tag handler (i.e., processed by the tag handler or sent through to the response body). This is also the default value, in case you omit the `<bodycontent>` element. The third alternative is `tagdependent`. This value means that possible JSP elements in the body will *not* be processed. Typically, this value is used when the body is processed by the tag handler, and the content may contain characters that could be confused with JSP elements, such as `SELECT * FROM MyTable WHERE Name LIKE '<%>'`. If a tag that expects this kind of body content is declared as `JSP`, the `<%>` is likely to confuse the web container. Use the `tagdependent` value to avoid this risk of confusion.

`<info>`
: This optional element can be used to describe the purpose of the action.

The `<tag>` element must also contain an `<attribute>` element for each action attribute. The `<attribute>` element contains the following nested elements to describe the attribute:

`<name>`
: This mandatory element contains the attribute name.

`<required>`
: This optional element tells if the attribute is required. The values `true`, `false`, `yes`, and `no` are valid, with `false` being the default.

`<rtexprvalue>`
: This optional element can have the same values as the `<required>` element. If it's `true` or `yes`, a request-time attribute expression can specify the attribute value; for instance, `attr="<%= request.getParameter("par") %>"`. The default value is `false`.

Packaging and Installing a Tag Library

During development, you may want to let the tag library classes and the TLD file reside as-is in the filesystem. This makes it easy to change the TLD and modify and recompile the classes. If you do so, make sure the class files are stored in a directory that's part of the classpath for the web container, such as the *WEB-INF/classes* directory for the web application. The TLD must also be available in a directory where the web container can find it. The recommended location is the *WEB-INF/tlds* directory. To identify the library with the TLD stored in this location, use a `taglib` directive like this in the JSP pages:

```
<%@ taglib uri="/WEB-INF/tlds/orataglib_1_0.tld"
  prefix="ora" %>
```

Here the `uri` attribute refers directly to the TLD file's location.

When you're done with the development, you may want to package all the tag handler classes, `TagExtraInfo` classes, and beans used by the tag handler classes, plus the TLD, in a JAR file. This makes it easier to install the library in an application. The TLD must be saved as */META-INF/taglib.tld* within the JAR file.

To create the JAR file, first arrange the files in a directory with a structure like this:

```
META-INF/
  taglib.tld
com/
  ora/
```

```
jsp/
  tags/
    generic/
      EncodeHTMLTag.class
      ...
  util/
    StringFormat.class
    ...
```

The structure for the class files must match the package names for your classes. A few of the classes in the tag library for this book are shown here as an example.

With the file structure in place, use the *jar* command to create the JAR file:

```
jar cvf orataglib_1_0.jar META-INF com
```

This command creates a JAR file named *orataglib_1_0.jar* containing the files in the *META-INF* and *com* directories. Use any JAR filename that makes sense for your own tag library. Including the version number for the library is a good idea, since it makes it easier for users to know which version of the library they are using.

You can now use the packaged tag library in any application. Just copy the JAR file to the application's *WEB-INF/lib* directory and use a `taglib` directive like this in the JSP pages:

```
<%@ taglib uri="/WEB-INF/lib/orataglib_1_0.jar"
  prefix="ora" %>
```

Note that the `uri` attribute now refers to the JAR file instead of the TLD file. A JSP 1.1 container is supposed to be able to find the TLD file in the JAR file, but this is a fairly recent clarification of the specification. If the web container you use doesn't support this notation yet, you have to extract the TLD file from the JAR file, save it somewhere else—for instance, in *WEB-INF/tlds*—and let the `uri` attribute refer to the TLD file instead.

An alternative to letting the `taglib` directive point directly to the TLD or JAR file is specifying a symbolic name as the `uri` attribute value and providing a mapping between this name

and the real location in the *WEB-INF/web.xml* file for the application:

```
<%@ taglib uri="/orataglib" prefix="ora" %>
```

The *WEB-INF/web.xml* file must then contain the following elements:

```
<web-app>
  ...
  <taglib>
    <taglib-uri>
      /orataglib
    </taglib-uri>
    <taglib-location>
      /WEB-INF/lib/orataglib_1_0.jar
    </taglib-location>
  </taglib>
  ...
</web-app>
```

The `<taglib-uri>` element contains the symbolic name, and the `<taglib-location>` element contains the path to either the JAR file or the extracted TLD file.

The Web Archive (WAR) File

The portable distribution and deployment format for a web application defined by the servlet specification is the Web Archive (WAR). All Servlet 2.2–compliant servers provide tools for installing a WAR file and associating the application with a servlet context.

A WAR file has a *.war* file extension and can be created with the Java *jar* command or a ZIP utility program such as *WinZip*. The internal structure of the WAR file is defined by the servlet specification as:

```
/index.html
/company/index.html
/company/contact.html
/company/phonelist.jsp
/products/searchform.html
/products/list.jsp
```

```
/images/banner.gif
/WEB-INF/web.xml
/WEB-INF/lib/bean.jar
/WEB-INF/lib/actions.jar
/WEB-INF/classes/com/mycorp/servlets/PurchaseServlet.class
/WEB-INF/classes/com/mycorp/util/MyUtils.class
/WEB-INF/tlds/actions.tld
```

The top level in this structure is the document root for all application web page files. This is where you place all your HTML pages, JSP pages, and image files. All these files can be accessed with a URI starting with the context path. For example, if the application was assigned the context path `/sales`, the URI `/sales/products/list.jsp` would be used to access the JSP page named *list.jsp* in the *products* directory.

The *WEB-INF* directory contains files and subdirectories for other types of resources. Two *WEB-INF* subdirectories have special meaning: *lib* and *classes*. The *lib* directory contains JAR files with Java class files; for instance, JavaBeans classes, custom action handler classes, and utility classes. The *classes* directory contains class files that are not packaged in JAR files. The servlet container automatically has access to all class files in the *lib* and *classes* directories (in other words, you don't have to add them to the `CLASSPATH` environment variable).

If you store class files in the *classes* directory, they must be stored in subdirectories mirroring the package structure. For example, if you have a class named `com.mycorp.util.MyUtils`, you must store the class file in *WEB-INF/classes/com/mycorp/util/MyUtils.class*.

The *WEB-INF* directory can also contain other directories. For instance, a directory named *tlds* is by convention used for tag library TLD files that are not packaged within the tag library JAR file.

During development, it's more convenient to work with the web application files in a regular filesystem structure than to create a new WAR file every time something changes. Most

containers therefore support the WAR structure in an open filesystem as well.

The *WEB-INF/web.xml* file is an important file. It is the application deployment descriptor that contains all the configuration information for an application. If your application consists of only JSP and HTML files, you typically do not need to worry about this file. But if the application also contains servlets or tag libraries, or uses the container-provided security mechanisms, you often need to define some configuration information in the *web.xml* file. A description of the elements in the *web.xml* file is beyond the scope of this reference. Please see the Servlet 2.2 specification instead.