CHAPTER 5

JSP Tag Libraries

THE TERM *TAG* IS CENTRAL to markup languages such as HTML and XML. A markup document grants meaning to text by surrounding or preceding it with a set of characters collectively called a *tag*. In HTML and XML documents, tags have the form `<text>` where *text* defines the meaning of the tag. For instance, a HTML first-level header may be created simply by surrounding the text with header tags: (`<h1>This is a first-level header</h1>`). JSP technology provides a standard mechanism whereby a programmer can create his/her own server-side tags. These *JSP tags* are evaluated on the server, and their results are sent back to the client browser.

This chapter describes how to create, use, and structure JSP tag libraries for the three existing types of JSP tags: standalone, iterative, and body content tags.

Describing Tags

Tags are displayed by a rendering engine on the client, which converts the cryptic text to a layout presented to the user. All HTML browsers contain a rendering engine—even the text-only browser Lynx highlights text made “bold” by a HTML tag pair. One may, therefore, regard the tag as an encapsulation of client-side rendering information; it is much simpler for the HTML author to include a tag than a full specification of what the client-side rendering engine should do with the text.

Pondering the situation a minute, you may quickly realize that the same principle holds true for encapsulating server-side information. Many server-side documents contain complex data manipulation instructions; hiding the complexity behind a simpler construct makes for a simpler overview of the document. Figure 5-1 illustrates two ways of inserting a string of characters into a JavaServer Pages (JSP) document; although both versions perform the same task, the custom tag hides most of the complexity (or, more to the point, the code mass) shown in the code block. In general, it is simpler to quickly get an overview of a JSP document containing several custom tags than multiple code blocks.

Of course, there has to be a catch to the beautiful story of the custom JSP tag. You may rightfully wonder where the information logic invoked by the custom tag illustrated in Figure 5-1 is placed. If it’s not in the JSP document, then where is it? The simple answer is that you may create *libraries* of tag handler classes, which must be placed in the classpath of the Web application to be usable. Frequently, one copies such a tag library—in the form of a JAR file—to the `WEB-INF/lib` directory of the Web

Code block (Unencapsulated)

```
<%
   // This block will output a length of
   // concatenated characters to the output
   // stream out (connected to the response
   // output writer).

   StringBuffer buf = new StringBuffer();
   for(char i ='a'; i < stopChar; i++)
   {
      // Printout the character with the
      // provided index in the alphabet
      buf.appendix(i);
   }

   // Print.
   out.println(buf.toString());
%>
```

Custom Tag (Encapsulated)

```
<homeMadeTags:printChars stopChar="m" />
```

Figure 5-1. Compare the two ways to specify printing a set of characters to the browser. If both ways accomplished the same thing, which notation would be simplest?

application. Using custom tags ensures low-impact portability between Web applications because the full functionality of a set of custom JSP tags resides in the JAR file. As an alternative, the bytecode files of the tag handler classes may be placed in the classpath of the Web application.

As a conclusion, there are several good reasons to use custom JSP tags in JSP documents, instead of raw code blocks:

- **Simplicity**. Custom JSP tags have better encapsulation and expose only the most relevant information. Reducing information clutter improves efficiency in learning and usage.
- **Protection**. Custom JSP tags hide the code statements; an information editor altering the text of a JSP document may accidentally modify text within a block of code—with disastrous effects on the Web application. Custom tags prevent such accidental modification.
- **Portability**. Custom JSP tags grant portability because their functionality (normally placed within an archive) may be moved between Web applications and/or Web application servers in a simple manner.
- **Familiarity**. Custom tags are well-known building blocks to information architects and HTML hackers—Java code is generally not. Presenting another building block that looks similar to a normal tag is likely not to summon a storm of protests; adding "strange code" in JSP documents may.

968c05.fm Page 365 Tuesday, October 30, 2001 6:56 PM

What Is a Custom Action?

You may have heard the phrase *custom action*, which is another name for a JSP custom tag. Because custom JSP tags look similar to a standard JSP action, it is plain to see how the nickname arose.

Having seen an example of a custom tag in Figure 5-1, let us now investigate how to describe custom JSP tags in general.

There are two custom JSP tag types: *empty actions,* which are standalone and contain no bodies, and *non-empty actions,* which are containers that have a body (see Figure 5-2). Of course, the body of a non-empty tag may be completely empty—but it must exist for a non-empty tag. To facilitate things, the terminology used within this book is the same as used within the Tag Library API (in other words, `package javax.servlet.jsp.tagext`).

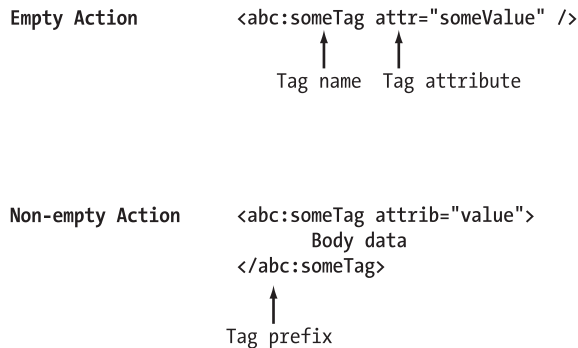

Figure 5-2. The definitions pertaining to the structure of empty (standalone) and non-empty (container) JSP tags. The definitions in this image (that follows the JSP 1.2 specification terms) will be used throughout this chapter.

Now investigate how to create powerful, platform-independent, and reusable custom JSP actions. In fact, creating a tag library is one of the better uses of encapsulation to guide developers in creating reusable JSP custom tags. In turn, custom tags reduce the amount of errors generated by JSP developers in server-side documents.

The JSP tag library system has four parts:

- *The tag handler class* implements the life-cycle methods required for all tag handlers by the JSP specification. The methods of the tag handler perform whatever custom behavior the developer has created. The tag handler class must be deployed into the classpath of the Web application which uses it; normally the tag handler classes are deployed into the `WEB-INF/classes` directory or packaged into a JAR file in the `WEB-INF/lib` directory.

- *The tag library descriptor* (TLD) document contains XML definitions mapping tag handler classes to logical tag names. Such logical names are the equivalent of an internal servlet name in the `web.xml` configuration document and must be unique within each TLD. The TLD document may have any name but is frequently called `taglib.tld`. In the JSP 1.1 specification, the TLD file must be placed in the `META-INF` directory and named `taglib.tld`, but the JSP 1.2 specification permits arbitrary placement and name.

- *The web.xml* configuration document may define several taglib constructs. Each taglib construct maps a TLD document to a logical Uniform Resource Identifier (URI) that must be unique within the Web application.

- The JSP document may contain one or more *JSP TLD directives* that map a URI defined in the `web.xml` to a unique prefix string for use within the document. This mapping is generally convenient because URIs may be rather long and awkward to type repeatedly within the JSP document; a prefix may be shorter and, consequently, more convenient.

XML in the J2EE World

Configuration settings used by Java 2 Enterprise Edition (J2EE) components and servers are frequently XML-formatted documents. XML documents, like HTML, are markup documents containing a set of tags that are defined in a document type definition (DTD). The DTD document is used as a template that defines an XML data tree, such as which tags are permitted within other tags. Although most of the XML examples within this book are explained in detail, you may want to visit `http://www.xml.org` for more information on XML.

Figure 5-3 shows a skeleton content version of the four files mentioned in the preceding bulleted list. A JSP document containing a particular `<%@ taglib … %>` directive uses the `uri` attribute of the taglib directive to locate the archive containing the TLD file. Unless the `uri` is a hyperlink to an actual TLD document,

968c05.fm Page 367 Tuesday, October 30, 2001 6:56 PM

the hyperlink contains a logical name for the TLD file. The `web.xml` configuration document is consulted to find the physical location of the TLD file requested. Using the definitions in the TLD file, a tag handler class is located. Moreover, the taglib directive defines a prefix (“`apress`” in the image) used in the JSP file to identify all tags from a particular TLD—when that prefix is encountered in a custom tag in a JSP document, the JSP engine knows which TLD to query for a tag definition. The class is resolved and instantiated to enable a method call to its life-cycle method.

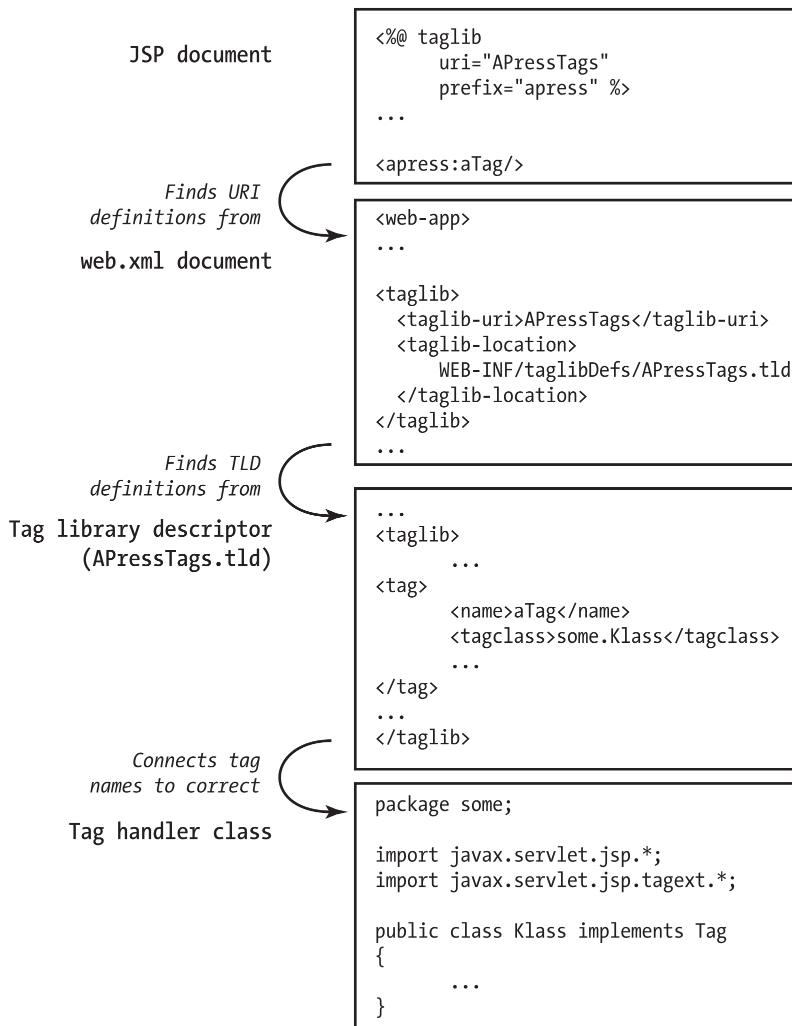

Figure 5-3. The tag handler class, TLD, `web.xml` *document, and JSP document work together. The JSP engine creates an instance of the correct tag handler class and invokes a set of methods within it whenever a particular tag is found within the JSP document. This figure shows the conversion between the name and prefix of the custom JSP tag and the tag handler class.*

The taglib directive may use four different URI attribute types to locate the TLD file. Those attributes may contain the following items:

- URL to a Web server from which the TLD file may be downloaded. An example of such a taglib URI is `<%@ taglib uri="http://www.apress.com/servletbook/tlds/examples.tld" prefix=”dummy” %>`.

- An absolute local Web application path, which originates from the root of the local Web application. Such URI attributes must start with a forward slash (`/`); an example is `<%@ taglib uri="/some/path/theExamples.tld" prefix="examples" %>`.

- Relative local Web application path, which originates from the directory where the JSP document resides. Such URI attributes may neither start with a forward slash nor a `http://` prefix. An example of a relative address taglib directive is `<%@ taglib uri="aDir/theExamples.tld" prefix="moreExamples” %>`.

- Arbitrary string, provided that the string is defined as a `<taglib-uri>` in the `web.xml` configuration file. Note that the `uri` attribute in the taglib directive must match the `web.xml` definition exactly, as in `<%@ taglib uri="accountingTags" prefix="accTags" %>`, providing that the `web.xml` file contains a taglib entry with the `taglib-uri` entry being `<taglib-uri>accounting Tags</taglib-uri>`.

The first of these options, loading the TLD document directly from a URL address somewhere on the Web, is considerably slower and more unpredictable than the other three options that loads the TLD from a file in the local file system. For industrial-strength systems, therefore, the second and fourth are used more frequently.

The JSP definition is quite liberal regarding the deployment placement of most documents described previously. Apart from the `web.xml` document that must be placed in the `WEB-INF` directory, all other documents may be placed arbitrarily. As the tag handler classes must reside in the `CLASSPATH` to be loaded by the Java Virtual Machine (JVM) running the servlet engine, they are frequently found in `WEB-INF/classes` or packaged in a JAR file in `WEB-INF/lib`.

You can see an example of a simple JAR file structure containing a full tag library in Figure 5-4. The `FirstTag.class` is a compiled tag handler class, which is invoked whenever a particular tag is found within the JSP document. Although it may be good practice to place TLD files in the `META-INF` directory, the JSP specification permits placing the TLD file in an arbitrary directory.

```
META-INF/
META-INF/MANIFEST.MF
META-INF/taglib.tld
se/jguru/tags/FirstTag.class
```

Figure 5-4. The content of a JAR file containing a tag library definition (`taglib.tld`*) and the tag handler class (*`FirstTag.class`*). In addition to the two files previously, the normal* `MANIFEST.MF` *file of a JAR is present. This structure is compliant with the JSP 1.1 deployment requirement because the TLD file is named* `taglib.tld` *and placed in the* `META-INF` *directory.*

When developing an industrial-strength library of custom tag handler classes, you should strive for providing a tag handler class name that would let the developer realize what function the tag performs. In the example shown in Figure 5-4, however, the main concern is to grant understanding for the deployment of all files being part of the custom JSP tag system, so we have relaxed the demand for usability in this example.

The Life Cycle of a Tag Handler Class

JSP tag handler classes behave much like servlets; they have a strict life cycle where methods are called in a specific order. The life cycle of JSP tag handlers is more difficult to grasp because the methods are found in four different interfaces/classes:

- The `Tag` interface defines life-cycle methods common to all tag handler classes. These life cycle methods define methods `doStartTag` and `doEndTag` that are called by the JSP engine to start and stop the evaluation of a tag. Also, the `Tag` interface defines JavaBean accessor methods to get and set `pageContext` and parent variables. The `Tag` interface should be implemented for empty tags.

- The `IterationTag` interface extends the `Tag` interface and provides an extra method (`doAfterBody`), which may be called repeatedly by the JSP engine when iterating over the body of the `IteratorTag`. Use the `IterationTag` to create a JSP custom tag that requires body iteration. `IterationTag`s are frequently used when generating HTML or XML lists that should be output to the client.

- The `BodyTag` interface extends the `Tag` interface and provides additional methods for non-empty tags only. The JSP tag handler class should implement the `BodyTag` interface if you want to manipulate the body content of the JSP tag.

- The tag handler implementation class provides implementations for the standard `Tag` life-cycle methods in addition to standard methods for setting (and getting) JavaBean properties. Such properties are automatically set by the JSP engine if the TLD file defines attributes for a tag. You'll learn more about tag attributes in sections "The Ideal Life Cycle for an Empty Tag with Attributes," "The Ideal Life Cycle for an IterationTag with Attributes," and "Tag Definitions."

The life cycle of a tag differs slightly depending on its type; empty tags without any attributes have a simpler life cycle than non-empty container tags with many attributes. Let's take a look at each type in turn to investigate the effect on tag life cycle that different constructs have.

Tag Interface

Figure 5-5 shows the life cycle of a handler class implementing the `javax.servlet.jsp.tagext.Tag` interface. During initialization, the tag handler class is created and configured, where all its JavaBean setter methods are called with the values provided in the TLD configuration document (`taglib.tld`, for instance). JavaBean setter methods are represented with the message "setXXX()". During runtime, the `doStartTag` and `doEndTag` methods are called and the actual processing of the tag is done. The four states of a tag handler class are:

- **Created**. This is the state entered after the default (parameterless) constructor is invoked. All internal variables in the tag handler instance just created have their default values, as set programmatically in the constructor. The created state is transient, in the sense that the tag handlers need additional setup methods called to be properly configured.

- **Configured**. Having created the tag handler instance, the JSP engine must configure it according to the directions found in the TLD configuration file. Configuration is performed by calling the default setter methods of a JSP engine, as well as any setter methods for JavaBean properties defined for the tag in the TLD configuration file.

- **Executing**. In runtime, the JSP engine invokes the `doStartTag` and `doEndTag` methods in the tag handler class. The return value of the `doStartTag` method indicates whether the body text (if existent) should be included in the JSP output. The `doStartTag` method is called before and the `doEndTag` after including the body text in the output, if applicable.

- **Undefined**. Having being released to an instance pool, the tag handler instance is set in an undefined state and cannot be used before again being properly configured by calls to `setParent`, `setPageContext`, and the declared JavaBean property setter methods. The JSP engine guarantees that the release method will be called in a tag handler before garbage collection sets in.

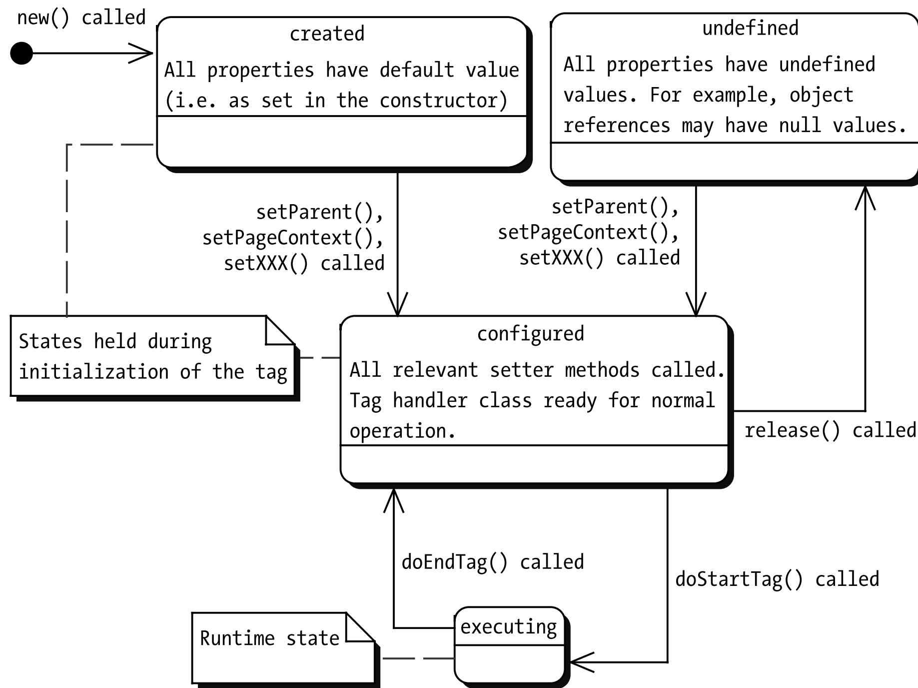

Figure 5-5. The life cycle for empty tags, where the tag handler class implements the `javax.servlet.jsp.tagext.Tag` *interface*

The Ideal Life Cycle for an Empty Tag without Attributes

State diagrams are wonderful for visualizing all possible scenarios for a particular instance. However, the simplicity of code may occasionally be lost in the wonderful box-and-line landscape of the Unified Modeling Language (UML). The ideal life cycle of a tag handler for a standalone tag without attributes is rather simple (see Figure 5-6).

The two first method calls provide the tag handler instance with references to the `PageContext` of the JSP document and the handler instance of a tag container being the parent of `theTagHandler`. A tag handler class is a parent of another if its JSP tag contains the second tag. In the following example, calling

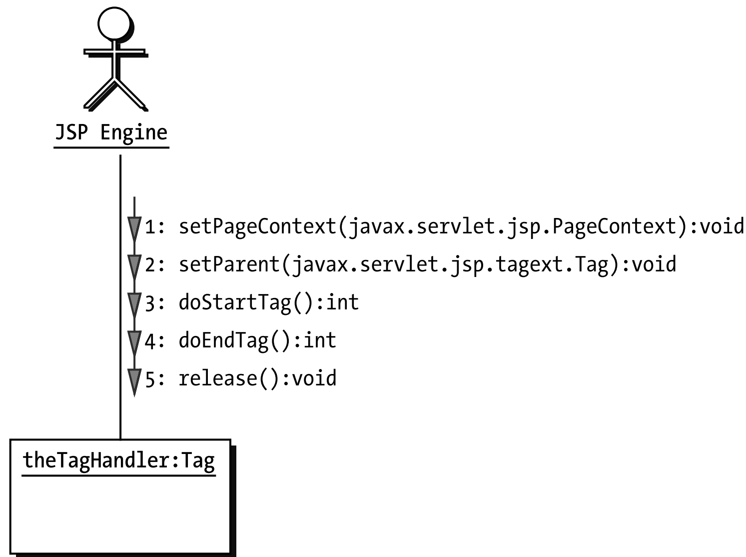

Figure 5-6. The life cycle of a standalone tag without any attributes, drawn as a UML collaboration diagram

`getParent` in the handler instance for `theChildTag` would return a reference to the handler instance for `theParentTag`:

```
<x:theParentTag>
        <!-- This is the body of theParentTag -->
        <x:theChildTag />
</x:theParentTag>
```

Methods 3 and 4, `doStartTag` and `doEndTag` from in Figure 5-6 are called when the JSP engine encounters the tag start and end respectively. For empty tags, there is no real difference between the two—but for non-empty tags the difference is obvious.

When done processing, the JSP engine calls the `release` method. This method has the same purpose as the `finalize` method of `java.lang.Object` in that it provides a standard place where the developer may free all objects held during the tag evaluation. Also, in the `release` method, the internal state of the tag handler instance is released, as shown in Figure 5-6.

The Ideal Life Cycle for an Empty Tag with Attributes

The typical life cycle of a tag handler for an empty tag accepting attributes differs slightly from the life cycle of an empty tag lacking attributes. In the case illustrated in Figure 5-7, the tag descriptor for the tag handler `attributeTag` has declared that

968c05.fm Page 373 Tuesday, October 30, 2001 6:56 PM

the tag has an attribute (in other words, a JavaBean property) called `output`. The state chart diagram in Figure 5-7 is identical for empty tags with and without attributes.

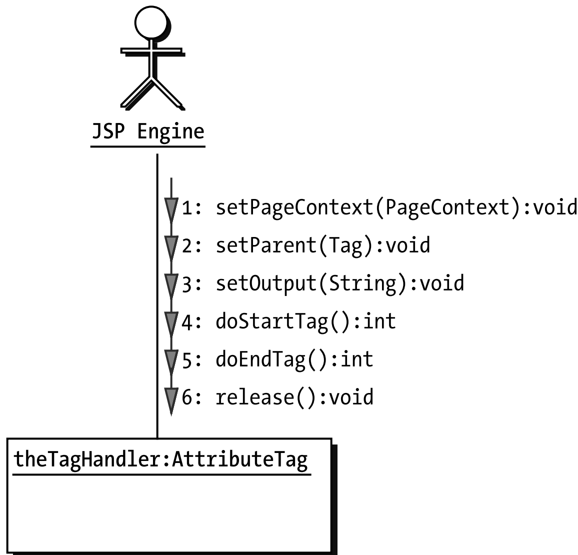

Figure 5-7. The life cycle of a standalone tag accepting one attribute (in other words, JavaBean property), drawn as a UML collaboration diagram. Note that the only difference between the two life cycles described in Figures 5-6 and 5-7 is the call to the `setOutput` *method, corresponding to setting a JavaBean property.*

Thus, between the `setParent` and the `doTagStartTag` methods, the JSP engine calls all JavaBean setter methods (in our case, `setOutput`) to initialize the state of the handler. Should any other JavaBean properties be declared in the tag library descriptor, the corresponding setter methods are invoked accordingly. Such setter methods are only invoked if an attribute is actually supplied within the JSP document, for example:

```
<!-- Setter method invoked. -->
<x:someTagName output="Some output" />

<!-- Setter method not invoked. -->
<x:someTagName />
```

From Figure 5-7 and the preceding listing, it is assumed that the tag `<x:someTagName />` name will invoke the life-cycle method calls on an instance of the `AttributeTag` class. However, the TLD linking the two for usage in the

968c05.fm Page 374 Tuesday, October 30, 2001 6:56 PM

previous code snippet is not provided here; the syntax of the TLD file will be discussed in "Tag Library Descriptors," later in this chapter.

Having dealt with tag handlers implementing the `Tag` interface that may or may not have registered attributes, proceed to study the life cycle for tag handlers that implement the `IterationTag` interface.

IterationTag Interface

The life cycle of a handler class implementing the `javax.servlet.jsp.tagext.IterationTag` interface is similar to the life cycle for tag handlers implementing the `Tag` interface. As shown in Figure 5-8, the initialization states of the `Tag` implementation class depend on the exact type of tag. The five states of an `IterationHandler` class are as follows:

- **Created**. This is the state entered after the default (parameterless) constructor is invoked. All internal variables in the tag handler instance just created have their default values, as set programmatically in the constructor. The created state is transient, in the sense that the tag handlers need additional setup methods called to be properly configured.

- **Configured**. Having created the tag handler instance, the JSP engine must configure it according to the directions found in the TLD configuration file. Configuration is performed by calling the default setter methods of a JSP engine, as well as any setter methods for JavaBean properties defined for the tag in the TLD configuration file.

- **Executing**. In runtime, the JSP engine invokes the `doStartTag` and `doEndTag` methods in the tag handler class. The return value of the `doStartTag` method indicates whether the body text (if existent) should be included in the JSP output. The `doStartTag` method is called before and the `doEndTag` after including the body text in the output, if applicable.

- **Undefined**. Having being released to an instance pool, the tag handler instance is set in an undefined state and cannot be used before again being properly configured by calls to `setParent`, `setPageContext`, and the declared JavaBean property setter methods. The JSP engine guarantees that the release method will be called in a tag handler before garbage collection sets in.

- **Evaluating body**. If the `doStartTag` method invoked in the configured state returned `Tag.EVAL_BODY_INCLUDE`, the body of the non-empty tag is evaluated. Having evaluated the body, the `doAfterBody` method is called—and the `IterationTag` will keep evaluating the body as long as the `doAfterBody` returns `IteratorTag.EVAL_BODY_AGAIN`. When done iterating, the `doAfterBody` method should return `Tag.SKIP_BODY`.

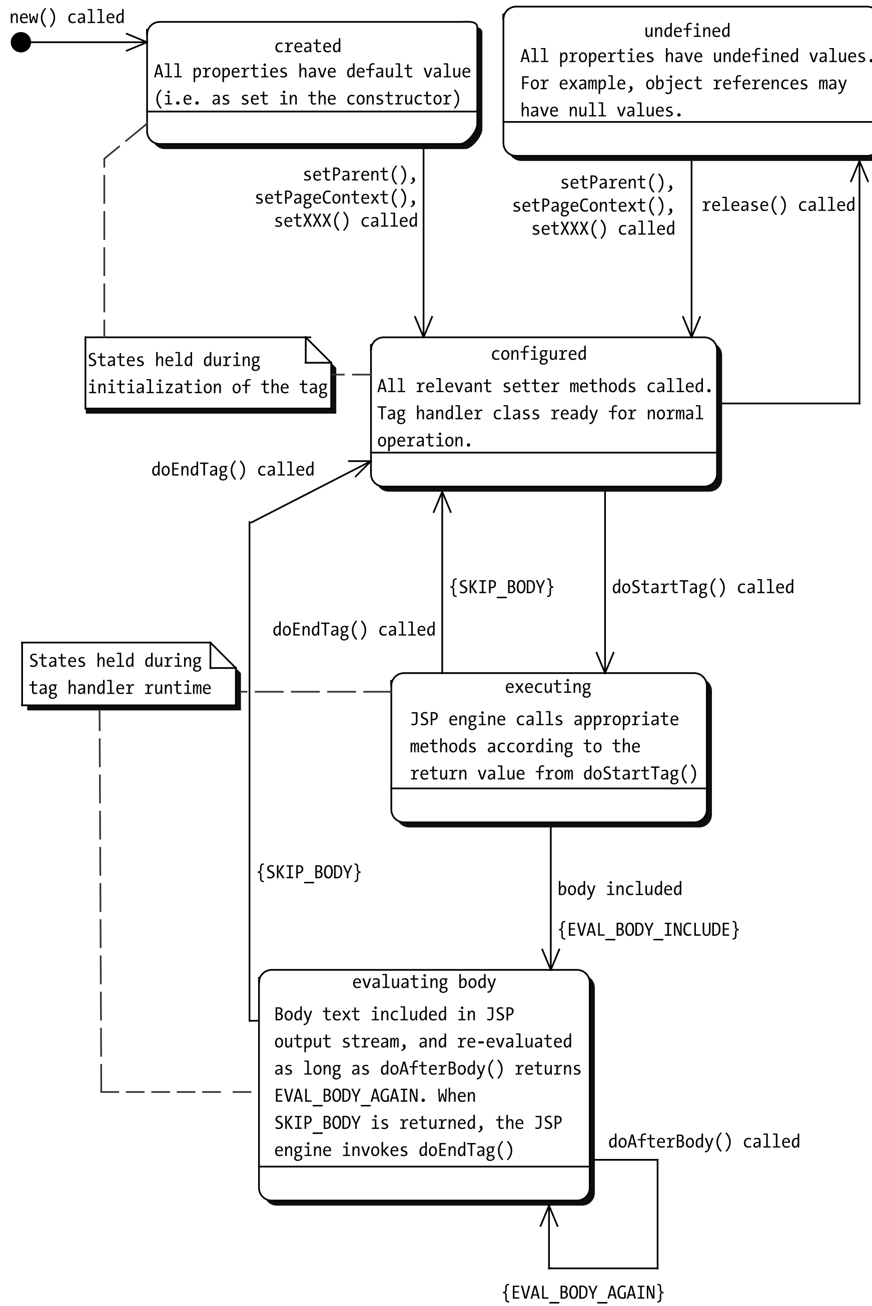

Figure 5-8. The five states of the `IterationTag` *handler class shown with the return values required to enter a given state. As shown, the evaluating body state will be preserved as long as the* `doAfterBody` *method returns* `IterationTag.EVAL_BODY_AGAIN`*. The main purpose of* `IterationTags` *is to simplify generating lists of output in the JSP view.*

The Ideal Life Cycle for an IterationTag without Attributes

The ideal life cycle of a tag handler for a non-empty tag (implementing the `IterationTag` interface) without attributes is rather simple (see Figure 5-9).

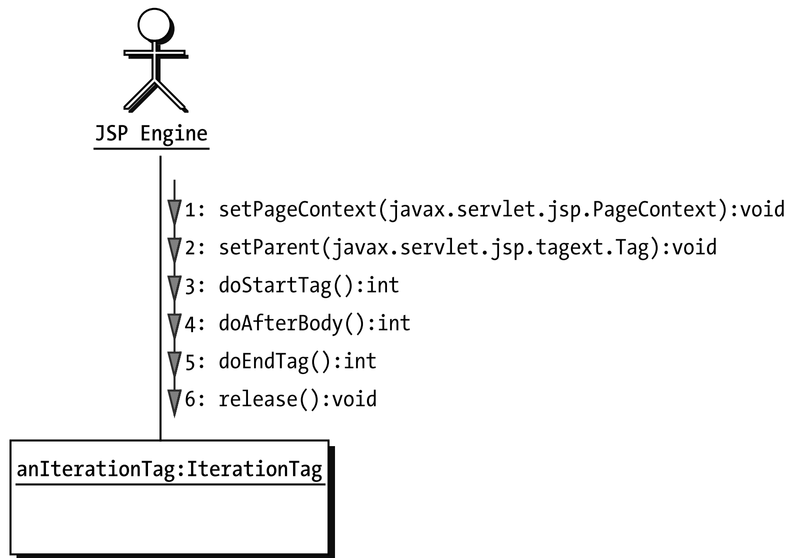

Figure 5-9. The life cycle of an `IterationTag` *without any declared attributes, drawn as a UML collaboration diagram. If you compare the ideal lifecycle to the one drawn for tag handler classes implementing the* `Tag` *interface, the* `doAfterBody` *method is added for* `IterationTags`.

Identical to the case of an empty custom JSP tag whose handler implements the `Tag` interface, the two first method calls provide the `IterationTag` handler instance with references to the `PageContext` of the JSP document and its parent. Methods 3 and 5 (`doStartTag` and `doEndTag` from Figure 5-9) are called when the JSP engine encounters the opening tag of the `IterationTag` and when the last iteration is just performed, respectively.

The major difference between tag handler classes that implement the `Tag` and `IterationTag` interfaces is, of course, the call to `doAfterBody` that is performed once after every evaluation of the body content. The return value of the `doAfterBody` method defines if the `IterationTag` will continue looping (`IterationTag.EVAL_BODY_AGAIN` is returned) or not (`Tag.SKIP_BODY` is returned).

When done iterating over the body text, the JSP engine calls the `release` method in the tag handler class, in an identical manner to the `Tag` handler.

The Ideal Life Cycle for an IterationTag with Attributes

As can be expected, the only difference between the ideal life cycles of IterationTags with and without attributes is that the JSP engine will attempt to call all setter methods for non-null JavaBean properties (see Figure 5-10).

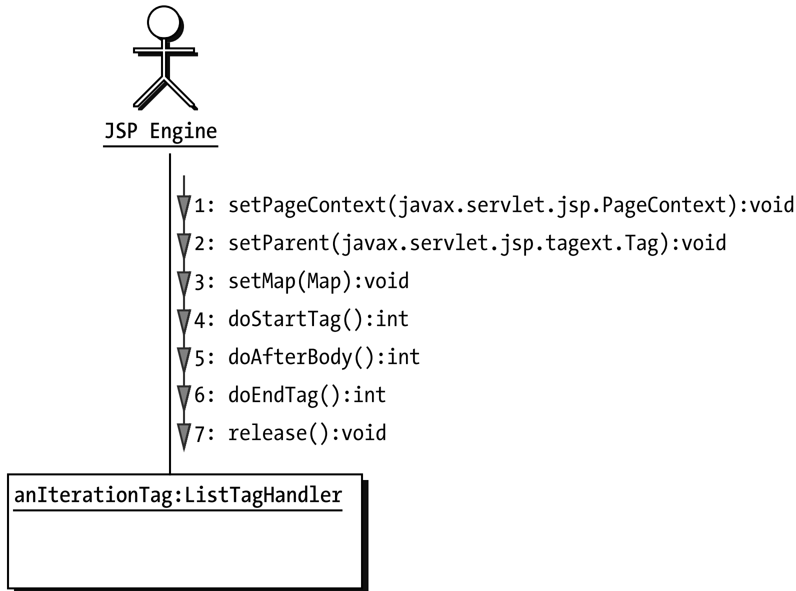

Figure 5-10. The life cycle of an IterationTag *with one declared attribute (the JavaBean property* map *of type* Map*), drawn as a UML collaboration diagram*

Identical to the case for a handler class implementing the Tag interface, all registered attributes for the IterationTag are set before the JSP engine calls the doTagStartTag method. The setter method, setOutput, is invoked to make the handler class do a state transition from *created* to *configured*, as shown in Figure 5-8. Of course, the setter methods are only invoked if an attribute is actually supplied within the JSP document. Assuming that the someIterationTagName is mapped to the ListTagHandler class shown in Figure 5-10, only the first of the two following code snippets will cause the JSP engine to call the setMap method:

```
<!-- Setter method invoked. -->
<x:someIterationTagName map="<%= aMap %>" />

<!-- Setter method not invoked. -->
<x: someIterationTagName />
```

The last subinterface to the `Tag` class is the `BodyTag` interface, which adds functionality that provides control over the body content.

BodyTag Interface

The life cycle of a handler class implementing the `javax.servlet.jsp.tagext.BodyTag` interface is similar to the life cycle for tag handlers implementing the `IterationTag` interface. As shown in Figure 5-11, the initialization states of the `BodyTag` handler are as follows:

- **Created**. This is the state entered after the default (parameterless) constructor is invoked. All internal variables in the tag handler instance just created have their default values, as set programmatically in the constructor. The created state is transient, in the sense that the tag handlers need additional setup methods called to be properly configured.

- **Configured**. Having created the tag handler instance, the JSP engine must configure it according to the directions found in the TLD configuration file. Configuration is performed by calling the default setter methods of a JSP engine, as well as any setter methods for JavaBean properties defined for the tag in the TLD configuration file.

- **Executing**. In runtime, the JSP engine invokes the `doStartTag` and `doEndTag` methods in the tag handler class. The return value of the `doStartTag` method indicates whether or not the body text (if existent) should be included in the JSP output. The `doStartTag` method is called before and the `doEndTag` after including the body text in the output, if applicable.

- **Undefined**. Having being released to an instance pool, the tag handler instance is set in an undefined state and cannot be used before again being properly configured by calls to `setParent`, `setPageContext`, and the declared JavaBean property setter methods. The JSP engine guarantees that the release method will be called in a tag handler before garbage collection sets in.

968c05.fm Page 379 Tuesday, October 30, 2001 6:56 PM

- **Setting body content**. If the `doStartTag` method returns `BodyTag.EVAL_BODY_BUFFERED`, the tag handler class proceeds to creating a buffer containing the body content of the tag, encapsulated in a `javax.servlet.jsp.tagext.BodyContent` object. The `BodyTag` handler class may then modify the buffered contents of the `BodyContent` and output the result to the JSP document, replacing the original body text. The `setBodyContent` method is not to be overridden by developers.

- **Preparing body**. The `doInitBody` method is called after the body content is set but before evaluation of the tag body has started. Thus, you may modify the buffered body text as retrieved from the JSP custom tag before starting to iterate and evaluate in the *Evaluating body* state.

- **Evaluating body**. If the `doStartTag` method invoked in the configured state returned `Tag.EVAL_BODY_INCLUDE`, the body of the non-empty tag is evaluated. Having evaluated the body, the `doAfterBody` method is called—and the `IterationTag` will keep evaluating the body as long as the `doAfterBody` returns `IteratorTag.EVAL_BODY_AGAIN`. When done iterating, the doAfterBody method should return `Tag.SKIP_BODY`.

For a non-empty tag, the state diagram and typical life cycle of its handler class differs from what has been described for `IterationTags` earlier in this chapter. Two new methods, `setBodyContent` and `doInitBody` are introduced; you are wise to leave the `setBodyContent` method as is, and focus on providing your own implementation for the `doInitBody` method only. The three methods `setBodyContent`, `doInitBody`, and `doAfterBody` are defined in the `BodyTag` interface, and the respective methods are called in order by the JSP engine.

Figure 5-11. The seven states of the `BodyTag` *handler class illustrated with the return values required to enter a given state. The state of the tag handler class is largely controlled by the return value from the* `doEndTag` *of the executing state. The* `BodyTag` *handler class may either skip the body, evaluate identical to an* `IterationTag`*, or create a buffer containing the tag body content usable for further evaluation.*

As shown in Figure 5-12, some methods in the life-cycle pattern return integers. Take a look at Table 5-1 to see which values are legal to return and what effect the return value has on further JSP evaluation.

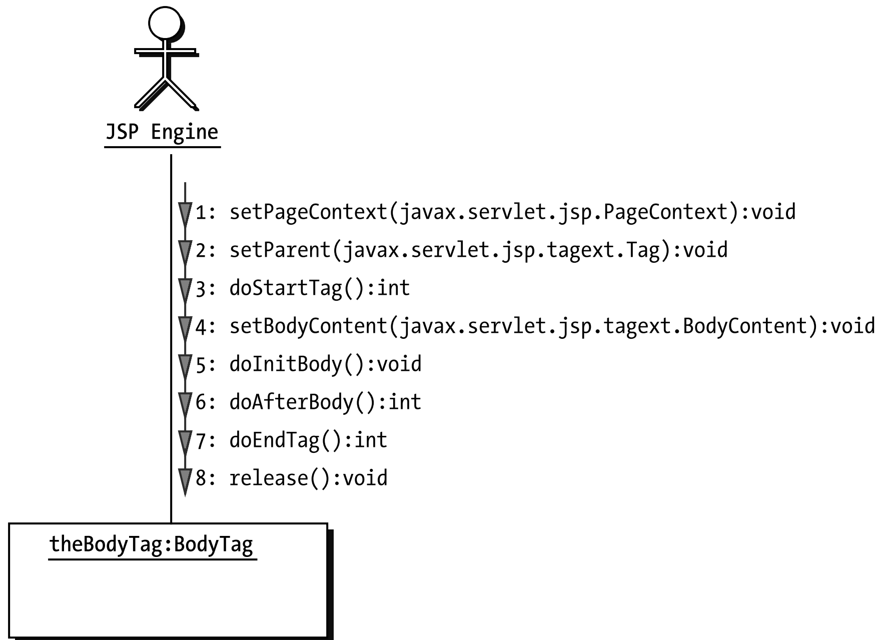

Figure 5-12. The life cycle of a body tag without any attributes, drawn as a UML collaboration diagram

Table 5-1. Values Returned from Lifecycle Methods of the `BodyTag` *Interface*

METHOD NAME	RETURN VALUE	EFFECT
doStartTag	Tag.EVAL_BODY_INCLUDE	The body contents of the container tag are evaluated into the existing out stream (in other words, unbuffered). This value is illegal if the tag handler class implements BodyTag; such tag handler classes must instead return BodyTag.EVAL_BODY_BUFFERED.
	BodyTag.EVAL_BODY_BUFFERED	A new BodyContent object (which contains a buffered copy of the original tag body content) is created and readied for evaluation. Note that BodyTag.EVAL_BODY_BUFFERED only is a legal return value from the doAfterBody method if the tag handler class implements BodyTag.
	BodyTag.EVAL_BODY_TAG	Deprecated in the JSP 1.2 specification; use BodyTag.EVAL_BODY_BUFFERED instead.
	Tag.SKIP_BODY	The body content of the non-empty tag is skipped and not evaluated. This value must be returned by empty tags (in other words, non-container tags).
doAfterBody	BodyTag.EVAL_BODY_TAG	Deprecated in the JSP 1.2 specification; use IterationTag.EVAL_BODY_AGAIN instead.

Table 5-1. Values Returned from Lifecycle Methods of the `BodyTag` *Interface (Continued)*

METHOD NAME	RETURN VALUE	EFFECT
	`IterationTag.EVAL_BODY_AGAIN`	Return this value to re-evaluate the body content of the `IterationTag` or `BodyTag`. For backwards compatibility with the JSP 1.1 version, the value of (the deprecated) `BodyTag.EVAL_BODY_TAG` is identical to `IterationTag.EVAL_BODY_AGAIN`.
	`Tag.SKIP_BODY`	Return this value to quit re-evaluation of the `IterationTag` or `BodyTag` and proceed with the JSP output generation.
`doEndTag`	`Tag.EVAL_PAGE`	Further contents of the JSP document are evaluated normally.
	`Tag.SKIP_PAGE`	No more JSP document content is evaluated; effectively returns from the `service` method of the generated servlet. Use this method to skip any output to the client browser after this tag.

Figure 5-13 provides a more detailed activity flow and state chart for custom tag evaluation.

The activity states from the flow diagram in Figure 5-13 represent activities performed by the JSP engine. The first activity state “Finding taglib definitions for tag” is completely handled by the JSP engine. The activities include reading metadata class definitions, instantiating metadata objects and calling methods within them to create the proper Java code placed within the servlet.

In the second activity state, “Evaluating body content,” the JSP engine performs normal variable substitution in the body content of the current `IterationTag` or `BodyTag`. When done substituting within the current iteration over the tag body, the method `doAfterBody` is called. As indicated in Figure 5-13, the iteration and substitution continues as long as the `doAfterBody` method returns `IterationTag.EVAL_BODY_AGAIN`. Iteration over the tag body is aborted when the `doAfterBody` method returns `Tag.SKIP_BODY`; the JSP engine then proceeds to invoke the `doEndTag` method.

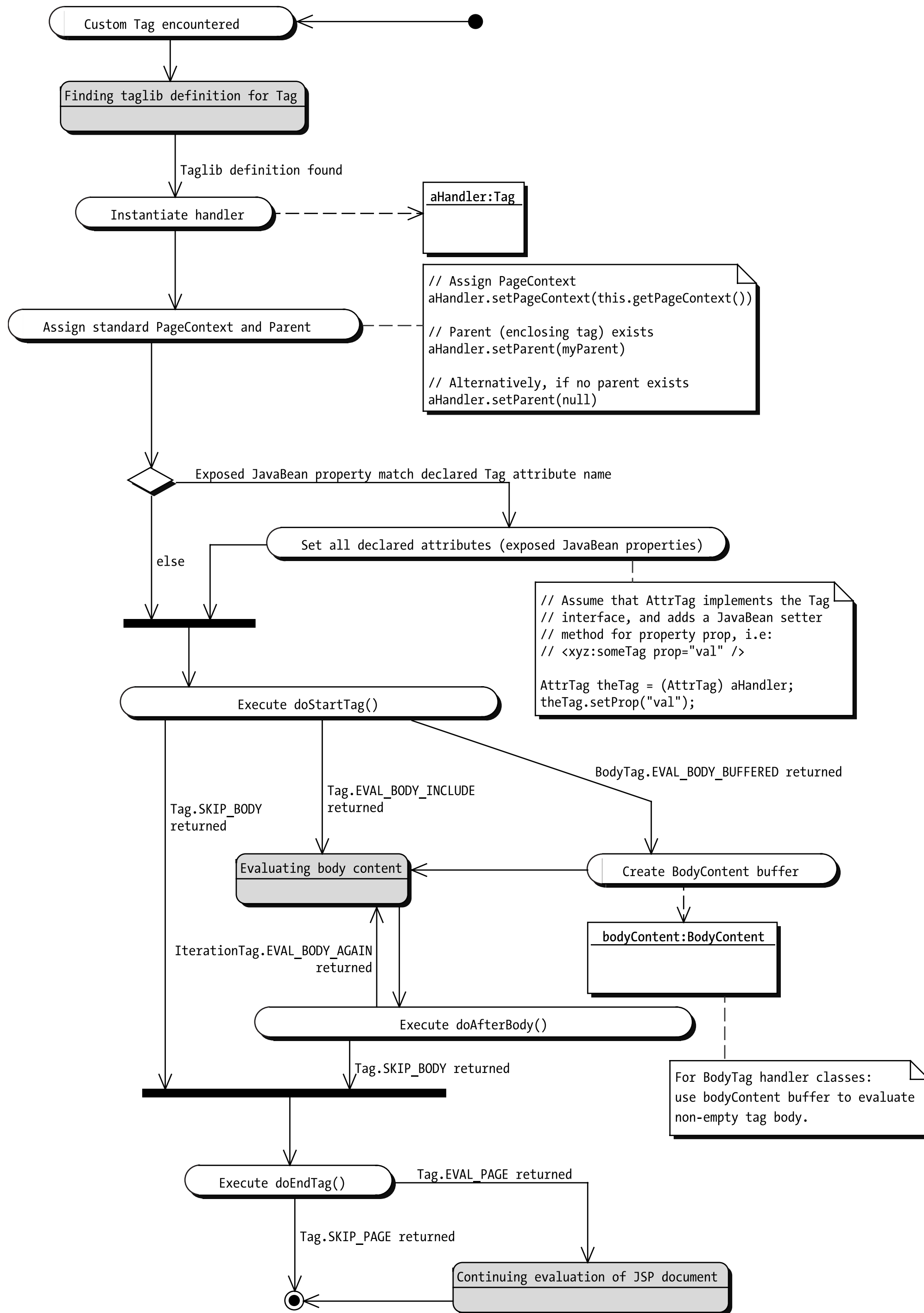

Figure 5-13. UML activity flow and state chart diagram for the life-cycle method of a tag handler class

The most common value returned by `doEndTag` is `Tag.EVAL_PAGE`, which instructs the JSP engine to keep parsing and evaluating the rest of the JSP document, found after the tag just evaluated. If, on the other hand, `Tag.SKIP_PAGE` is returned, the execution of the JSP document is aborted.

Before plunging into the details and principles of creating customized tag libraries, start with a small, warm-up example.

The firstTag.jsp Example

The objective of the `firstTag.jsp` example shown in Figure 5-14 is to create an empty custom JSP tag without any attributes that prints, "This is the First Tag." to the standard `JspWriter`.

Figure 5-14. The output result of the `firstTag.jsp` *document*

Study the four documents involved in the process:

- `firstTag.jsp` contains the view and the JSP taglib directive.
- `WEB-INF/web.xml` contains the mapping between the URI provided in the `firstTag.jsp` document and the taglib definition file.
- `WEB-INF/taglibs/APressTags` contains the tag library definitions.
- `WEB-INF/classes/se/jguru/tags/FirstTag.class` contains the compiled bytecode for the tag handler class.

The JSP document is the origin of all tag library activity, and its relevant parts are shown in Listing 5-1. Note that the taglib directive pinpoints a JAR file, and identifies the tags defined within using the prefix "`apress`".

Listing 5-1. `firstTag.jsp`

```
<%--
        Include all tags defined in the Tag Library identified by the
        URI APressServletBookTags. To distinguish them from
        tags defined in other places, we identify them with the name "apress".
--%>
<%@ taglib uri="APressServletBookTags" prefix="apress" %>

<HTML>

...

<tr>
        <td>
<%--
        Output the HTML equivalent of the tag to invoke.
        This is done only for viewing purposes; to identify
        on the client view what is being invoked within
        the JSP document.
--%>
        &lt;apress:FirstTag/&gt;
        </td>

        <td>
<%--
        Invoke the handler class for the tag FirstTag, defined
        In the taglib having the prefix apress.
--%>
        <apress:firstTag />
        </td>

...
```

The JSP document uses the `<apress:firstTag />` tag and provides a URI for the prefix `apress`. The next document in the definition chain that leads to the `Tag` implementation is the `web.xml` descriptor document, as shown in Figure 5-3 previously.

Including a TLD Archive

If your custom tag libraries are distributed in a JAR file, you are recommended to name the TLD file `taglib.tld` and place it in the directory `META-INF` within the JAR. When packaging your custom tag handler classes in a JAR file, you may skip providing a mapping in the `web.xml` configuration file.

The JSP file would pinpoint the tag library definition file in the following way:

```
<%--
        Include all tags defined within the META-INF/taglib.tld file of
        The JAR file /taglibs/APressTags.jar. To distinguish them from
        tags defined in other places, we identify them with the name "apt".
--%>
<%@ taglib uri="/taglibs/APressTags.jar" prefix="apt" %>
...
<%--
        Invoke the handler class for the tag firstTag, defined
        In the taglib having the prefix dummy.
--%>
        <apt:FirstTag />
...
```

All other properties of the JSP custom tags (such as attributes and validators) apply identically to tags deployed outside of a JAR file.

The web.xml Configuration File

When the JSP file has defined a taglib URI that cannot be found as a file in the Web application's file system, the `web.xml` configuration file is consulted. The `web.xml` file may contain several tag library mappings, similar to Listing 5-2, that map a taglib URI defined in the JSP page to a TLD file.

Listing 5-2. The `web.xml` *deployment descriptor*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
```

```
<web-app>
    <taglib>
        <taglib-uri>
        APressServletBookTags
        </taglib-uri>
        <taglib-location>
           /WEB-INF/taglibs/ApressTagLibs.tld
        </taglib-location>
    </taglib>
</web-app>
```

The URI "APressServletBookTags" in the JSP taglib directive is mapped to the taglib definition file `/WEB-INF/taglibs/ApressTagLibs.tld`. That tag library definition file is consulted to find the tag handler classes used for each found custom JSP tag.

> **NOTE** *The Tomcat reference implementation engine uses the default name* `taglib.tld` *to identify a taglib definition file, unless another name is specified.*

The Tag Library Definition File

The TLD file contains all mappings between custom JSP tags and tag handler classes. The TLD may either be referenced directly from the URI attribute of the `<%@ taglib %>` directive, or by using a logical name (`<taglib-uri>`) defined in the `web.xml` configuration file.

In this example, the TLD (`/WEB-INF/taglibs/ApressTagLibs.tld`) is defined in the `web.xml` file. See Listing 5-3.

Listing 5-3. The taglib definition file, `ApressTagLibs.tld`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE taglib PUBLIC
   "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
   "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>
     <tlib-version>1.2</tlib-version>
     <jsp-version>1.2</jsp-version>
```

```
    <short-name>
        J2EE frontend technologies; servlets, JSPs and
        EJBs tutorial tag library
    </short-name>
    <uri>http://localhost/taglib/tagz.jar</uri>
    <description>Sample Tutorial Tag Library</description>
    <tag>
            <name>FirstTag</name>
            <tag-class>se.jguru.tags.FirstTag</tag-class>
            <body-content>empty</body-content>
            <description>The first tutorial tag</description>
    </tag>
</taglib>
```

Note that the `DOCTYPE` element provides the version specification for this taglib (JSP tag library 1.2) and that the versions required by the tag handlers of this TLD are included in the `<tlib-version>` and `<jsp-version>` tags.

The `taglib.tld` file contains three sections:

- Standard XML header, defining this document to adhere to the XML JSP Tag Library 1.2 DTD. If you are interested in taking a closer peek at this DTD, it can be downloaded from the URI provided in the header.

- Tag library container tag (`<taglib>`), which contains:

    - Metadata section providing information about the tag library itself, rather than any of its defined tags.

    - A `<tag>` definition linking this custom tag called `FirstTag` (which is empty and has no body content) to the tag handler class `se.jguru.tags.FirstTag`. Thus, whenever this tag library is used and a `FirstTag` tag is encountered by the JSP engine, an instance of `se.jguru.tags.FirstTag` is created and life-cycle methods are invoked in that instance to produce any desired output.

TIP *If you are deploying your tag handler classes into a JAR file, be sure to use the name* `taglib.tld` *for the TLD file, as some commercial implementations of Web containers will start searching for the TLD assuming its Web application path to be* `META-INF/taglib.tld`*. For convenience, the JAR file should be placed in* `WEB-INF/lib`*. That way, all tag handler classes will always be included in the Web application classpath.*

Tag library definition files will be discussed in greater detail in “Tag Library Descriptors,” later in this chapter.

The TLD File for JSP version 1.1

Should you be using a JSP engine that does not support version 1.2, you must create a TLD file complying with JSP version 1.1. Refer to “Tag Libraries in JSP 1.1,” later in this chapter, for information on TLD under JSP version 1.1.

Having defined a mapping between the tag handler class and the tag name, you must now examine the Tag handler class for the `FirstTag` custom JSP tag.

The Tag Handler Class

The last part of the four-file system is the handler class, `se.jguru.tags.FirstTag`, which is a JavaBean with the structure you see in Figure 5-15.

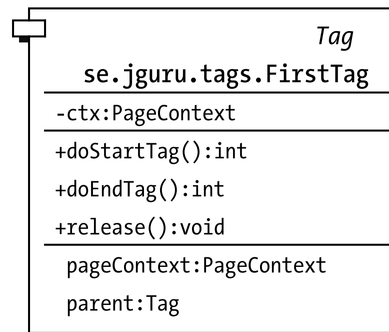

Figure 5-15. The UML diagram of the tag handler class `FirstTag`

The `FirstTag` class is a tag handler because it implements the interface `javax.servlet.jsp.tagext.Tag`, which contains the life-cycle methods of a tag handler class. These methods are *automagically* called in a certain order by the JSP engine whenever a custom tag is encountered, as shown in Figure 5-15.

Taking a quick peek at a pseudo-code compilation of the relevant parts of the generated JSP servlet reveals the order in which these methods are called (see Listing 5-4). Note that the variable name has been abbreviated (to `theTag`) and the absolute class name `se.jguru.tags.FirstTag` has been abbreviated to `FirstTag`, for

purposes of readability. In Listing 5-4 the `Tag` life-cycle methods have been highlighted in bold.

Listing 5-4. The generated servlet code for a JSP document containing the `<apress:firstTag />` *tag*

```
// begin
/* ----  dummy:firstTag ---- */
    FirstTag theTag = new FirstTag();
    theTag.setPageContext(pageContext);
    theTag.setParent(null);

    try
    {
        int eval = theTag.doStartTag();
        if (eval == BodyTag.EVAL_BODY_BUFFERED)
            throw new JspTagException("Since tag handler class "
            + "se.jguru.tags.FirstTag does not implement BodyTag, it can't
            + "return BodyTag.EVAL_BODY_TAG");
        if (eval != Tag.SKIP_BODY)
        {
            do {
                    // end
                    // begin
            } while (false);
        }
        if (theTag.doEndTag() == Tag.SKIP_PAGE)
            return;
    }
    finally
    {
        theTag.release();
    }
// end
```

The JSP 1.1 Code Equivalent

If you are using a JSP 1.1-compliant engine, the code generated by the JSP engine to call the life-cycle methods of the `FirstTag` handler class becomes slightly different than the 1.2 equivalent. Refer to "Tag Libraries in JSP 1.1," later in this chapter, for a discussion of JSP 1.1-compliant tag library definition files.

You may simplify the pseudo-code further to reveal the principal life cycle of a tag handler class for an empty tag without attributes. See Listing 5-5.

Listing 5-5. Servlet code for invoking a JSP 1.1-style taglib tag

```
// ### 1)          Create the tag instance, and set its internal
//                      handler objects, which allow communication with
//                      the pageContext and any enclosing(parent) tags,
//                      should this tag be placed within another.
FirstTag theTag = new FirstTag();
theTag.setPageContext(pageContext);
theTag.setParent(null);

// ### 2)           Call the doStartTag() method in the handler class.
//                       (For non-empty tags, this method is called prior
//                       to evaluating any body content).
theTag.doStartTag();

// ### 3)           Call the doEndTag() method in the handler class.
//                     (For non-empty tags, this method is called after
//                      evaluating the body content).
theTag.doEndTag();

// ### 4)           Call the release() method to free up any used resources
/                        after the tag has been completely processed.
theTag.release();
```

Thus, you can see that the JSP engine calls most of the available methods of the tag handler class. Most of those methods are defined in the interface `javax.servlet.jsp.tagext.Tag`, which must be implemented by all tag handler classes. A better walkthrough of the `java.servlet.jsp.tagext` package and its interfaces and classes will be done in “Touring the javax.servlet.jsp.tagext Package,” later in this chapter. Listing 5-6 provides the complete code of the tag handler.

Listing 5-6. The `FirstTag` *implementation class. Life-cycle methods are highlighted in bold.*

```
/*
 * Copyright (c) 2000 jGuru Europe AB
 * All rights reserved.
 */

package se.jguru.tags;
```

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

/**
 * Tag handler class which simply replaces the
 * occurrence of the tag with the text "This is
 * the First Tag.".
 *
 * This tag is empty (i.e. it does not have a
 * body or body content).
 */
public class FirstTag implements Tag
{
    // Internal state
    private PageContext ctx;
    private Tag parent;

    /**
     * Method called by the JSP engine before the tag body is
     * evaluated. Since this is a standalone tag, simply return
     * Tag.SKIP_BODY to indicate that we will not evaluate any
     * body content.
     */
    public int doStartTag() throws JspException
    {
        // Do not do anything when the tag starts.
        return SKIP_BODY;
    }

    /**
     * Method called by the JSP engine after any body evaluation
     * has taken place.
     */
    public int doEndTag() throws JspException
    {
        try
        {
            // Get the JspWriter connected to the
            // client browser output. This is received from
            // the pageContext, ctx.
            JspWriter outWriter = ctx.getOut();
```

968c05.fm Page 394 Tuesday, October 30, 2001 6:56 PM


```
            // Write something to the JspWriter.
            outWriter.write("This is the First Tag.");
        }
        catch(Exception ex)
        {
            throw new JspException("[doEndTag]: " + ex);
        }

        // Tell the JSP engine to continue evaluating the
        // rest of the JSP page, rather than aborting all
        // evaluation here.
        return EVAL_PAGE;
    }

    /**
     * Since this handler has allocated no resources, we don't need
     * to release any when the JSP engine is done with the tag.
     */
    public void release() {}

    /**
     * Method called by the JSP engine before any evaluation
     * has taken place. The purpose of it is to provide a handle
     * to the PageContext of the running page to the tag.
     */
    public void setPageContext(PageContext context)
    {
        this.ctx = context;
    }

    /**
     * Method called by the JSP engine before any evaluation
     * has taken place. The purpose of it is to provide a handle
     * to the parent (enclosing) tag handler of this tag handler.
     */
    public void setParent(final Tag parent)
    {
        this.parent = parent;
    }
```

```
    /**
     * Method returning the parent Tag handler class of this one.
     */
    public Tag getParent()
    {
        return this.parent;
    }
}
```

The code of the JSP-compiled servlet in Listing 5-6, shows the two relevant evaluation methods of this tag handler: `doStartTag` and `doEndTag`. The former is called before and the latter after any body content of the tag is evaluated—but because the `firstTag` has no body content, you need not do anything special to the `out JspWriter` here. To tell the JSP engine to skip any evaluation of body content, you return `Tag.SKIP_BODY` from the method.

The latter of the two methods, `doEndTag`, is called after any evaluation of body content. In this case, you obtain a reference to the `out JspWriter` connected to the `HttpResponse` of the generated servlet. After this, you are able to write any content to the client from within the handler class—in our case the static string "This is the First Tag." is printed to the output stream.

Tag Library Descriptors

The TLD file is a document that maps tag handler classes to tag names and provides metadata about the tag library itself and all tags defined within it. The TLD is an XML document—so the document structure is defined within the DTD provided in the XML header ("`http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd`"). An XML visualization of the TLD DTD may give greater understanding.

The TLD DTD 1.2 specification, illustrated in Figure 5-16, requires the following information to be present in the TLD file:

- XML version definition (`<?xml version="1.0" encoding="ISO-8859-1" ?>`).
- DTD telling the JSP engine what tag library definition structure is used in this TLD. For example, the 1.2 specification has the following structure: `<!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>` container tag, declaring a tag library definition. This is a container that encloses all real specifications in the TLD file.

968c05.fm Page 396 Tuesday, October 30, 2001 6:56 PM

- Tag library metadata, containing version information and other data common to the entire tag library. Referring to the TLD file listing below, the taglib metadata elements provided are `<tlib-version>`, `<jsp-version>`, `<short-name>`, and `<description>`.

- Tag definitions for all custom tags defined within this tag library. The TLD file in Listing 5-7 defines one empty custom JSP tag, `FirstTag`. The `Tag` definition is a container XML element; its children defines all aspects of the custom JSP tag, according to the DTD illustrated in Figure 5-16.

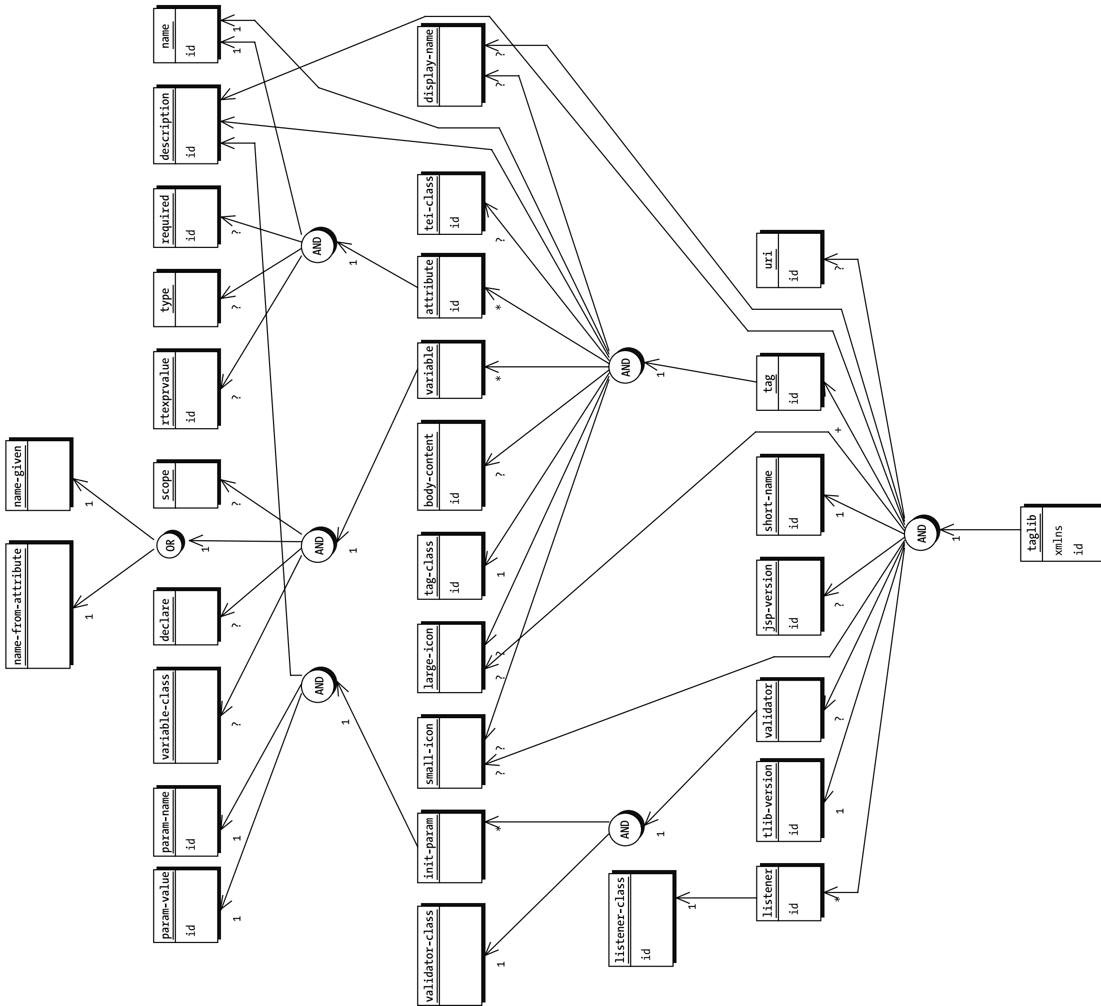

Figure 5-16. The TLD DTD 1.2 specification for the tag library definition file
Listing 5-7 shows what a typical `.tld` *file looks like.*

968c05.fm Page 397 Tuesday, October 30, 2001 6:56 PM

Listing 5-7 shows what a typical `.tld` file looks like.

Listing 5-7. A typical `.tld` *file*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE taglib PUBLIC
   "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
   "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>
             <tlib-version>1.2</tlib-version>
             <jsp-version>1.2</jsp-version>
             <short-name>APress tutorial tags</short-name>
             <description>Sample Tutorial Tag Library</description>

             <tag>
                <name>FirstTag</name>
                <tag-class>se.jguru.tags.FirstTag</tag-class>
                <body-content>empty</body-content>
                <description>The first tutorial tag</description>
             </tag>
</taglib>
```

Figure 5-16 shows all elements permitted within a Tag Library Descriptor XML container. All XML documents use data found within their DTD to define permitted XML elements. Understanding the tag library XML DTD is the best way to realize the possibilities of the XML DTD, so take a peek at a fully populated TLD document and correlate its contents with the Taglib DTD, starting with TLD metadata.

Tag Library Metadata

All tag libraries contain a set of metadata information tags, some of which are required. Listing 5-8 is an excerpt from a taglib definition containing fully populated metadata entries.

Listing 5-8. Excerpt from a taglib definition

```
<!--
      Mandatory Metadata Container Tags
      ---------------------------------

      tlib-versionTag Library version of this tag library, given
            in Dewey decimal notation (i.e: 1.25-1.6) A regular
            expression of the Dewey decimal otation is [0-9]*{ "."[0-9] }0..3,
            which denotes a maximum of 3 dots.
            Example: 1.2
      jsp-versionMinimum JSP version required for this tag
            library to function, given in Dewey decimal noration.
            Example: 1.2
      short-nameFreetext description of this tag library, which
            may be used by builder tools to generate a default taglib prefix.
            According to the DTD, one should not use white space, and not
            start with digits or underscore.

      Optional Metadata Container Tags
      --------------------------------

      uri          Unique identifier of this tag library
      display-name Human-readable name intended for use by
                   taglib construction tools
      description  Human-readable string describing the "use" of this taglib
      small-icon   Filename of an icon image for use by taglib construction tools
      large-icon   Filename of an icon image for use by taglib construction tools
      validator    Classname of a validator class which performs runtime
                   validation on the <%@ taglib ...%> directives connected
                   to this tag library definition.
      listener     Classname of an event listener to this Tag.
-->
      <tlib-version>1.2</tlib-version>
      <jsp-version>1.2</jsp-version>
      <short-name>APress custom IterationTag</short-name>
      <uri>taglib/apressTags.tld</uri>
      <description>Sample Tutorial Tag Library</description>
      <!-- Mandatory Custom Tag definitions go here -->

</taglib>
```

An XML element definition corresponds to a container tag in Listing 5-8. Thus, reading from the taglib DTD, you find the following `<taglib>` element definition:

```
<!ELEMENT taglib (tlib-version, jsp-version, short-name, uri?, display-name?,
small-icon?, large-icon?, description?, validator?, listener*, tag+) >
```

Interpreting XML DTD Entries

The `<taglib>` element may contain a well-defined series of child elements, the names of which are given within the parentheses. Let's originate from the taglib XML element definition:

```
<!ELEMENT taglib (tlib-version, jsp-version, short-name, uri?,
   display-name?, small-icon?, large-icon?, description?,
   validator?, listener*, tag+) >
```

The XML DTD defines the four meanings listed in Table 5-2 to the child elements of an XML container, depending on the character appended to the child element name.

Table 5-2. Character Definitions

CHARACTER APPENDED	CARDINALITY	STATUS	DESCRIPTION
<none>	1	Mandatory	A mandatory element must occur exactly once within its parent. The `<tlib-version>` element is mandatory within its `<taglib>` parent.
?	0..1	Optional	An optional element may occur exactly once within its parent. The `<display-name>` element is mandatory within its `<taglib>` parent.

Table 5-2. Character Definitions (Continued)

CHARACTER APPENDED	CARDINALITY	STATUS	DESCRIPTION
*	0..n	Optional, list	An optional list element may occur several times within its parent. The <listener> element is an optional list within its <taglib> parent.
+	1..n	Mandatory, list	A mandatory list element must occur at least once within its parent. The <tag> element is a mandatory list within its <taglib> parent.

Figure 5-16 shows the element specification of the TLD DTD. The element cardinalities are implied in Figure 5-16 by printing the character from the Character Appended column in the preceding table. Can you find the two mandatory child elements of the `<tag>` element?

Before taking a look at the `<tag>` element definitions in detail, study the optional `<validator>` element a little closer. The functionality provided by a tag library validator instance is powerful; it may be beneficial—especially in larger projects.

The <validator> Element

Each tag library may contain one (optional) tag library validator object. The purpose of the validator is to verify that the `<@% taglib … %>` directive is well-formed and potentially perform a custom validation on the JSP document. The `<validator>` element is new to JSP version 1.2, with no corresponding TLD 1.1 equivalent.

The tag library validator instance must extend the class `javax.servlet.jsp.tagext.TagLibraryValidator`, which is shown in Figure 5-17. The developer must override at least one method—`validate`. Frequently, one makes use of initialization parameters sent to the validator from the TLD file. If so, the `setInitParameters` and `getInitParameters` methods are also of interest to the development. The JSP container will invoke the `setInitParameters` method before calling validate, ensuring that all `init` parameters will be available during validation.

The TLD DTD specifies that the `<validator>` element has the following structure:

```
<!ELEMENT validator (validator-class, init-param*, description?) >
```

Figure 5-17. The `TagLibraryValidator` *class encapsulates a minimalistic but powerful functionality. The only two methods requiring the attention of a developer are* `release` *and* `validate`*. The JavaBean setter and getter for the* `initParameters` *JavaBean property are merely convenience methods, which do not require overriding on the part of the developer.*

Listing 5-9 is a sample validator specification from a TLD file.

Listing 5-9. A validator definition

```
<taglib>
      ...
<!--
      Mandatory Validator Container Tags
      ---------------------------------
      validator-class     The class which should be instantiated to
            obtain the tag library validator.

      Optional Validator Container Tags
      --------------------------------
      init-param     An initialization parameter specification
            to the Validator instance. All parameters defined in
            the TLD file will be parsed and inserted into a Map which
            may be retrieved with the getInitParameters(); method.
      -->
      <validator>
            <validator-class>
               se.jguru.tags.BasicTagLibValidator
            </validator-class>
            <init-param>
                  <param-name>logFileName</param-name>
                  <param-value>validatorLog.txt</param-value>
            </init-param>
```

968c05.fm Page 402 Tuesday, October 30, 2001 6:56 PM


```
            <init-param>
                  <param-name>dateFormat</param-name>
                  <param-value>-hh:mm:ss.SSSS-</param-value>
            </init-param>
      </validator>
      ...
</taglib>
```

The simplest way to understand the function and use of a TagLibraryValidator subclass is to show a simple example of its use. Therefore, take a look at the three entries that join forces to create a validator service:

- JSP file contains the <@% taglib ... %> directive. Let's assume that the taglib directive is <%@ taglib uri="apressTags" prefix="apt" %>.

- TLD file contains the <validator> specification. Let's use the previous TLD snippet as a point of origin in this small example.

- javax.servlet.jsp.tagext.TagLibraryValidator subclass contains the validation definition. According to the previous TLD file, you will use a validator class called se.jguru.tags.BasicTagLibValidator (see Figure 5-18).

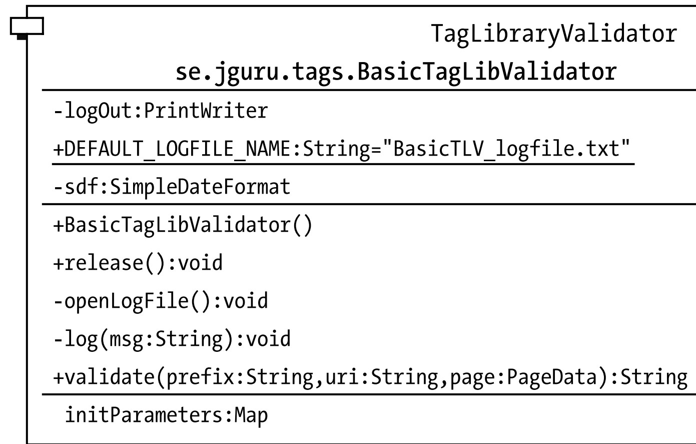

Figure 5-18. The taglib validator class of this small example. Besides overriding the validate *and* release *methods, the* BasicTagLibValidator *may log data to a log file—and the* logOut Writer *is connected to the log file.*

Listing 5-10 provides the resulting output to the log file. Note that the output in the topmost section, where the initialization parameters names and values are

968c05.fm Page 403 Tuesday, October 30, 2001 6:56 PM

printed, correspond perfectly to the values provided in the TLD file in Listing 5-9. Note also, that the *prefix* and *uri* printouts are read from the `<@% taglib ... %>` directive in the JSP document. The `PageData` output is simply the XML version of the JSP document, which is sent to the validator as an argument to the `validate` method.

Listing 5-10. Log file excerpt

```
[-02:04:38.0739-]: Log file opened. Validator in normal operational mode.

[-02:04:38.0739-]:  -- Printing all init parameters
[-02:04:38.0759-]:  [dateFormat]: -hh:mm:ss.SSSS-
[-02:04:38.0759-]:  [logFileName]: validatorLog.txt
[-02:04:38.0759-]:  -- All init parameters printed.

[-02:04:38.0819-]:  prefix: apt
[-02:04:38.0819-]:  uri: apressTags

[-02:04:38.0819-]:  --> Begin PageData
[-02:04:38.0819-]: (1): <jsp:root
[-02:04:38.0819-]: (2):   xmlns:jsp="http://java.sun.com/JSP/Page"
[-02:04:38.0819-]: (3):   version="1.2"
[-02:04:38.0819-]: (4):   xmlns:apt="apressTags"
[-02:04:38.0819-]: (5): >
[-02:04:38.0829-]: (6): <jsp:text><![CDATA[
[-02:04:38.0829-]: (7):
[-02:04:38.0829-]: (8): <html>
[-02:04:38.0829-]: (9):         <head>
[-02:04:38.0829-]: (10):        <title>AnotherBody Tag Sample</title>
[-02:04:38.0829-]: (11):        </head>
[-02:04:38.0829-]: (12):
[-02:04:38.0829-]: (13):        <body>
[-02:04:38.0829-]: (14):        <center>
[-02:04:38.0829-]: (15):
[-02:04:38.0829-]: (16):        <h1>Body Tag Example</h1>
[-02:04:38.0829-]: (17):
[-02:04:38.0829-]: (18):        <table border=2>
[-02:04:38.0829-]: (19):        <tr>
[-02:04:38.0829-]: (20):                <td>JSP source code</td>
[-02:04:38.0829-]: (21):                <td>Resulting output</td>
[-02:04:38.0839-]: (22):        </tr>
[-02:04:38.0839-]: (23):
[-02:04:38.0859-]: (24):        <tr>
[-02:04:38.0859-]: (25):                <td>&lt;apt:SimpleBodyTag>This is a
                                body text.&lt;/apt:SimpleBodyTag></td>
```

```
[-02:04:38.0859-]: (26):                  <td>]]>
[-02:04:38.0879-]: (27): </jsp:text>
[-02:04:38.0879-]: (28): <apt:SimpleBodyTag><jsp:text><![CDATA[
[-02:04:38.0879-]: (29): This is a body text.]]>
[-02:04:38.0899-]: (30): </jsp:text>
[-02:04:38.0899-]: (31): </apt:SimpleBodyTag>
[-02:04:38.0899-]: (32): <jsp:text><![CDATA[
[-02:04:38.0919-]: (33): </td>
[-02:04:38.0919-]: (34):         </tr>
[-02:04:38.0919-]: (35):
[-02:04:38.0919-]: (36):         </table>
[-02:04:38.0919-]: (37):
[-02:04:38.0919-]: (38):         </center>
[-02:04:38.0919-]: (39):         </body>
[-02:04:38.0919-]: (40): </html>]]>
[-02:04:38.0919-]: (41): </jsp:text>
[-02:04:38.0929-]: (42): </jsp:root>
[-02:04:38.0929-]:  <-- End PageData
```

The full source code of the `BasicTagLibValidator` shows that the `validate` method has access to the initialization parameters, as well as a `PageData` object that encapsulates the XML version of the JSP document. The `PageData` class has one single method of interest, `getInputStream`, which returns an `InputStream` connected to the XML stream of the document. The developer may therefore validate documents originating from the data found in the JSP file, the taglib directive, or the initialization parameters.

In Listing 5-11, the `TagLibraryValidator`-specific method calls appear in bold text.

Listing 5-11. The `BasicTagLibValidator` *class*

```
/*
 * Copyright (c) 2000,2001 jGuru Europe.
 * All rights reserved.
 */

package se.jguru.tags;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.util.*;
import java.io.*;
import java.text.SimpleDateFormat;
```

968c05.fm Page 405 Tuesday, October 30, 2001 6:56 PM


```
public class BasicTagLibValidator extends TagLibraryValidator
{
    // Writer to a log file where the log messages will go.
    private PrintWriter logOut;

    // Define the default logfile name
    public static final String DEFAULT_LOGFILE_NAME = "BasicTLV_logfile.txt";

    // The simple date format of the log messages
    private SimpleDateFormat sdf;

    public BasicTagLibValidator()
    {
        // Call the constructor of our superclass
        super();
    }

    /**
     * Free up all held resources; i.e. close the log file.
     */
    public void release()
    {
        // Close the logFile
        try
        {
            this.logOut.close();
        }
        catch (Exception ex)
        {
            // Whoops.
            System.err.println("[BasicTagLibValidator::release]: "
            + "Could not close the log file Writer: " + ex);
        }

        // Call the release of our superclass
        // to continue the release of the state
        // in this validator.
        super.release();
    }
```

```
public void setInitParameters(Map params)
{
    // Proceed with the normal setInitParameters
    super.setInitParameters(params);

    // Setup the log file...
    this.openLogFile();
}

private void openLogFile()
{
    // Don't redo the openLogFile tasks
    // if the log file is already open.
    if (this.logOut != null)
    {
        // Log the attempt.
        this.log("Log file already opened.");

        // Bail out.
        return;
    }

    // Get the configuration parameters
    Map params = this.getInitParameters();

    // Get the log file name
    String logFileName = "" + params.get("logFileName");
    if (logFileName == null || logFileName.equals(""))
        logFileName = DEFAULT_LOGFILE_NAME;

    // LogFile exists?
    File logfile = new File(logFileName);
    if (logfile.exists())
        logfile.renameTo(
            new File(logfile.getName() + ".old"));

    // Open the logfile Writer
    try
    {
        this.logOut = new PrintWriter(
            new FileWriter(logfile));
    }
```

968c05.fm Page 407 Tuesday, October 30, 2001 6:56 PM


```
    catch (IOException ex)
    {
        // Whoops...
        System.out.println("Could not open the logging Writer: " + ex);
    }

    // Get the format string for the simple date format
    String sdfString = "" + params.get("dateFormat");
    if (sdfString == null || sdfString.equals(""))
            sdfString = "hh:mm:ss.SSSS";

    // Create the internal log format
    this.sdf = new SimpleDateFormat(sdfString);

    // Done.
    this.log("Log file opened. Validator in normal operational mode.");
    this.log(" -- Printing all init parameters");
    for (Iterator it = params.keySet().iterator(); it.hasNext(); )
    {
        // Get the current key
        Object key = it.next();

        // Log the key/value pair
        this.log(" [" + key + "]: " + params.get(key));
    }

    // Done.
    this.log(" -- All init parameters printed.");
}

/**
 * Private logging method, which writes a log message
 * to the standard log stream of this validator, i.e.
 * logOut.
 */
private void log(String msg)
{
    // Check sanity
    if (this.logOut == null) this.openLogFile();
    if (msg == null || msg.equals("")) return;

    // Get the current timestamp
    Date now = new Date();
```

```
        // Potentially sane. Log.
        String completeMessage = "[" + this.sdf.format(now) + "]: " + msg;
        this.logOut.println(completeMessage);
        this.logOut.flush();
    }

    /**
     * Validate a JSP page. This will get invoked once per directive in the
     * JSP page. This method will return a null String if the page passed
     * through is valid; otherwise an error message.
     */
    public String validate(String prefix, String uri, PageData page)
    {
        // Log
        this.log(" prefix: " + prefix);
        this.log(" uri: " + uri);

        // Get the content of the JSP document from the PageData argument
        LineNumberReader lnr = new LineNumberReader(
            new InputStreamReader(page.getInputStream()));
        String aReadLine = "";

        // Output the content of the PageData
        this.log(" --> Begin PageData");
        try
        {
            while ((aReadLine = lnr.readLine()) != null)
            {
                // Output the current line
                this.log("(" + lnr.getLineNumber() + "): " + aReadLine);
            }
        }
        catch (IOException ex)
        {
            this.log("Error reading PageData: " + ex);
        }
        this.log(" <-- End PageData");

        // If something displeases us, return a String with an
        // error message. If all went well, return null.
        return null;
        // Or: return "This is an error message from the validator.";
    }
}
```

A `TagLibraryValidator` instance is global for all directives referring to a particular TLD file. Its fully qualified class is defined as an XML element in the TLD file, as seen in Listing 5-11.

Having taken a look at the metadata structure of tag libraries, now turn your attention to the tag definitions themselves.

Tag Definitions

Each `<tag>` element defined in the TLD file maps a handler class to a tag name. Each `<tag>` element is a container, grouping tag metadata and possibly attribute definitions (contained in `<attribute>` elements) for all attributes known (in other words, automatically set by the tag handler instance). All `<tag>` definition elements must be contained within their `<taglib>` parent. The `<tag>` elements in Listing 5-12 are therefore assumed to be contained in a `<taglib>` parent.

Listing 5-12 builds on Listing 5-8. Because you did not actually create any `<tag>` containers, the TLD file does not currently define any custom actions. However, the comment `<!-- Mandatory Custom Tag definitions go here -->` indicates that more information is to be inserted into the TLD file. Create a `<tag>` entry that defines a new tag called `FirstTag`. The comments within the code listing explain the effect of each entry.

Listing 5-12. The tag definition section of the TLD file

```
<!--
          All tag metadata and attribute definitions must be contained
          within the tag element, and each tag element must in turn
          be contained within the taglib parent element.
-->

<tag>

     <!--
          Mandatory metadata container tags
          ---------------------------------

          name       := The (unique) name of the tag, found in the action
                        tag name of the JSP action. (i.e: For the JSP action
                        <x:someName />, the name attribute is someName ).

          tag-class  := Fully qualified class name of the Tag handlerclass
                        of this tag. The handler must implement interface
                        javax.servlet.jsp.tagext.Tag
```

```
 -->
              <name>FirstTag</name>
              <tag-class>se.jguru.tags.FirstTag</tag-class>

<!--
 Optional metadata container tags
 --------------------------------
tei-class   The Tag Extra Information class is a fully
            qualified class name to a class which contains
            metadata describing the attributes of a
            tag. The teiclass must implement
            javax.servlet.jsp.tagext.TagExtraInfo

body-content     Defines whether or not the tag should
            be used in standalone or container mode.
            3 different values are permitted:
                a) JSP. This (default) value indicates
                   that the body of the tag contains normal
                   JSP data.
                b) empty. This value Indicates that the
                   tag cannot have any body content (i.e.
                   the tag Is empty).
                c) tagdependent. This value indicates that
                   the body of the tag contains statements
                   in some language other than JSP. An example
                   could be SQL statements for execution within
                   a DB.
display-name  Human-readable name intended for use by tag
              construction tools
small-icon    Filename of a 16x16 icon image for use by tag
              construction tools
large-icon    Filename of a 32x32 icon image for use by tag
              construction tools
description   Human-readable string describing the "use" of this tag
variable      Provides information about any scripting variables declared
              by this tag. If variable definitions are present, the tag
              must have a TagExtraInfo companion class, defined in the
             <tei-class> element.
attribute    Each tag element may contain 0 or more attribute
             definitions, specifying all the attributes which should be
             recognized by the tag in question. All attributes definitions
             are XML containers, which fully defines the attribute as a
             Java object. Tag attributes are covered in detail shortly.
example      Human-readable string providing an example of
             this tag used
```

968c05.fm Page 411 Tuesday, October 30, 2001 6:56 PM


```
-->

<tag>
      <name>FirstTag</name>
      <tag-class>se.jguru.tags.FirstTag</tag-class>
      <body-content>empty</body-content>
      <description>The first tutorial tag</description>

            <!-- Attribute definitions go here. -->
</tag>
```

The tag defined in Listing 5-12 is an empty tag; its definition elements are seen in the last part of the definition file. A graphical listing of the element nodes introduced in the `<tag>` element reveals the structure (see Figure 5-19).

This is the XML definition for the `tag` element:

```
<!ELEMENT tag (name, tag-class, tei-class?, body-content?, display-name?,
  small-icon?, large-icon?, description?, variable*, attribute*, example?) >
```

Note that the order of the elements within their parent container could be—but rarely is—irrelevant. Some well-known XML processors do not create a full-dependency tree like a regular compiler—the programmer must therefore still pay close attention to the element order, rather than simply focus on the content. Undoubtedly, this situation will improve in the future, but the current state of XML parsers is such that the development community is required to mind yet another formatting detail of the taglib configuration.

The element order within the `<tag>` element container is clear from the element definition. Although it may be trivial to deduct the required element order from a look at the element definition, it certainly is cumbersome to constantly keep an eye at the tag definition at all times while generating a TLD file. In fact, one of the better uses of integrated development environments is the sanity checking and automatic re-ordering of XML tags to comply with the DTD in question. If you find yourself getting frequent XML parse exceptions, I recommend finding a development environment that has tools for strict generation and parsing of XML files.

As shown in the preceding element tag definition (and in Figure 5-18), the only two indexed (list) subelements of `<tag>` are attribute and variable. The former defines an attribute known to the JSP custom tag, and the latter defines a variable declared for use by other tags in the JSP page. Take a look at each in turn.

968c05.fm Page 412 Tuesday, October 30, 2001 6:56 PM

Figure 5-19. The XML structure of the tag element. All mandatory elements have been shaded. Although a detailed description for all elements in the image hasn't been given, you can see that all the elements are subelements to the `<tag>` *container.*

The <attribute> Element

The `<attribute>` element, where attributes of a JSP custom action are defined. The `<attribute>` element definition is small and simple to understand, as shown here and in Figure 5-20:

```
<!ELEMENT attribute (name, required? , rtexprvalue?, type?, description?) >
```

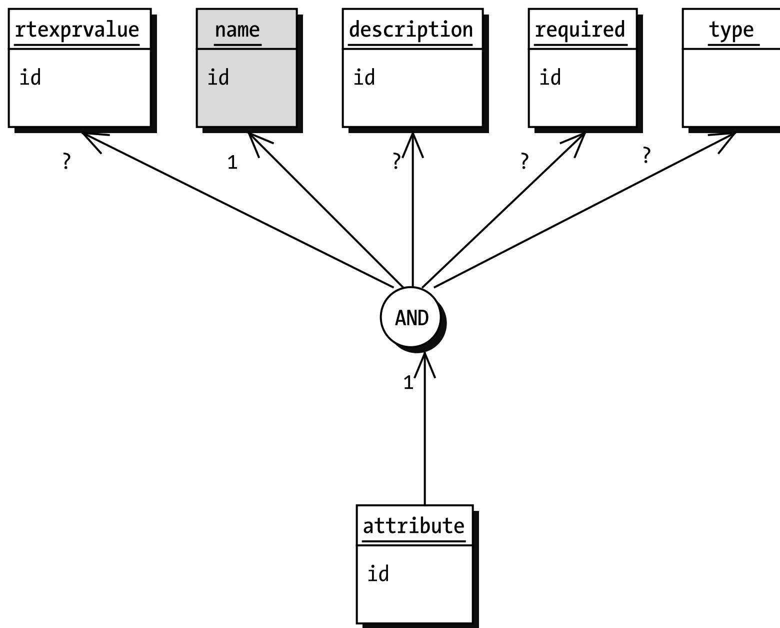

Figure 5-20. Visual representation of the `<attribute>` *element. Note that the only mandatory subelement of the attribute tag is* `<name>`*. If you choose to use types other than the default (*`java.lang.String`*), the* `<type>` *element must be defined as well.*

Attribute definitions are straightforward; all properties in the attribute definition amounts to providing a Java code snippet that declares a variable. All elements of the attribute container are explained in the comments of the `taglib.tld` excerpt shown in Listing 5-13.

Listing 5-13. The tag attribute specification

```
<!--
       All tag attribute metadata and must be contained
       within the tag container.
-->
<attribute>

       <!--
              Mandatory metadata container tags
              --------------------------------

          name       The name of the attribute, found in the action tag name of the
                     JSP action.
                     (i.e: The JSP action <x:someName someKey="apa" />
                      has the attribute name someKey).

       -->

       <name>allClients</name>

       <!--
              Optional metadata container tags
              -------------------------------

          required       Indicates whether or not this attribute is mandatory.
                         If this value is false or no, a default value must be
                         defined in the code of the Tag handler class.
                         Default value: false.
                         Legal values are { yes | true | no | false }
          rtexprvalue    Indicates whether or not the value of this attribute
                         can be calculated during runtime as a JSP expression,
                         as opposed to a static (pre-defined) value set during
                         compilation.
                         Default value: false.
                         Legal values are { yes | true | no | false }
          type           The type of the attribute, expressed as a fully
                         qualifiedclass, such as java.util.Map.
                         Default value: java.lang.String.
          description    Human-readable string describing the purpose of
                         this attribute
```

```
    -->
    <required>yes</required>
    <rtexprvalue>yes</rtexprvalue>
    <type>java.util.List</type>
    <description>The List of all Clients</description>

</attribute>
```

As shown in Listing 5-13, the definition of an attribute to a custom `Tag` implementation class is straightforward.

Java Code for a JSP Custom Tag Attribute

It seems tragic that it is much simpler to understand the generated Java code deriving from a TLD attribute than the TLD attribute definition itself. In other words, the TLD definition does a great job of making a rather simple task a good deal more complex. For instance, the XML definition:

```
<attribute>
    <name>map</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
    <type>java.util.Map</type>
</attribute>
```

and the JSP document:

```
<apt:ListTag map="<%= aMap %>"></apt:ListTag>
```

creates the Java call:

```
listTagInstance.setMap( aMap );
```

It would appear that the TLD definition is more complex (and involves more typing than) the JSP document and the corresponding generated Java code.

The remaining optional list element of the `<tag>` container is the `<variable>` container element that defines scripting variables declared by the custom JSP action.

The <variable> Element

JSP custom tags may define variables that may be accessed from the JSP document or other custom tags. This mechanism is useful to share information from a custom JSP parent tag to its contained children. That way, the child custom action needs not know about the existence of its parent at all, which permits complete separation of the two handler classes. The `<variable>` element definition is small, as shown in the following definition and in Figure 5-21:

```
<!ELEMENT variable ( (name-given | name-from-attribute), variable-class?,
  declare?, scope?, description?) >
```

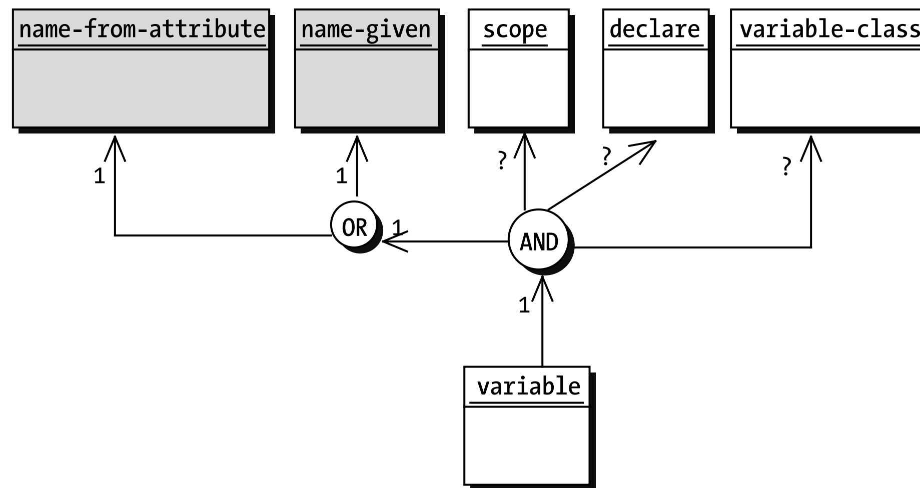

Figure 5-21. Visualization of the variable container element of the TLD 1.2 DTD. Mandatory elements are shaded; note that each variable must include either a `<name-given>` *or a* `<name-from-attribute>` *container element.*

It is easy to get confused by the naming element dualism of the `<variable>` container, but it's a simple process to understand the vision of the TLD DTD design engineers. Each variable may be named in either of two ways:

- The script variable name is assigned by the Web application deployment engineer, who provides a `<name-given>` attribute in the TLD file.
- The script variable name is provided as a translation-time value of an attribute. The name of the attribute is given in the `<name-from-attribute>` element.

Listing 5-14 contains a full description of the variable element in the TLD.

Listing 5-14. A variable element definition

```
<!--
       All tag variable metadata and must be contained
       within the tag container.
-->
<variable>

       <!--
              Mandatory metadata container elements
              ---------------------------------
              name-given      Name of the scripting variable. Either a
                              name-given or a name-from-attribute element
                              must be supplied to the variable - but both
                              cannot be used at the same time.
              name-from-attribute     Name of the attribute whose value
                              is the name of this scripting variable. Either
                              a name-given or a name-from-attribute element
                              must be supplied to the variable – but both
                              cannot be used at the same time.
       -->
       <name-given>firstName</name-given>

       <!--
              Optional metadata container tags
              --------------------------------

              variable-class    Class name of the variable declared. Defaults
                                to String if not provided.
              declare           true if the variable should be declared (as
                                opposed to simply used) by the JSP container,
                                false otherwise. The default value is true.
              scope             One of three values indicating which scope the
                                declared element should have. The values are
                                VariableInfo.AT_BEGIN, VariableInfo.AT_END and
                                VariableInfo.NESTED. Refer to the section
                                "Class javax.servlet.jsp.tagext.VariableInfo"
                                for a detailed walkthrough on variable scoping,
                                and to the sidebar below for the associated code.
              description       Human-readable description of this variable.
```

```
    -->
    <variable-class>String</variable-class>
    <declare>true</declare>
    <scope>AT_BEGIN</scope>
    <description>The first name of the current client</description>
</variable>
```

Listing 5-14 defines the variable `String firstName` with a visibility scope starting from the opening tag delimiter.

Java Code for the <variable> Element

The code generated for a `<variable>` definition in a custom JSP page is a small matter; the following snippet was generated by the Tomcat4.0-b5 Web container. Paths and variable names have been abbreviated for readability; the Catalina engine of Tomcat has an affinity for always using the fully qualified class path for all type definitions and variables. Although this is good indeed for system stability, the autogenerated code is somewhat tricky to read unedited.

The code pertaining to the variable definition is bolded in the following snippet. Note that the variable has nested scope (in other words, it is accessible only within the body of the tag) because it is declared between the `doStartTag` and `doEndTag` method calls). The visibility of a script variable is indicated by one of three constants, defined in the `javax.servlet.jsp.tagext.VariableInfo` class. When the variable is nested within the body of the enclosing custom JSP tag, its corresponding `VariableInfo` constant is `VariableInfo.NESTED`. Refer to “Class javax.servlet.jsp.tagext.VariableInfo,” later in this chapter, for more information about the `VariableInfo` class.

```
/* ----  apt:VariableTag ---- */
VariableDefinitionTag vt = new VariableDefinitionTag();
vt.setPageContext(pageContext);
vt.setParent(null);
try
{
   int startResult = vt.doStartTag();
   if (startResult == BodyTag.EVAL_BODY_BUFFERED)
      throw new JspTagException("Since tag handler class VariableDefinitionTag "
       + "does not implement BodyTag, it can't return BodyTag.EVAL_BODY_TAG");
```

```
    if (startResult != Tag.SKIP_BODY)
    {
        do
        {
            String foo = null;
            foo = (String) pageContext.findAttribute("foo");
        } while (vt.doAfterBody() == BodyTag.EVAL_BODY_AGAIN);
    }
    if (vt.doEndTag() == Tag.SKIP_PAGE)
        return;
}
finally
{
    vt.release();
}
```

Note that the value of the variable is retrieved from an attribute bound in the `pageContext` with the same name as the `<variable>` element.

When altering the scope of a declared variable, such as in `<scope>AT_BEGIN</scope>`, the declaration is done in another place; but all other aspects of the invocation is identical to the `<scope>NESTED</scope>` case illustrated in the previous code snippet. Taking a look at a piece of the resulting JSP code, generated by the Tomcat-4.0 engine, you see two alterations: the variable is declared in another place, and it is initialized in two places. This rather odd behavior is done to permit `IterationTags` and `BodyTags` to modify and re-read the value of the variable with each iteration.

```
try
{
    String foo = null;
    int startResult = vt.doStartTag();
    foo = (String) pageContext.findAttribute("foo");

    if (startResult == BodyTag.EVAL_BODY_BUFFERED)
       throw new JspTagException("Since tag handler class VariableDefinitionTag "
        + "does not implement BodyTag, it can't return BodyTag.EVAL_BODY_TAG");

    if (startResult != Tag.SKIP_BODY)
    {
        do
        {
            foo = (String) pageContext.findAttribute("foo");
        } while (vt.doAfterBody() == BodyTag.EVAL_BODY_AGAIN);
```

Using the `<scope>AT_END</scope>` setting for declaring a variable alters the code produced by the JSP compiler yet again, placing the final assignment of the `foo` variable after the `while` statement marking the end of the tag body, as shown in the previous code snippet. Feel free to verify the code generated by your favorite Web container.

Of course, the full XML DTD can be found and downloaded using a regular Web browser, after navigating to `http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd`.

> **NOTE** *Chapter 6 provides some thorough examples of creating new Tag libraries.*

For those of you still using the 1.1 TLD specification, take a brief look at the 1.1 TLD file.

Tag Libraries in JSP 1.1

The TLD specification version 1.1 contains most options found in version 1.2, but it lacks many descriptor elements of JSP 1.2. In fact, the full XML DTD of the TLD for JSP 1.1 is a lot simpler than the corresponding for 1.2. Figure 5-22 shows the DTD for the TLD document version 1.1.

The difference between TLD DTD versions 1.2 and 1.1 is rather big; apart from the obvious alteration of the `DOCTYPE` element at the TLD document top, and the values of the `<tlibversion>` and `<jspversion>` elements, more subtle differences occur frequently in the document. The 1.1 versions of the TLD DTD uses concatenated element names (for example, `jspversion`), but the 1.2 TLD DTD uses hyphens to separate word parts from one another (for example, `jsp-version`).

Also, some element names in the JSP 1.1 TLD have been altered in the JSP 1.2 release. For instance, the `<info>` element has been renamed `<description>` in the JSP 1.2 TLD DTD. For a full listing of the changes between the 1.1 and 1.2 TLD DTDs, refer to the JSP specification, section "Changes."

To illustrate the differences between the TLD versions, the `FirstTag` custom action discussed in "The Tag Library Definition File" earlier in this chapter is re-created in Listing 5-15 TLD version 1.1.

968c05.fm Page 421 Tuesday, October 30, 2001 6:56 PM

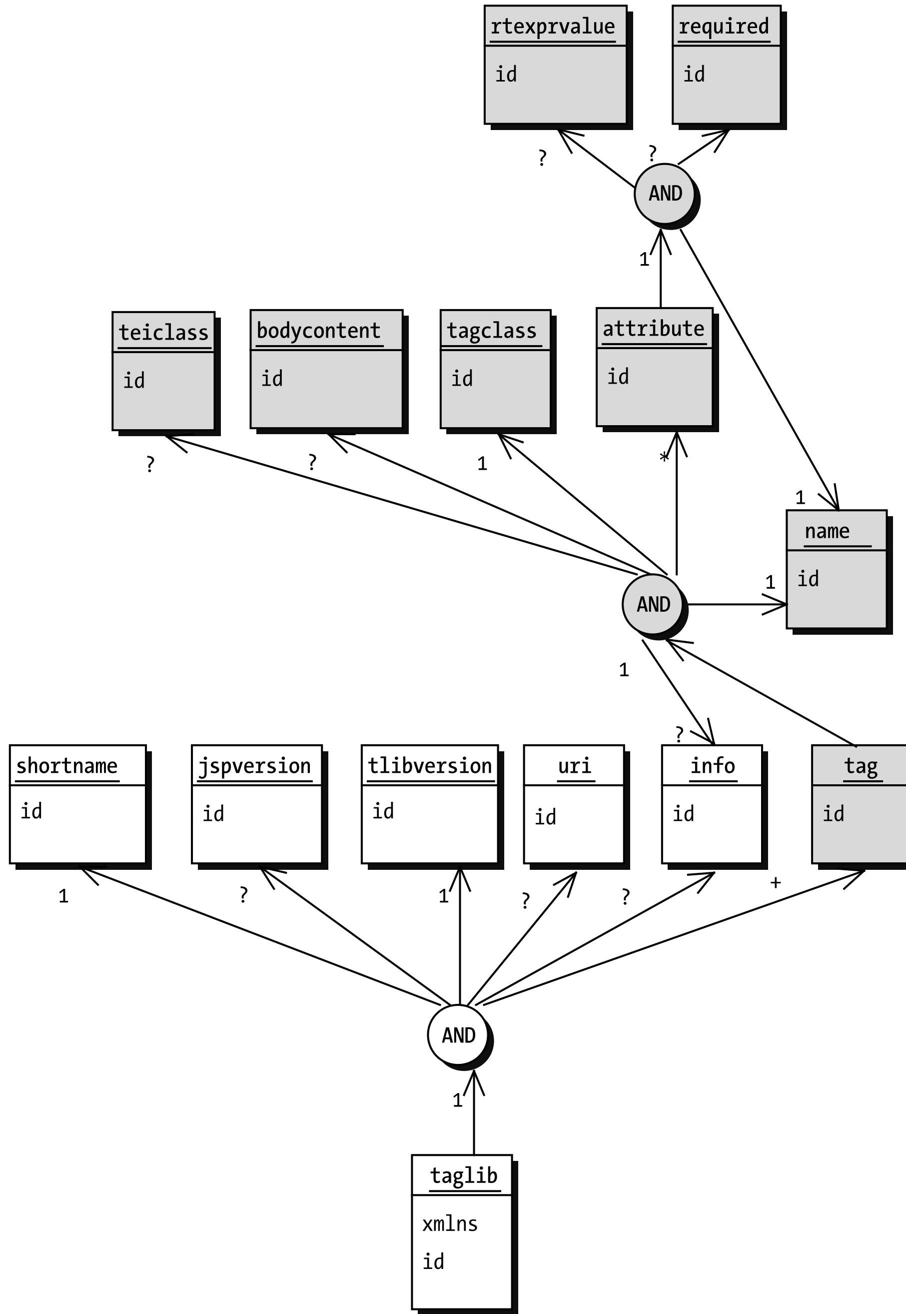

Figure 5-22. Visualization of the TLD DTD for the JSP 1.1 specification. All elements pertaining to a tag definition has been shaded. Note that several of the elements from the TLD DTD version 1.2 are missing or spelled differently. For instance, the `<info>` *element corresponds to the description element of TLD DTD version 1.2.*

968c05.fm Page 422 Tuesday, October 30, 2001 6:56 PM

Listing 5-15. The TLD document

```
<?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>
                J2EE frontend technologies; servlets, JSPs and
                EJBs tutorial tag library
        </shortname>
        <uri>http://localhost/taglib/tagz.jar</uri>
        <info>Sample Tutorial Tag Library</info>

        <tag>
                <name>FirstTag</name>
                <tagclass>se.jguru.tags.FirstTag</tagclass>
                <bodycontent>empty</bodycontent>
                <info>The first tutorial tag</info>
        </tag>
</taglib>
```

The XML `<tag>` element definition from the TLD DTD version 1.1 (shown in the code line below) is smaller than its 1.2 counterpart. Note that the only two mandatory subelements are `<name>` and `<tagclass>`:

```
<!ELEMENT tag (name, tagclass, teiclass?, bodycontent?, info?, attribute*)>
```

A graphical illustration of the tag element shows the simplicity even better (see Figure 5-23).

The code generated by the JSP engine to call the life-cycle methods of the `FirstTag` handler class becomes slightly different than the 1.2 equivalent. The only significant difference is that the named constant in one boolean expression is declared deprecated in 1.2, as shown here:

```
if (eval == BodyTag.EVAL_BODY_TAG)
throw new JspTagException("Since tag handler class se.jguru.tags.FirstTag "
+ "does not implement BodyTag, it can't return BodyTag.EVAL_BODY_TAG");
```

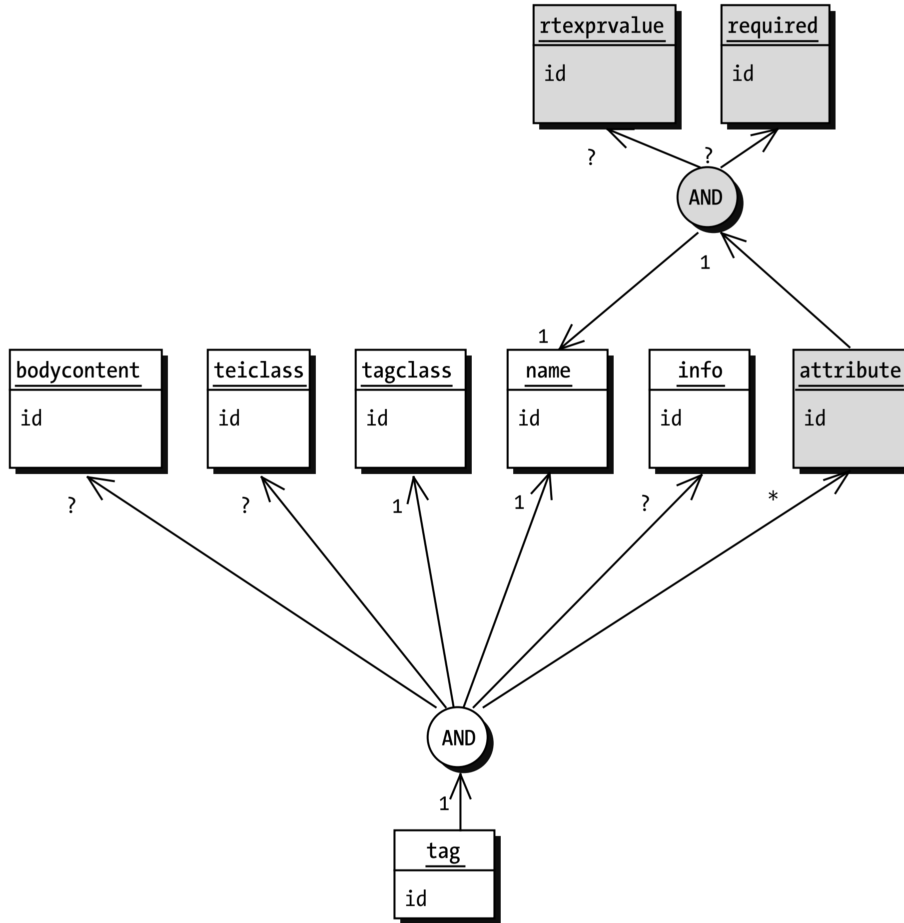

Figure 5-23. XML structure of the `<tag>` *element in the JSP 1.1 version of the TLD DTD*

Thus, although the JSP 1.2 specification may have altered quite a lot with regards to specification files and DTDs, the autogenerated code compiled to a servlet is practically the same between the two revisions.

Touring the javax.servlet.jsp.tagext Package

All classes for implementing the functionality of JSP custom tags are found in the `javax.servlet.jsp.tagext` package. The package contains two types of classes:

- Required interfaces of tag handler classes and their abstract implementation classes. Tags are separated into two categories, corresponding to the three tag handler interfaces (`javax.servlet.jsp.tagext.Tag`, `javax.servlet.jsp.tagext.IterationTag`, and `javax.servlet.jsp.tagext.BodyTag`) with their corresponding helper classes (`TagSupport` and `BodyTagSupport`).

- Metadata classes describing tag handlers, their required and optional attributes, and the operation of the tag. The information provided by the metaclasses includes: optional or mandatory body content, required parent tag, and any attributes required or variables declared.

The tagext metadata classes are numerous but well-designed, where each class encapsulates specific information about the tag. The main task of these metadata classes is encapsulating information from the TLD file and exposing it to the JSP engine. This metadata information allows the JSP engine to create the Java code of the autogenerated servlet and permits the developer to provide specifications to the JSP engine. Such specifications include valid value ranges for attributes, parameters, and so on.

Standalone tags are handled by a class implementing the `javax.servlet.jsp.tagext.Tag` interface. Body tags (possibly containing text) are handled by a class implementing the `javax.servlet.jsp.tagext.BodyTag` interface if the body content of the tag container should be modified during runtime or the `javax.servlet.jsp.tagext.IterationTag` interface otherwise.

All three interfaces are implemented by a related support class with empty method bodies, similar in construction to the `Adapter` classes of the `java.awt.event` package. Thus, when implementing a tag handler class, the developer may choose to implement the proper interface or extend its support class (which is frequently done, because it facilitates and speeds up the development). Note that both the `IterationTag` and the `BodyTag` interfaces extends the `Tag` interface, thus providing additional sets of methods that may operate on the body content of a container tag.

These three interfaces contain life cycle method definitions of all non-tag specific methods called by the JSP engine when it executes the code defined for a particular tag. Put simpler, the JSP engine will call only the methods defined in the three interfaces in Figure 5-24, plus any methods implied by descriptor entries in the TLD file. Although not complex in nature, the indirect calling convention of the `Tag`, `IterationTag` and `BodyTag` methods make these classes worthwhile to study.

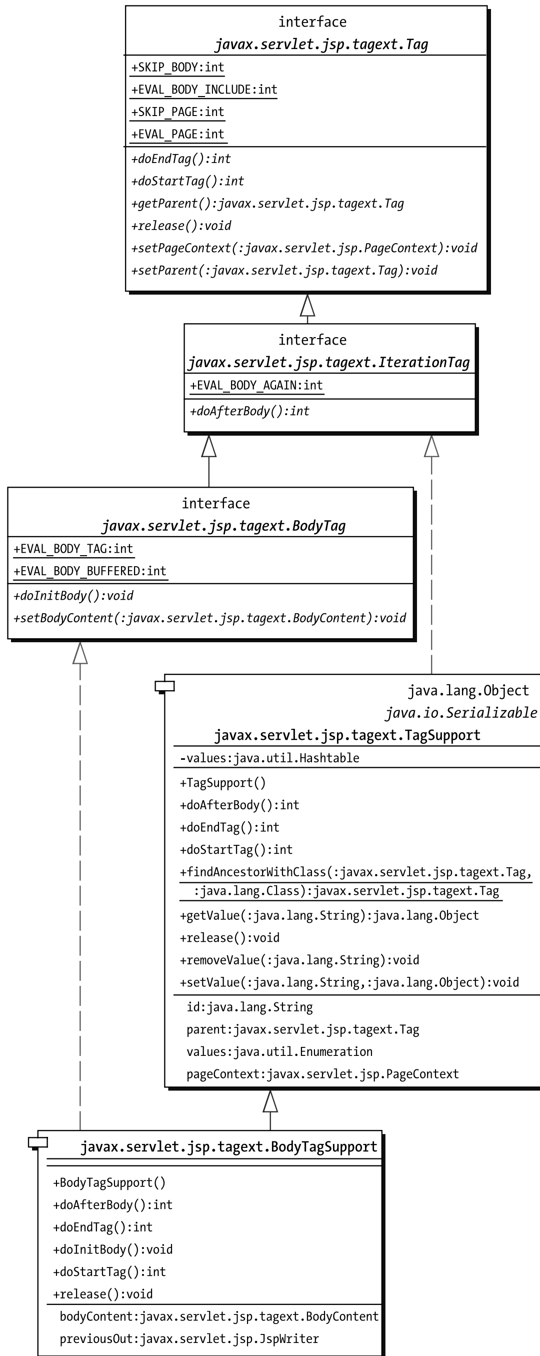

Figure 5-24. Structure of the tag specification interfaces (`Tag`*,* `IterationTag` *and* `BodyTag`*) and their implementation class counterparts (*`TagSupport` *and* `BodyTagSupport`*). Note that both the* `Tag` *and* `IterationTag` *interfaces are implemented by the* `TagSupport` *class.*

Interface javax.servlet.jsp.tagext.Tag

The Tag interface contains methods and constants defining the life cycle of a tag handler class (see Figure 5-25).

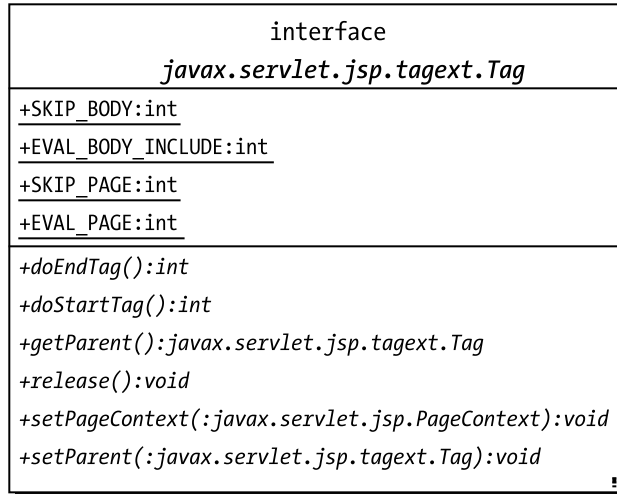

Figure 5-25. The Tag *interface*

Mainly, the methods fall in one of the two following categories:

- JavaBean property setter methods, called by the JSP engine to set the state of the tag handler instance. The corresponding JavaBean getter method is generally called from within the tag handler. The methods are `setParent(Tag t)` and `setPageContext(PageContext obj)`.

- Life-cycle methods called by the JSP engine at specific points in the execution of the tag handler. The methods are `doEndTag`,`doStartTag`, `release`.

Being root interface of all tag handler classes, the Tag interface defines the primary life-cycle methods called by the JSP engine when executing the methods belonging to a custom tag, as defined previously.

The JSP engine is responsible for completely setting up all internal data (by calling the JavaBean `setXXX` setter methods) prior to evaluating any tag content. This content is evaluated by calling the `doStartTag` and `doEndTag` methods.

968c05.fm Page 427 Tuesday, October 30, 2001 6:56 PM

Class javax.servlet.jsp.tagext.TagSupport

`TagSupport` is an `Adapter` class implementing an abstract infrastructure that supports all methods in the `Tag` interface, as shown in Figure 5-26. Thus, its main development benefit is reducing the amount of specific code that the developer must create.

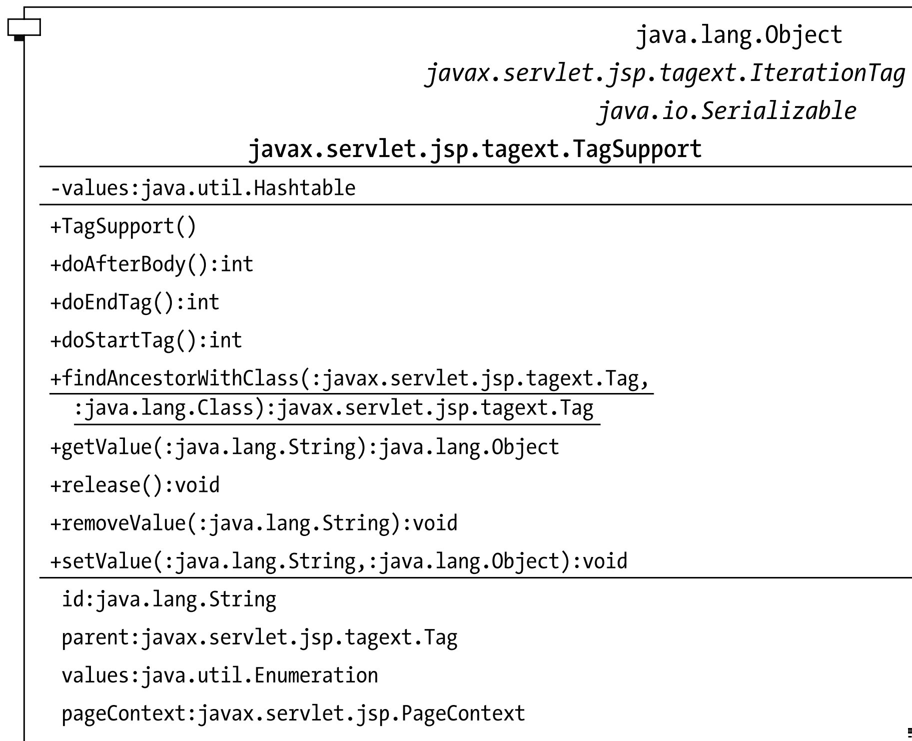

Figure 5-26. The `TagSupport` *class*

Tag-wide values, which should be accessible from all parts and methods, are stored within the `values` hashtable and retrieved using the `getValue`, `getValues`, `setValue`, and `setValues` methods.

Most methods of the `TagSupport` class are quite self-explanatory, given their good-fashioned naming convention. The only method that deserves some special attention within the `TagSupport` class is `findAncenstorWithClass`, which locates the closest enclosing tag handler of the particular class provided. Take a look at Listing 5-16, which illustrates the use of the `findAncestorWithClass` method.

968c05.fm Page 428 Tuesday, October 30, 2001 6:56 PM

Listing 5-16. The `findAncestorWithClass` *method*

```
// Let us assume that the JSP document has a structure like so:
//
//       <xyz:parentTag >
//             <xyz:firstChildTag >
//                    <xyz:innerChildTag />
//             </ xyz:firstChildTag >
//      </ xyz:parentTag >
//
// A nice way to access a reference to either of the enclosing tags
// from within the methods of the innerChildTag handler class is:

public void someMethodInInnerChildTagHandler()
{
       // Get a reference to the handler class of parentTag
       ParentTagHandler myParent = (ParentTagHandler) findAncestorWithClass(
              this, ParentTagHandler.class);

       // Check sanity
       if(myParent == null) throw new JspException("The innerChildTag must" +
              "be contained within a parentTag.");

       // Get a reference to the handler class of firstChildTag
       Child1TagHandler bigSister = (Child1TagHandler) findAncestorWithClass(
              this, Child1TagHandler.class);

       // Check sanity
       if(bigSister == null) throw new JspException("The innerChildTag must"
              + "be contained within a firstChildTag.");

       // Call some methods within the handler instances.
       ...
}
```

As shown in Listing 5-16, the `findAncestorWithClass` method may be used to obtain a reference to an enclosing parent having a particular type. This is useful to avoid type mismatches and corresponding exceptions when calling the parent from the child.

968c05.fm Page 429 Tuesday, October 30, 2001 6:56 PM

Interface javax.servlet.jsp.tagext.IterationTag

The `IterationTag` interface extends the `Tag` interface and defines an additional life-cycle method called by the Web container at the end of the `IterationTag` body evaluation. The `IterationTag` interface introduces but one new method, `doAfterBody`, as shown in Figure 5-27.

```
javax.servlet.jsp.tagext.Tag
interface
javax.servlet.jsp.tagext.IterationTag
-------------------------------------
+EVAL_BODY_AGAIN:int
-------------------------------------
+doAfterBody():int
```

Figure 5-27. The `IterationTag` *interface*

The `doAfterBody` method is called once after each pass through the body text. As long as the `doAfterBody` method returns `BodyTag.EVAL_BODY_AGAIN`, the iteration over the body text continues.

In practice, it is uncommon to implement the IterationTag interface directly; most tag handler classes requiring the functionality of the IterationTag would instead extend the `TagSupport` class (see "Class javax.servlet.jsp.tagext.TagSupport" later in this chapter).

Interface javax.servlet.jsp.tagext.BodyTag

The `BodyTag` interface extends the `IterationTag` interface, defining additional life-cycle methods that facilitate modifying the body text contained in the tag, as shown in Figure 5-28.

When the JSP engine generates the servlet, the calling order of the methods will always be:

1. The JSP engine creates a `BodyContent` object from the text in the container body and calls `setBodyContent` using the newly created object. Using methods within the `BodyContent`, the text may be retrieved or sent to the `JspWriter` of the enclosing tag handler.

2. `doInitBody` is called once, prior to first pass through the body text.

3. `doAfterBody` (defined in the `IterationTag` interface) is called once after each pass through the body text. As long as the `doAfterBody` method returns `BodyTag.EVAL_BODY_AGAIN`, the iteration over the body text continues.

```
          javax.servlet.jsp.tagext.IterationTag
          interface
       javax.servlet.jsp.tagext.BodyTag
+EVAL_BODY_TAG:int
+EVAL_BODY_BUFFERED:int
+doInitBody():void
+setBodyContent(:javax.servlet.jsp.tagext.BodyContent):void
```

Figure 5-28. The `BodyTag` *interface*

In practice, it is uncommon to implement the `BodyTag` interface directly; most tag handler classes requiring the functionality of the `BodyTag` would instead extend the `BodyTagSupport` class (refer to the following section for a walkthrough of the `BodyTag` class).

> **NOTE** *Refer to Chapter 6 for examples of extending the BodyTag class.*

Class javax.servlet.jsp.tagext.BodyTagSupport

Similar to the `TagSupport` class, the main benefit of extending from `BodyTagSupport` instead of directly implementing the `BodyTagSupport` interface is that the amount of trivial code one needs to create is somewhat reduced. Figure 5-29 shows the `BodyTagSupport` class.

Note that the added convenience JavaBean properties `bodyContent` and `previousOut` may simplify the development further. The `getBodyContent` method returns a reference to information about the body text, and the `getPreviousOut` method returns a reference to the `out JspWriter` of the enclosing context or tag. Normally, you need to use the `previousOut` to write text to the JSP document—which, in turn, is connected to the client browser.

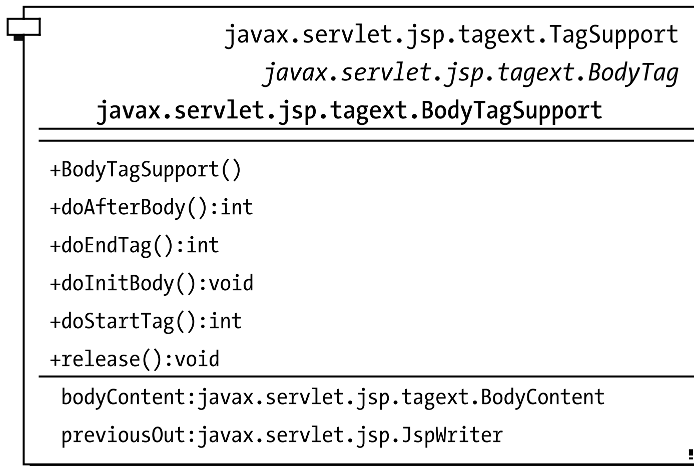

Figure 5-29. The `BodyTagSupport` *class*

Interface javax.servlet.jsp.tagext.TryCatchFinally

The `TryCatchFinally` interface shown in Figure 5-30 declares two methods, `doCatch` and `doFinally`, which are called from within automatically generated `catch` and `finally` blocks, respectively.

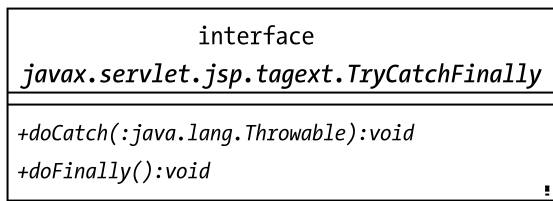

Figure 5-30. The `TryCatchFinally` *interface*

Thus far, the assumption has been that all code execution would progress according to plan. If the code of a tag handler class throws an exception during its execution, a somewhat tricky situation arises. How can the programmer handle the conditional execution that takes place in the `catch` and `finally` blocks following a `try` block? The simple answer is that such coding has been left up to the programmer before JSP version 1.2.

Although it is a simple task to encapsulate all text of a JSP document in a catch-all `try` block, it is not recommended, as the JSP document becomes burdened with a

mandatory `<% try { %> ... <% } catch(Exception ex) { // Handler goes here } %>` scriptlet. The JSP document content should replace the … for this pattern to work. Apart from making the document quite unreadable, the potential for errors in the JSP document using this structure is quite high.

The better alternative is to let your tag handler class implement the `TryCatchFinally` interface, having the two methods `doCatch` and `doFinally` as described in Table 5-3.

Table 5-3. The Function of Methods `doCatch` *and* `doFinally`

METHOD	DECLARATION
Handler method invoked from within the mandatory `catch` of the JSP body	`void doCatch(Throwable t)` throws `Throwable`.
Handler method invoked from within the `finally` block of the JSP body (in other words, this method is always invoked)	`void doFinally()`

NOTE *It is important that the* `doFinally` *method does not throw any exceptions because these cannot normally be handled by the autogenerated Java class.*

Java Code Generated from a TryCatchFinally

The stereotypical implementation of the JSP-generated code for a tag handler class implementing the `TryCatchFinally` interface reveals little new functionality. Note, however, the calls to the `doCatch(Throwable)` and `doFinally()` methods, which are inserted into the `catch` and `finally` blocks.

For improved readability, the `doCatch` and `doFinally` methods in the following listing appear in bold, and the full type paths converted into simple ones:

```
 /* ----  apt:VariableTag ---- */
VariableDefinitionTag vt = new VariableDefinitionTag();
vt.setPageContext(pageContext);
vt.setParent(null);
```

```
try
{
   int startTagResult = vt.doStartTag();
   if (startTagResult == BodyTag.EVAL_BODY_BUFFERED)
      throw new JspTagException("Since tag handler class VariableDefinitionTag "
      + "does not implement BodyTag, it can't return BodyTag.EVAL_BODY_TAG");
   if (startTagResult != Tag.SKIP_BODY)
   {
      do
      {
      } while (vt.doAfterBody() == BodyTag.EVAL_BODY_AGAIN);
   }
   if (vt.doEndTag() == Tag.SKIP_PAGE) return;
}
catch (Throwable t)
{
   vt.doCatch(t);
}
finally
{
   vt.doFinally();
   vt.release();
}
String foo = null;
foo = (String) pageContext.findAttribute("foo");
```

Extra bonus: The `VariableDefinitionTag` defines one variable, named `foo` of type `String`. From the previous code, can you determine the value of its `<scope>` element in the TLD?

Class javax.servlet.jsp.tagext.BodyContent

Non-empty tags, which enclose text between their opening and closing tag delimiters, may manipulate the enclosed text and optionally react to its content. The `javax.servlet.jsp.tagext.BodyContent` class, shown in Figure 5-31, encapsulates a buffer that holds the body text content and also contains the methods used to interact with the body text.

At construction time, the `BodyContent` instance is given a `JspWriter` object argument; this object should be the `JspWriter` connected to the browser output. One may thereafter get a reference to the contained `JspWriter` by calling the `getEnclosingWriter` method.

```
javax.servlet.jsp.JspWriter
javax.servlet.jsp.tagext.BodyContent
-enclosingWriter:javax.servlet.jsp.JspWriter
#BodyContent(:javax.servlet.jsp.JspWriter)
+clearBody():void
+flush():void
+getEnclosingWriter():javax.servlet.jsp.JspWriter
+getReader():java.io.Reader
+getString():java.lang.String
+writeOut(:java.io.Writer):void
```

Figure 5-31. The BodyContent interface

The `BodyContent` class extends `JspWriter`, so you may regard it as a local wrapper containing extra convenience methods that extracts the buffered contents from the wrapped writer. The methods can be divided into the two following logical groups:

- Retrieving a representation of the `BodyContent` string. If you wish to retrieve a String representation, use the `getString` method. Should you prefer a Reader representation, use the `getReader` method.
- Writing data to the `JspWriter` of the enclosing JSP page. This `JspWriter` is obtained by calling the `getEnclosingWriter` method. The methods are `writeOut(Writer out)`, `clearBody()`, and `flush()`.

The usage of the `BodyContent` and `BodyTagSupport` classes is simplest demonstrated with an example. You'll develop a custom JSP action called `SimpleBodyTag` and a JSP page called `simpleBodyTag.jsp`. The purpose of the `SimpleBodyTag` handler is to provide information about the text that was enclosed inside it. You can see the resulting output of running the `simpleBodyTag.jsp` page in Figure 5-32.

The relevant source code of the JSP document, which produced the output in Figure 5-32, is rather minimalistic. Note that the content of the leftmost cell simply illustrates the code in the JSP document and does not perform any actual server-side code invocations. The relevant code of the `simpleBodyTag.jsp` document is provided here, and the custom JSP action appears in bold text:

968c05.fm Page 435 Tuesday, October 30, 2001 6:56 PM

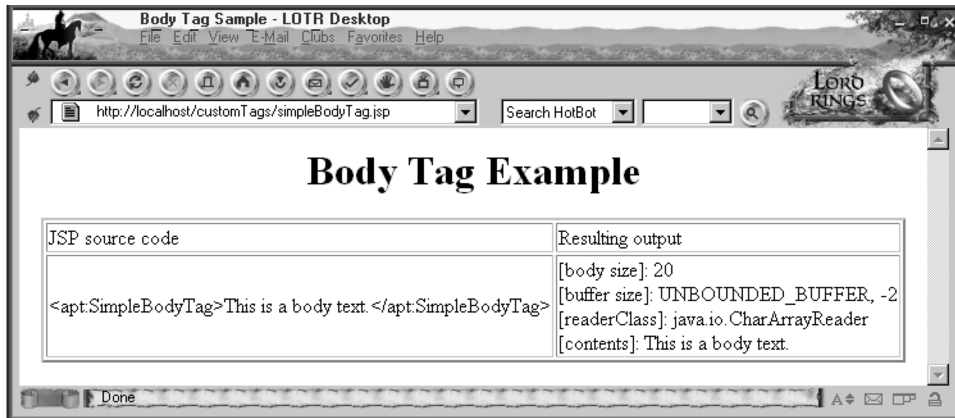

Figure 5-32. The resulting output of the `SimpleBodyTag` *example custom JSP action. Only the output in the table cell to the bottom right originate from the* `SimpleBodyTag` *handler; all other output comes from the JSP document,* `simpleBodyTag.jsp`.

```
<%@ taglib uri="apressTags" prefix="apt" %>
...
<tr>
<td>&lt;apt:SimpleBodyTag>This is a body text.&lt;/apt:SimpleBodyTag></td>
<td><apt:SimpleBodyTag>This is a body text.</apt:SimpleBodyTag></td>
</tr>
...
```

The JSP-to-servlet compiler generated the code in Listing 5-17 for the JSP document in the preceding snippet. Again, as the code generated may be less than perfectly simple to read for humans; its variable names, class paths, and indentation have been cleaned up for easier reading. The bold code in Listing 5-17 corresponds to the execution of the tag body. Note that the body content from the JSP document has been converted into the `out.write(“This is a body text.”);` statement.

Listing 5-17. The autogenerated code for the `SimpleBodyTag`

```
/* ----  apt:SimpleBodyTag ---- */
SimpleBodyTag sbt = new SimpleBodyTag();
sbt.setPageContext(pageContext);
sbt.setParent(null);
```

```
try
{
   int startResult = sbt.doStartTag();
   if (startResult != Tag.SKIP_BODY)
   {
      try
      {
         if (startResult != Tag.EVAL_BODY_INCLUDE)
         {
            out = pageContext.pushBody();
            sbt.setBodyContent((BodyContent) out);
         }
         sbt.doInitBody();
         do
         {
            out.write("This is a body text.");
         } while (sbt.doAfterBody() == BodyTag.EVAL_BODY_AGAIN);
      }
      finally
      {
         if (startResult != Tag.EVAL_BODY_INCLUDE)
         out = pageContext.popBody();
      }
   }
   if (sbt.doEndTag() == Tag.SKIP_PAGE) return;
}
finally
{
   sbt.release();
}
// end
```

The full code of the SimpleBodyTag handler class is smaller one than might think, as simply extending the BodyTagSupport class can perform powerful tasks without large amounts of custom code. In this case, the only custom code of the SimpleBodyTag has been supplied in the doAfterBody method. The simple UML diagram of the tag handler class is shown in Figure 5-33.

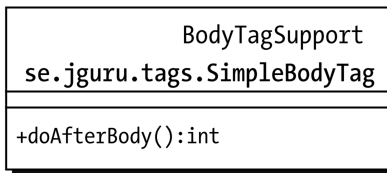

Figure 5-33. The structure of the `SimpleBodyTag` *handler class is minimalistic—only the* `doAfterBody` *method needs be overridden from the* `BodyTagSupport` *superclass. All other methods are run with the default behaviour from* `BodyTagSupport`.

Listing 5-18 displays the full code for the `SimpleBodyTag` handler class.

Listing 5-18. The `SimpleBodyTag` *handler class*

```
/*
 * Copyright (c) 2000 jGuru.se
 * All rights reserved.
 */

package se.jguru.tags;

import java.util.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class SimpleBodyTag extends BodyTagSupport
{
    /**
     * Method which prints out some status messages regarding its
     * body contents. Therefore, the BodyTagSupport
     * class is used as a superclass of this Tag handler.
     */
    public int doAfterBody() throws JspException
    {
        // Get the BodyContent.
        BodyContent bc = this.getBodyContent();

        // Get the body content as a string.
        String bodyAsString = bc.getString();

        // Get the JspWriter connected to the JSP document
        JspWriter out = this.getPreviousOut();
```

```
        // Find out some Reader metadata
        int bufferSize = bc.getBufferSize();
        String strBufSize = "UNKNOWN, size=" + bufferSize;
        if (bufferSize == out.DEFAULT_BUFFER)
                strBufSize = "DEFAULT_BUFFER, " + bufferSize;
        if (bufferSize == out.NO_BUFFER)
                strBufSize = "NO_BUFFER, " + bufferSize;
        if (bufferSize == out.UNBOUNDED_BUFFER)
                strBufSize = "UNBOUNDED_BUFFER, " + bufferSize;

        // Get the fully qualified class name of the BodyContent reader
        String readerClass = bc.getReader().getClass().getName();

        // Clear the original body content from the JSP document.
        // If we do not, the tag body from the JSP document will
        // be written to the client browser. However, our intention
        // is to output onlyprocessed text with metadata about
        // the body text. We must therefore clear the original body,
        // and create a new one.
        bc.clearBody();

        // Now, generate the new body content of this tag.
        try
        {
            out.println("[body size]: " + bodyAsString.length() + "<br>");
            out.println("[buffer size]: " + strBufSize + "<br>");
            out.println("[readerClass]: " + readerClass + "<br>");
            out.println("[contents]: " + bodyAsString + "<br>");
        }
        catch (Exception ex)
        {
            System.out.println("Whoops! " + ex);
        }

        // Done.
        return Tag.SKIP_BODY;
    }
}
```

The `doAfterBody` method of Listing 5-18 prints metadata corresponding to the `BodyContent` held by the tag. The explanation of each individual statement in Listing 5-18 can be found in its comments.

Class javax.servlet.jsp.tagext.TagExtraInfo

The `TagExtraInfo` class shown in Figure 5-34, is used as a superclass from which descriptions about tag handlers are derived.

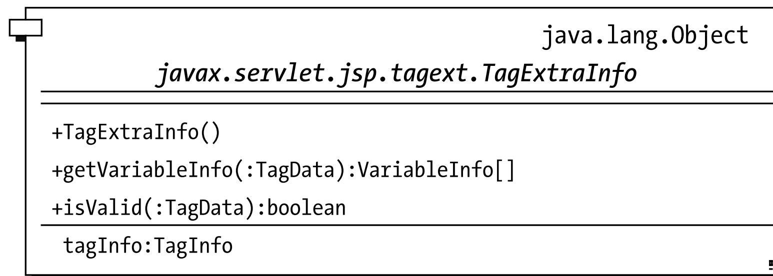

Figure 5-34. The `TagExtraInfo` *class*

The extra information about a tag handler provided by the `TagExtraInfo` class is required for proper operation in two situations:

- When your tag handler defines scripting variables (in other words, variables existing within the boundaries of a container tag or from the tag onwards within the JSP document). Type information about scripting variables is given through the method `public VariableInfo[] getVariableInfo(TagData data)`, where the JSP engine encapsulates attribute metadata information in the `TagData` object and passes it as a method argument. The return array contains information and about all variables/attributes created or modified by the tag handler.

- When you want to perform validation checking of attributes, provided within a JSP tag. Such validation is achieved by the method `public boolean isValid(TagData data)`.

Both these methods return or handle arguments that encapsulate attribute or data information in a similar manner. Their specific details are described in the following sections on `TagData` and `VariableInfo`, respectively.

In addition, the `TagExtraInfo` class contains a `TagInfo` object, which contains information about the attributes declared in the TLD file. This `TagInfo` object can be retrieved from the `TagExtraInfo` class by calling `public TagInfo getTagInfo()`.

Class javax.servlet.jsp.tagext.TagInfo

The TagInfo class, shown in Figure 5-35, is used to retrieve runtime metadata about the attributes of a tag handler class.

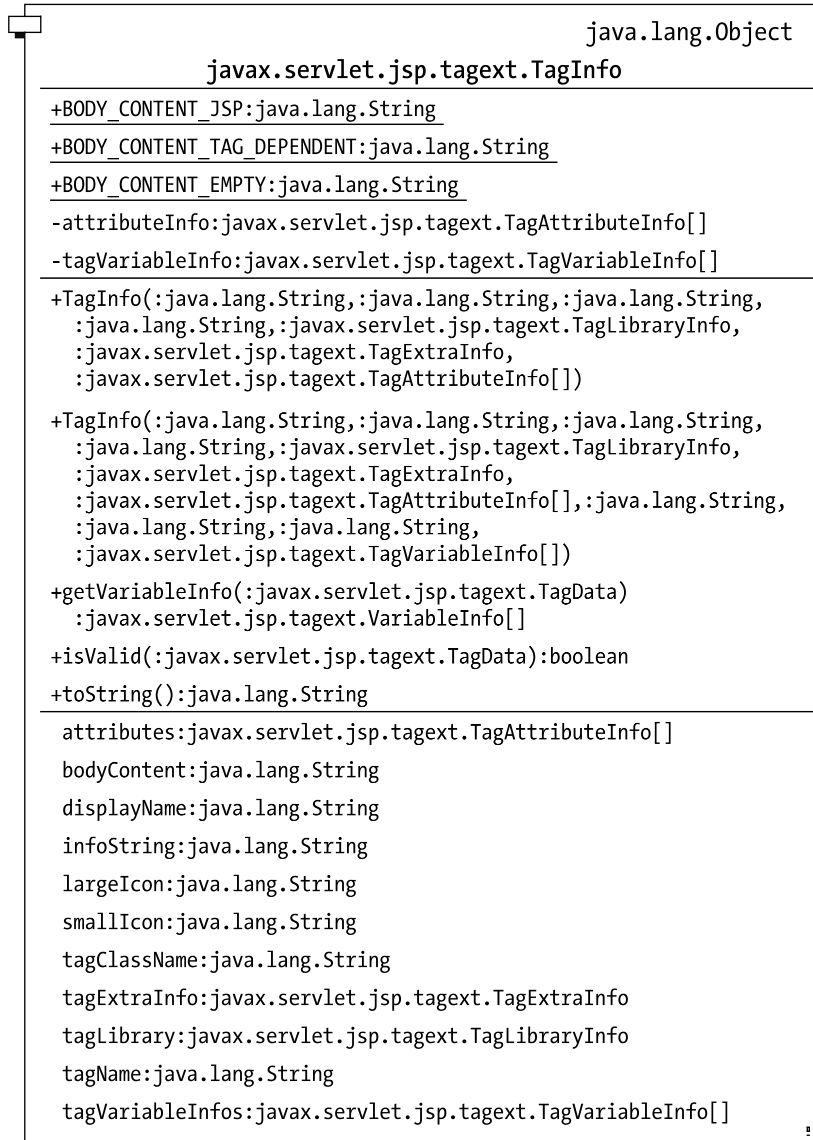

Figure 5-35. The TagInfo *class*

The metadata retrieved from calling methods in the `TagInfo` class is interpreted form the TLD file. The methods of the `javax.servlet.jsp.tagext.TagInfo` are of three main types:

- Methods that return a string form of the data found in the `taglib.tld`. The methods are `getAttributes`, `getTagName`, `getBodyContent`, and `getTagClassName`.
- Methods that return containers holding metadata describing other JSP entities. The methods are `getTagExtraInfo` and `getTagLibrary`.
- Methods that return runtime information from parsing the JSP document. The method is `getVariableInfo`.

The `toString` method of a `TagInfo` object returns a `String`, which provides some information about the contents of a `TagInfo` object.

Listing 5-19 is the output result of calling the `toString` method on a `TagInfo` instance. Compare the output in the listing with the provided information in the `taglib.tld` file. Can you find any discrepancies?

Listing 5-19. Printout of a `TagInfo.toString()` *call*

```
name = printTypeTree
class = se.jguru.tags.PrintTypeTreeTag
body = JSP
info = Tag iterating over a Map of instances, exposing each instance using its
target JavaBean property.
attributes = { name = allTargets type = null reqTime = true required = true}
```

The relevant parts of the TLD file which was used as the source for the listing in 5-19, is reprinted in Listing 5-20.

Listing 5-20. The tag definition in the TLD file

```
<tag>
        <name>printTypeTree</name>
        <tag-class>se.jguru.tags.PrintTypeTreeTag</tag-class>
        <tei-class>se.jguru.tags.PrintTypeTreeExtraInfo</tei-class>
        <body-content>JSP</body-content>
        <description>Tag iterating over a Map of instances,
                exposing each instance using its target JavaBean
                property.</ description>
```

```
        <attribute>
                <name>allTargets</name>
                <required>yes</required>
                <rtexprvalue>yes</rtexprvalue>
        </attribute>
</tag>
```

As you can see, the `TagInfo` class simply contains the parsed version of the TLD file.

Class javax.servlet.jsp.tagext.TagData

`TagData` objects contain metadata information describing attribute names and values used by tags (see Figure 5-36).

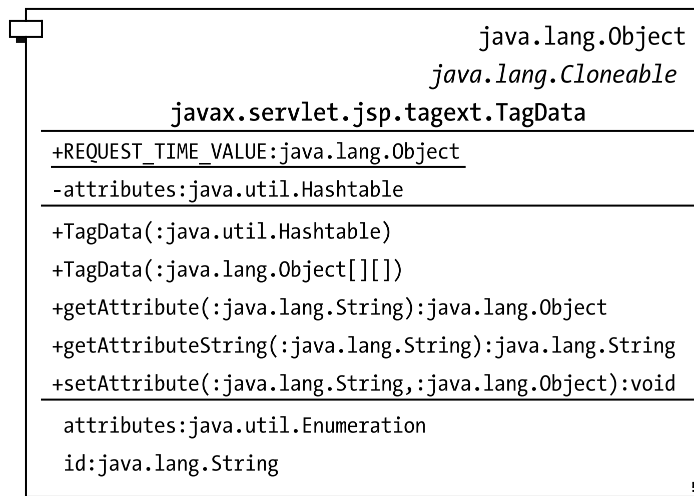

Figure 5-36. The `TagData` *class*

The `TagData` instance vaguely mimics the behavior of a Hashtable from which data may be retrieved using the `getAttribute` methods. `TagData` instances are created and populated by the JSP engine for your convenience.

If the scripting variable described by a particular `TagData` object has not yet been given a value (such as, for instance, is the case with runtime variables being set by the Tag handler), the scripting variable has the value `TagData.REQUEST_TIME_VALUE`. This value simply distinguishes variables having been set from those not yet given a value.

The three relevant getter methods of the `TagData` class are:

- `public Object getAttribute(String attributeName)` is a normal hashtable getter method, which retrieves the value of the attribute with the provided name.

- `public String getAttributeString(String attributeName)` is a normal hashtable getter method, which retrieves the value of the attribute with the provided name. The difference between the `getAttributeString` and the `getAttribute` methods is simply that the former will typecast the results to a string.

- `public String getId()` returns the identifier of the tag (in other words, its `id` attribute value).

Class javax.servlet.jsp.tagext.VariableInfo

Each `VariableInfo` instance contains information about a scripting variable created or modified by a tag handler. All methods in this class, shown in 5-37, are getter JavaBean methods that return information about a declared variable.

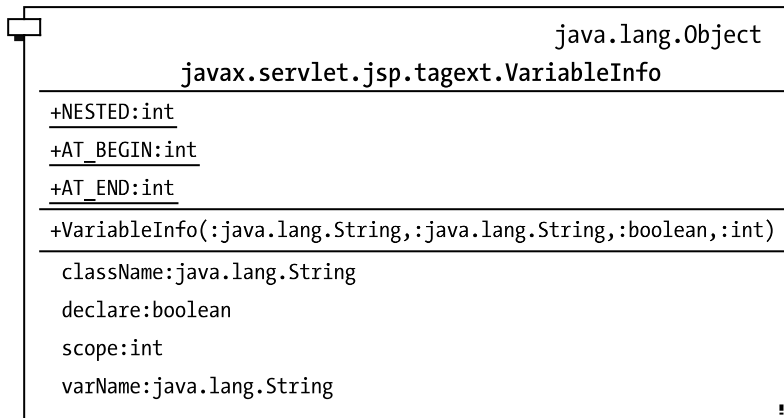

Figure 5-37. The `VariableInfo` *class*

The following methods of the `VariableInfo` class have simple modes of operation:

- `public String getVarName()` returns the name of the variable described. Say that a `VariableInfo` instance `obj` describes a variable that was declared with the statement `String aCuteString;`. The call to `obj.getVarName()` would return the value `"aCuteString"` describing the variable name.

968c05.fm Page 444 Tuesday, October 30, 2001 6:56 PM

- `public String getClassName()` returns the name of the class of the variable. Say that a `VariableInfo` instance `obj` describes a variable that was declared with the statement `String aCuteString;`. The call to `obj.getClassName()` would return the value "`java.lang.String`" describing the class name of the declared variable.

- `public String getDeclare()` returns `true` if the variable is declared by the tag handler class for which this `VariableInfo` object is provided.

- `public int getScope()` returns one of the constants defined in the `VariableInfo` class and reveals which scope the declared variable will have in the JSP document.

- `VariableInfo.NESTED` denotes that the variable exists only between the start and end tag delimiters of the tag whose variable is described by the `VariableInfo` object.

- `VariableInfo.AT_BEGIN` indicates that the variable is defined from the start tag delimiter to the end of the JSP document.

- `VariableInfo.AT_END` indicates that the variable is defined from the end tag delimiter to the end of the JSP document.

Figure 5-38 depicts the areas of the JSP document where a declared variable will be visible. Of course, the variable must be declared by a tag handler class; examples of classes declaring variables will be provided in the next chapter.

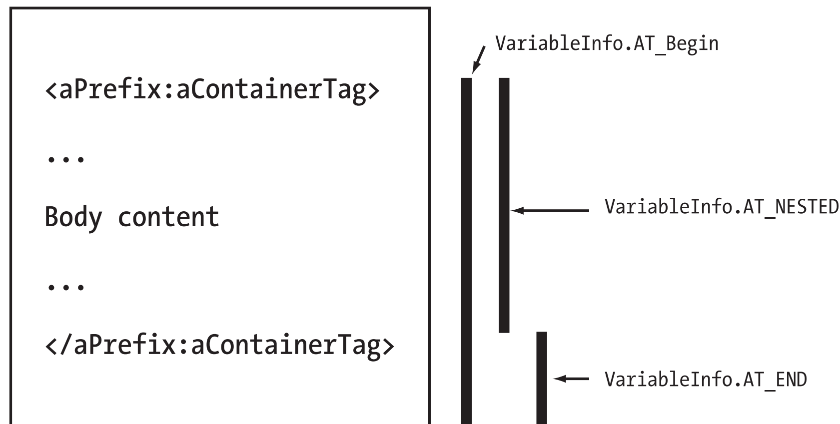

Figure 5-38. The different declaration scopes of a declared variable

968c05.fm Page 445 Tuesday, October 30, 2001 6:56 PM

Class javax.servlet.jsp.tagext.TagAttributeInfo

The `TagAttributeInfo` class, shown in Figure 5-39, is a simple storage class for parsed data, and its methods retrieve information about attributes of a tag handler, as defined in the `taglib.tld` file.

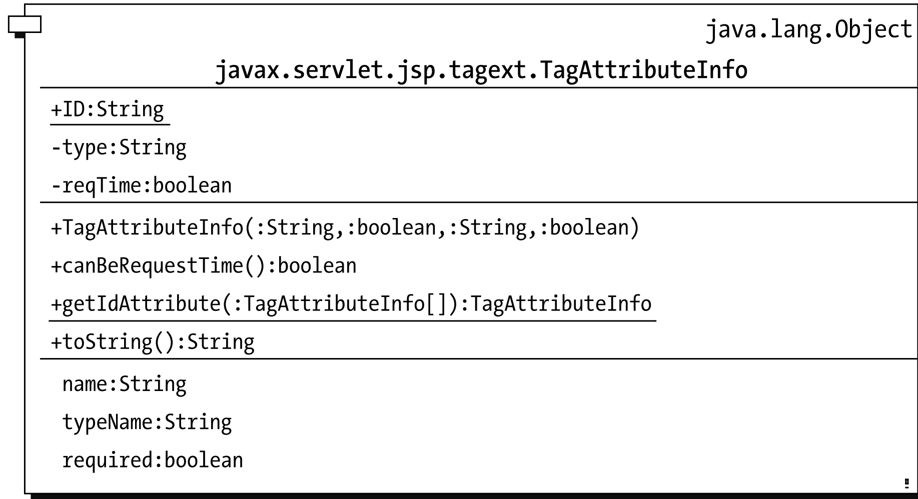

Figure 5-39. The `TagAttributeInfo` *class*

The methods fall in the two following categories:

- Methods returning parsed and interpreted values from the attribute definitions provided in the `taglib.tld` file. The methods are `canBeRequestTime`, `getName`, `getTypeName`, `isRequired`, and `toString`.

- Static convenience method that returns the `TagAttributeInfo` object for the `id` attribute, corresponding to the variable name within the generated servlet. The method is `getIdAttribute(TagAttributeInfo a[])`.

Class javax.servlet.jsp.tagext.TagLibraryInfo

The `TagLibraryInfo` class shown in Figure 5-40 is a simple storage class for parsed data, and its methods retrieve data provided in the taglib descriptor file.

All methods are JavaBean getter methods, which return the data in the variable with the same name. For instance, the `getShortName` method returns the value kept within the `shortname` variable.

968c05.fm Page 446 Tuesday, October 30, 2001 6:56 PM


```
                                        java.lang.Object
          javax.servlet.jsp.tagext.TagLibraryInfo
-----------------------------------------------------------------
#prefix:java.lang.String
#uri:java.lang.String
#tags:javax.servlet.jsp.tagext.TagInfo[]
#tlibversion:java.lang.String
#jspversion:java.lang.String
#shortname:java.lang.String
#urn:java.lang.String
#info:java.lang.String
-----------------------------------------------------------------
#TagLibraryInfo(:java.lang.String,:java.lang.String)
+getInfoString():java.lang.String
+getPrefixString():java.lang.String
+getReliableURN():java.lang.String
+getRequiredVersion():java.lang.String
+getShortName():java.lang.String
+getTag(:java.lang.String):javax.servlet.jsp.tagext.TagInfo
+getTags():javax.servlet.jsp.tagext.TagInfo[]
+getURI():java.lang.String
```

Figure 5-40. The `TagLibraryInfo` *class*

The need to manually extract data from the `TagLibraryInfo` class is rather limited, as it is mainly intended to be used by the JSP engine in its metadata extraction.

Class javax.servlet.jsp.tagext.TagLibraryValidator

As the developer of a tag library, you may want to make sure that the <@% taglib ... %> directive is printed in a proper way. Before JSP version 1.2, there was no simple way of performing such validation—the `TagLibraryValidator` class, shown in Figure 5-41, has been added to the JSP version 1.2 class library to provide a simple way of validating a `TagLib` directive within a JSP document.

A `TagLibraryValidator` instance is global for all directives referring to a particular TLD file. Its fully qualified class is defined as an XML element in the TLD file, as seen in earlier “The <validator> Element” section.

The four methods in the class fall in the two following categories:

968c05.fm Page 447 Tuesday, October 30, 2001 6:56 PM

Figure 5-41. The `TagLibraryValidator` *class*

- Methods manipulating the initialization parameters and state (instance variables) of the `TagLibraryValidator` instance. (All parameters are provided in the TLD file). These methods are `setInitParameter(Map aMap)`; `Map getInitParameters()`; and `release()`.
- Methods handling life-cycle validation for the `TagLibraryValidator` instance. This method is `validate(String prefix, String uri, PageData page)`.

Any non-null result from the `validate` method is regarded as an error message by the JSP engine. If the validate method returns a non-null string, it is wrapped in an `Exception` object and thrown by the JSP engine. Be sure to return `null` from the validate method to indicate successful validation. If running the Tomcat-4.0 reference implementation, the root exception prints the following message:

```
org.apache.jasper.JasperException: TagLibraryValidator in APress
   tutorial TagLib - invalid page: This is an error message from
   the validator.
```

The last part of the message ("This is an error message from the validator") is the string that was returned from the `validate` method, and the ("APress tutorial TagLib") is the short name (content of the `<short-name>` element) of the TLD file.

The Role of JSP Tag Libraries in Middle-Tier System Development

JSP taglib definitions are indeed flexible and powerful to use in the development of server-side customized JSP functionality. The massive increase in complexity to plan, create, and implement JSP tag libraries compared using the standard JSP functionality is evident to most developers investigating taglib functionality for

the first time. Thus, the relevancy and major benefit of taglibs require a bit of evaluating.

Many server-side systems have front-end functionality that mimics the Model View Controller (MVC) design pattern to divide the separate functionalities of the server. Thus, the middleware front ends commonly use either of the design patterns (view or controller) illustrated in Figure 5-42.

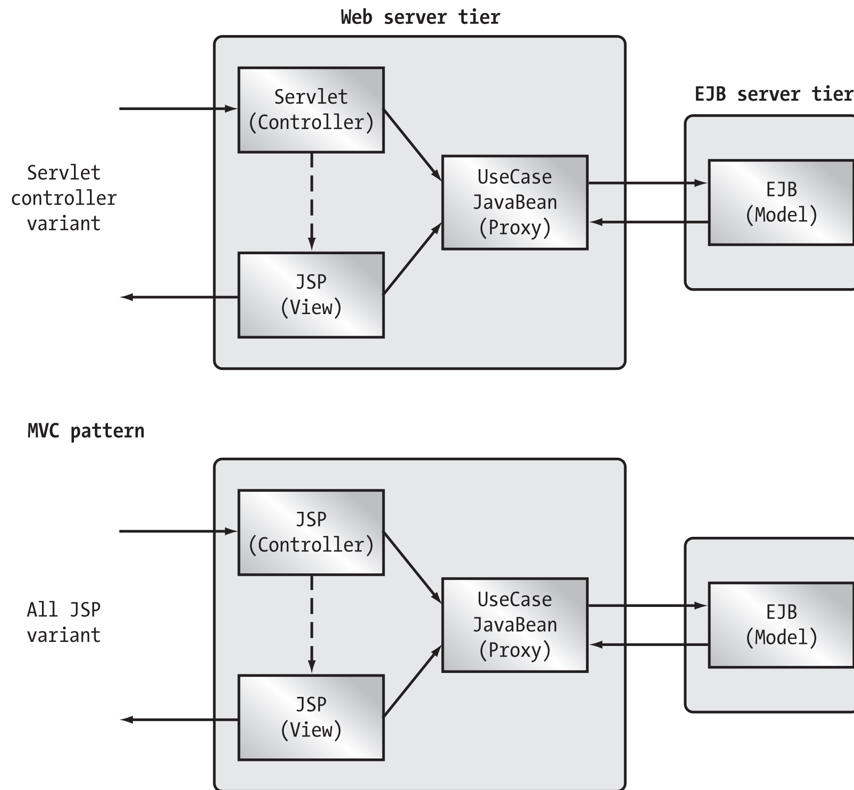

Figure 5-42. Frequently used system-wide design pattern for collaborating components or class clusters when developing a Web application system

Briefly consider the functionalities and call order of the two front-end entities (in other words, the controller and view of the MVC design pattern). The tasks of the three objects in the pattern are:

- *Controller* reacts to the input and commands sent by the user. Any capture of user-supplied data or command parameters is handled by the controller servlet. When finished in reading all data provided by the user, the controller servlet updates the model to incorporate the commands from the user.

968c05.fm Page 449 Tuesday, October 30, 2001 6:56 PM

- *Model* holds the business object state and behavior. All system entities should be held and manipulated within an application server. Figure 5-42 illustrates the standard J2EE model that assumes an enterprise JavaBeans (EJB) server as the application server. An application server of some type should always be used, unless the system is sufficiently small that the model can be included in the controller or view. Such design is recommended only for the smallest of systems. The dynamic data required by the view is read from the model and sent to the view encapsulated within a JavaBean instance.

- *View* defines non-dynamic HTML code areas and joins it with the dynamic data extracted from the model. The actual joining is performed by calling JavaBean getter property methods in the value object, holding model data.

The joining of dynamic and static data presents a fundamental problem, as two professional roles must work together to create the final result. The developer is responsible for the dynamic data, and the Web designer/content manager is responsible for the static data of the view. The system code must be shielded from improper modifications (such as tampering with the JSP tag syntax) by mistake—enforcing data encapsulation in the server tier reduces the risk of having somebody corrupting the JSP document.

Tag library definitions and customized server-side tags provide a good way of encapsulating the data from the model. In general, more encapsulated and data-wise hidden modes of operations tend to be more stable in the long run. Also, simpler models and modes of usage tend to have a rewarding quality about themselves.

The Apache framework Struts uses the model illustrated in Figure 5-42. Refer to `http://jakarta.apache.org/struts/` for a reference to the Struts architecture—or the next chapter for a walkthrough.