6 Tag development techniques

In this chapter

- Common programming tasks with tags
- Integrating tags with their environment
- Tag cleanup
- Base classes for future examples

Just when you thought you might be able to settle in and start using all your newfound knowledge about writing tags, the custom tag API, and life cycles, you discover that there needs to be something more. Mastering the details of a TLD's anatomy and its role in making tags known to the JSP runtime engine may mean the ability to build and deploy simple tags; unfortunately, as with any technology, knowing the basics will only get you so far. To build tags that can actually make a difference in your projects, you need a few key programming techniques that will prove highly beneficial for most of the tags you build. You can think of these techniques as tools in the toolbox that holds all your tag development knowledge. The tools we discuss in this chapter will be the mainstays in your daily tag construction.

6.1 Reusable tag programming techniques

Once you've tried out the examples in the previous chapters and built some basic tags, you're likely to find yourself in need of additional techniques pertaining to tag behavior, initialization, configuration, and cleanup. It's vital that you know how to reliably write content to the user, pass parameters to tags, and share information between tags. You may also benefit from knowing how to configure an entire tag library via a single application variable, and how to create tags that inspect and modify their body. Of course, after adding all this functionality, you'll have to be able to write your tags in a way that they can be properly cleaned up after the runtime has finished with them. Though these techniques vary greatly, the common theme is that each is a typical component in the construction of a production tag library.

6.1.1 The techniques you'll use most

What kinds of applications will benefit from the techniques in this chapter? Most of them! This chapter covers the programming techniques most commonly used when building a tag-based application. They are:

- Writing content back to the user
- Logging and messaging
- Using tag attributes
- Using the Servlet API
- Configuring tags
- Writing tags that modify or use their body
- Properly cleaning up state after your tag has executed.

Let's discuss these techniques and how they fit into a typical web application.

Writing content back to the user

Almost any web application you build with tags is likely to have at least one tag that performs the task of getting a value and returning it to the user. This value may come from a database, a cookie, another web server, or perhaps an Enterprise JavaBean (EJB). This kind of tag can be used to echo a username, as in "Welcome back, Cole," report a bank balance, or present some other piece of data stored in the database.

Using tag attributes

Sometimes a tag will take a parameter from the JSP author so that it may behave differently under different circumstances. The ability of tags to take parameters (typically made possible by tag attributes) is what makes your tags flexible enough to be reused across projects.

Logging and Messaging

Like any programming project, writing tags will require the ability to log error messages as they occur and propagate them to the developer/administrator.

Using the servlet API

Since JSP custom tags run in the same environment as servlets and JSPs, namely the Web, it is also to be expected that your tags need to interact with the same kinds of web-related information that most servlets and JSPs use. This includes reading from and writing to cookies, looking at HTTP headers, redirecting requests, and so forth. Almost any web application has a need for some of these functions, and web applications built on custom tags are no different.

Configuring tags

When building a web application that uses tags, you might like to configure some aspects of a tag (or group of tags) in a central place, rather than in each and every JSP that uses those tags. We may, for example, wish to build a suite of tags that send email and use a particular mail server to do so. It would, in such a case, be ideal to indicate that server name once in a central place, and allow any JSP that has our email tags to pick up and use that property. This type of configuration, though optional, can often clean up your design and make implementing changes fairly painless. Not all tag libraries will require this kind of configuration, but even most small libraries will have at least one or two settings that would benefit from being configured in a central place.

Writing tags that modify/use their body

For many (if not most) tags, looking at or changing their bodies is not necessary. This tactic is useful for tags that need to take parameters too complex to be passed as attributes, or for tags that want to operate on a block of HTML or text.

Cleaning up

Of all the techniques, learning how to properly clean up after your tags is the most important we'll discuss, since any and all applications should have tags that leave resources and state clean after they've run.

Now that we know what these techniques can do for us, let's look at each technique in detail.

6.2 *Writing content back to the user*

Returning content from your tags is probably the most widely known technique. Generally, custom tags write a bit of content to the page for the user to see. Though some tags may run without creating any user-visible output, such as a tag that iterates through a set of parameters or exports new beans into the page, the majority of tags will ultimately write text into the response flowing back to the user. We saw three such tags in chapter 3.

To facilitate this requirement, the JSP infrastructure provides tags with a special `Writer` class called `JspWriter`. With this class, a tag can include any text you choose in the web server's response to a user. The advantage of including text in this way is that it appears in the proper place on the page in the user's browser.[1] The methods that are of greatest interest to us in the `JspWriter` are in table 6.1.

Table 6.1 Important methods in the `JspWriter` class

Methods use for printing to the user	Methods used for buffer manipulation
abstract public void newLine()	abstract public void clear()
abstract public void print(boolean b)	abstract public void clearBuffer()
abstract public void print(Object obj)	
abstract public void println()	abstract public void flush()
abstract public void println(Object x)	public int getBufferSize()

[1] Much of our discussion in this chapter assumes a classic web application model with a standard HTML browser for a client. It should be mentioned that, like any servlet or JSP, custom tags can return data to anything that issues an HTTP request. This includes WAP browsers, Internet spiders, or any other process that asks the web server for a page.

Table 6.1 Important methods in the `JspWriter` class (continued)

Methods use for printing to the user	Methods used for buffer manipulation
	public boolean isAutoFlush()

As you can see, the `JspWriter` offers the following facilities in addition to those available in the simple `Writer` we already know:

- Print—The original Java `Writer` only supports writing chunks of data from an array. `JspWriter`, on the other hand, adds those methods that usually exist in the `PrintWriter` class. This way you can easily print data (such as `String`, primitive types, and Objects) to the response.
- Buffer manipulation—The output returned by a servlet or JSP typically is buffered. This buffering is implemented by the Servlet/JSP container. The `JspWriter` class provides methods to query the buffer's internal state and to clear its contents.

A tag may obtain a reference to the `JspWriter` to use for output in the current page by calling the method `PageContext.getOut()`.

NOTE The `JspWriter` returned by `PageContext.getOut()` is not always connected directly to the user. The JSP runtime can use multiple `JspWriters` to collect the output of certain page fragments. For this reason, the returned `JspWriter` may change from call to call; in fact, the JSP engine is holding a stack of `JspWriters` that correspond to the file structure of an individual JSP. The contents of all these individual `JspWriters` are concatenated after the processing and it's that concatenated content that is sent to the user. This is explained in greater detail later in this chapter.

Now, we look at how to use the `JspWriter` to manipulate the response.

6.2.1 Adding data to the output

Listing 6.1 shows a code fragment taken from `ShowFormParamTag` which demonstrates writing data to the user.

Listing 6.1 Printing output to the user

```
public class ShowFormParamTag extends TagSupport {

    // Some other code was omitted here.

    public int doStartTag()
```

```
            throws JspException
    {
        try {
            HttpServletRequest req =
                (HttpServletRequest)pageContext.getRequest();
            String value = req.getParameter(name);
            if(null != value) {
                writeHtml(pageContext.getOut(), value);        ❶
            }
            return SKIP_BODY;
        } catch(java.io.IOException ioe) {                     ❷
            // User probably disconnected ...
           //log an error and throw a JSPTagException
            //...
        }
    }
    // Some other code was omitted here too.
}
```

❶ **Performs HTML special tags filtering and writes the output back to the user** `ShowFormParamTag` prints the value of a particular form parameter sent by the user, but for this discussion we've tried to omit all the code that is not directly related to the actual printing. From this code fragment you can see that the `JspWriter` is obtained through a call to `pageContext.getOut()`.

❷ **Handles the ever annoying `IOException`. Logs the exception and interrupts the JSP execution by throwing a `JSPTagException`.**

Writing HTML properly with writeHtml()

You'll note that we are not using the `JspWriter` directly, as we did in chapter 3. Instead of calling `pageContext.getOut().print(value)` we are calling a method we've written called `writeHtml()` to print the parameter `value` to the response. Why take this extra step? If you look at `writeHtml()` (listing 6.2) you'll see that it simply applies the proper escape sequences for special HTML characters such as "<", ">", and "&". The incidents in which we'll need to pass our output through `writeHtml()` will be those when we aren't sure if the String we are writing to the user contains any of these special characters. We want to make sure the user reads the String as it was intended to be; and not allow it to be accidentally interpreted by the browser as HTML. Consider, for example, a case where our `ShowFormParamTag` is being used to echo an individual's username that was just submitted on a previous form. A malicious user could enter the username as the following:

```
<script>
    alert("this is a big bad virus, the site is not protected!!!")
</script>
```

If we write this username back to the response unescaped, the browser will interpret it as standard Javascript and the user's evil alert message will pop up. By passing the parameter through `writeHTML()` instead, we convert all of the "<" and ">" characters to their escaped equivalents. The user then sees the text they've typed echoed back to them verbatim, instead of the ill-intentioned JavaScript message.

Listing 6.2 The writeHtml() method defined.

```
protected void writeHtml(JspWriter out,
                         String html)
    throws IOException
{
    if((html.indexOf('<') == -1) &&
       (html.indexOf('>') == -1) &&
       (html.indexOf('&') == -1)) {
        out.print(html);
    } else {
        int len = html.length();
        for(int i = 0 ; i < len ; i++) {
            char c = html.charAt(i);
            if('<' == c) {
                out.print("&lt;");
            } else if('>' == c) {
                out.print("&gt;");
            } else if('&' == c) {
                out.print("&amp;");
            } else {
                out.print(c);
            }
        }
    }
}
```

Defining base classes for our tags

Since `writeHtml()` is a useful method to have in any tag that returns content to a user, we'll want to use it in most of the tags we develop. Now is a good time to define a base class for our tags where we can place fuctionality like this. We'll call this class `ExTagSupport` and it will serve as the base for most of our tag examples for the remainder of the book. Throughout the upcoming chapters, we'll add to `ExTagSupport` as we encounter logic that we want inherited by all our tags. For now, `ExTagSupport` will define only one method, `writeHtml()`, and, of course, extend the `TagSupport` utility class. We'll need a base class for our BodyTags which will extend `BodyTagSupport`. We'll call that class `ExBodyTagSupport`. You'll see references to both these classes throughout examples in the remainder of this book.

NOTE We don't need to list `ExBodyTagSupport` separately here because, for now, it is identical in listing 6.3, except for the class name and the fact that it extends `BodyTagSupport` instead of `TagSupport`.

Listing 6.3 Our base class for future tag development: ExTagSupport

```
package book.util;

import java.util.Enumeration;
import java.io.Exception;

import javax.servlet.ServletContext;
import javax.servlet.ServletConfig;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;

public class ExTagSupport extends TagSupport {

    protected void writeHtml(JspWriter out,
                             String html)
    throws IOException
    {
      if((html.indexOf('<') == -1) &&
         (html.indexOf('>') == -1) &&
         (html.indexOf('&') == -1)) {
        out.print(html);
      } else {
          int len = html.length();

          for(int i = 0 ; i < len ; i++) {
            char c = html.charAt(i);
            if('<' == c) {
                out.print("<");
            } else if('>' == c) {
                out.print(">");
            } else if('&' == c) {
                out.print("&");
            } else {
                out.print(c);
            }
          }
       }
    }
}
```

6.2.2 Exceptions and writing to the user

Note that the `print` methods in the `JspWriter` may throw `IOExceptions`. This is nothing new. IO-related methods throw `IOExceptions` all the time, but if you understand what is going on you can react accordingly. In the context of custom tags, why should there be an `IOException`? There are several reasons, some of which have to do with problems in connection with the user. For example, an `IOException` would be thrown if the user's browser crashed or the user pressed Reload while we were writing back to them. Other reasons stem from the implementation of JSPs. For example, if the JSP output buffer overflows and the autoFlush directive is set to `false`, the JSP runtime will generate an `IOException`.

No matter what the reason for the exception, we handle it properly and in a way consistent with the policy acceptable for our web server. To begin with, you'll definitely want to abort the page execution. To do so, throw a `JspException` (or, even better, a `JspTagException`) from your tag and the JSP runtime will do the rest. Unfortunately, aborting the page is typically not enough. `JspExceptions` can be a symptom of several problems, such as poor design (buffer overflow), slow site (a reason why the user pressed Reload), or denial of service attacks. To help identify any of these potential problems, we should also log this exception to the servlet container's log file for later analysis.

6.2.3 Flushing the JspWriter's internal buffer

The `JspWriter` is heavily buffered, as that allows the servlet container and the JSP runtime to provide services such as error pages and improved performance. Also, when an error occurs, the JSP runtime can erase the content of the buffer and forward the request to the error page. Despite its benefits, buffering also has the drawback of delaying the receipt of the user's response. Imagine that you are writing a JSP file that will access several databases, and that each database query provides enough information to build a portion of the output. Since JSP uses buffering, the user may need to wait a long time until the page preparation is completed (many database queries). In the meantime, the waiting user could become bored and switch to another site.

To keep this from happening, most developers flush the response buffer whenever a significant portion of the page is ready. Flushing the buffer causes its current contents to be sent immediately to the user. The `JspWriter` facilitates this by exposing a method named `flush()`, which allows an override of the normal buffering behavior of a JSP and assures that the user receives the buffered content immediately.

NOTE The `flush()` method does not work when tags are executed within the body of other tags. Body-modifying tags like to collect the contents of their bodies and manipulate them. The JSP runtime implements that behavior by creating an instance of `BodyContent` (a special derivative of `JspWriter`) in which to hold the processed body. Since `BodyContent` is created only for the purpose of collecting the content of the tag's body, it doesn't really represent the stream of content flowing back to the user. It is, rather, a holding tank for the content in a tag's body. It makes sense then that flushing a `BodyContent` has no meaning and, therefore, any call to `BodyContent.flush()` will result in an `IOException`. It is therefore important that, before flushing the `JspWriter`, you verify that it is not actually an instance of `BodyContent`.

FlushWriterTag example

The next tag we'll look at is called `FlushWriterTag`, whose job is to flush the `JspWriter` to the user. With this tag, a JSP author can specify places in the page where the output of the processing (up to that point) will be flushed to the user. Since we are placing all of the necessary logic for a flush within this custom tag, a page author can use its functionality without knowing anything about Java, the internals of the `JspWriter`, or the exact type of `JspWriter` currently in use. `FlushWriterTag`'s source code appears in listing 6.4.

Listing 6.4 Source code for FlushWriterTag's handler class

```
package book.simpletasks;

import book.util.LocalStrings;
import book.util.ExTagSupport;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.BodyContent;

public class FlushWriterTag extends ExTagSupport {

    //some code was omitted for clarity

    public int doStartTag()
               throws JspException
    {
        try {
            JspWriter out = pageContext.getOut();
            if(!(out instanceof BodyContent)) {          ❶
                out.flush();
            }
        } catch(java.io.IOException ioe) {              ❷
            // User probably disconnected ...
```

```
            // log an error and throw a JspTagException
            // …
        }
        return SKIP_BODY;
    }
}
```

❶ **Check if the `JspWriter` can be flushed and flush it to the user (if applicable)** We first check if the `JspWriter` in use is actually an instance of `BodyContent`. If so, flushing it would throw an exception since the method is not implemented. But in cases in which the `JspWriter` is not a `BodyContent`, it will trigger a flush call that will immediately write the buffer contents to the user.

❷ **Handle the `IOException`. Log the exception and abort the JSP execution by throwing a `JSPTagException`** As always, methods executing on the `JspWriter` can throw `IOExceptions`. We handle them here by throwing a `JspTagException` that causes the JSP runtime engine to abort the processing of the page.

6.3 Setting tag attributes

The second technique is one you will use in most of the tags you develop. In order to make tags more flexible and reusable it is often necessary to let JSP authors pass parameters to them. One way to do this is through tag attributes. As noted in chapte 3, attributes are very common in HTML tags. One example is the HTML `<font>` tag, in which we see usage like the following:

```
<font face="verdana" size="3">Manning Press</font>
```

In this case, `face` and `size` are attributes that allow the page author to specify how the tag should format the text in its body. Imagine how useless the `<font>` tag would be if it always formatted text in the same size, style, and face.

The custom tags we build likely need attributes as well. We saw the use of attributes in chapter 3 with our `CookieValueTag`, but we did not, up until now, conduct a serious discussion on how attributes are implemented, nor did we discuss the different options available with the custom tags attributes mechanism. Now is a good time to start, because almost any tag (including most of our future samples) requires a great deal of configuration, and attributes are the prime tool for that.

Before describing the JSP runtime behavior when it tackles a tag attribute, let's think of the possible requirements associated with custom attributes for custom tags:

1 We need to specify, for a given tag, all of the attributes valid for it and indicate which are mandatory and which are not. This information must be

available at translation time to the JSP engine so that it can make decisions about whether a particular tag is being used properly.

2. Some criteria have to be specified for validating the values an author sets for a particular attribute. For example, if we are writing a custom version of the HTML `<font>` tag, we want to specify logic that checks if the `size` attribute is a positive integer.
3. In some pages and tags you may want to pass dynamic value, such as the results of a JSP scriptlet, as attribute values. Functionality is required to support this.
4. We need a standard way to define methods in our tag handler class that can be called to set an attribute's value. The JSP specification could mandate a special method with the signature of `void setAttribute(String name, Object value)` in all tags, and pass the values this way. But this is a brute force technique requiring additional work by the tag developer (something that specification writers prefer to avoid).

All of these requirements are met through the following conventions described by the JSP specification:

- Special entries in the TLD indicate the valid attributes (by name) for a particular tag, as well as whether or not each attribute is mandatory. The entry can also specify whether a particular attribute is the result of evaluating Java code embedded in the JSP. Recall the case in which we are writing our own version of the HTML `<font>` tag. We might want our `size` attribute to equal the result of some arithmetic we perform on local variables within the page.
- A special helper class that lets the tag writer code attribute validity checks being performed by the JSP runtime during translation time.
- JavaBeans coding conventions for defining methods in the tag handler to be used in setting methods.

We will review each of these in detail.

6.3.1 *Specifying tag attributes in the TLD*

In addition to the tag name and implementing class, each tag entry in the TLD file can contain attribute information (if nothing is specified, the tag cannot have any attributes). For each attribute the tag supports, a name must be specified, whether or not the attribute is mandatory (defaults to no), and whether the attribute's value is the result of runtime expression (again, the default is no). For example, assume that we have a tag called `Greeting` that will greet a returning user to our site.

`Greeting` will be implemented by a class called `book.simpletasks.GreetingTag`. Its TLD entry should look something like this:

```
<tag>
   <name>Greeting</name>
   <tagclass>book.simpletasks.GreetingTag</tagclass>
<tag>
```

Assume that `Greeting` has the following set of attributes:

- user
  The person to greet. This is a mandatory attribute whose value can be the outcome of a runtime expression.
- type
  The type of greeting. This is a mandatory attribute whose value must be hard-coded into the JSP file. Some possible values for this might be `tip` (to show a helpful tip with this greeting), `promotion` (to include a link to the current promotion on the site), `standard` (to output the standard greeting), and so forth.
- tip
  A tip to include with the greeting (used only if its type attribute equals `tip`). This might be a tip about how to navigate the site or help for the current page the user is on. This is an optional attribute whose value must be hard-coded into the JSP file.

In order to support these tag attributes, our TLD entry should have three attribute entries and will look like:

```
<tag>
    <name>Greeting</name>
    <tagclass>book.simpletasks.GreetingTag</tagclass>
    <attribute>
        <name>user</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
        <name>tip</name>
    </attribute>
    <attribute>
        <name>type</name>
        <required>true</required>
    </attribute>
<tag>
```

As you can see, each definition is enclosed within an `<attribute>` tag that wraps the following three subtags: `<name>` to specify a name, `<required>` to specify

whether an attribute is mandatory (`true`) or optional (`false`, the default), and `<rtexprvalue>` specifying whether the value can result from a runtime expression (`true`) or is a hard-coded value (`false`, the default).

NOTE The runtime expression assigned to an attribute value must be a JSP language expression of the form `<%= expression %>`. If you want to provide dynamic input whose complexity exceeds a scriptlet, you will need to employ another way such as using the tag's body to provide the complex dynamic input. Future sections in this chapter will deal with body processing and custom tags.

Now that each possible attribute for the tag is specified, the JSP runtime can perform syntactic checks when translating the JSP file to a servlet. These checks determine whether any required attributes are missing or if a certain attribute is not legal for a tag (meaning it's not listed in the TLD). The JSP runtime also determines whether a certain attribute is allowed to contain the results of a runtime expression and handles it accordingly.

Introducing the attribute information into the TLD solved many of the translation time syntax problems associated with custom tag attributes. This affords us a basic level of control, but what if we desire some specific conditions with which to validate our attributes? For example, in our `Greeting` tag, we might want to be sure that when the greeting includes a tip, that the page author provides its text. Recall that specifying a value of `tip` for the `type` attribute in our tag will indicate that this greeting should include some helpful text along with our standard "Good afternoon, so and so" message. We could make our `tip` attribute mandatory, but then page authors using the `Greeting` tag would be required by the JSP runtime to include a tip even when it won't be used. What is optimal is to make the `tip` attribute required in some cases (namely, when `type` equals `tip`) and optional in others (when `type` is anything besides `tip`). This type of conditional check is commonly needed for tags and, luckily, the authors of the JSP specification made provisions for it. For such a complex check, the JSP specification allows tag developers to define a `TagExtraInfo` object which specifies the logic for our condition.

6.3.2 *Providing validity checks at translation time*

You say you want to provide extra syntax checks on your attribute data? No problem. The way we accomplish this is by coding the checks in Java and injecting that code into the JSP runtime by overriding a method in a class called `TagExtraInfo`. But first, let's take a look at `TagExtraInfo` and how the JSP runtime uses it.

TagExtraInfo

The JSP runtime associates each custom tag with a set of metadata objects derived from the information stored in the TLD. These metadata objects contain all the information specified about a tag such as its name, implementing class, valid attributes, and so forth. During the translation phase, the JSP translator consults the data stored in these objects and, based on that, determines how to invoke a tag handler and whether or not a tag is being used properly. One of these metadata objects is `TagExtraInfo` but, unlike all other metadata objects the translator uses, `TagExtraInfo` does not simply echo data that is in the TLD. Instead, it is written explicitly by the tag developer and then registered with the JSP runtime for a particular tag. This `TagExtraInfo` object provides extra attribute checks and scripting variables information to the JSP runtime.

NOTE `TagExtraInfo` is not mandatory and most tags manage without it. But if you want your tag to perform special syntax checks or export scripting variables (a feature we'll talk about in the next chapter), you need it.

Table 6.2 shows the methods in `TagExtraInfo`.

Table 6.2 `TagExtraInfo's` methods

Method name	Description
public VariableInfo[] getVariableInfo(TagData data)	Used to expose new scripting variables into the JSP. This method will be discussed in the next chapter.
public boolean isValid(TagData data)	The method we'll override to check conditions on our tag attributes. We return `true` if the attributes are valid or `false` otherwise.
public final void setTagInfo(TagInfo tagInfo)	Setter method for the TagInfo object (discussed later in the book).

The only method we'll need to use for now is `isValid()`. This is where we will place the code for the JSP runtime to use in evaluating our attributes.

Here are the steps to follow if we want our tag to have its own attribute checks:

1. Create a class that extends `javax.servlet.jsp.tagext.TagExtraInfo`. This class will serve the JSP runtime during the translation phase of the page and provide it with the extra tag-related information.
2. In the new class, override the method `boolean isValid(TagData data)`. The JSP runtime will call this method with the attribute information inside

the `data` parameter, and you will need to check these attributes and return `true` if you approve them (`false` if not).

3 Inform the JSP runtime that the custom tag has a `TagExtraInfo` associated with it. You will need to add a `<teiclass>` entry for your tag description in the TLD.

Attribute validation in GreetingTag

To clarify, let's relate this to our `Greeting` tag. Remember, the rule is that if the `Greeting` tag's `type` attribute is `tip`, then the tag user must specify a value for the `tip` attribute. This new requirement forces us to implement a `TagExtraInfo` for `Greeting` tag (let's name it `GreetingTagInfo`). We associate the `GreetingTagInfo` class with the `Greeting` tag in the TLD file:

```
<tag>
    <name>Greeting</name>
    <tagclass>book.simpletasks.GreetingTag</tagclass>
    <teiclass>book.simpletasks.GreetingTagInfo</teiclass>
    <attribute>
        <name>user</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
        <name>tip</name>
    </attribute>
    <attribute>
        <name>type</name>
        <required>true</required>
    </attribute>
<tag>
```

And the implementation of `GreetingTagInfo` will be:

```
import javax.servlet.jsp.tagext.TagData;
import javax.servlet.jsp.tagext.TagExtraInfo;

public class GreetingTagInfo extends TagExtraInfo {

    public boolean isValid(TagData data)
    {
        String greetType = data.getAttributeString("type");    ❶
        if(greetType.equals("tip")) {
            String tip = data.getAttributeString("tip");    ❷
            if(null == tip || 0 == tip.length()) {
                return false;
            }
        }

        return true;
    }
}
```

❶ **Gets the String value of type** isValid() uses TagData.getAttributeString() to collect the values of the attributes. Once collected, we determine whether the values satisfy our condition and return either true or false. Normally, the values stored in the TagData are instances of String, with runtime expression attributes as the exception. These attributes, by their very nature, have no value until runtime, whereas the TagExtraInfo is used for checks at translation time. Because of this, runtime expressions are assigned a value for a plain Java Object[2] to represent them in TagData.

❷ **Performs the syntactic check on type and tip.**

With our GreetingTagInfo class in place we are now assured that our JSP runtime engine will enforce proper usage of our tag. If a JSP author attempts to set the attribute type to the value tip and not set a value to the tip attribute, the compiler will produce an error when it tries to translate the JSP that this tag is in. The error will produce output that varies slightly from vendor to vendor, but will ultimately inform the developer via some message in his or her web browser that the attributes are invalid for this tag.

Specifying attributes and their associated syntax and content constraints should be clear by now. The last piece left in the puzzle is how we write our tag handler to accept and use these attributes.

NOTE In JSP1.2, the JSP file is translated into an XML document, then the JSP runtime translates this XML document into a servlet. A JSP1.2-compliant library can provide a validator class to work on the intermediate XML document representing the JSP file, and in this way perform a more rigorous validation spanning a whole document instead of one tag at a time. However, the majority of tags do not require the power, nor the complexity, of this validator.

6.3.3 *Using the JavaBeans coding conventions*

Setting the attributes of Java objects is not a new problem, so the designers of the JSP specification selected a tried-and-true solution—having the tag attribute setters follow the JavaBeans coding conventions. JavaBeans, as you know, is the Java

---

[2] To facilitate working with non-String attributes, TagData also has a method named getAttribute() that returns an Object value. If your attribute is the result of a runtime expression, this is the method to use.

component model, and its specification defines the way to set a property into an object based on the property's name. JavaBeans uses a simple coding convention that implies that a Bean with a writable property named `foo` should have a setter method named `setFoo()`. Using this convention, a JavaBean's environment can discern the method to call for every property value it needs to set.

Instructing the tags to expose attribute properties as JavaBeans properties solves the problem in a very pleasant way, since the names of the attributes are known in advance. Let's look at what the attribute setters of the `Greeting` tag should look like:

```
public class GreetingTag extends TagSupport  {

// Omitted code
    String user;
    String type;
    String tip;

    public void setUser(String user) {
        this.user = user;
    }

    public void setType(String type) {
        this.type = type;
    }

    public void setTip(String tip) {
        this.tip = tip;
    }

// Some more omitted code
}
```

Nice and intuitive. After we've defined our setter methods, we need only refer to those local variables in any of the tag methods where attributes are required. As you might guess by their usefulness, the majority of the custom tags you build will include attributes. This is also true for most of the custom tags we build in the remainder of this book, and you'll see plenty of examples of this in the coming chapters.

6.4 *Logging and messaging*

Another important practice in any tag development project is logging error and informational messages and handling errors. It is important for debugging and troubleshooting, especially with web applications, to be able to review log files or inspect error pages to determine where things went wrong. Virtually all the tags we write in this book will need to have this ability. Here is an approach that will prove useful in future tag development.

6.4.1 Logging

Logging messages to a file is a very common practice in software development and, as such, is already integrated into the language or runtime environment. A JSP runtime container is no exception, with built-in logging facilities at your disposal. The actual location of the log file (as well as other, more advanced features, such as whether or not they can be rolling logs) typically varies depending on the runtime container vendor. The method for logging, however, is the same for any web container and is done via `log()` of the `ServletContext` object. This method allows either the logging of a simple `String` message or a `String` message and a `Throwable` in which case the stack trace for the `Throwable` is printed to the log.

It would be best not to have to write the logging code in every tag we develop, so we'll add two simple methods to our tag base classes (`ExTagSupport` and `ExBodyTagSupport`). These methods are:

```
protected void log(String msg)
{
    getServletContext().log(msg);
}

protected void log(String msg,
                   Throwable t)
{
    getServletContext().log(msg, t);
}

protected ServletContext getServletContext()
{
    return pageContext.getServletContext();
}
```

These methods are basically delegates to the log methods in the `ServletContext`. They remove the step of having to explicitly get the `ServletContext` in each tag and provide a place where we can enhance our logging, for example, by adding logging levels or checking debug flags.

6.4.2 Handling and throwing exceptions

Now that we have logging functionality, we can log any exceptions caught within our tags, but merely logging an error typically isn't enough. Many times, an exception will mean that the action the user was trying to process in the JSP, such as saving registration information or performing a search, has failed. In these cases, we want to log the problem and handle the exception properly so that the user is aware that the intended action failed and can contact technical support or otherwise correct the situation.

Once again, we find that the functionality for handling errors this way is built into any JSP/Servlet container. For any JSP we write we can indicate easily where the users's browser should be redirected should an error occur. We do this through the `errorPage` attribute of the `page` directive. For example:

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

indicates that if an uncaught exception is thrown during the JSP's execution, the user should be redirected to `errorpage.jsp`, which can either show a default message to warn the user that there is a problem and/or inspect the exception that was thrown and display information about it.[3] By specifying an error page in our JSP's in this way, we only need to throw a `JspTagException` when an error occurs (such as the following).

```
public int doStartTag()
    throws JspException
{
    try {
      JspWriter out = pageContext.getOut();
      //some code that could create an exception
      out.println("Look Ma, no errors!");
    } catch(Exception e) {
      // Log the error and throw a JspTagException
      log("An error occurred");
      throw new JspTagException("Yikes!");
    }
    return SKIP_BODY;
}
```

If a problem is encountered, we log the error and throw a `JspTagException` which, assuming the JSP has the `errorPage` defined, will cause the user to be redirected to the proper error page.

6.4.3 Improving logging

The logging and error handling code we've written thus far is pretty straightforward. It satisfies our needs but we could improve it slightly by getting our messages from a resource file to gain flexibility for changing our messages and retrieving support internationalization in our tags. This ability is achieved with the addition of two simple helper classes: `LocalStrings`, which will read the properties file with our messages and make them available to the tags and `Constants`, which will provide tag-specific keys with which to refer to messages in `LocalStrings`.

---

[3] For more information on how to write an error page, see the Sun tutorial at http://developer.java.sun.com/developer/onlineTraining/JSPIntro/exercises/ErrorHandling/.

LocalStrings

Essentially, all our `LocalStrings` class will do is read the LocalStrings.properties file from the classpath which will hold name-value pairs of keys and messages. The format for the LocalStrings.properties file is:

Key=message or, for example:

```
IO_ERROR=Error: IOException while writing back to the user
```

Defining error messages in this way lets us create LocalStrings.properties files for every locale in which our application is deployed and lets us quickly change, add, or delete messages. As the implementation of this class is not specifically relevant to tag development as a whole, we will forgo an in-depth look here. You can, however, download the source for this class from the book's web site.

Constants

The keys to the messages in the LocalStrings.properties files will be stored in tag-specific classes that we'll call `Constants` (one `Constants` class for each package, since each package is likely to have different error or information message needs). For the previous `IO_ERROR` example, this key would be stored in a `Constants` class, such as:

```
public class Constants {

    public static final String IO_ERROR = "IO_ERROR";
    //other keys follow
}
```

Putting it together

How does our revised error handling look with the addition of our two new classes? See listing listing 6.5.

Listing 6.5 Improved error handling in ShowFormParamTag

```
public class ShowFormParamTag extends TagSupport {
    // Some other code was omitted here.
    static LocalStrings ls =
        LocalStrings.getLocalStrings(ShowFormParamTag.class);    ❶

    public int doStartTag()
        throws JspException
    {
        try {
            HttpServletRequest req =
              (HttpServletRequest)pageContext.getRequest();
            String value = req.getParameter(name);
            if(null != value) {
```

```
                writeHtml(pageContext.getOut(), value);
            }
            return SKIP_BODY;
        } catch(java.io.IOException ioe) {
            // User probably disconnected ...
            log(ls.getString(Constants.IO_ERROR), ioe);      ❷
            throw new
              JspTagException(ls.getStr(Constants.IO_ERROR));      ❸
        }
    }
}
```

❶ **Loads the key-value pairs in the LocalStrings.properties file in which this class is deployed.**

❷ **Gets the proper message string for an `IO_ERROR`.**

❸ **Gets the proper message string for an `IO_ERROR` and throws a `JspTagException` with it.**

We now have a simple and clean logging and messaging interface that lets us handle errors in our tags and send those errors to the client (with built-in internationalization). This approach is used in the examples throughout the book.

6.5 Using the Servlet API

Another technique central to custom tag development is interacting with the Servlet API. If you've had any experience with JSP or servlet development (or you have read through chapters 1 and 2), you are familiar with the classes and interfaces in the Servlet API that enable web development. These are the objects that make web programming possible by allowing access to request parameters, session variables, the HTTP response, the user session, and so forth. To be of any use in a web environment, custom tags must be able to access these same objects to do their work. Since we know that custom tags and JSPs are ultimately compiled into servlets, it is no wonder that all Java web technologies (serlvets, JSPs, or tags) eventually interact with the same classes to do their jobs. The only difference is the way in which each technology gains access to the objects.

In servlets, these objects are retrieved via method parameters and local variables. In JSPs, the objects are always available (in scope) and can be referred to by name anywhere in the file (i.e., request, response, etc.). For JSPs, these ever present objects are referred to as the implicit JSP objects. Since tags actually sit within JSPs, we refer to this group of objects as the implicit JSP objects in the context of tags as well. This simple naming convention mustn't distract you from the fact that we're talking about a few key classes that reside in the Servlet API and in which all three technologies share an interest.

What then are the implicit JSP objects and what are they used for? They are:

- The `request` object—To obtain request parameters and other information.
- The `response` object—To add headers and redirect the request.
- The `session` object—If we want the tag to manipulate the session directly (e.g., when we want to perform metaactions on the session, such as invalidation).
- The `application (ServletContext)` and `ServletConfig` objects
To obtain context and page-level initialization variables.
- All of the JSP attribute objects in the four scopes used by a JSP (application/session/request/page)—This way the tag can interact with other portions of the web application. For example, one tag may open a JDBC connection and place it as an attribute in the page scope; later on another tag can take this connection and use it to query a database.

If servlets have variables and methods to access these objects and JSPs can refer to them by name, how do custom tags obtain them? The solution is straightforward: all of these variables are made available to custom tags via the `PageContext`.

Each tag has two mandatory attributes: its parent and the `PageContext`assigned to the current JSP execution. The `PageContext` has many roles, but as far as JSP tags are concerned, the most important ones are to connect the tag to the JSP environment and to provide access to this environment's services and the implicit objects. Let's look at how tags can use the `PageContext` to get a reference to the different objects in the environment.

6.5.1 *Accessing the implicit JSP objects*

A JSP implicit object represents a key object in the Servlet API and it is always available. Table 6.3 shows the available JSP implicit objects and how a tag can attain reference to each:

Table 6.3 Implicit JSP objects and their tag counterpart

JSP implicit objects	Custom tags counterpart	Typical use by the tags
pageContext	The pageContext attribute of the tag. This attribute is set on tag initialization by the page implementation.	Obtains other implicit variables. Obtains JSP attribute. Accesses `RequestDispatcher` type services.
request	Calling pageContext.getRequest()	Queries request information; for example, query form parameters or in-bound cookies.

Table 6.3 Implicit JSP objects and their tag counterpart (continued)

JSP implicit objects	Custom tags counterpart	Typical use by the tags
response	Calling pageContext.getResponse()	Manipulates the response; for example, add cookies, redirect, etc.
session	Calling pageContext.getSession()	Manipulates the session directly; for example, invalidate the session or set a different inactivity timeout.
config	Calling pageContext.getServletConfig()	Obtains configuration parameters for this page.
application	Calling pageContext.getServletContext()	Obtains configuration parameters for this application and uses its utility method (for example, `log()`).
out	Calling pageContext.getOut()	Writes data into the page.
page	Calling pageContext.getPage()	Usually not in use. Unless coded specifically for a certain page, the tag cannot know the services exposed by the page class.
exception	Calling pageContext.getException()	Analyzes and displays in the response.

All the implicit JSP objects are accessible for the custom tags and the key to all of them is the tag's `pageContext` attribute. We'll now show how to use these variables through a few examples.

ShowFormParam tag example

In web development, we know that the only way to pass information from the browser to the server (other than a cookie) is through the use of POST variables or query string parameters. Accessing these parameters is, therefore, one of the most important tasks we need to perform in a JSP. Regular JSPs can access the form parameters through the implicit `request` object (usually by means of the method `String req.getParameter(String name)`). How does a tag do this?

The answer recalls the workings of a JSP or servlet except, in custom tags we don't have the request object at our fingertips, so we must first get a reference to it. Looking back at the `ShowFormParamTag`, we remember that it prints the value of a named form parameter into the response that is returning to the user. Since we need to print a named value, `ShowFormParamTag` has an attribute that specifies the name of the parameter to print. And, of course, we will need to have the implementation of this tag's unique logic actually fetch the form parameter and print its value to the user. The resulting tag source is in listing 6.6:

Listing 6.6 Source code for ShowFormParamTag's handler class

```
package book.simpletasks;

import book.util.LocalStrings;
import book.util.ExTagSupport;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;

public class ShowFormParamTag extends ExTagSupport {

    static LocalStrings ls =
        LocalStrings.getLocalStrings(ShowFormParamTag.class);

    protected String name = null;                        ❶

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

    public int doStartTag()
                throws JspException
    {
        try {
            HttpServletRequest req =
                (HttpServletRequest)pageContext.getRequest();   ❷
            String value = req.getParameter(name);
            if(null != value) {
                writeHtml(pageContext.getOut(), value);
            }
            return SKIP_BODY;
        } catch(java.io.IOException ioe) {
            // User probably disconnected ...
            log(ls.getStr(Constants.IO_ERROR), ioe);
            throw new
                JspTagException(ls.getStr(Constants.IO_ERROR));
        }
    }

    protected void clearProperties()
    {
        name = null;
        super.clearProperties();
    }
}
```

❶ **Implements the tag's `name` attribute** The tag starts by defining a setter for the name attribute (`setName()`), and continues by implementing the `doStartTag()`

method that simply fetches the `request` object from the `pageContext` and queries it for the named parameter. The tag ends with an odd-looking method named `clearProperties()` that we will discuss in the section dealing with tag cleanup.

❷ **Fetches the request object from the `pageContext` and obtains the needed form parameter.**

After creating this tag, the next step is to put together a TLD for it and test drive it using a JSP file. Listing 6.7 is the result.

Listing 6.7 Tag library descriptor for ShowFormParamTag

```
<?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>simp</shortname>
           <uri> http://www.manning.com/jsptagsbook/simple-taglib </uri>
           <info>
           A simple sample tag library
           </info>

    <tag>
        <name>formparam</name>
        <tagclass>book.simpletasks.ShowFormParamTag</tagclass>
        <bodycontent>empty</bodycontent>
              <info> Show a single named form parameter</info>
              <attribute>
                        <name>name</name>                  ❶
                        <required>true</required>
                        <rtexprvalue>true</rtexprvalue>
              </attribute>
           </tag>
</taglib>
```

❶ **The `name` attribute is required and can be the result of a runtime expression, providing of flexibility in listing the parameters.**

The JSP can be found in listing 6.8.

Listing 6.8 Sample page employing ShowFormParamTag

```
<%@ page errorPage="error.jsp" %>    ❶
<%@ taglib
```

```
    uri="http://www.manning.com/jsptagsbook/simple-taglib"    ❷
    prefix="simp" %>

<html>
<body>

Here are your FORM request parameters:

<table>
<tr><th>Name</th> <th>Value</th> </tr>
<% java.util.Enumeration e = request.getParameterNames();    ❸
   while(e.hasMoreElements()) {
       String paramname = (String)e.nextElement();
%>
    <tr>
        <td> <%= paramname %></td>
        <td><simp:formparam name='<%= paramname %>'/> </td>    ❹
    </tr>
<% } %>
</table>
That's all for now.
</body>
</html>
```

❶ **Uses the error page that we developed in our first hello chapter.**

❷ **Instructs the page to use the simple tags library.**

❸ **Walks through all the request parameters** ❹ **Prints a named request parameter based on its runtime value** The test JSP simply gets the list of FORM parameters and iterates on them, printing the different values for each. This is also a demonstration of how runtime expressions in attributes can come in handy. Since we'd like this JSP to work with any HTML form, each with any number of parameters, we couldn't possibly hard-code a value for the tag's `name` attribute. Because we specified in the TLD that the `name` attribute can be the result of a runtime expression, the JSP engine evaluates `<%= paramname %>` first and then passes the results of this evaluation to our tag handler (by calling `setName()` with the result).

Figure 6.1 shows the results of accessing showform.jsp with a few parameters. The output of our JSP is a table displaying the names and values of the FORM parameters.

RedirectTag example

Once you know how to manipulate values in the `request`, it is time to look at the `response`. To do so, we'll look at a tag we'll call `RedirectTag` which redirects the user's browser to another location. Since we'll want the JSP author to specify which URL to redirect to, the tag will have an attribute called `location`.

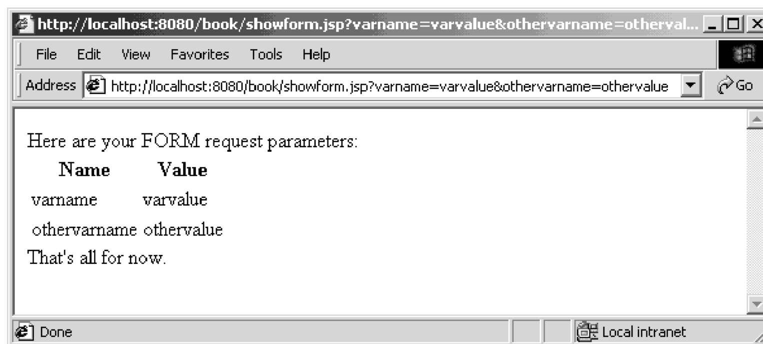


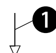
Figure 6.1 The results of accessing showform.jsp

To ensure that our redirect tag works reliably, we need to build it with one key nuance of HTTP in mind. An HTTP redirect response includes a standard HTTP redirect response code as well as unique redirection headers. Once a response to the user begins, it is too late to modify the headers and response code (and thus, too late to send a redirect). The `RedirectTag` must watch closely to make sure it is not too late to modify the HTTP header. If it is, we should inform the executing JSP by throwing an exception.

Fortunately, the JSP infrastructure is buffering the response, which makes it possible to ask for a redirection at any time, as long as the buffer hasn't already been flushed. The buffer can be flushed explicitly by calling `pageContext.getOut().flush()`, or automatically when it becomes full. Once the response is flushed to the user it is considered committed, and you will be unable to modify the headers. listing 6.9 presents the source code for the `RedirectTag`.

Listing 6.9 Source code for RedirectTag's handler class

```
package book.simpletasks;

import book.util.LocalStrings;
import book.util.ExTagSupport;

import javax.servlet.http.HttpServletResponse;
import javax.servlet.jsp.JspException;

public class RedirectTag extends ExTagSupport {

    static LocalStrings ls =
        LocalStrings.getLocalStrings(RedirectTag.class);

    protected String location = null;

    public void setLocation(String location)
    {
```



```
        this.location = location;              ❶
    }

    public int doStartTag()
               throws JspException
    {
        HttpServletResponse res =                                       ❷
            (HttpServletResponse)pageContext.getResponse();
        if(res.isCommitted()) {
            throw new JspException(ls.getStr(Constants.COMMITTED));
        }

        try {                                                        ❸
            res.sendRedirect(res.encodeRedirectURL(location));
            return SKIP_BODY;
        } catch(java.io.IOException ioe) {
            // User probably disconnected ...
            // log an error and throw a JspTagException
            // …
        }
    }

    public int doEndTag()
               throws JspException                ❹
    {
        super.doEndTag();
        return SKIP_PAGE;
    }

    protected void clearProperties()
    {
        location = null;
        super.clearProperties();
    }
}
```

❶ **Implements the tag's location attribute. This is the location to which we redirect the browser.**

❷ **Fetches the response object from `pageContext` and checks to see if it is commited (which is an error).**

❸ **Uses the response object to redirect the browser (keeps URL-based rewrite session state in place)** Since Servlet/JSP-based applications have two methods to keep session state, namely cookies and URL encoding, one must support URL encoding when redirecting the user from one page to another. To facilitate this, the `request` object exposes a method (`encodeRedirectURL()`) whose job is to rewrite the redirected URL according to the URL encoding specifications. Calling this method is exactly what we are doing prior to calling the utility redirect method. Remember also to call `encodeURL()` any time you print a URL or FORM action field into the output sent to the user.

NOTE URL encoding is a method wherein session tracking is accomplished by encoding the user's session `id` inside all the JSP file's URLs (each user therefore receives a slightly different set of URLs in his content). In most web servers, this approach is a backup to the preferred method of placing session information in a cookie. Some users choose not to use cookies, or their firewalls prevent it, so embedding the session `id` in a URL is a fallback approach. For more information about URL encoding, refer to the Servlet API specification of any servlet book.

❹ **Terminates the execution of the page by returning `SKIP_PAGE` from `doEndTag`** This is the first time any of our tags has implemented `doEndTag()`. We can usually leave `doEndTag()` out of our tag handlers since it is implemented by our `ExTagSupport` base class; however, in this tag we must alter the value returned from `doEndTag()` to tell the JSP runtime engine to stop page execution after the redirection. The default implementation of `doEndTag` returns `EVAL_PAGE`, a constant value that instructs the JSP runtime to continue executing the remainder of the JSP page. This default behavior is not appropriate for our redirect tag, because a redirection means that we do not want to continue with this JSP file execution. We would like to instruct the JSP runtime to stop the execution and return immediately. As you recall from chapter 3, this can be accomplished by overriding the default `doEndTag()` method and returning `SKIP_PAGE`.

Testing RedirectTag

It is useful to test the `RedirectTag` in cases in which the output is already committed (to see how the tag works in case of an error), and fortunately we can accomplish that by using a tag-only approach. Earlier we developed `FlushWriterTag` whose job was to flush the output to the user, so combining these two tags serves as a good test case for both of them.

Listing 6.10 presents the TLD we are using which includes two tags. We define the two tags, redirect and flush, their respective attributes (flush does not have any), and we're through.

Listing 6.10 Tag library descriptor for the redirect tag

```
<?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>simp</shortname>
    <uri> http://www.manning.com/jsptagsbook/simple-taglib </uri>
    <info>
      A simple sample tag library
    </info>

    <tag>    ❶
        <name>redirect</name>
        <tagclass>book.simpletasks.RedirectTag</tagclass>
        <bodycontent>empty</bodycontent>
        <info>
          Redirect the browser to another site. Stop the response.
        </info>
        <attribute>
            <name>location</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
    </tag>

   <tag>    ❷
    <name>flush</name>
    <tagclass>book.simpletasks.FlushWriterTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info> Flush the JSP output stream. </info>
    </tag>
</taglib>
```

❶ **Defining the `redirect` tag.**

❷ **Defining the `flush` tag.**

After naming each of the tags, we can also write the JSP driver (flushredirect.jsp) as presented in listing 6.11:

Listing 6.11 Sample page employing RedirectTag and causing an exception

```
<%@ page errorPage="error.jsp" %>    ❶
<%@ taglib
    uri="http://www.manning.com/jsptagsbook/simple-taglib"
    prefix="simp" %>
<html>
<body>

Here is some text before the redirection. <br>
<simp:flush/>                              ❷
<simp:redirect location="/"/>
Here is some text after the redirection. <br>
</body>
</html>
```

❶ **Assigns an error page**

❷ **Flushes the output before trying to rediret. This should cause an exception** Note that we force the committing of the response prior to the redirect via our `<simp:flush/>` tag. The redirect tag is forced to throw an exception because the response will already have been committed by the time it is executed. With the response committed, the user's browser has received our server's response and it is too late to perform a redirect.

Note that we set a standard JSP error page in the very first line of listing 6.11. In most cases, doing so will cause the user's browser to be directed to the specified error page whenever an exception is thrown within the JSP. In this case, will the exception that is thrown by the redirect tag cause this type of error handling? The answer is, of course, no. Since we can't redirect with our tag because the response is already committed, it would only make sense that we cannot redirect to an error page either. What we will see instead is a notification from the container that something went awry (figure 6.2).

6.5.2 Accessing the JSP attributes

JSPs are executed in the context of a Java web application, as defined in the Servlet API specification. This specification defines a set of scopes wherein the entities taking part in the application (i.e., servlets, JSPs, custom tags, etc.) can store Java objects. We store objects in this way in order to share them between entities and from request to request. These Java objects are called attributes, but should not be confused with the tag attributes we discussed earlier in this chapter. The scopes defined for attributes are request, session, application, and page. Let's look at each scope in detail.

Figure 6.2 The results of accessing flushredirect.jsp

Request scope

Web application entities can store objects within the `ServletRequest` object. This scope makes an attribute available to all the entities taking part in the service of a particular request. For example, a servlet can retrieve data from the database, embed that data in a JavaBean, set the Bean as an attribute in the `request` object and, finally, forward the user to a JSP. A custom tag in the JSP can then retrieve the bean from the request and format its data for presentation to the user as HTML.[4] In this case, the servlet and the custom tag are functioning within the same HTTP request and, therefore, the `request` scope is the proper choice for their data sharing.

Session scope

When web application entities are associating attributes with a particular user, they can store objects within the `HttpSession` object. This scope allows all the entities taking part in the service of a user session (typically more than one page request) to exchange information. For example, in an ecommerce application, a certain request may put the shopping cart as a session attribute, and the next request may perform the checkout operation based on the previously stored cart session attribute. This scope differs from the `request` scope in that it renders stored attributes available for the life of a user's visit (their session) instead of a single page request.

Application scope

Web application entities can store objects within the `ServletContext`. Associating objects in this way makes them available to any entity, no matter what session or request it is serving. Setting attributes in the application scope means that all entities taking part in the application can exchange information. For example, a certain servlet can initialize a database connection pool and store it inside the `ServletContext`; later on, other parts of the application can fetch the initialized connection pool attribute and use it to query values from the database.

Page scope

The scripting elements within a certain JSP file may need to share information between themselves. For example, we may want a custom tag to produce information and a JSP scriptlet in the same page to display it. How can these elements share this information? Of the scopes we've covered, the most appropriate one for such a need is the

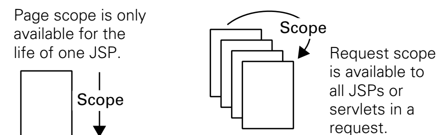


Figure 6.3 Page scope vs. request scope

[4] This facilitates the popular Model-2 approach described in chapter 2.

request. Using the request scope means, however, that the shared information will be available through all the different stages of the request. Though a single JSP will often handle the entire life of a client request, there are times when the request will be forwarded to or included by another JSP or servlet. In such a case, storing attributes in the request scope may pollute our request scope and attributes of the same name from different pages may clash. To avoid this rare but possible case, the JSP specification adds a new page scope that is implemented by the PageContext instance. The PageContext instance for a given page holds a table with attribute names and their values and, whenever a page's scoped attribute is required, the PageContext is used to fetch/store it.

Accessing attributes through PageContext

We've now discussed four different scopes, each with its own job, and our custom tags need to access objects in all of them. How will the tags do that? One simple way is to fetch the needed JSP implicit object (the request, session, application, or pageContext) and ask that object for the attribute. The problem with this approach is that it forces tight coupling between the tags and the different implicit objects and their methods which (from a design and reusability perspective) is not a good idea. Since the access methods for getting and setting attributes on each object are so similar, a better way to handle attribute interaction might be to have uniform access to all the different scopes. This design goal was considered in the implementation of the JSP specification and, as was realized in several methods, exposed by PageContext. The role of these methods is to provide a common interface to all the variable scopes. These methods are shown in table 6.4.

Table 6.4 Attribute control methods in PageContext

Method	Description
public Object getAttribute(String name, int scope)	Fetches a named attribute from a specific scope. Possible scopes (in all the methods listed in this section) are: PageContext.PAGE_SCOPE, PageContext.REQUEST_SCOPE, PageContext.SESSION-_SCOPE, and PageContext.APPLICATION_SCOPE.
public Object getAttribute(String name)	Sets/adds a named attribute in a specific scope.
public void setAttribute(String name, Object attribute, int scope)	Sets/adds attribute in the page scope.
public void removeAttribute(String name, int scope)	Removes a named attribute from a specific scope.
public void removeAttribute(String name)	Removes a named attribute from the page scope.

Table 6.4 Attribute control methods in `PageContext` (continued)

Method	Description
public Object findAttribute(String name)	Fetches a named attribute by searching for it in all scopes; starting with the page scope, continuing with request and session, and ending with application.

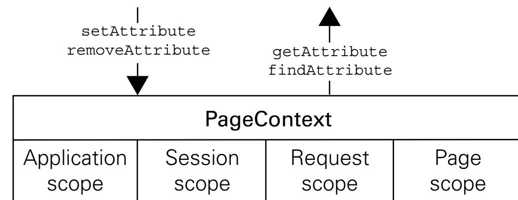


Figure 6.4 `PageContext` provides access to all four scopes

`PageContext` also provides methods to enumerate the names of the attributes in a specific scope and to find the scope of a specific attribute; but these methods are of less importance to us. Also note that all the methods in table 6.4 are actually abstract in the formal `PageContext` class definition. When we manipulate a `PageContext` instance within our tags (or JSPs), we are referring to a subclass that is implemented by the JSP runtime vendor.

ShowObjectTag example

Since all the needed functionality is easily available through the `PageContext`, there is no longer a reason to use the implicit objects for attribute interaction. Let us now look at an example tag to illustrate the concepts introduced here. We'll build a simple tag to access JSP attributes based on their name and scope which we'll call `ShowObjectTag`.

`ShowObjectTag` prints the value of a named (and optionally scoped) JSP attribute into the response returned to the user. In many ways, it is similar to `ShowFormParamTag`, except that it prints real JSP attribute objects and not simple request parameters. `ShowObjectTag` has two tag attributes that provide it with (1) the name of the JSP attribute to show and (2) an optional scope for this attribute. From these two attributes, the tag will fetch the matching object and present it. The source code for `ShowObjectTag` is displayed in listing 6.12.

Listing 6.12 Source code for ShowObjectTag handler class

```
package book.simpletasks;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;

import book.util.ExTagSupport;
import book.util.LocalStrings;
```

```
public class ShowObjectTag extends ExTagSupport {

    public static final String PAGE_ID = "page";                    ❶
    public static final String REQUEST_ID = "request";
    public static final String SESSION_ID = "session";
    public static final String APPLICATION_ID = "application";

    static LocalStrings ls =
        LocalStrings.getLocalStrings(ShowObjectTag.class);

    protected String name = null;
    protected String scope = null;

    public void setName(String newName) {     ❷
        name = newName;
    }

    public void setScope(String newScope)     ❷
    {
        scope = newScope;
    }

    public int doStartTag()
            throws JspException
    {
        Object o = getPointedObject(name, scope);                ❸
        try {
            writeHtml(pageContext.getOut(), o.toString());
            return SKIP_BODY;
        } catch(java.io.IOException ioe) {
            // User probably disconnected ...
            // signal that by throwing a JspException
            //
        }
    }

        protected Object getPointedObject(String name,
                                          String scope)
            throws JspException
    {
        Object rc = null;     ❹
        if(null != scope) {
            rc = pageContext.getAttribute(name,
                                          translateScope(scope));
        } else {
            rc = pageContext.findAttribute(name);     ❺
        }
        if(null == rc) {
            // No such object, this is probably an error
            // signal that by throwing a JspTagException
            ...
        }

        return rc;
    }
```

```
    protected int translateScope(String scope)
              throws JspException
    {
        if(scope.equalsIgnoreCase(PAGE_ID)) {
            return PageContext.PAGE_SCOPE;
        } else if(scope.equalsIgnoreCase(REQUEST_ID)) {
            return PageContext.REQUEST_SCOPE;
        } else if(scope.equalsIgnoreCase(SESSION_ID)) {
            return PageContext.SESSION_SCOPE;
        } else if(scope.equalsIgnoreCase(APPLICATION_ID)) {
            return PageContext.APPLICATION_SCOPE;
        }

        // No such scope, this is probably an error maybe the
        // TagExtraInfo associated with this tag was not configured
        // signal that by throwing a JspTagException
        //
    }

    protected void clearProperties()
    {
        name = null;
        scope = null;
        super.clearProperties();
    }
}
```

❶ **The scope names, page.**

❷ **The tag properties: name and scope.**

❸ **Getting the JSP attribute object pointed by the name and scope and printing it to the result.**

❹ **When both name and atributes are provided, we are using `getAttribute()` to locate the pointed attribute** ❺ **When only the name is provided, `findAttribute()` is the best way to locate an attribute in a consistent way** `getPointedObject()` is where the tag looks for the JSP attribute (and returns it). The method has two parameters: the `name` of the attribute (mandatory) and the `scope` (recommended, but optional). When the `scope` is given, we translate its name to its `PageContext` identifier (as in `translateScope()`) and call the `PageContext` method `getAttribute()`. Doing so will cause the `PageContext` to seek the named attribute in a specified scope. Assuming the parameter can be found in one of the four scopes, `findAttribute` will return it.

❻ **Translates the scope name to the integer `id` that the `pageContext` understands.**

To ensure proper behavior from our tag, we insist that the user provide a valid scope in our tag's attribute. This is a case in which we apply the tactics we just discussed for validating tag attributes. To do so, we associate a `TagExtraInfo` deriva-

tive (`ShowObjectTagExtraInfo`) that will add a semantic check on the value the JSP author passes to the `scope` attribute. This check will verify that the value is one of the four legal scope names, or null (if not specified at all). `ShowObjectTagExtraInfo` is displayed in listing 6.13.

Listing 6.13 Source code for the ShowObjectTagExtraInfo class

```
package book.simpletasks;

import javax.servlet.jsp.tagext.TagData;
import javax.servlet.jsp.tagext.TagExtraInfo;

public class ShowObjectTagExtraInfo extends TagExtraInfo {

    public boolean isValid(TagData data)
    {
        String scope = data.getAttributeString("scope");
        if(null == scope) {
            return true;
        }
        if(scope.equals(ShowObjectTag.PAGE_ID) ||
           scope.equals(ShowObjectTag.REQUEST_ID) ||
           scope.equals(ShowObjectTag.SESSION_ID) ||
           scope.equals(ShowObjectTag.APPLICATION_ID)) {
            return true;
        }

        return false;
    }
}
```

Note that `isValid()` does not assume the existence of the `scope` attribute; in fact it is all right for the `scope` to be missing. A problem could arise, however, if the scope name has any value other than those defined, and in such a case the method will notify the JSP runtime by returning `false`.

Now that we have the tag's implementation available, we create a TLD entry for it (listing 6.14) and a driver JSP file (listing 6.15).

Listing 6.14 Tag library descriptor for ShowObjectTag

```
<tag>
    <name>show</name>
    <tagclass>book.simpletasks.ShowObjectTag</tagclass>
    <teiclass>book.simpletasks.ShowObjectTagExtraInfo</teiclass>
    <bodycontent>empty</bodycontent>
    <info>
                Show a certain object by its name.
    </info>
```

```
    <attribute>
                  <name>name</name>
                  <required>true</required>
                  <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
                  <name>scope</name>
                  <required>false</required>
    </attribute>
</tag>
```

Listing 6.15 JSP file employing ShowObjectTag

```
<%@ page errorPage="error.jsp" %>
<%@ taglib
    uri="http://www.manning.com/jsptagsbook/simple-taglib"
    prefix="simp" %>
<html>
<body>

Here are your application attributes:
<table>     ❶
<tr><th>Name</th> <th>Value</th> </tr>
<% java.util.Enumeration e = application.getAttributeNames();     ❷
   while(e.hasMoreElements()) {
       String attname = (String)e.nextElement();
%>
    <tr>
        <td> <%= attname %></td>
        <td><simp:show name='<%= attname  %>'     ❸
                       scope="application" /> </td>
    </tr>
<% } %>
</table>
And here they are again (scope not given this time):
<table>
<tr><th>Name</th> <th>Value</th> </tr>
<% e = pageContext.getAttributeNamesInScope(     ❹
                                PageContext.APPLICATION_SCOPE);
    while(e.hasMoreElements()) {
        String attname = (String)e.nextElement();
%>
    <tr>
        <td> <%= attname %></td>
        <td><simp:show name='<%= attname  %>' /> </td>     ❺
    </tr>
<% } %>
</table>
That's all for now.
</body>
</html>
```

❶ **Lists the attributes given the name and the scope.**

❷ **Enumerates the names of the application attributes** To enumerate, we are using the application object directly. For now we need to use a script to enumerate the attribute names.

❸ **Shows the named attribute using its name and scope (application).**

❹ **Enumerates the names of the application attributes using the `PageContext's getAttributeNamesInScope()`** The results are the same as using the application object directly.

❺ **Shows the named attribute using its name only.**

The JSP driver enumerates the application-scoped attributes in two ways. These techniques are interesting on their own since they demonstrate the manner in which to use the `PageContext` attribute's manipulation methods:

- In the first enumeration, the JSP driver uses the `application` object to enumerate its attributes. Accessing the `application` object makes it possible to call `application.getAttributeNames()`, which retrieves an enumeration of the application-scoped attribute names. Later, the driver will print these attributes to the result returned to the user, using the name and the scope.
- The second shows how to use the `PageContext.getAttributeNamesInScope()` method, instead of directly using the `application` object. In doing so, we gain the use of uniform code when we want to access the different scopes and the end results are the same. This time the driver shows the application attributes only by name (the scope is not provided), yet the results are the same since the attribute names are unique.

The end results of running our JSP driver on Tomcat 3.1 are shown in figure 6.5.

In figure 6.5, the generated page presents two identical tables filled with Tomcat's built-in application attributes (which point to the server's temporary

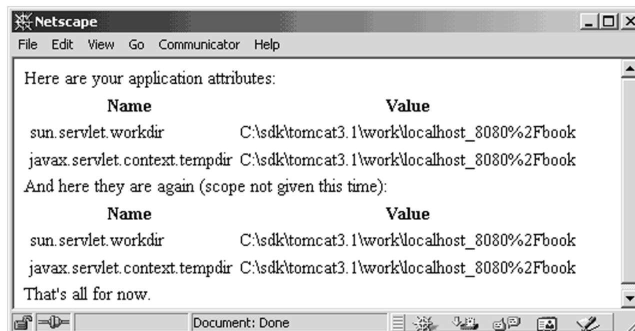


Figure 6.5 Accessing the application attributes

application directory). The attribute `javax.servlet.context.tempdir` is actually part of the Servlet API 2.2 specification, and the `sun.servlet.workdir` is a proprietary implementation attribute used within Tomcat.

In chapter 8 we will show how to use the `TagExtraInfo` class to add scripting variables to the page. Doing so allows you to define and set a variable within a tag and make that variable available to any scriptlets that follow the tag. Adding new variables in this way requires adding new attributes to the environment. We'll discover in chapter 8 that this is done via `setAttribute()` that was neglected in this section.

6.6 Configuring tags and bundling resources

Earlier in this chapter, we learned the technique of using tag attributes when passing parameters to our tags. This is a great tool, but tag attributes alone aren't always enough to let us really control tag configuration. Sometimes it's preferable to hide the more complex configuration from the page programmers, instead of burdening them with it. We want to be able to define and modify some applicationwide parameters in a central place and have all of our tags use those parameters. For example, tags sometimes need access to a database. In a data-driven application, it is likely that more than one tag in our library will have to use a single database, in which case it would be beneficial to configure the database properties in a single place (within the application) and have all the tags access this centralized configuration data. It would be an extra burden (not to mention, prone to error) to require JSP page authors to include database configuration in the attributes of every tag on every page.

There are two clear approaches to configuring tags in this way:

- Use the web application built-in configuration. In this way, the tags read configuration items from the `ServletContext` initialization parameters to configure themselves. This approach is very appealing when you need to provide a relatively limited and focused configuration. An example could be a certain application attribute, such as the name of the database connection pool.
- Use homegrown configuration. There are several variants to this approach: (1) Placing the configuration file in the application classpath and using `Class.getResourceAsStream()` to get a reference to the file's contents. (2) Placing the configuration file in the application directories and using `ServletContext.getResourceAsStream()` to get a reference to the file contents. (3) Placing the configuration file somewhere in the file system and informing the tag (using the web application built-in configuration) where this file is. The homegrown configuration is very useful when you need to provide a big

chunk of relatively constant information, such as the default behavior of the tags, product license keys, and so forth.

We'll discuss using the web application built-in configuration here. So-called home-grown configurations can offer more control, but vary greatly and are beyond the scope of this book.

6.6.1 Configuring a web application

Since version 2.2 the Servlet API defines two configuration scopes, `ServletContext` and `ServletConfig`, as well as an API to access them, in the application scope (accessible via a `ServletContext` object) you can provide configuration items that all the servlets or JSPs (including tags) can access. In the servlet/JSP scope, accessible via a `ServletConfig` object, only an individual servlet or JSP file can access the configuration items. The servlet scope holds the most interest for servlet developers. For tags, however, the application scope is much more useful, because it allows a tag to be configured once for the entire application, no matter how many times or on how many pages it is used.

The APIs used by tags to access the configuration parameters (as well as the exact configuration syntax to be used in the web application deployment descriptor) are defined in the Servlet API 2.2 specification. A tag may use the `ServletContext` object to access the broader, application-scoped configuration, and the `ServletConfig` object for individual JSP file-scoped configuration. In both objects the methods to be used are:

- `getInitParameterNames()` — Gets an enumeration with the names of the configuration parameters.
- `getInitParameter()` — Gets the string value of a certain named configuration parameter.

Note that all the parameters are string values. If you want a different type (such as Boolean) in your parameter, you simply need to convert the string value to the desired type.

Accessing the configuration parameters is not available through the `PageContext`, which makes accessing the various configuration parameters needlessly painful (you need to access the appropriate object and call the needed method). Since accessing configuration parameters is a relatively common practice, we've added a set of initialization parameters handling methods to `ExTagSupport` and `ExBodyTagSupport` (our previously defined tag handler base classes) as shown in listing 6.16:

Listing 6.16 Initialization parameter handling in ExTagSupport and ExBodyTagSupport

```
// Some of the class implementation is available above...
protected String getInitParameter(String name)     ❶
{
    return getInitParameter(name,
                            PageContext.APPLICATION_SCOPE);
}

protected Enumeration getInitParameterNames()     ❷
{
    return getInitParameterNamesForScope(
                            PageContext.APPLICATION_SCOPE);
}

protected String getInitParameter(String name,     ❸
                                  int scope)
{
    switch(scope) {
        case PageContext.PAGE_SCOPE:
            return getServletConfig().getInitParameter(name);

        case PageContext.APPLICATION_SCOPE:
            return getServletContext().getInitParameter(name);

        default:
            throw new IllegalArgumentException("Illegal scope");
    }
}

protected Enumeration getInitParameterNamesForScope(int scope)     ❹
{
    switch(scope) {
        case PageContext.PAGE_SCOPE:
            return getServletConfig().getInitParameterNames();

        case PageContext.APPLICATION_SCOPE:
            return getServletContext().getInitParameterNames();

        default:
            throw new IllegalArgumentException("Illegal scope");
    }
}

protected ServletContext getServletContext()
{
    return pageContext.getServletContext();
}

protected ServletConfig getServletConfig()
{
    return pageContext.getServletConfig();
}
// Some of the class implementation continues below...
```

❶ **Shortcut method which fetches a named configuration parameter from the default scope (application).**

❷ **Shortcut method which enumerates the configuration parameter in the default scope (application).**

❸ **Fetches a configuration parameter based on the parameter name and scope.**

❹ **Enumerates the configuration parameter in a specified scope.**

We've kept to the spirit of `PageContext` by providing scope-based methods to fetch the initialization parameters, as well as shortcut methods for the most common scope (application). This way we can use initialization parameters in our tags while we think in terms of scopes instead of having to remember which object (`ServletConfig` or `ServletContext`) exposes a certain scope.

ShowConfigTag

To test drive the configuration methods and illustrate how to integrate configuration parameters into your tags, we've written `ShowConfigTag` (listing 6.17). This tag accesses named configuration parameters and prints their values to the result. `ShowConfigTag` has a single tag attribute that allows us to set the name of the configuration property whose value it should display. The tag will look for the value in the page scope first; but, if it is not there, `ShowConfigTag` will look for the value in the application scope.

Listing 6.17 Source code for the ShowConfigTag's handler class

```
package book.simpletasks;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;

import book.util.ExTagSupport;
import book.util.LocalStrings;

public class ShowConfigTag extends ExTagSupport {

    static LocalStrings ls =
        LocalStrings.getLocalStrings(ShowConfigTag.class);

    protected String name = null;

    public void setName(String newName) {
        name = newName;
    }

    public int doStartTag()
                throws JspException
    {
        String conf = getInitParameter(name,
                                       PageContext.PAGE_SCOPE);
```

```
        if(null == conf) {
            conf = getInitParameter(name,
                                    PageContext.APPLICATION_SCOPE);
        }

        try {
            writeHtml(pageContext.getOut(), conf);
            return SKIP_BODY;
        } catch(java.io.IOException ioe) {
            // User probably disconnected...
        }
    }

    protected void clearProperties()
    {
        name = null;
        super.clearProperties();
    }
}
```

The next thing to look into is the JSP driver for `ShowConfigTag` (listing 6.18). You should be familiar with the driver's general structure, as it is a modification to the driver used by `ShowObjectTag`. In this case, however, instead of enumerating the JSP attributes in a certain scope, the driver is enumerating the configuration parameters (first in the application scope, then in the page scope).

Listing 6.18 JSP file employing ShowConfigTag

```
<%@ page errorPage="error.jsp" %>
<%@ taglib
    uri="http://www.manning.com/jsptagsbook/simple-taglib"
    prefix="simp" %>
<html>
<body>

Here are your application initialization parameters:
<table>
<tr><th>Name</th> <th>Value</th> </tr>
<% java.util.Enumeration e = application.getInitParameterNames();
   while(e.hasMoreElements()) {
       String name = (String)e.nextElement();      (1)
%>
    <tr>
        <td> <%= name %></td>
        <td> <simp:conf name='<%= name %>' /> </td> </td>
    </tr>
<% } %>
</table>
And here they are again (scope not given this time):
```

```
<table>
<tr><th>Name</th> <th>Value</th> </tr>
<% e = config.getInitParameterNames();
   while(e.hasMoreElements()) {
       String name = (String)e.nextElement();    ❷
%>
    <tr>
        <td> <%= name %></td>
        <td> <simp:conf name='<%= name %>' /> </td>
    </tr>
<% } %>
</table>
That's all for now.
</body>
</html>
```

❶ **Enumerates the names and shows the values of the configuration parameter in the application scope.**

❷ **Enumerates the names and shows the values of the the configuration parameter in the page scope.**

The web application descriptor

A more interesting aspect of the JSP driver is the web application deployment descriptor that was generated to provide initialization parameters. Until now we have not provided configuration parameters in any of the samples. This example's web.xml, available in listing 6.19, configures two application-scoped parameters and two page-level parameters.

Listing 6.19 Web application descriptor for the ShowConfigTag JSP driver

```
<?xml version="1.0"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <context-param>    ❶
        <param-name>contextparam_name_1</param-name>
        <param-value>contextparam_value_1</param-value>
    </context-param>
    <context-param>
        <param-name>contextparam_name_2</param-name>
        <param-value>contextparam_value_2</param-value>
    </context-param>

    <servlet>    ❷
        <servlet-name>showconfig</servlet-name>
        <jsp-file>/showconfig.jsp</jsp-file>
```

```
        <init-param>    ❸
            <param-name>pageparam_name_1</param-name>
            <param-value>pageparam_value_1</param-value>
        </init-param>
        <init-param>
            <param-name>pageparam_name_2</param-name>
            <param-value>pageparam_value_2</param-value>
        </init-param>
    </servlet>

    <taglib>
        <taglib-uri>
             http://www.manning.com/jsptagsbook/simple-taglib
        </taglib-uri>
        <taglib-location>
             /WEB-INF/simpletags.tld
        </taglib-location>
    </taglib>
</web-app>
```

❶ **Defines a context (application-scoped) configuration parameter** Each application-scoped configuration parameter is defined in the web deployment descriptor with a `<context-param>` tag. The `<context-param>` wraps two other tags: `<param-name>`, which encloses the name of the configuration parameter name; and `<param-value>`, which encloses the value of the named configuration parameter.

❷ **To provide a configuration parameter for a JSP file, the file needs to be associated with a servlet** ❸ **Defines a servlet (page-scoped) configuration parameter** To associate a JSP file to a servlet name, define a servlet and, in lieu of an implementing class, specify the JSP file as the entity that implements the servlet (later you will be able to bind the JSP-implemented servlet to some arbitrary URLs, instead of the URL that represents the JSP file). When you specify a servlet for the JSP file, add initialization parameters to this servlet by adding an `<init-param>` tag to the servlet definitions. This `<init-param>` will again enclose `<param-name>` and `<param-value>` tags as defined for the application-scoped parameter.

When we are ready[5], we can deploy the JSP driver and the tag. After it has executed, we will have two tables, one with the application-scoped parameters and the other with the page-scoped parameters.

---

[5] We omitted the tag entry in the TLD because we aren't introducing anything new to it here.

NOTE The example in this section did not use the configuration parameters for configuration, but rather showed you how to access them. Later in this book we will use these techniques for actual configuration purposes.

6.7 Working with the tag's body

Until now, our tags paid little attention to their body. In fact, most of the tags we've seen so far simply returned `SKIP_BODY` from `doStartTag()`, thereby instructing the JSP environment to disregard their body content altogether. This practice is not, however, always the case. Tags often find body content manipulation to be a very useful tool. Some examples are:

- A tag that displays some data values (like those implemented in this chapter) may need to have its body contain alternative content to be presented in the absence of its intended item.
- A tag that performs the equivalent of an "if-condition" statement needs to have a body that it can execute conditionally.
- A tag that performs looping needs to repeat its body execution until a certain condition is meet.
- A filter/translator type of tag needs to get a reference to its body and replace certain tag occurrences with some specified values, or translate the body into some other format. An example of this is the `LowerCaseTag` we created in chapter 3.
- A tag that executes a query could have the SQL for its query specified in its body.

These are just a few of the possible instances in which body manipulation in a tag is desirable.

Generally, we can make a clear distinction between:

- Tags that need to enable/disable their entire body evaluation conditionally. Tags that belong in this group either don't include their body, or include it unchanged, after the JSP engine has processed it.
- Tags that need to obtain their body content, either to send a modified version of the body to the user or to use it as an input to another application component.

These two cases differ greatly in the APIs that enable them, and also in the way that the JSP runtime executes the tag. The next few sections are going to tackle these issues, starting with the simple conditional body execution.

6.7.1 Tag body evaluation

As explained in chapter 4, enabling and disabling a tag's body evaluation is performed using the `doStartTag()` return code protocol. As a rule, whenever `doStartTag()` returns a value of `SKIP_BODY`, the JSP runtime will ignore the tag's body (if there is one) and neither evaluate it nor include it in the response to the user. Alternatively, a tag can enable its body evaluation by returning a value of `EVAL_BODY_INCLUDE` (for simple tags) or `EVAL_BODY_TAG` (for `BodyTags`, that is, tags that implement the `BodyTag` interface).

To illustrate this, we'll modify the `ShowFormParamTag` such that its body can contain text to be shown if the parameter cannot be found (similar to the "alt" attribute for images in the HTML `<img>` tag). Our goal is to add functionality to the `ShowFormParamTag` that enables us to specify alternative content like this:

```
<td>
<simp:formparam name="username">  Username was not found
</simp:formparam>
</td>
```

In this JSP fragment, we would expect the tag to send the client the `"Username was not found"` message when the form parameter `username` isn't found.

NOTE You may be asking why you would use the tag's body to specify an alternative content and not some other attribute (e.g., `<simp:formparam name="paramname" alt="\"paramname\" was not provided"/>`). Using an attribute to specify alternative values is possible, but not as flexible as using the body. The body lets the alternative content be as complex and dynamic as necessary; tag attributes are much more limited. It can also be looked at as a style issue as well, where you can easily wrap your alternative content between start and end tags rather than burying it in an attribute and worrying about quote delimiting and other tedious formatting issues.

To enable this feature in `ShowFormParamTag`'s handler class requires a minimal change (confined to a single method `doStartTag()`), as illustrated in listing 6.20.

Listing 6.20 Modifying ShowFormParamTag's handler class to make it body aware

```
public class ShowFormParamBodyTag extends ExTagSupport {

    // Some code was removed

    public int doStartTag()
                throws JspException
    {
```

```
        try {
            HttpServletRequest req =
                (HttpServletRequest)pageContext.getRequest();
            String value = req.getParameter(name);
            if(null != value) {
                writeHtml(pageContext.getOut(), value);
                return SKIP_BODY;      ❶
            }
        } catch(java.io.IOException ioe) {
            // User probably disconnected ...
            // log an error and throw a JspTagException
            // …
        }
        return EVAL_BODY_INCLUDE;      ❷
    }
    // Some more code was removed
}
```

❶ **We managed to print, do not show the body.**

❷ **The variable is not available, show the alternative text contained in the body.**

BodyTags and the TLD

When instructing the JSP runtime engine in handling a tag's body, changing the tag handler is only one of the procedures required. Each tag must also provide information regarding its body in the TLD.

Each tag element in the TLD should reflect how its body looks by providing an optional `<bodycontent>` entry with one of the following three possible values:

- `JSP`—Specifies that the body of the tag contains JSP. In this case, if the body is not empty, the JSP runtime will process it the same as any other content in a JSP. Choosing this option means that we can include any Java scriptlet or variable references we wish within the tag's body and it will be processed first. The outcome of this processing is passed to the tag as its body or included in the response to the user. If the `<bodycontent>` entry is missing, the runtime assumes that its value is JSP.
- `tagdependent`—Specifies that the body of the tag contains tag-dependent data that is not JSP and not to be processed by the JSP runtime.
- `empty`—The tag body must be empty.

Now it would be advantageous to create a TLD tag entry for `ShowFormParamBodyTag` and specify its `<bodycontent>` type. To allow the body to contain Java scriptlets (if the tag user chooses), we will assign the value `JSP` to the tag's `<bodycontent>` entry. A value of `JSP` is probably the most widely used `bodycontent` type since it provides the greatest flexibility. Using it, the body can either be empty,

contain static content, or contain legal Java scriptlets. The new TLD tag element is provided in listing 6.21.

Listing 6.21 Tag library descriptor for the body aware ShowFormParamTag

```
<tag>
    <name>bformparam</name>
    <tagclass>book.simpletasks.ShowFormParamBodyTag</tagclass>
    <bodycontent>JSP</bodycontent>     ❶
    <info>
        Show a single named form parameter or an alternate content
        taken from the tag's body
    </info>
    <attribute>
        <name>name</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>
```

❶ **Defines the body of the tag to be JSP.**

With our modified tag handler and TLD, we can now develop JSP code such as the one in listing 6.22. In it you see how we take advantage of the fact that alternative values can be used when the required form variable is not available.

Listing 6.22 JSP file employing ShowFormParamBodyTag

```
<%@ page errorPage="error.jsp" %>
<%@ taglib
    uri="http://www.manning.com/jsptagsbook/simple-taglib"
    prefix="simp" %>
<html>
<body>
Checking for a form variable named <tt>"varname"</tt>:<br>

Value of variable "varname" is
<simp:bformparam name="varname"> unspecified </simp:bformparam><br>    ❶

That's all for now.
</body>
</html>
```

❶ **Specifing an alternative content in case the variable “varname” was not sent by the form.**

This section provided a small step toward implementing conditional body evaluation. We will discuss this issue again, in somewhat greater detail, in chapter 9.

6.7.2 Referencing your tag's body

The previous section showed an example of conditional body evaluation; that is, letting a tag choose whether or not to include its body based on some logic in the tag. As you noticed, our tag either ignored its body or included it in the response verbatim. There are times when we want to take it one step further and have our tags inspect and modify their body. In chapter 4 we noted that tags with this ability need to implement an interface called `BodyTag` (which augments the simpler `Tag` interface). We saw an example of a `BodyTag` usage in the very simple `LowerCaseTag` example from chapter 3. Let's take a closer look at this technique as well as some more meaningful examples.

The primary difference between a `BodyTag` and a simple `Tag` is that it has access to the content between its opening and closings markups (the tag's body). We covered the `BodyTag` API and life cycle in chapter 4, but let's recap the important details of the API here.

- `BodyTag` introduces an additional method called `doAfterBody()` which is called on a tag handler after the JSP engine reads the tag's body and processes it. This is the method in which the tag handler can inspect and/or change its processed body.
- A tag handler accesses its processed body through its `BodyContent` object, which can be retrieved simply by calling `getBodyContent()` as long as the tag handler extends `BodyTagSupport`.
- Calling `getString()` on the `BodyContent` object for a tag returns a `String` containing the processed body of the tag.
- Calling `getEnclosingWriter()` on the `BodyContent` object for a tag returns a `JspWriter` which can be used to write back to the user. Note that this is different than the `pageContext.getOut()` method we used in simpler tags. We'll discuss the reason for this difference later in this section.
- `doAfterBody()` can return `EVAL_BODY_TAG` to cause the JSP runtime engine to process the body again and call `doAfterBody()` once more (useful in tags that iterate). It can return `SKIP_BODY` to inform the JSP engine to proceed to the `doEndTag()` method.

A discussion of some details of how the runtime engine manages `BodyTags` will clarify what is happening when a `BodyTag` is executed, and will also answer the question about why we must use a different `JspWriter` (accessed via `BodyContent.getEnclosingWriter()`) to write to the user than we did with standard tags. This section is pretty technical and discusses some of the intricacies of the JSP runtime that you might happily take for granted. Knowing these details will, however, help you truly understand what happens to the tags you are building.

BodyTags and the JSP runtime engine (behind the scenes)

Having gone through an in-depth discussion of the life cycle of `BodyTags` in chapter 4, you might think we know everything possible about JSP engines handling `BodyTags`, right? Although we learned when and why the runtime engine calls the methods of a `BodyTag`, what we didn't cover was how the engine manages the output of `BodyTags`. Since `BodyTags` can modify the contents of their body (which can contain other tags or scriptlets) these modifications must be managed by the engine until the tags are finished changing it. At that point they can be aggregated and sent to the user. This process requires a little juggling by the runtime engine in order to produce the predicted results for pages containing `BodyTags`. Let's take a look at that juggling act.

No matter what the content of a tag's body, whether it be scriptlets, static HTML, or other custom tags, the JSP engine will first process this content (as if it were anywhere else in the JSP) and then pass the results of that processing to the tag as its `BodyContent`. This is not such a simple task. How can all the scriptlets and tags suddenly hand over their results to the `BodyTag`? Redirecting all this output to a new location seems to be a daunting task, but the solution chosen by the JSP specification made it all much simpler than might be imagined.

The JSP specification's solution works on the premise that all output flowing to the user must be written to the implicit `out` object. When the JSP engine begins processing the body of a `BodyTag`, it swaps the implicit `out` with a new `JspWriter` that writes to a temporary holding tank. All the code and/or tags within a `BodyTag`'s body that "think" they are writing to the user, are really writing to some storage managed by the JSP engine. Later, when the body processing is completed, the engine gives the enclosing `BodyTag` access to this storage which now contains all the processed output of the tag's body. Indeed, the JSP specification defines a special `JspWriter` derivative called `BodyContent`, whose role is to serve as this holding tank and to be the implicit `out` variable during the processing of a `BodyTag`'s body. The `BodyContent` provides methods that let its developer access the content written into the `BodyContent`, as well as erase this content when needed. The problem becomes more complex in the face of `BodyTag` recursion—meaning `BodyTag` whose body encloses yet another `BodyTag`, and whose body encloses yet another `BodyTag`, and so forth. All these tags together force the JSP runtime to remember each tag's `BodyContent` and return to it when the enclosed tag is finished. To solve that, the JSP runtime is managing a stack of all the active `JspWriter` instances in the current page. In this way it can always pop the enclosed `BodyContent` out of the stack and return to the previous `JspWriter`.

NOTE This solution breaks down if one of the page developers breaks the rules and uses the `Writer/OutputStream` exported by the implicit response object. This is one of the reasons you must not use these `Writer/OutputStreams`.

This JSP fragment demonstrates how the JSP runtime uses the writer stack and the `out` implicit variable:

```
<%@ taglib
    uri="http://www.manning.com/jsptagsbook/simple-taglib"
    prefix="simp" %>

<html>
<body>
<simp:BodyTag1>
    Some text
    <simp:BodyTag2>
        Some other text
    </simp:BodyTag2>
</simp:BodyTag1>
</body>
```

Figure 6.6 shows the values taken by `out` and the use of the writer stack at any given moment. In this figure there are five phases in the JSP execution. In the first phase (a) the JSP runtime is passing through the file and approaches the tag named BodyTag1. At this time, the output generated from the JSP execution goes to the original `out` variable (generated by the JSP runtime) and the writer stack is empty.

The next phase (b) occurs when the JSP runtime tackles BodyTag1 and starts to process its body. At this point, the JSP runtime creates a `BodyContent` (out1) to be used inside the body, pushes the original `out` on the writer stack, and sets out1 to be the current implicit `out` variable. From this moment forward, the JSP output goes into out1.

Phase (c) occurs when the JSP runtime tackles BodyTag2 and begins processing its body. The JSP runtime will create yet another `BodyContent`, out2, to be used inside the body, push out1 onto the writer stack (there are now two writers on the stack), and set out2 to be the current implicit `out` variable. Now the JSP output goes into out2.

The finalization of BodyTag2 triggers the next phase (d) and the JSP runtime should return the writer state to the way it was before phase (c). To do that, the JSP runtime pops out1 from the writer stack and sets it to be the implicit `out` variable. The JSP output goes again into out1.

In the final phase (e), when BodyTag1 completes its execution, the JSP runtime should return the output state to its original form in phase (a). To facilitate this, the

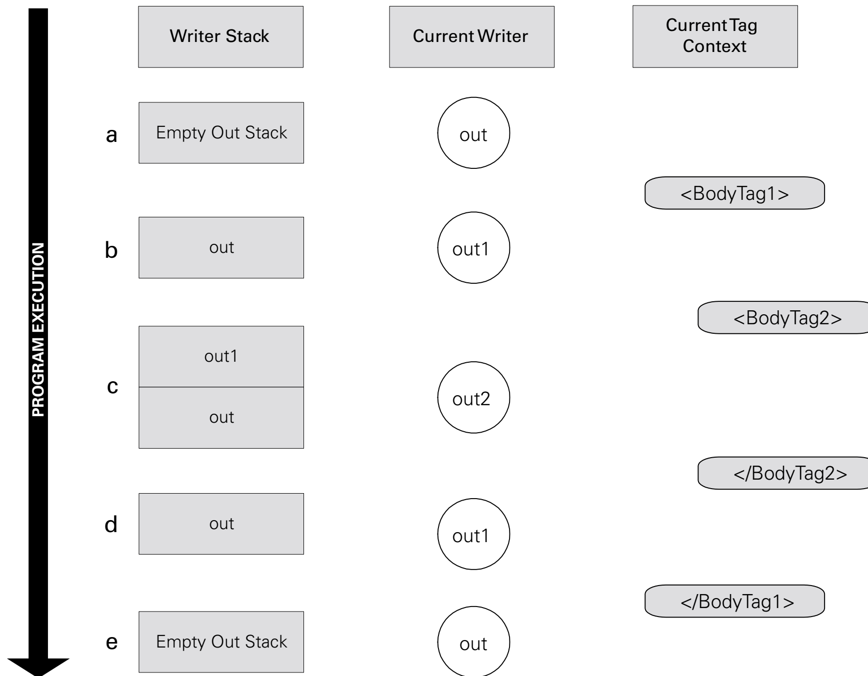


Figure 6.6 Body processing performed by the JSP runtime

JSP runtime pops the original `out` from the writer stack and sets it into the implicit `out` variable. The writer stack is empty again and whatever is written by the JSP goes again into the original `Writer out`.

In general, if the tag decides it wants to process its body, it informs the JSP runtime that it wants its body processed by returning the value `BodyTag.EVAL_BODY_TAG`. This causes the JSP runtime to do the following:

- The JSP runtime takes the current `JspWriter` object and pushes it onto the `JspWriter` stack.
- The JSP runtime takes the new `BodyContent` and sets it into the `out` implicit object. This way the new `BodyContent` will collect all the processed JSP that is written into the `out` object.
- The JSP runtime sets the new `BodyContent` object into the `BodyTag`, allowing the tag to get a reference to its processed body.
- The JSP runtime calls the tag's method `doInitBody()`. This lets the tag prepare for the body execution. The preparation can be initializing a variable

that is needed through the body processing or introducing new JSP scripting variables (discussed later on in the book).

- The JSP runtime processes the tag's body and writes all the results into the current `JspWriter`.
- When the body evaluation is completed, the JSP runtime needs to know if the tag wants it to repeat the body evaluation one more time. To find out, it will call the tag's method `doEndBody()`. A return code of `BodyTag.EVAL_BODY_TAG` instructs the runtime to repeat the body processing one more time, or a return code of `Tag.SKIP_BODY` instructs the runtime to stop evaluating the body.
- When finished processing the body, the JSP runtime pops the previous `JspWriter` from the stack and assigns its value to the implicit `out` object.

Handling the body obviously involves many steps and is a relatively complex operation that relies on cooperation between the tag and the JSP runtime. Fortunately, most of this complexity is not an issue since it is all handled by the JSP engine.

It should be clear why the `JspWriter` returned by `BodyContent.getEnclosingWriter()` is not always the one returned by `pageContext.getOut()`. In cases in which our tag is enclosed within yet another tag (as BodyTag2 was within BodyTag1) the enclosing writer is the `BodyContent` associated with the enclosing tag; for example, the enclosing writer for out2 in our previous example was out1.

6.7.3 A BodyTag example–logging messages

Having covered the low-level details of how `BodyTags` are executed, let's now look at some examples. We can break the usage patterns for `BodyTags` into two logical groups:

1. Tags that inspect and optionally modify their body one time.
   Tags in this group do so by returning `Tag.SKIP_BODY` from `doAfterBody()` the first time it is called.
2. Tags that return `BodyTag.EVAL_BODY_TAG` from `doAfterBody()` until a certain condition is met (at which point it returns `Tag.SKIP_BODY` to mark the end of the processing).
   In this way the tag may repeat its body processing over and over again, possibly while iterating some data source such as an array or a database.

Clearly both of these patterns fulfill two extremely useful cases, and we will deal with both of them at length.

We will now develop a sample tag that uses the first body pattern and offer several examples of the second pattern in chapter 10. This tag will log whatever information is located in its body into the servlet log. With this tag you can log errors in

your JSP files as well as improve error handler by logging the thrown exception to the servlet log. Let's first see how a tag can log information to the servlet logger.

Logging in servlets or JSPs

Logging in a servlet or a JSP is quite simple. Tags access the servlet log in the same way servlets do, by using the `ServletContext`. The `ServletContext` has two log methods that accept:

- A message string to be logged
- A message string and a `Throwable` object to be logged

A tag can use the `pageContext` to access the `ServletContext` by calling the method `pageContext.getServletContext()`, after which it can simply call any of the log methods.

Writing the Logging tag

If we want to design a simple tag to log messages we *could* just have it require two tag attributes: a message and a `Throwable` runtime object. The problem with this naïve approach is that it lacks flexibility. For example, there is a limit to what we can put in an attribute; hence, the message is limited and we will not be able to combine scriptlets (or other tags) with constant text. Moreover, you cannot have more than one message line since tag attributes cannot include multiple lines of text.

A preferable solution is to have the log tag take its input from its body. This way we can have any JSP content inside the body (including dynamic entities such as scriptlets and tags) and the log tag will use the processed output without a problem. Based on this argument, we'll build our tag so that it gets its log message from its body, rather than through a tag attribute.

NOTE Since logging is such a common practice, we implemented log methods in our superclasses `ExBodyTagSupport` and `ExTagSupport`. These log methods simply fetch the `ServletContext` object associated with this page and call the matching log method.

As we approach our log writer implementation, we see that this will not be the last tag we develop that needs to access its body. It will be useful then to have a base class to provide the functionality of body reading so that more specialized tags (such as the log writer) can just inherit from it. To accomplish this, we've built the abstract class `BodyReaderTag` (listing 6.23).

Listing 6.23 Source code for the BodyReaderTag abstract handler

```
package book.util;

import javax.servlet.jsp.JspException;

public abstract class BodyReaderTag extends ExBodyTagSupport {

    public int doAfterBody()
        throws JspException
    {
        processBody(bodyContent.getString());
        return SKIP_BODY;     ❶
    }

    protected abstract void processBody(String content)
        throws JspException;
}
```

❶ **Returning `SKIP_BODY` instructs the JSP runtime not to repeat processing the body** In essence, the first body pattern is implemented here.

The role of `BodyReaderTag` is to read the body and send it as a string to be processed by the method `processBody()`, implemented by an extending class. With `BodyReaderTag` it is now very easy to implement our log writer tag as presented in llisting 6.24.

Listing 6.24 Source code for the LogWriterTag's tag handler

```
package book.simpletasks;

import book.util.LocalStrings;
import book.util.BodyReaderTag;

import javax.servlet.jsp.JspException;

public class LogWriterTag extends BodyReaderTag {

    protected void processBody(String content)
        throws JspException
    {
        log(content);     ❶
    }
}
```

❶ **The method `log()` is implemented by the superclass `ExBodyTagSupport`.**

Listing 6.24 shows how the simple code that is the guts of `LogWriterTag` is dropped nicely into `processBody()`. Other tags may require more complex body

processing or initialization before entering the tag's body (through the use of `doInitBody()`), but they will still do the bulk of their processing in our `processBody()` method.

Using the logger tag we can create a useful error handler page. We'll forgo showing the TLD for this tag, since we've already seen several examples of this. Our tag's name will be "log" for the following example (which implements an error handling page):

```
<%@ page isErrorPage="true" %>
<%@ taglib
    uri="http://www.manning.com/jsptagsbook/simple-taglib"
    prefix="simp" %>
<html>
<head>
</head>
<body>
Sorry but your request was terminated due to errors:
<pre>
<simp:viewError/>
</pre>

<simp:log>
The following exception was thrown: <simp:viewError/>     ❶
</simp:log>
</body>
</html>
```

❶ **Uses the error viewer to print the thrown exeption into the logger's body** This way the exception is reported into the servlet log.

In this example, we assume that the `<simp:viewError>` tag simply writes the current exception out to the page. Let's review what happens during a request to this page and how our new tag is used. Note: we mention only the methods in this process that are important to this example; some life cycle calls have been omitted.

- The page is requested and passed an exception from some other servlet or JSP.
- The HTML at the top of the page is returned to the user.
- Our `<simp:log>` tag is encountered and its body is processed.
- The body is written to a `BodyContent` object, including the static message ("The following …") and the result of the evaluation of `<simp:viewError/>`, which is just the text of the exception.
- `doAfterBody()` is called, which is now handled at our base class. This, in turn, gets the `BodyContent` as a string and invokes `processBody()` on our subclass.
- Our log method is called with the stringified `BodyContent` as its parameter—thereby logging the message and the exception text to the servlet log.

We've now built a useful `BodyTag` and a reusable base class that will be helpful in upcoming examples. Later chapters will build on this section to provide high-level functionality such as conditioning (chapter 9) and iteration (chapter 10) but the fundamental principles we've learned here will not change.

6.8 Tag cooperation through nesting

A very powerful technique, though not as widespread as attribute use or even body manipulation, is tag nesting. Until now, none of our tags cooperated directly with any others. Admittedly, sometimes the execution of one tag affected another (such as in the case of the flush and redirect tags, and in our previous logger example), but this was not explicit cooperation. One tag acting alone cannot solve many real-world cases, which brings us to the need for some cooperation between different tags.

One obvious way that tags can cooperate is by using the JSP attributes (not to be confused with tag attributes). In this technique, tags use the JSP attributes as a shared memory space where they can exchange data. However, simple data exchange using the JSP attributes is not always sufficient. For example, what if we have a complex containment relation between two tags such that one tag has meaning only within the body of another? We surely cannot force such relations using the JSP attributes. When JSP attributes are used to coordinate two different tags, the JSP developer is typically required to name the different attributes (usually by providing an ID to the produced attribute) and to link the consumer of the attribute by again providing its name. Sometimes this is unnecessary extra work that can be resolved by another coordination technique. Indeed, JSP custom tags offer an implicit coordination technique by using parent-child relations among tags and the tag's parent attribute. This is known as nesting.

In chapter 4 we said that when a certain tag is contained within the body of another, the containing tag is the parent of the contained one (for instance, in the error handler presented in the previous section, `<simp:log>` was the parent tag of `<simp:viewError>`). Each tag has an attribute named `parent` that holds reference to its parent tag (set by the JSP runtime). This way the tag can traverse its parent list, searching for a specific tag with which it needs to cooperate.

This traversing and searching for a specific class is already implemented by `findAncestorWithClass()` in the class `TagSupport`. This method takes two parameters: a reference to the tag from which it should start to search (in many cases it will be `this`), and a class representing the type of tag handler we are seeking. For example, the following code fragment uses `findAncestorWithClass()` to find a tag in `this` tag parent chain whose class is `WrapperTag`.

```
WrapperTag wrapper =
        (WrapperTag)findAncestorWithClass(this,
                                          WrapperTag.class);
```

The class `TagSupport` provides yet another set of methods to ease tag cooperation through nesting, and these are the methods that deal with value manipulation. In many cases, contained tags will set values into their parents. One way is to have a setter method for each such value. `TagSupport`, however, provides an alternative group of value manipulation methods (including setting, removing, and getting value) that allow tags to exchange values with others without implementing setter methods. (All this value manipulation is implemented by keeping the values in a tag internal hash table and exposing its `set()` and `get()` methods.) So we can now take the wrapped class and set values into it in the following manner:

```
WrapperTag wrapper =
        (WrapperTag)findAncestorWithClass(this,
                                          WrapperTag.class);
wrapper.setValue("valuekey", "some value object");
```

Cooperation through nesting as shown in the previous code fragment is extremely useful when you design a tag family with specific syntactic structure (e.g., tag *x* should be contained within tag *y*), and it provides very easy coordination requiring nothing from the JSP developer. In the next chapter, when we implement a set of email-sending tags, we will see a more concrete example of the benefits and syntax for implementing this powerful feature.

6.9 Cleaning up

It's no accident that the final technique to discuss corresponds with the last stage of the tag's life cycle: cleanup. Most nontrivial tags collect state while executing, and these tags must free their state or else resource leaks will happen (Armageddon for an application server). Cleaning resources can be a tricky proposition for components managed by an external environment such as the JSP runtime engine. With tags, however, resource management is not the only motivation for cleanup. Tags are defined as reusable objects and, therefore, any tag is a candidate for pooling and reuse. Not only do tags need to free accumulated state, they also need to recycle themselves so that each reuse starts with all the properties in the exact same states.

To facilitate state cleanup and tag recycling, the JSP specification defines the tag method calls wherein these steps should occur. Cleaning after your tags is not rocket science, but doing it correctly requires a few considerations that we will explore soon. We will begin with a short reminder of the tag's life cycle and then

discuss how this life cycle affects your tag design cleanup. We will then see how the tags developed for this book implement cleanup and reuse.

6.9.1 Review of tag life cycle

Looking back at the tag's life cycle as explained in chapter 4, we can divide the tag life cycle into five stages:

- A tag is **created**. It should then have some initial state that allows it to be used by the JSP environment as needed.
- The JSP environment **initializes** the tag. At this time, the JSP environment sets various properties into the tag, starting with the `pageContext` and `parent` properties, and ending with other properties as specified by the tag attributes and the TLD.
- The JSP environment puts the tag into **service** by calling `doStartTag()`. The tag is now starting to collect state needed for the current execution.
- The JSP environment informs the tag that the current **service** is done by calling `doEndTag()`. The tag should now free all the resources accumulated for the ended service phase. At the end of `doEndTag()`, the tag should be in a state that allows it to be reused again at the same JSP page.
- The JSP environment puts the tag into **reuse** by calling its `release()` method. The tag should now recycle itself, returning to the same state as when it was **created.**

All this life cycle discussion makes it clear that there are two cleanup points:

1. `doEndTag()`—The tag must free all the state allocated for its current service period.
2. `release()`—The tag must recycle itself. This usually entails clearing the tag's properties (for example `pageContext` and `parent`), since all other state was probably part of the service phase.

6.9.2 Exceptions and cleanup

What happens if an exception is thrown somewhere within this life cycle? Most of the tag's methods can throw a `JspException`, but the method may (of course) throw a runtime exception such as `java.lang.NullPointerException`. What then? The answer is rather simple. If `doStartTag()`, `doEndTag()`, or some other tag callback method was called and threw an exception, in JSP1.1 the JSP runtime would immediately call `release()`, not `doEndTag()`. This is not per the specification, but is the common practice and a reasonable solution since the tag should not

gather state until the call to `doStartTag()`. For example, look at the following pseudocode fragment that is similar to that generated by Tomcat's JSP1.1 translator.

```
Sometag _t = get Sometag ();
t.setPageContext(pageContext);
t.setParent(null);
t.setSomeProperty(…);

try {
    t.doStartTag();
    // some code was omitted …
    t.doEndTag();
} finally {
    t.release();
}
```

As you can see, `release()` is executing within a `finally` block, assuring us that it will be called even in the face of exceptions.

JSP1.2 offers an improved and regulated exception handling capability by providing the `TryCatchFinally` interface. Tags that implement `TryCatchFinally` inform the JSP runtime that they want to be notified when exceptions occur during their run. The JSP runtime will assure that the `TryCatchFinally` methods in the tags will be called in the appropriate time.

TryCatchFinally and the JSP runtime

As stated in chapter 4, the `TryCatchFinally` interface exports the following methods:

- `doCatch()` allows the JSP runtime to inform the tag that an exception was thrown while executing it.
  The tag can then respond to the exceptional condition based on the exception parameter and the general state of the tag.
- `doFinally()` is called by the JSP runtime whenever the tag finishes its service phase.
  This way the tag can free the state it accumulated when serving the request.

But how will the JSP runtime assure that?

The answer is elementary. The JSP translator surrounds the tag with code fragments as demonstrated in the following listing;

```
// Execute the tag lifecycle
h = get a Tag();  // get a tag handler

h.setPageContext(pc);  // initialize as desired
h.setParent(null);
h.setFoo("foo");

// Call the lifecycle methods inside a try-catch-finally
// fragment.
```

```
try {
  doStartTag()...
  ....
  doEndTag()...
} catch (Throwable t) {
  // react to exceptional condition
  // assure that doCatch() get called
  h.doCatch(t);
} finally {
  // restore data invariants and release per-invocation resources
  // assure that doFinally() get called
  h.doFinally();
}

... other invocations perhaps with some new setters
...
h.release();  // release long-term resources
```

The code emitted by the JSP runtime makes sure that the tag will be notified of exceptions, no matter what happens in or out of the tag.

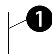
Using the `TryCatchFinally` interface, implementing cleanup for our tag is very simple. All we need is to make sure that we use the method `doFinally()` and `doCatch()` to clean up the tag, and the JSP runtime will assure us that these methods are called, even in the face of exceptions.

6.9.3 *Improving our base classes to handle cleanup*

The end result is that a tag should be ready to perform two tasks: cleaning its accumulated state and recycling itself. Although these two tasks will usually occur in different methods, sometimes (when an exception is thrown) both will happen in `release()` (or in `doFinally()`). All this rather complex cleanup logic would fit best in some superclass and the extending classes should just implement their own resource deallocation and recycling. So, in all the samples in this book, most cleanup logic is buried inside `ExTagSupport` and `ExBodyTagSupport`,[6] as seen in listing 6.25.

Listing 6.25 Cleanup logic in ExTagSupport and ExBodyTagSupport

```
public int doEndTag()
    throws JspException
{
    clearServiceState();
    return super.doEndTag();
}
```



[6] For JSP1.2 we replace `release()` with `doFinally()` in the `TryCatchFinally` interface.

```
public void release()
{
    clearServiceState();     ❷

    clearProperties();       ❸
    super.release();
}

protected void clearProperties()     ❹
{
}

protected void clearServiceState()     ❹
{
}
```

❶ **Implementing service state cleanup, calling `clearServiceState`, and informing that the state is clear.**

❷ **Implementing service state cleanup in the face of exceptions. False value in `isServiceStateClear` means that an exception prevented the execution of `clearServiceState`.**

❸ **Clearing the tag's properties.**

❹ **Placeholder methods for clearing tag's properties and service state.**

The idea behind the presented cleanup logic is to release the developer from thinking in terms of "OK, release is getting called, what should I do?" and instead, think in terms of a specific tag logic ("OK, lets clear these two attributes."). To help, we extend the `ExTagSupport` and `ExBodyTagSupport` classes to expose two methods the tag developer may wish to override. The rules for overriding these methods are the following:

- `clearProperties()`—Overriding this lets the tag clear its specific custom properties.
  If the tag inherits yet another tag with properties of its own, it should add a call to `super.clearProperties()` at the end of its `clearProperties()` method (always do that by default, just to make sure).
- `clearServiceState()`—Overriding this allows the tag to free its service phase state.
  In cases of tag inheritance, the rules for `clearProperties()` are also applicable for `clearServiceState()`. Note that in many cases `clearServiceState()` is being called twice, once in `doEndTag()` and again in `release()`. This is because we need to make sure that `clearServiceState()` will be called in case of an exception.

Using these two methods frees the tag developer to think in terms of the specific tag state, but for the cleanup logic to work, the developer would be wise to follow these rules:

- Tags that override `release()` should call `super.release()` to activate the cleanup logic.
- Tags that override `doEndTag()` should call `super.doEndTag()`.

These rules are not complex or restrictive as most of the tags do not need to override `doEndTag()` and `release()`. For that reason, all the tags developed for this book are going to use this autocleanup mechanism.

JSP1.2 NOTE The ExXXXSupport classes for JSP1.2 implement the cleanup logic using the `TryCatchFinally` interface by putting the call to `clearServiceState()` and `clearProperties()` inside `doFinally()`.

As a final note, you do not have to use this proposed cleanup logic in your own tags (although we recommend it), but it's a good idea to observe the basic cleanup guidelines described in this section.

6.10 Summary

We've covered a wealth of useful techniques here, including how to write content to the user, how to use tag attributes, using the servlet API, initializing tags, sharing data in different scopes, `BodyTags`, and cleaning up your tags' resources. The common theme throughout these techniques is providing tags the facilities they need to make them effective. With these skills alone, you can begin building production quality tag libraries for your applications.

There are still several aspects of tag development with which we've only flirted so far. The remainder of the book will focus on strengthening your grasp of the concepts learned here and applying them to real-world examples and scenarios. In the next chapter we apply many of these techniques as we build our first, real-world tag library for sending email.