

Help for Server-Side Java Developers

3rd Edition
Covers JSP 2.0 & JSTL 1.1

JavaServer Pages™



O'REILLY®

Hans Bergsten

Developing Custom Tag Libraries as Tag Files

Starting with the 2.0 version of the JSP specification, custom tag library actions can be implemented in two ways: as Java classes or as regular text files containing JSP elements. In prior versions, custom actions could only be implemented as Java classes, putting them out of the reach of nonprogrammers. Another problem with the Java implementation of custom actions is that you're forced to print out HTML code with `println()` calls to produce complex content—the very problem JSP was supposed to solve.

In this chapter I show you how to develop custom actions as plain text files and package them as tag libraries that can be used in JSP pages.

Creating and Using a Tag File

A tag file is a text file that contains JSP elements implementing the functionality of a custom action. You must use a `.tag`* filename extension to identify this type of file to the web container. All JSP elements that you can use in a JSP file can also be used in a tag file, with exception to the `page` directive (a tag file is not a page). There are also a few JSP directives that are only allowed in a tag file, as you will see shortly. Apart from that, creating a tag file is no different than creating a JSP page. Once created and installed, a tag file is used the same as the custom actions implemented in Java that you've seen in previous chapters.

Example 11-1 shows a very simple tag file.

Example 11-1. Simple tag file (copyright.tag)

```
<%@ tag body-content="empty" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

* If you write the tag file in XML format, as described in Chapter 17, you must instead use the `.tagx` extension.

Example 11-1. Simple tag file (copyright.tag) (continued)

```
<jsp:useBean id="now" scope="application" class="java.util.Date" />  
Copyright &copy; ${now.year + 1900} My Company
```

This tag file inserts a copyright statement with the current year in the calling page. The first line is a tag directive. You may use attributes of this directive to specify a description, icon, or an example that a page-authoring tool can show the designer. Other attributes let you specify whether EL expressions should be processed, as well as various information related to scripting code, i.e., the same type of information as you specify with the page directive in JSP pages. All of these attributes are described in Appendix A.

In most cases, tag file authors only care about the attribute used in Example 11-1: `body-content`. This attribute defines how the custom action element's body should be handled, and that it must have one of these values: `empty`, `scriptless` (the default), or `tagdependent`. If it's `empty` (as in Example 11-1), trying to use a body for the custom action element results in a syntax error. The `scriptless` value means that the body can contain any JSP elements except the type of scripting elements described in Chapter 16. In other words, template text, EL expressions, standard actions, and custom actions are all allowed. As you will see later, the tag file can ask the container to process the actions in a scriptless body when and how often as it wants through the use of standard action named `<jsp:doBody>`. If the `body-content` attribute is set to `tagdependent`, the action element body is treated as pure template text (i.e., action elements and EL expressions in the body are not processed, just handled as plain text).

The rest of the tag file in Example 11-1 looks just like an ordinary JSP page. It declares that it uses the JSTL core library, a `<jsp:useBean>` standard action to create an instance of the `java.util.Date` class representing the current time (if it isn't already available in the application scope), and finally outputs static template text mixed with a dynamic value (the current year) generated by an EL expression: `${now.year + 1900}`.*

Tag files can be placed directly in the web application structure under the `WEB-INF/tags` directory or a subdirectory. Each directory containing tag files represents a separate tag library:

```
WEB-INF/tags/  
  mytags/  
    copyright.tag  
    forEvenAndOdd.tag  
    htmlFormat.tag  
    motd.tag  
  myothertags/
```

* The year property of a `java.util.Date` (represented by the `getYear()` method) contains the current year minus 1900, so here I add 1900 to get the real year.

```
foo.tag  
bar.tag
```

Here we have two tag libraries: `mytags` and `myothertags`. The `mytags` library contains the `copyright.tag` file from Example 11-1 plus three other tag files. By default, the name of the custom action implemented by the tag file is the filename minus the `.tag` extension, so the `copyright.tag` file represents a custom action named `copyright` in the `mytags` library.

A JSP page must declare that it uses a tag library represented by tag files in the web application structure with a slightly different `taglib` directive than what we've used in earlier chapters:

```
<%@ page contentType="text/html" %>  
<%@ taglib prefix="my" tagdir="/WEB-INF/tags/mytags" %>  
  
<html>  
  <body bgcolor="white">  
    ...  
    <my:copyright/>  
  </body>  
</html>
```

Note that the `tagdir` attribute is used instead of the `uri` attribute. The value of the `tagdir` attribute is the context-relative path to the directory that contains the tag files for the library. It may seem redundant to have to specify the `/WEB-INF/tags` part of the path, since all tag library directories must start with this path. Regardless, the JSP specification group decided to require this to be consistent with other attributes taking path values.

When the JSP container processes this JSP page, it locates the `copyright.tag` file in the `WEB-INF/tags/mytags` directory and turns it into a format that the container can invoke. The conversion details are left open by the JSP specification, allowing container vendors to compete with smart implementations. Tomcat turns the tag file into a Java class and compiles it, but other implementations are possible (e.g., converting it to a proprietary data structure).

Tag files can also be packaged in a JAR file. It requires a bit more work and is primarily of interest for tag files intended to be reused in many applications, so let's defer the details to the end of this chapter. One thing to note at this time, though, is that when the tag files are packaged in a JAR file, the `taglib` directive is used with the `uri` attribute exactly as in the previous chapters. This means that tag files packaged in a JAR file are indistinguishable from custom actions implemented as Java classes. You can therefore implement the actions as tag files initially (because it's easier) and convert them to Java classes later (maybe to gain better performance) without having to make any changes in the JSP pages that use them.

Accessing Attribute Values

The tag file in Example 11-1 is too simple to illustrate all that you can do with tag files. For instance, most real-world tag files are controlled through attribute values set by the page author. You may recall from Chapter 7 that the `<ora:motd>` custom action has a `category` attribute for selecting the message category that messages should be picked from. Example 11-2 shows how a tag file implementation of the `<ora:motd>` action declares, accesses, and uses this attribute value.

Example 11-2. Using attributes in a tag file (motd.tag)

```
<%@ tag body-content="empty" %>
<%@ attribute name="category" required="true" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<jsp:useBean id="mmb" class="com.ora.jsp.beans.motd.MixedMessageBean" />
<c:set target="${mmb}" property="category" value="{category}" />
${mmb.message}
```

Each attribute must be declared with an attribute directive in a tag file. In Example 11-2, the `category` attribute is declared using an attribute directive with the `name` attribute set to `category`. The `required` attribute is set to `true`, meaning that the page author must specify a value for the `category` attribute; the container complains if the attribute is missing. The default value for `required` is `false`, so you can leave it out for attributes that are optional.

Another attribute of the attribute directive, not used in Example 11-2, is `rtexprvalue`. A value of `true` means that the author can specify the value either as a static string or as a *request-time attribute value*, such as an EL expression; `false` means the value must be a static string. The default value is `true`, so you only need to use this attribute if you absolutely require a static value.*

The value the page author assigns to an attribute shows up as a page scope variable in the tag file, with the same name as the attribute. This makes it easy to use it in an EL expression. In Example 11-2, a `<c:set>` action sets the `category` property in a `MixedMessageBean` (which contains the list of messages). The EL expression used as the value gets the `category` page scope variable that represents the `category` attribute.

It's important to note, however, that the page scope seen by the tag file is not the same as the page scope seen by the page that invokes the tag file—I sometimes call the page scope seen by the tag file the *tag scope* to make this distinction. By giving the tag file its own local page scope, there's no chance for confusion between the calling page and the tag file if they use the same names for page scope variables.

* The convention established by JSTL is that only `var` and `scope` attributes should have `rtexprvalue` set to `false`. These attributes may need to be available in the translation phase (hence, have static string values) in a future version of the JSP specification to allow for additional syntax checking and optimizations.

Using Undeclared Attributes

Occasionally, declaring all attributes for a tag file can be a hassle. Say you want to develop a tag file that generates an HTML table, and you want the page author to be able to specify all standard attributes that an HTML table element supports. That's a lot of attributes and the tag file would need to test for the existence of each one. A better approach for this scenario is to use the tag directive's `dynamic-attributes` attribute. This attribute declares that the tag file accepts any custom action element attribute. The attribute value is the name of a local page scope variable that holds a collection (a `Map`) with all undeclared attribute names and values. Example 11-3 shows an example of a tag file that uses this approach to generate a table with all request header values.

Example 11-3. Using undeclared attributes in a tag file (headers.tag)

```
<%@ tag body-content="empty" dynamic-attributes="dynattrs" %>
<%@ attribute name="caption" required="true" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<table
  <c:forEach items="${dynattrs}" var="a">
    ${a.key}="${a.value}"
  </c:forEach>
>
  <caption>${caption}</caption>
  <tr>
    <th>Name</th>
    <th>Value</th>
  </tr>
  <c:forEach items="${header}" var="h">
    <tr>
      <td>${h.key}</td>
      <td>${h.value}</td>
    </tr>
  </c:forEach>
</table>
```

The `dynamic-attributes` attribute declares a variable named `dynattrs` to hold the undeclared attributes, and a JSTL `<c:forEach>` action loops through the collection and adds the name and value for each to the HTML `<table>` element's attribute list. As shown in Example 11-3, you can declare regular attributes in the same tag file. This example declares a mandatory attribute named `caption`, used to add a caption text for the table.

This is how you can use the tag file, shown in Example 11-3, in a JSP page:

```
<%@ page contentType="text/html" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags/mytags" %>

<html>
  <head>
```

```

    <title>Headers</title>
  </head>
  <body bgcolor="white">
    <my:headers caption="Request Headers"
      border="1" cellspacing="0" cellpadding="5" />
  </body>
</html>

```

The action element for the tag file defines values for the mandatory caption attribute plus three undeclared attributes: border, cellspacing, and cellpadding.

Processing the Action Body

So far, the tag files we've looked at ignore (or actually forbid) the body of the custom action element used to invoke them, but the body is often an important part of the equation. One example is a conditional custom action, such as a variation of the `<c:if>` JSTL action. It needs to process the body if the condition is true. Another example is a custom action that transforms the body in some way or simply uses it as input.

Let's develop a custom action that transforms its body content. It first converts all characters that have special meaning in HTML and XML to the corresponding character entity codes (e.g., `<` to `<`), and then converts special proprietary codes into HTML elements. A custom action like this can be used to process user input in an online forum to protect it against cross-site scripting attacks while still allowing for limited formatting of the messages. Here's how you can use this custom action in a JSP page:

```

<%@ page contentType="text/html" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags/mytags" %>

<!-- Create test data -->
<c:set var="message">
  This is just a lot of text that the browser will format to
  fit the browser window. Attempts to <blink> add HTML elements
  are dealt with by conversion to character entities.
  [code]
  This part I want the browser to leave alone, so that
  all my indentations are left intact:

  public class Foo {
    public String getBar() {
      return bar;
    }
  }
  [/code]
  And then some regular text again.
</c:set>
<html>

```

```

<head>
  <title>Online Forum</title>
</head>
<body bgcolor="white">
  <h1>Online Forum</h1>
  Here's a formatted message:
  <p>
    <my:htmlFormat>
      ${message}
    </my:htmlFormat>
  </p>
</body>
</html>

```

This page first saves test data containing text, an HTML element, and the proprietary formatting codes in a variable named `message`. In a real application, the text would likely come from a database or some other external source. It then processes the text with the `<my:htmlFormat>` custom action.

The result is shown in Figure 11-1. Note how the `<blink>` HTML element is displayed instead of causing most of the text to blink, and how the formatting is preserved for all text between the proprietary `[code]` and `[/code]` tags.

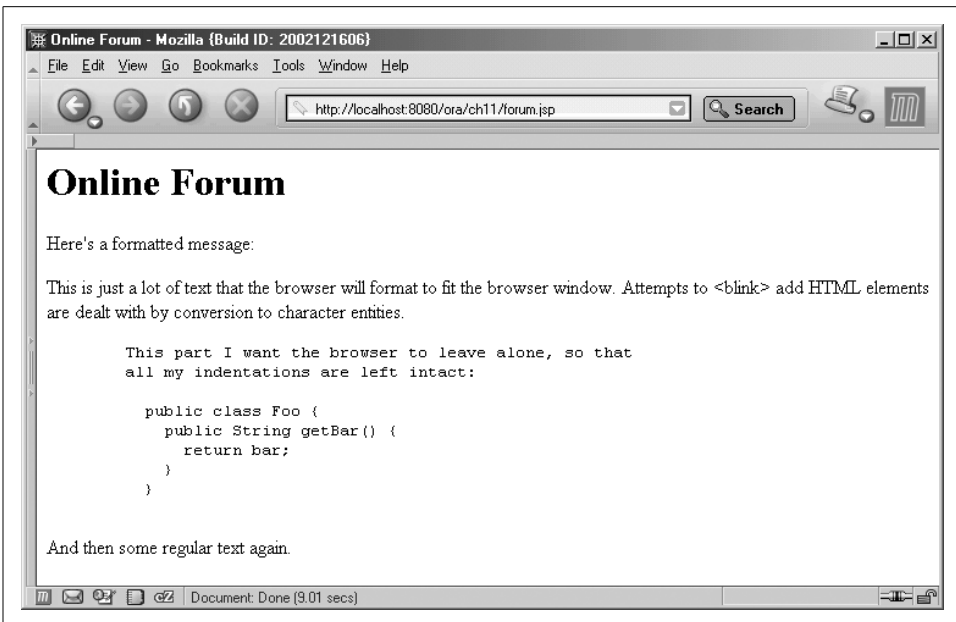


Figure 11-1. Result of text processing with a custom action

Example 11-4 shows the tag file that implements the `<my:htmlFormat>` custom action.

Example 11-4. Processing the body (*htmlFormat.tag*)

```
<%@ tag body-content="scriptless" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!-- Capture the body evaluation result in a variable -->
<jsp:doBody var="bodyRes" />

<!-- Convert special characters to character entities -->
<c:set var="escapedBody" value="{fn:escapeXml(bodyRes)}" />

<!-- Replace "[code]/[/code]" with "<pre></pre>" -->
<c:set var="convBody"
  value="{fn:replace(escapedBody, '[code]', '<pre>')}" />
<c:set var="convBody"
  value="{fn:replace(convBody, '[/code]', '</pre>')}" />

<!-- Output the result -->
${convBody}
```

Note that the tag directive in Example 11-4 sets the `body-content` attribute to `scriptless`. As I mentioned earlier, this means that the page author is allowed to put template text, standard actions and custom actions, in the body but not scripting elements (i.e., Java code).

It's after the directive elements that this example gets interesting; here's a standard action that we have not discussed before: `<jsp:doBody>`. This action can only be used in tag files. It evaluates the body of the custom action element, meaning that all action elements (if any) in the body are called and the output they produce is mixed with the template text (if any). The result is saved in a variable, using the `var` attribute to name the variable. This attribute is optional, as shown in Table 11-1, and you can use the `varReader` attribute as an alternative. If you don't specify any of these attributes, the result is added to the page invoking the custom action.

Table 11-1. Attributes for `<jsp:doBody>`

Attribute name	Java type	Dynamic value accepted	Description
<code>var</code>	<code>String</code>	No	Optional. The name of the variable to hold the body evaluation result as a <code>String</code> .
<code>varReader</code>	<code>String</code>	No	Optional. The name of the variable to hold the body evaluation result as a <code>java.io.Reader</code> .
<code>scope</code>	<code>String</code>	No	Optional. The variable scope; one of <code>page</code> , <code>request</code> , <code>session</code> , or <code>application</code> . Default is <code>page</code> .

The difference between the `var` and `varReader` attributes is the type of Java object used for capturing the result. The `var` attribute captures it as a `String` and is sufficient for most cases. When the `varReader` attribute is used, the result is captured as a `java.io.Reader` object instead. For large results, this can be slightly more efficient when combined with an action or function for the transformation that reads its input from a `Reader`. Along with one of `var` or `varReader`, you can also specify the scope for the variable with the `scope` attribute.

The rest of the tag file in Example 11-4 transforms the captured body. First it uses the JSTL `fn:escapeXml()` function to convert all special characters to character entity codes, and then it replaces all occurrences of `[code]` and `[/code]` with the HTML `<pre>` and `</pre>` tags using the JSTL `fn:replace()` function, to preserve formatting in these sections. Finally, the converted body evaluation result is added to the calling page with a simple EL expression.

Processing Fragment Attributes

Processing the custom action body is easy and powerful as you can see, but wait, there's more! The custom action body is actually just a special case of what's called a *JSP fragment* in the JSP specification. A JSP fragment is an executable representation of a set of dynamic elements (actions and EL expressions), optionally mixed with template text. When the tag file invokes the fragment, all the dynamic elements in the fragment are executed. Since the elements have access to the current values of all scoped variables, the result typically differs from invocation to invocation, and the tag file can invoke it any number of times (e.g., once or none for a conditional action or multiple times for an iteration action).

In Example 11-4, the `<jsp:doBody>` action invokes the special fragment representing a custom action element body, but named fragments can also be provided as custom action attributes and be invoked by the tag file. Such fragments are invoked with the `<jsp:invoke>` action, described in Table 11-2.

Table 11-2. Attributes for `<jsp:invoke>`

Attribute name	Java type	Dynamic value accepted	Description
<code>fragment</code>	<code>String</code>	No	Mandatory. The name of the fragment to invoke.
<code>var</code>	<code>String</code>	No	Optional. The name of the variable to hold the body evaluation result as a <code>String</code> .
<code>varReader</code>	<code>String</code>	No	Optional. The name of the variable to hold the body evaluation result as a <code>java.io.Reader</code> .
<code>scope</code>	<code>String</code>	No	Optional. The variable scope; one of <code>page</code> , <code>request</code> , <code>session</code> , or <code>application</code> . Default is <code>page</code> .

Let's develop a variant of the JSTL `<c:forEach>` action to illustrate how you can use named fragments. Say you want to loop through all the elements in a collection to generate an HTML table, and you want to render even rows one way and odd rows another. Here's a page that solves this problem by using a custom action with separate fragment attributes for even and odd rows:

```
<%@ page contentType="text/html" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags/mytags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
  <head>
    <title>Even and Odd Rows</title>
  </head>
  <body bgcolor="white">
    <h1>Even and Odd Rows</h1>
    <table>
      <my:forEvenAndOdd items="a,b,c,d,e">
        <jsp:attribute name="even">
          <c:set var="counter" value="{counter + 1}" />
          <tr bgcolor="red"><td>${counter}: Even Row</td></tr>
        </jsp:attribute>
        <jsp:attribute name="odd">
          <c:set var="counter" value="{counter + 1}" />
          <tr bgcolor="blue"><td>${counter}: Odd Row</td></tr>
        </jsp:attribute>
      </my:forEvenAndOdd>
    </table>
  </body>
</html>
```

A fragment attribute value is defined using the `<jsp:attribute>` action introduced earlier. The body of this action element makes up the content of the fragment. In the page shown here, each fragment attribute values contain a JSTL `<c:set>` action for incrementing a counter and HTML table row and cell elements for showing the counter's value plus the static text "Even Row" and "Odd Row", respectively. The fragments also set different row background colors to make the differences clear. The result of processing this page is shown in Figure 11-2.

Note how the current value of the counter page scope variable is used for each new row, and how the rows alternate between the even and odd fragments. Example 11-5 shows the tag file for the `<my:forEvenAndOdd>` custom action.

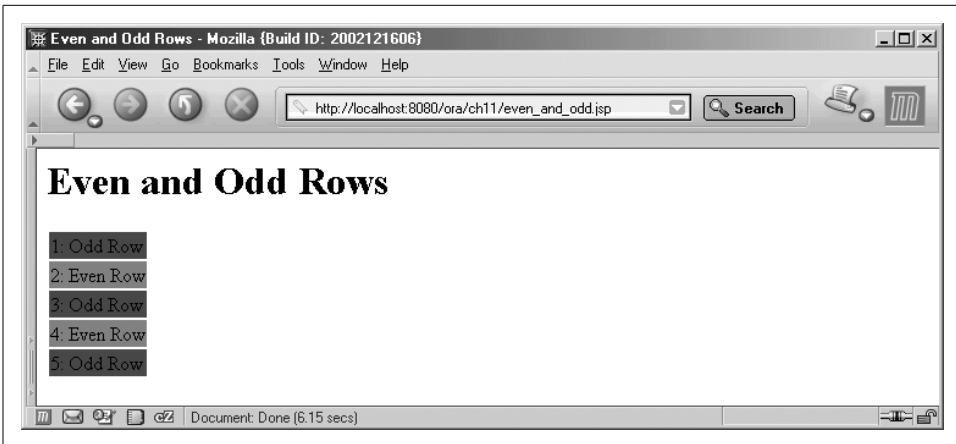


Figure 11-2. Representing even and odd rows as fragments

Example 11-5. Using fragment attributes (forEvenAndOdd.tag)

```
<%@ tag body-content="empty" %>
<%@ attribute name="items" rtexprvalue="true" required="true" %>
<%@ attribute name="even" fragment="true" required="true" %>
<%@ attribute name="odd" fragment="true" required="true" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:forEach items="${items}" varStatus="status">
  <c:choose>
    <c:when test="${status.count % 2 == 0}">
      <jsp:invoke fragment="even" />
    </c:when>
    <c:otherwise>
      <jsp:invoke fragment="odd" />
    </c:otherwise>
  </c:choose>
</c:forEach>
```

The tag directive specifies that the body must be empty; in this example, it must only contain the `<jsp:attribute>` elements (no template text or other elements), and they are considered alternatives to regular element attributes, not body content.

To tell the container to use an executable fragment as the attribute value, the attribute must be declared as such. Note that the attribute directive's `fragment` attribute is set to `true` for both the `even` and `odd` attributes. Otherwise the container evaluates the `<jsp:attribute>` body once and sets the attribute to the resulting value, as described in Chapter 6.

After the directives in Example 11-5, JSTL actions are used to loop through the list of items, and decide whether it's an even or odd row. The `<jsp:invoke>` action then invokes the appropriate fragment. The result is what you see in Figure 11-2.

Exposing Data to the Calling Page Through Variables

Attributes provide input to a custom action, but sometimes you also need to give the page that contains the custom action access to data produced by the custom action. For instance, the `<my:forEachEvenAndOdd>` action is not all that useful unless the page can access the current iteration value in the fragments for even and odd rows. To handle this requirement, data can be passed from a custom action to the caller by exposing it through declared variables.

Example 11-6 shows a version of the tag file from Example 11-5 that's been extended to expose the current iteration value as a variable named `current`. All differences between the examples are highlighted.

Example 11-6. Exporting data through variables (forEvenAndOdd2.tag)

```
<%@ tag body-content="empty" %>
<%@ attribute name="items" rtexprvalue="true" required="true" %>
<%@ attribute name="even" fragment="true" required="true" %>
<%@ attribute name="odd" fragment="true" required="true" %>
<%@ variable name-given="current" variable-class="java.lang.Object"
scope="NESTED" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:forEach items="${items}" varStatus="status" var="current">
  <c:choose>
    <c:when test="${status.count % 2 == 0}">
      <jsp:invoke fragment="even" />
    </c:when>
    <c:otherwise>
      <jsp:invoke fragment="odd" />
    </c:otherwise>
  </c:choose>
</c:forEach>
```

The `variable` directive declares the variable. The `name-given` attribute specifies its name and the `variable-class` attribute its type. (Here I use the most generic class possible, `java.lang.Object`, because the collection to iterate over can contain elements of any type.)

The `scope` attribute accepts one of three values: `AT_BEGIN`, `AT_END`, or `NESTED`. It controls where the caller sees the variable. Despite its name, it has nothing to do with the scopes we've talked about earlier (page, request, session, and application), so *visibility* would have been a better name for this attribute. If it's set to `AT_BEGIN`, the variable is visible to the caller immediately after the start tag for the custom action element. If the attribute is set to `AT_END`, the variable is visible after the end tag. `NESTED` means it's only visible between the start and end tags.

To make the data visible to the caller, the tag file sets a page scope variable with the name declared by the variable directive. I told you earlier that the tag file has its own page scope, separate from the caller, so the container must do a bit of magic for this to work. For a variable declared as `AT_BEGIN` or `NESTED`, it copies the value of the variable in the tag file's page scope to the caller's page scope before invoking a fragment. If the variable is declared as `AT_BEGIN` or `AT_END`, it copies the value before exiting the tag file. In the case of a `NESTED` variable, it also saves and restores the value of the caller's page scoped variable with the same name, if any, before entering and exiting the tag file. Don't worry if this sounds confusing at first; it actually ends up working as you would expect it to.

The tag file in Example 11-6 exposes a variable named `current`, containing the value of the current iteration value. The local variable is set indirectly with help of the `var` attribute of the `<c:forEach>` action. As you may recall, the `<c:forEach>` action makes the current iteration value available in the page scope variable named by this attribute. By setting the name of the `<c:forEach>` variable to the name of the declared tag file variable, the variable value set by the `<c:forEach>` action is also exposed to the caller.

With the new version of the tag file, I can use it to display the current iteration value in each row:

```
<%@ page contentType="text/html" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags/mytags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
  <head>
    <title>Even and Odd Rows</title>
  </head>
  <body bgcolor="white">
    <h1>Even and Odd Rows</h1>
    <table>
      <my:forEvenAndOdd2 items="a,b,c,d,e">
        <jsp:attribute name="even">
          <c:set var="counter" value="{counter + 1}" />
          <tr bgcolor="red"><td>${counter}: Even Row: ${current}</td></tr>
        </jsp:attribute>
        <jsp:attribute name="odd">
          <c:set var="counter" value="{counter + 1}" />
          <tr bgcolor="blue"><td>${counter}: Odd Row: ${current}</td></tr>
        </jsp:attribute>
      </my:forEvenAndOdd2>
    </table>
  </body>
</html>
```

Note how the exposed variable is used in EL expressions in both fragments.

There's still a problem here: the exposed variable name is hardcoded into the tag file. This may be okay in some cases, but it's better if the variable name can be specified

using an attribute, just as you can pick a name with the var attribute for all JSTL actions that expose data. Fortunately, there's a solution, shown in Example 11-7.

Example 11-7. Letting the page author specify the variable name (forEvenAndOdd3.tag)

```
<%@ tag body-content="empty" %>
<%@ attribute name="items" rtexprvalue="true" required="true" %>
<%@ attribute name="var" rtexprvalue="false" required="true" %>
<%@ attribute name="even" fragment="true" required="true" %>
<%@ attribute name="odd" fragment="true" required="true" %>
<%@ variable name-from-attribute="var" alias="current"
    variable-class="java.lang.Object" scope="NESTED" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:forEach items="${items}" varStatus="status" var="current">
  <c:choose>
    <c:when test="${status.count % 2 == 0}">
      <jsp:invoke fragment="even" />
    </c:when>
    <c:otherwise>
      <jsp:invoke fragment="odd" />
    </c:otherwise>
  </c:choose>
</c:forEach>
```

Instead of the name-given attribute used in the previous example, I use the name-from-attribute and alias attributes of the variable directive in Example 11-7. The name-from-attribute attribute value is the name of the custom action attribute used to name the variable. The named attribute (var in this example) must be declared as required and must not accept a request time value. The alias attribute value declares the name of the tag file's local page scope variable, which the JSP container copies to the caller's page scope as described earlier. The aliasing trick is needed because the page author can assign any name for the variable when she uses the custom action, but a fixed name must be used when developing the tag file.

The rest of Example 11-7 is identical to Example 11-6, but I can now specify the variable name in the calling page like this:

```
<%@ page contentType="text/html" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags/mytags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
  <head>
    <title>Even and Odd Rows</title>
  </head>
  <body bgcolor="white">
    <h1>Even and Odd Rows</h1>
    <table>
      <my:forEvenAndOdd3 items="a,b,c,d,e" var="anyName">
        <jsp:attribute name="even">
          <c:set var="counter" value="{counter + 1}" />
```

```

        <tr bgcolor="red"><td>${counter}: Even Row: ${anyName}</td></tr>
    </jsp:attribute>
    <jsp:attribute name="odd">
        <c:set var="counter" value="${counter + 1}" />
        <tr bgcolor="blue"><td>${counter}: Odd Row: ${anyName}</td></tr>
    </jsp:attribute>
</my:forEvenAndOdd3>
</table>
</body>
</html>

```

Aborting the Page Processing

In Chapter 9, I described how using the `<jsp:forward>` action or the JSTL `<c:redirect>` action shifts processing from the current page to the page specified by the page attribute, effectively aborting processing of the current page. Custom actions implemented as Java classes can cause the same thing to happen.

There's no directive or similar mechanism that a tag file can use to explicitly abort processing, but using `<jsp:forward>`, `<c:redirect>`, or a custom action that aborts page processing in a tag file has the same effect; both the tag file processing and the processing of the page that invokes the tag file stop after it aborts the processing. You can use this feature to, for instance, develop a smart forwarding action that decides which page to forward to based on runtime conditions, such as the time of the day, the current user, or the type of browser accessing the page.

Packaging Tag Files for Easy Reuse

All examples in this chapter use a `taglib` directive with a `tagdir` attribute to specify the directory that contains the tag files. While this is handy for custom actions implemented as tag files in an application you control, it's not as easy as one would want for deployment and use of the tag library in third-party applications. As you may recall from Chapter 7, it's very easy to deploy a tag library packaged as a JAR file; just put the JAR file in the `WEB-INF/lib` directory and use the default URI as the `uri` attribute value in the `taglib` directive.

You can do the same with a tag library developed as tag files, but in this case you must also create a Tag Library Descriptor (TLD) and include it in the JAR file. I described the purpose of the TLD briefly in Chapter 7, but let's take a closer look at it here. Example 11-8 shows the TLD for a tag library with some of the tag files we've developed in this chapter.

Example 11-8. TLD for tag files

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

Example 11-8. TLD for tag files (continued)

```
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
version="2.0">

<tlib-version>1.0</tlib-version>
<short-name>my</short-name>
<uri>mytaglib</uri>

<tag-file>
  <name>copyright</name>
  <path>/META-INF/tags/mytags/copyright.tag</path>
</tag-file>

<tag-file>
  <name>forEvenAndOdd</name>
  <path>/META-INF/tags/mytags/forEvenAndOdd.tag</path>
</tag-file>

<tag-file>
  <name>htmlFormat</name>
  <path>/META-INF/tags/mytags/htmlFormat.tag</path>
</tag-file>

</taglib>
```

As you can see, this file is an XML document. Don't worry about the `<taglib>` element attributes; just copy the element exactly as its shown here into your TLD. I describe it in more detail in Chapter 21, along with the TLD elements not covered here.

The first elements provide information about the tag library itself. The `<tlib-version>` element contains the version of this tag library and the `<short-name>` element contains the default namespace prefix for this library. An authoring tool may use the default namespace prefix when it generates the `taglib` directive and action elements, but a page author can pick different prefixes if needed, as described in Chapter 7.

The `<uri>` element is important. This element declares the default URI (identifier) for the library. The value you use for this element is the value that must be used as the `uri` attribute value for the `taglib` directive in the JSP pages to take advantage of the auto-deploy feature, as I described in Chapter 7.

Next comes a `<tag-file>` element for each tag file. The nested `<name>` element gives the name for the custom action. It's typically the same as the filename (minus the `.tag` extension) but you can specify a different name if you want. The `<path>` element holds the path within the JAR file to the tag file. It must start with `/META-INF/tags/`. The TLD file itself must also be located in the `/META-INF` directory in the JAR file, so you need to create a directory structure like this for the tag files and the TLD:

```
META-INF/
  mytags.tld
```

```
tags/  
  mytags/  
    copyright.tag  
    forEvenAndOdd.tag  
    htmlFormat.tag
```

Then create the JAR file with the *jar* command (included with the Java SDK) like this:

```
C:\> jar cvf mytags.jar META-INF
```

This creates a JAR file named *mytags.jar* containing the contents of the *META-INF* directory structure. You can now place this JAR file in the *WEB-INF/lib* directory of any web application that needs the library. When you restart the web container, it will locate the JAR file and its TLD, so that you can identify the tag library in a JSP page with a *taglib* directive like this:

```
<%@ taglib prefix="my" uri="mytags" %>
```

In other words, use the *uri* attribute with the default URI for the library (declared in the TLD), just as for the JSTL libraries we've used in previous chapters, instead of the *tagdir* attribute.

You can also use a TLD to identify tag files placed directly in the filesystem, i.e., not packaged in a JAR file. Doing this allows you to use the *uri* attribute instead of the *tagdir* attribute for the *taglib* directive, potentially saving you from changing the *taglib* directives in a number of JSP pages if you eventually decide to package the tag files in a JAR file for easier reuse in other applications. With this approach, you put the tag files under *WEB-INF/tags* (or a subdirectory) just as in the first examples in this chapter, but you also place a TLD in the *WEB-INF* directory (or a subdirectory, like *tlds*):

```
WEB-INF/  
  tlds/  
    mytags.tld  
tags/  
  mytags/  
    copyright.tag  
    forEvenAndOdd.tag  
    htmlFormat.tag
```

In this case, the *<path>* elements in the TLD must specify the context-relative path to the tag files in the filesystem:

```
<tag-file>  
  <name>copyright</name>  
  <path>/WEB-INF/tags/mytags/copyright.tag</path>  
</tag-file>
```

Other than that, the TLD is the same as in Example 11-7.