CREATING CUSTOM JSP TAG LIBRARIES

Topics in This Chapter

- Tag handler classes
- Tag library descriptor files
- The JSP `taglib` directive
- Simple tags
- Tags that use attributes
- Tags that use the body content between their start and end tags
- Tags that modify their body content
- Looping tags
- Nested tags

Chapter 14

JSP 1.1 introduced an extremely valuable new capability: the ability to define your own JSP tags. You define how the tag, its attributes, and its body are interpreted, then group your tags into collections called *tag libraries* that can be used in any number of JSP files. The ability to define tag libraries in this way permits Java developers to boil down complex server-side behaviors into simple and easy-to-use elements that content developers can easily incorporate into their JSP pages.

Custom tags accomplish some of the same goals as beans that are accessed with `jsp:useBean` (see Chapter 13, "Using JavaBeans with JSP")—encapsulating complex behaviors into simple and accessible forms. There are several differences, however. First, beans cannot manipulate JSP content; custom tags can. Second, complex operations can be reduced to a significantly simpler form with custom tags than with beans. Third, custom tags require quite a bit more work to set up than do beans. Fourth, beans are often defined in one servlet and then used in a different servlet or JSP page (see Chapter 15, "Integrating Servlets and JSP"), whereas custom tags usually define more self-contained behavior. Finally, custom tags are available only in JSP 1.1, but beans can be used in both JSP 1.0 and 1.1.

At the time this book went to press, no official release of Tomcat 3.0 properly supported custom tags, so the examples in this chapter use the beta version of Tomcat 3.1. Other than the support for custom tags and a few efficiency improvements and minor bug fixes, there is little difference in the

behavior of the two versions. However, Tomcat 3.1 uses a slightly different directory structure, as summarized Table 14.1.

Table 14.1 Standard Tomcat Directories

	Tomcat 3.0	*Tomcat 3.1*
Location of `startup` and `shutdown` Scripts	*`install_dir`*	*`install_dir`*`/bin`
Standard Top-Level Directory for Servlets and Supporting Classes	*`install_dir`*`/webpages/WEB-INF/classes`	*`install_dir`*`/webapps/ROOT/WEB-INF/classes`
Standard Top-Level Directory for HTML and JSP Files	*`install_dir`*`/webpages`	*`install_dir`*`/webapps/ROOT`

14.1 The Components That Make Up a Tag Library

In order to use custom JSP tags, you need to define three separate components: the tag handler class that defines the tag's behavior, the tag library descriptor file that maps the XML element names to the tag implementations, and the JSP file that uses the tag library. The rest of this section gives an overview of each of these components and the following sections give details on how to build these components for various different styles of tags.

The Tag Handler Class

When defining a new tag, your first task is to define a Java class that tells the system what to do when it sees the tag. This class must implement the `javax.servlet.jsp.tagext.Tag` interface. This is usually accomplished by extending the `TagSupport` or `BodyTagSupport` class. Listing 14.1 is an example of a simple tag that just inserts "`Custom tag example (coreservlets.tags.ExampleTag)`" into the JSP page wherever the corresponding tag is used. Don't worry about understanding the exact behavior of this class; that will be made clear in the next section. For now, just note that it is in the

coreservlets.tags class and is called ExampleTag. Thus, with Tomcat 3.1, the class file would be in *install_dir*/webapps/ROOT/WEB-INF/classes/coreservlets/tags/ExampleTag.class.

Listing 14.1 ExampleTag.java

```
package coreservlets.tags;

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

/** Very simple JSP tag that just inserts a string
 *  ("Custom tag example...") into the output.
 *  The actual name of the tag is not defined here;
 *  that is given by the Tag Library Descriptor (TLD)
 *  file that is referenced by the taglib directive
 *  in the JSP file.
 */

public class ExampleTag extends TagSupport {
  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
      out.print("Custom tag example " +
                "(coreservlets.tags.ExampleTag)");
    } catch(IOException ioe) {
      System.out.println("Error in ExampleTag: " + ioe);
    }
    return(SKIP_BODY);
  }
}
```

The Tag Library Descriptor File

Once you have defined a tag handler, your next task is to identify the class to the server and to associate it with a particular XML tag name. This task is accomplished by means of a tag library descriptor file (in XML format) like the one shown in Listing 14.2. This file contains some fixed information, an arbitrary short name for your library, a short description, and a series of tag descriptions. The nonbold part of the listing is the same in virtually all tag library descriptors and can be copied verbatim from the source code archive at http://www.coreservlets.com/ or from the Tomcat 3.1 standard examples (*install_dir*/webapps/examples/WEB-INF/jsp).

The format of tag descriptions will be described in later sections. For now, just note that the `tag` element defines the main name of the tag (really tag suffix, as will be seen shortly) and identifies the class that handles the tag. Since the tag handler class is in the `coreservlets.tags` package, the fully qualified class name of `coreservlets.tags.ExampleTag` is used. Note that this is a class name, not a URL or relative path name. The class can be installed anywhere on the server that beans or other supporting classes can be put. With Tomcat 3.1, the standard base location is *`install_dir`*`/webapps/ROOT/WEB-INF/classes`, so `ExampleTag` would be in *`install_dir`*`/webapps/ROOT/WEB-INF/classes/coreservlets/tags`. Although it is always a good idea to put your servlet classes in packages, a surprising feature of Tomcat 3.1 is that tag handlers are *required* to be in packages.

Listing 14.2 `csajsp-taglib.tld`

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

<!-- a tag library descriptor -->

<taglib>
  <!-- after this the default space is
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    A tag library from Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <tag>
    <name>example</name>
    <tagclass>coreservlets.tags.ExampleTag</tagclass>
    <info>Simplest example: inserts one line of output</info>
    <bodycontent>EMPTY</bodycontent>
  </tag>
  <!-- Other tags defined later... -->

</taglib>
```

The JSP File

Once you have a tag handler implementation and a tag library description, you are ready to write a JSP file that makes use of the tag. Listing 14.3 gives an example. Somewhere before the first use of your tag, you need to use the `taglib` directive. This directive has the following form:

```
<%@ taglib uri="..." prefix="..." %>
```

The required `uri` attribute can be either an absolute or relative URL referring to a tag library descriptor file like the one shown in Listing 14.2. To complicate matters a little, however, Tomcat 3.1 uses a `web.xml` file that maps an absolute URL for a tag library descriptor to a file on the local system. I don't recommend that you use this approach, but you should be aware of it in case you look at the Apache examples and wonder why it works when they specify a nonexistent URL for the `uri` attribute of the `taglib` directive.

The `prefix` attribute, also required, specifies a prefix that will be used in front of whatever tag name the tag library descriptor defined. For example, if the TLD file defines a tag named `tag1` and the `prefix` attribute has a value of `test`, the actual tag name would be `test:tag1`. This tag could be used in either of the following two ways, depending on whether it is defined to be a container that makes use of the tag body:

```
<test:tag1>
Arbitrary JSP
</test:tag1>
```

or just

```
<test:tag1 />
```

To illustrate, the descriptor file of Listing 14.2 is called `csajsp-taglib.tld`, and resides in the same directory as the JSP file shown in Listing 14.3. Thus, the `taglib` directive in the JSP file uses a simple relative URL giving just the filename, as shown below.

```
<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>
```

Furthermore, since the `prefix` attribute is `csajsp` (for *Core Servlets and JavaServer Pages*), the rest of the JSP page uses `csajsp:example` to refer to the `example` tag defined in the descriptor file. Figure 14–1 shows the result.

Listing 14.3 SimpleExample.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>

<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<TITLE><csajsp:example /></TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1><csajsp:example /></H1>
<csajsp:example />

</BODY>
</HTML>
```

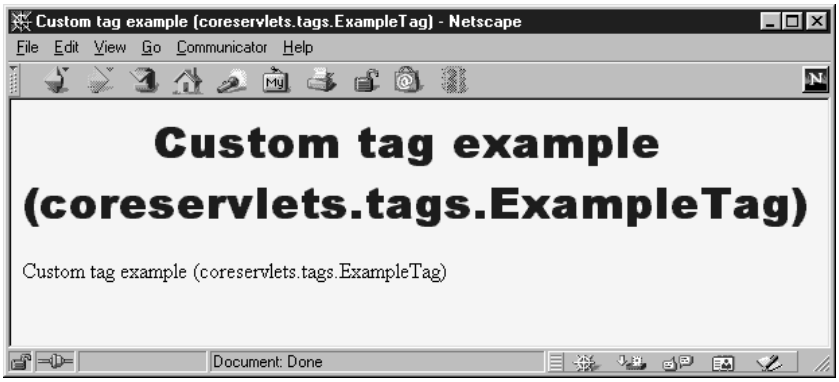

Figure 14–1 Result of SimpleExample.jsp.

14.2 Defining a Basic Tag

This section gives details on defining simple tags without attributes or tag bodies; the tags are thus of the form <prefix:tagname />.

The Tag Handler Class

Tags that either have no body or that merely include the body verbatim should extend the `TagSupport` class. This is a built-in class in the `javax.servlet.jsp.tagext` package that implements the `Tag` interface and contains much of the standard functionality basic tags need. Because of other classes you will use, your tag should normally import classes in the `javax.servlet.jsp` and `java.io` packages as well. So, most tag implementations contain the following `import` statements after the package declaration:

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

I recommend that you download an example from `http://www.coreservlets.com/` and use it as the starting point for your own implementations.

For a tag without attributes or body, all you need to do is override the `doStartTag` method, which defines code that gets called *at request time* where the element's start tag is found. To generate output, the method should obtain the `JspWriter` (the specialized `PrintWriter` available in JSP pages through use of the predefined `out` variable) from the `pageContext` field by means of `getOut`. In addition to the `getOut` method, the `pageContext` field (of type `PageContext`) has methods for obtaining other data structures associated with the request. The most important ones are `getRequest`, `getResponse`, `getServletContext`, and `getSession`.

Since the `print` method of `JspWriter` throws `IOException`, the `print` statements should be inside a `try`/`catch` block. To report other types of errors to the client, you can declare that your `doStartTag` method throws a `JspException` and then throw one when the error occurs.

If your tag does not have a body, your `doStartTag` should return the `SKIP_BODY` constant. This instructs the system to ignore any content between the tag's start and end tags. As we will see in Section 14.5 (Optionally Including the Tag Body), `SKIP_BODY` is sometimes useful even when there is a tag body, but the simple tag we're developing here will be used as a stand-alone tag (`<prefix:tagname />`) and thus does not have body content.

Listing 14.4 shows a tag implementation that uses this approach to generate a random 50-digit prime through use of the `Primes` class developed in Chapter 7 (Generating the Server Response: HTTP Response Headers) — see Listing 7.4.

Listing 14.4 SimplePrimeTag.java

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.math.*;
import coreservlets.*;

/** Generates a prime of approximately 50 digits.
 *  (50 is actually the length of the random number
 *  generated -- the first prime above that number will
 *  be returned.)
 */

public class SimplePrimeTag extends TagSupport {
  protected int len = 50;

  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
      BigInteger prime = Primes.nextPrime(Primes.random(len));
      out.print(prime);
    } catch(IOException ioe) {
      System.out.println("Error generating prime: " + ioe);
    }
    return(SKIP_BODY);
  }
}
```

The Tag Library Descriptor File

The general format of a descriptor file is almost always the same: it should contain an XML version identifier followed by a DOCTYPE declaration followed by a taglib container element. To get started, just download a sample from http://www.coreservlets.com/. The important part to understand is what goes *in* the taglib element: the tag element. For tags without attributes, the tag element should contain four elements between <tag> and </tag>:

1. **name**, whose body defines the base tag name to which the prefix of the taglib directive will be attached. In this case, I use
   <name>simplePrime</name>
   to assign a base tag name of simplePrime.

2. **`tagclass`**, which gives the fully qualified class name of the tag handler. In this case, I use
   `<tagclass>coreservlets.tags.SimplePrimeTag</tagclass>`
3. **`info`**, which gives a short description. Here, I use
   `<info>Outputs a random 50-digit prime.</info>`
4. **`bodycontent`**, which should have the value `EMPTY` for tags without bodies. Tags with normal bodies that might be interpreted as normal JSP use a value of `JSP`, and the rare tags whose handlers completely process the body themselves use a value of `TAGDEPENDENT`. For the `SimplePrimeTag` discussed here, I use `EMPTY` as below:
   `<bodycontent>EMPTY</bodycontent>`

Listing 14.5 shows the full TLD file.

Listing 14.5 `csajsp-taglib.tld`

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

<!-- a tag library descriptor -->

<taglib>
  <!-- after this the default space is
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    A tag library from Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- Other tags defined earlier... -->
```

Listing 14.5 csajsp-taglib.tld (continued)

```
  <tag>
    <name>simplePrime</name>
    <tagclass>coreservlets.tags.SimplePrimeTag</tagclass>
    <info>Outputs a random 50-digit prime.</info>
    <bodycontent>EMPTY</bodycontent>
  </tag>

</taglib>
```

The JSP File

JSP documents that make use of custom tags need to use the `taglib` directive, supplying a `uri` attribute that gives the location of the tag library descriptor file and a `prefix` attribute that specifies a short string that will be attached (along with a colon) to the main tag name. Listing 14.6 shows a JSP document that uses

```
<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>
```

to use the TLD file just shown in Listing 14.5 with a prefix of `csajsp`. Since the base tag name is `simplePrime`, the full tag used is

```
<csajsp:simplePrime />
```

Figure 14–2 shows the result.

Listing 14.6 SimplePrimeExample.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some 50-Digit Primes</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1>Some 50-Digit Primes</H1>

<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<UL>
  <LI><csajsp:simplePrime />
  <LI><csajsp:simplePrime />
  <LI><csajsp:simplePrime />
  <LI><csajsp:simplePrime />
</UL>

</BODY>
</HTML>
```

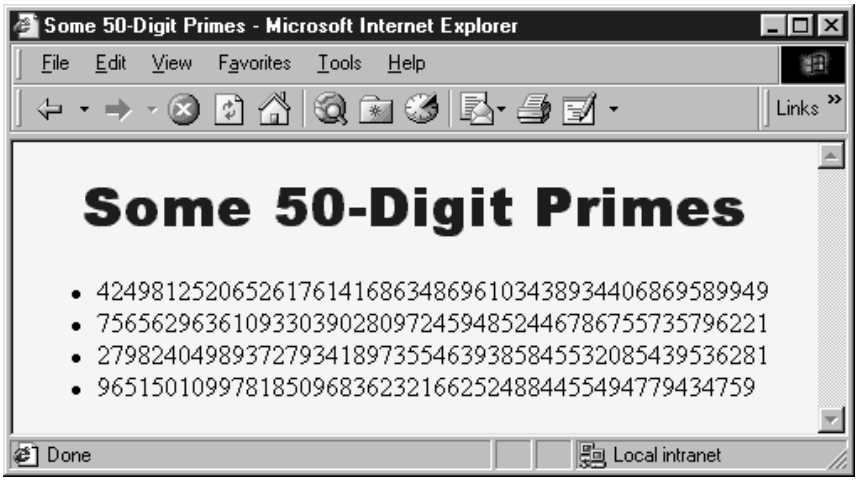

Figure 14–2 Result of `SimplePrimeExample.jsp`.

14.3 Assigning Attributes to Tags

Allowing tags like

```
<prefix:name attribute1="value1" attribute2="value2" ... />
```

adds significant flexibility to your tag library. This section explains how to add attribute support to your tags.

The Tag Handler Class

Providing support for attributes is straightforward. Use of an attribute called `attribute1` simply results in a call to a method called `setAttribute1` in your class that extends `TagSupport` (or otherwise implements the `Tag` interface). The attribute value is supplied to the method as a `String`. Consequently, adding support for an attribute named `attribute1` is merely a matter of implementing the following method:

```
public void setAttribute1(String value1) {
  doSomethingWith(value1);
}
```

Note that an attribute of `attributeName` (lowercase `a`) corresponds to a method called `setAttributeName` (uppercase `A`).

One of the most common things to do in the attribute handler is to simply store the attribute in a field that will later be used by `doStartTag` or a similar method. For example, following is a section of a tag implementation that adds support for the `message` attribute.

```
private String message = "Default Message";

public void setMessage(String message) {
  this.message = message;
}
```

If the tag handler will be accessed from other classes, it is a good idea to provide a `getAttributeName` method in addition to the `setAttributeName` method. Only `setAttributeName` is required, however.

Listing 14.7 shows a subclass of `SimplePrimeTag` that adds support for the `length` attribute. When such an attribute is supplied, it results in a call to `setLength`, which converts the input `String` to an `int` and stores it in the `len` field already used by the `doStartTag` method in the parent class.

Listing 14.7 `PrimeTag.java`

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.math.*;
import coreservlets.*;

/** Generates an N-digit random prime (default N = 50).
 *  Extends SimplePrimeTag, adding a length attribute
 *  to set the size of the prime. The doStartTag
 *  method of the parent class uses the len field
 *  to determine the approximate length of the prime.
 */

public class PrimeTag extends SimplePrimeTag {
  public void setLength(String length) {
    try {
      len = Integer.parseInt(length);
    } catch(NumberFormatException nfe) {
      len = 50;
    }
  }
}
```

The Tag Library Descriptor File

Tag attributes must be declared inside the `tag` element by means of an `attribute` element. The `attribute` element has three nested elements that can appear between `<attribute>` and `</attribute>`.

1. **`name`**, a required element that defines the case-sensitive attribute name. In this case, I use `<name>length</name>`
2. **`required`**, a required element that stipulates whether the attribute must always be supplied (`true`) or is optional (`false`). In this case, to indicate that `length` is optional, I use `<required>false</required>` If you omit the attribute, no call is made to the `setAttributeName` method. So, be sure to give default values to the fields that the method sets.
3. **`rtexprvalue`**, an optional attribute that indicates whether the attribute value can be a JSP expression like `<%= expression %>` (`true`) or whether it must be a fixed string (`false`). The default value is `false`, so this element is usually omitted except when you want to allow attributes to have values determined at request time.

Listing 14.8 shows the complete `tag` element within the tag library descriptor file. In addition to supplying an `attribute` element to describe the `length` attribute, the `tag` element also contains the standard `name` (`prime`), `tagclass` (`coreservlets.tags.PrimeTag`), `info` (short description), and `bodycontent` (`EMPTY`) elements.

Listing 14.8 `csajsp-taglib.tld`

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

<!-- a tag library descriptor -->

<taglib>
  <!-- after this the default space is
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->
```

Listing 14.8 csajsp-taglib.tld (continued)

```
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    A tag library from Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- Other tag defined earlier... -->

  <tag>
    <name>prime</name>
    <tagclass>coreservlets.tags.PrimeTag</tagclass>
    <info>Outputs a random N-digit prime.</info>
    <bodycontent>EMPTY</bodycontent>
    <attribute>
      <name>length</name>
      <required>false</required>
    </attribute>
  </tag>

</taglib>
```

The JSP File

Listing 14.9 shows a JSP document that uses the `taglib` directive to load the tag library descriptor file and to specify a prefix of `csajsp`. Since the `prime` tag is defined to permit a `length` attribute, Listing 14.9 uses

```
<csajsp:prime length="xxx" />
```

Remember that custom tags follow XML syntax, which requires attribute values to be enclosed in either single or double quotes. Also, since the `length` attribute is not required, it is permissible to use

```
<csajsp:prime />
```

The tag handler is responsible for using a reasonable default value in such a case.

Figure 14–3 shows the result of Listing 14.9.

Listing 14.9 `PrimeExample.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some N-Digit Primes</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1>Some N-Digit Primes</H1>

<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<UL>
  <LI>20-digit: <csajsp:prime length="20" />
  <LI>40-digit: <csajsp:prime length="40" />
  <LI>80-digit: <csajsp:prime length="80" />
  <LI>Default (50-digit): <csajsp:prime />
</UL>

</BODY>
</HTML>
```

Figure 14–3 Result of `PrimeExample.jsp`.

14.4 Including the Tag Body

Up to this point, all of the custom tags you have seen ignore the tag body and thus are used as stand-alone tags of the form

```
<prefix:tagname />
```

In this section, we see how to define tags that use their body content, and are thus used in the following manner:

```
<prefix:tagname>body</prefix:tagname>
```

The Tag Handler Class

In the previous examples, the tag handlers defined a `doStartTag` method that returned `SKIP_BODY`. To instruct the system to make use of the body that occurs between the new element's start and end tags, your `doStartTag` method should return `EVAL_BODY_INCLUDE` instead. The body content can contain JSP scripting elements, directives, and actions, just like the rest of the page. The JSP constructs are translated into servlet code at page translation time, and that code is invoked at request time.

If you make use of a tag body, then you might want to take some action *after* the body as well as before it. Use the `doEndTag` method to specify this action. In almost all cases, you want to continue with the rest of the page after finishing with your tag, so the `doEndTag` method should return `EVAL_PAGE`. If you want to abort the processing of the rest of the page, you can return `SKIP_PAGE` instead.

Listing 14.10 defines a tag for a heading element that is more flexible than the standard HTML `H1` through `H6` elements. This new element allows a precise font size, a list of preferred font names (the first entry that is available on the client system will be used), a foreground color, a background color, a border, and an alignment (`LEFT`, `CENTER`, `RIGHT`). Only the alignment capability is available with the `H1` through `H6` elements. The heading is implemented through use of a one-cell table enclosing a `SPAN` element that has embedded style sheet attributes. The `doStartTag` method generates the `TABLE` and `SPAN` start tags, then returns `EVAL_BODY_INCLUDE` to instruct the system to include the tag body. The `doEndTag` method generates the `</SPAN>` and `</TABLE>` tags, then returns `EVAL_PAGE` to continue with normal page processing. Various `setAttributeName` methods are used to handle the attributes like `bgColor` and `fontSize`.

Listing 14.10 HeadingTag.java

```
package coreservlets.tags;

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

/** Generates an HTML heading with the specified background
 *  color, foreground color, alignment, font, and font size.
 *  You can also turn on a border around it, which normally
 *  just barely encloses the heading, but which can also
 *  stretch wider. All attributes except the background
 *  color are optional.
 */

public class HeadingTag extends TagSupport {
  private String bgColor; // The one required attribute
  private String color = null;
  private String align="CENTER";
  private String fontSize="36";
  private String fontList="Arial, Helvetica, sans-serif";
  private String border="0";
  private String width=null;

  public void setBgColor(String bgColor) {
    this.bgColor = bgColor;
  }

  public void setColor(String color) {
    this.color = color;
  }

  public void setAlign(String align) {
    this.align = align;
  }

  public void setFontSize(String fontSize) {
    this.fontSize = fontSize;
  }

  public void setFontList(String fontList) {
    this.fontList = fontList;
  }

  public void setBorder(String border) {
    this.border = border;
  }

  public void setWidth(String width) {
    this.width = width;
  }
```

Listing 14.10 `HeadingTag.java` (continued)

```
  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
      out.print("<TABLE BORDER=" + border +
                " BGCOLOR=\"" + bgColor + "\"" +
                " ALIGN=\"" + align + "\"");
      if (width != null) {
        out.print(" WIDTH=\"" + width + "\"");
      }
      out.print("><TR><TH>");
      out.print("<SPAN STYLE=\"" +
                "font-size: " + fontSize + "px; " +
                "font-family: " + fontList + "; ");
      if (color != null) {
        out.println("color: " + color + ";");
      }
      out.print("\"> "); // End of <SPAN ...>
    } catch(IOException ioe) {
      System.out.println("Error in HeadingTag: " + ioe);
    }
    return(EVAL_BODY_INCLUDE); // Include tag body
  }

  public int doEndTag() {
    try {
      JspWriter out = pageContext.getOut();
      out.print("</SPAN></TABLE>");
    } catch(IOException ioe) {
      System.out.println("Error in HeadingTag: " + ioe);
    }
    return(EVAL_PAGE); // Continue with rest of JSP page
  }
}
```

The Tag Library Descriptor File

There is only one new feature in the use of the `tag` element for tags that use body content: the `bodycontent` element should contain the value `JSP` as below.

```
<bodycontent>JSP</bodycontent>
```

The `name`, `tagclass`, `info`, and `attribute` elements are used in the same manner as described previously. Listing 14.11 gives the code.

Listing 14.11 csajsp-taglib.tld

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

<!-- a tag library descriptor -->

<taglib>
  <!-- after this the default space is
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    A tag library from Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- Other tags defined earlier... -->

  <tag>
    <name>heading</name>
    <tagclass>coreservlets.tags.HeadingTag</tagclass>
    <info>Outputs a 1-cell table used as a heading.</info>
    <bodycontent>JSP</bodycontent>
    <attribute>
      <name>bgColor</name>
      <required>true</required> <!-- bgColor is required -->
    </attribute>
    <attribute>
      <name>color</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>align</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>fontSize</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>fontList</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>border</name>
      <required>false</required>
    </attribute>
```

Listing 14.11 csajsp-taglib.tld (continued)

```
    <attribute>
      <name>width</name>
      <required>false</required>
    </attribute>
  </tag>

</taglib>
```

The JSP File

Listing 14.12 shows a document that uses the heading tag just defined. Since the bgColor attribute was defined to be required, all uses of the tag include it. Figure 14–4 shows the result.

Listing 14.12 HeadingExample.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some Tag-Generated Headings</TITLE>
</HEAD>

<BODY>
<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<csajsp:heading bgColor="#C0C0C0">
Default Heading
</csajsp:heading>
<P>
<csajsp:heading bgColor="BLACK" color="WHITE">
White on Black Heading
</csajsp:heading>
<P>
<csajsp:heading bgColor="#EF8429" fontSize="60" border="5">
Large Bordered Heading
</csajsp:heading>
<P>
<csajsp:heading bgColor="CYAN" width="100%">
Heading with Full-Width Background
</csajsp:heading>
<P>
<csajsp:heading bgColor="CYAN" fontSize="60"
                fontList="Brush Script MT, Times, serif">
Heading with Non-Standard Font
</csajsp:heading>
<P>
</BODY>
</HTML>
```

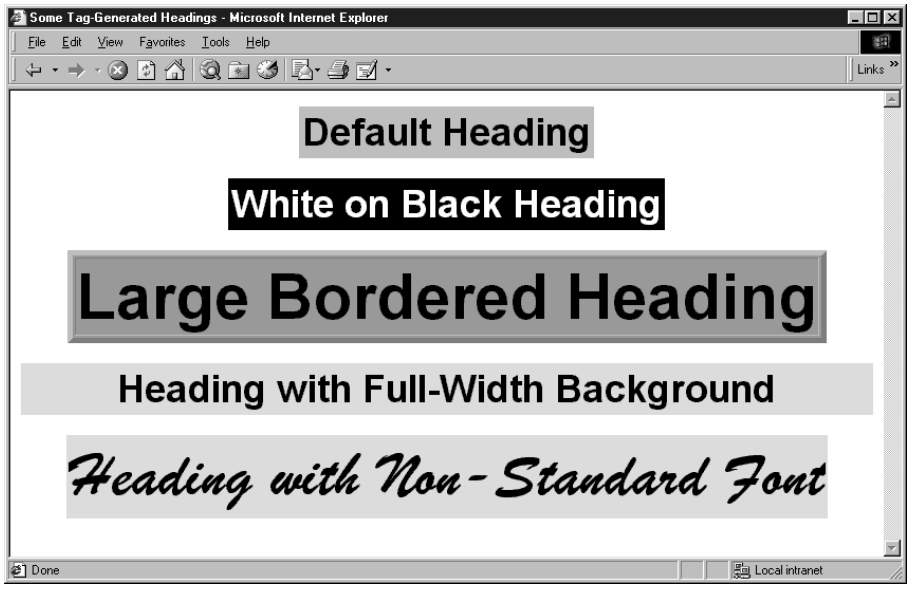

Figure 14–4 The custom `csajsp:heading` element gives you much more control over heading format than does the standard `H1` through `H6` elements in HTML.

14.5 Optionally Including the Tag Body

Most tags either *never* make use of body content or *always* do so. This section shows you how to use request time information to decide whether or not to include the tag body. Although the body can contain JSP that is interpreted at page translation time, the result of that translation is servlet code that can be invoked or ignored at request time.

The Tag Handler Class

Optionally including the tag body is a trivial exercise: just return `EVAL_BODY_INCLUDE` or `SKIP_BODY` depending on the value of some request time expression. The important thing to know is how to discover that request time information, since `doStartTag` does not have `HttpServletRequest` and `HttpServletResponse` arguments as do `service`,

_jspService, doGet, and doPost. The solution to this dilemma is to use getRequest to obtain the HttpServletRequest from the automatically defined pageContext field of TagSupport. Strictly speaking, the return type of getRequest is ServletRequest, so you have to do a typecast to HttpServletRequest if you want to call a method that is not inherited from ServletRequest. However, in this case I just use getParameter, so no typecast is required.

Listing 14.13 defines a tag that ignores its body unless a request time debug parameter is supplied. Such a tag provides a useful capability whereby you embed debugging information directly in the JSP page during development, but activate it only when a problem occurs.

Listing 14.13 DebugTag.java

```
package coreservlets.tags;

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

/** A tag that includes the body content only if
 *  the "debug" request parameter is set.
 */

public class DebugTag extends TagSupport {
  public int doStartTag() {
    ServletRequest request = pageContext.getRequest();
    String debugFlag = request.getParameter("debug");
    if ((debugFlag != null) &&
        (!debugFlag.equalsIgnoreCase("false"))) {
      return(EVAL_BODY_INCLUDE);
    } else {
      return(SKIP_BODY);
    }
  }
}
```

The Tag Library Descriptor File

If your tag *ever* makes use of its body, you must provide the value `JSP` inside the `bodycontent` element. Other than that, all the elements within `tag` are used in the same way as described previously. Listing 14.14 shows the entries needed for `DebugTag`.

Listing 14.14 `csajsp-taglib.tld`

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

<!-- a tag library descriptor -->

<taglib>
  <!-- after this the default space is
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    A tag library from Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- Other tags defined earlier... -->

  <tag>
    <name>debug</name>
    <tagclass>coreservlets.tags.DebugTag</tagclass>
    <info>Includes body only if debug param is set.</info>
    <bodycontent>JSP</bodycontent>
  </tag>

</taglib>
```

The JSP File

Listing 14.15 shows a page that encloses debugging information between <csajsp:debug> and </csajsp:debug>. Figures 14–5 and 14–6 show the normal result and the result when a request time debug parameter is supplied, respectively.

Listing 14.15 DebugExample.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Using the Debug Tag</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1>Using the Debug Tag</H1>

<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

Top of regular page. Blah, blah, blah. Yadda, yadda, yadda.
<P>

<csajsp:debug>
<B>Debug:</B>
<UL>
  <LI>Current time: <%= new java.util.Date() %>
  <LI>Requesting hostname: <%= request.getRemoteHost() %>
  <LI>Session ID: <%= session.getId() %>
</UL>
</csajsp:debug>

<P>
Bottom of regular page. Blah, blah, blah. Yadda, yadda, yadda.

</BODY>
</HTML>
```

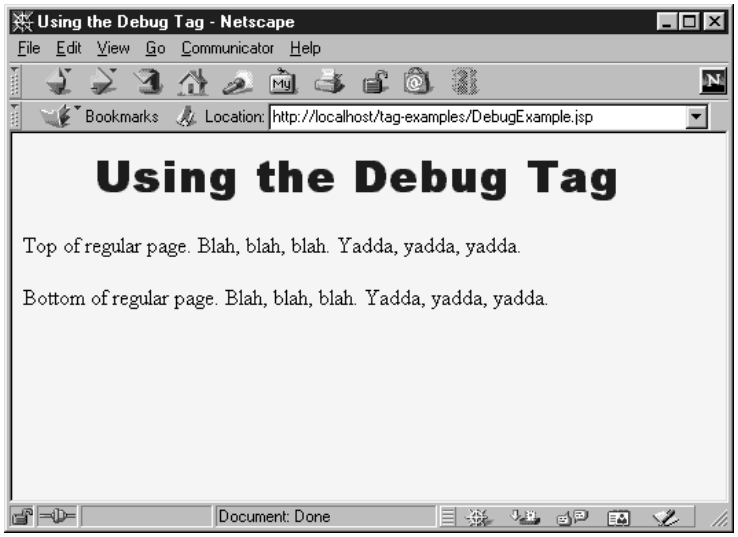

Figure 14–5 The body of the `csajsp:debug` element is normally ignored.

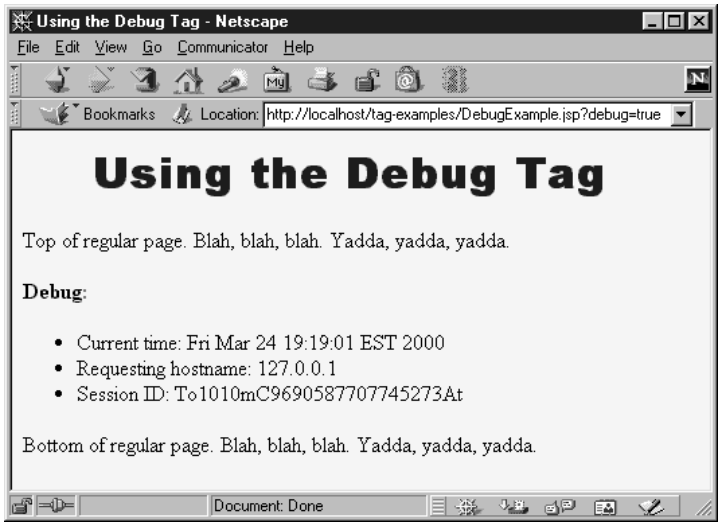

Figure 14–6 The body of the `csajsp:debug` element is included when a `debug` request parameter is supplied.

14.6 Manipulating the Tag Body

The `csajsp:prime` element (Section 14.3) ignored any body content, the `csajsp:heading` element (Section 14.4) used body content, and the `csajsp:debug` element (Section 14.5) ignored or used it depending on a request time parameter. The common thread among these elements is that the tag body was never modified; it was either ignored or included verbatim (after JSP translation). This section shows you how to process the tag body.

The Tag Handler Class

Up to this point, all of the tag handlers have extended the `TagSupport` class. This is a good standard starting point, as it implements the required `Tag` interface and performs a number of useful setup operations like storing the `PageContext` reference in the `pageContext` field. However, `TagSupport` is not powerful enough for tag implementations that need to manipulate their body content, and `BodyTagSupport` should be used instead.

`BodyTagSupport` extends `TagSupport`, so the `doStartTag` and `doEndTag` methods are used in the same way as before. The two important new methods defined by `BodyTagSupport` are:

1. **`doAfterBody`**, a method that you should override to handle the manipulation of the tag body. This method should normally return `SKIP_BODY` when it is done, indicating that no further body processing should be performed.
2. **`getBodyContent`**, a method that returns an object of type `BodyContent` that encapsulates information about the tag body.

The `BodyContent` class has three important methods:

1. **`getEnclosingWriter`**, a method that returns the `JspWriter` being used by `doStartTag` and `doEndTag`.
2. **`getReader`**, a method that returns a `Reader` that can read the tag's body.
3. **`getString`**, a method that returns a `String` containing the entire tag body.

In Section 3.4 (Example: Reading All Parameters), we saw a static `filter` method that would take a string and replace <, >, ", and & with `<`, `>`,

", and &, respectively. This method is useful when servlets output strings that might contain characters that would interfere with the HTML structure of the page in which the strings are embedded. Listing 14.16 shows a tag implementation that gives this filtering functionality to a custom JSP tag.

Listing 14.16 FilterTag.java

```
package coreservlets.tags;

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

/** A tag that replaces <, >, ", and & with their HTML
 *  character entities (&lt;, &gt;, &quot;, and &amp;).
 *  After filtering, arbitrary strings can be placed
 *  in either the page body or in HTML attributes.
 */

public class FilterTag extends BodyTagSupport {
  public int doAfterBody() {
    BodyContent body = getBodyContent();
    String filteredBody =
      ServletUtilities.filter(body.getString());
    try {
      JspWriter out = body.getEnclosingWriter();
      out.print(filteredBody);
    } catch(IOException ioe) {
      System.out.println("Error in FilterTag: " + ioe);
    }
    // SKIP_BODY means I'm done. If I wanted to evaluate
    // and handle the body again, I'd return EVAL_BODY_TAG.
    return(SKIP_BODY);
  }
}
```

The Tag Library Descriptor File

Tags that manipulate their body content should use the bodycontent element the same way as tags that simply include it verbatim; they should supply a value of JSP. Other than that, nothing new is required in the descriptor file, as you can see by examining Listing 14.17.

Listing 14.17 csajsp-taglib.tld

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

<!-- a tag library descriptor -->

<taglib>
  <!-- after this the default space is
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    A tag library from Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- Other tags defined earlier... -->

  <tag>
    <name>filter</name>
    <tagclass>coreservlets.tags.FilterTag</tagclass>
    <info>Replaces HTML-specific characters in body.</info>
    <bodycontent>JSP</bodycontent>
  </tag>

</taglib>
```

The JSP File

Listing 14.18 shows a page that uses a table to show some sample HTML and its result. Creating this table would be tedious in regular HTML since the table cell that shows the original HTML would have to change all the < and > characters to `<` and `>`. Doing so is particularly onerous during development when the sample HTML is frequently changing. Use of the `<csajsp:filter>` tag greatly simplifies the process, as Listing 14.18 illustrates. Figure 14–7 shows the result.

Listing 14.18 FilterExample.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>HTML Logical Character Styles</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1>HTML Logical Character Styles</H1>
Physical character styles (B, I, etc.) are rendered consistently
in different browsers. Logical character styles, however,
may be rendered differently by different browsers.
Here's how your browser
(<%= request.getHeader("User-Agent") %>)
renders the HTML 4.0 logical character styles:
<P>

<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<TABLE BORDER=1 ALIGN="CENTER">
<TR CLASS="COLORED"><TH>Example<TH>Result
<TR>

<TD><PRE><csajsp:filter>
<EM>Some emphasized text.</EM><BR>
<STRONG>Some strongly emphasized text.</STRONG><BR>
<CODE>Some code.</CODE><BR>
<SAMP>Some sample text.</SAMP><BR>
<KBD>Some keyboard text.</KBD><BR>
<DFN>A term being defined.</DFN><BR>
<VAR>A variable.</VAR><BR>
<CITE>A citation or reference.</CITE>
</csajsp:filter></PRE>

<TD>
<EM>Some emphasized text.</EM><BR>
<STRONG>Some strongly emphasized text.</STRONG><BR>
<CODE>Some code.</CODE><BR>
<SAMP>Some sample text.</SAMP><BR>
<KBD>Some keyboard text.</KBD><BR>
<DFN>A term being defined.</DFN><BR>
<VAR>A variable.</VAR><BR>
<CITE>A citation or reference.</CITE>

</TABLE>
</BODY>
</HTML>
```

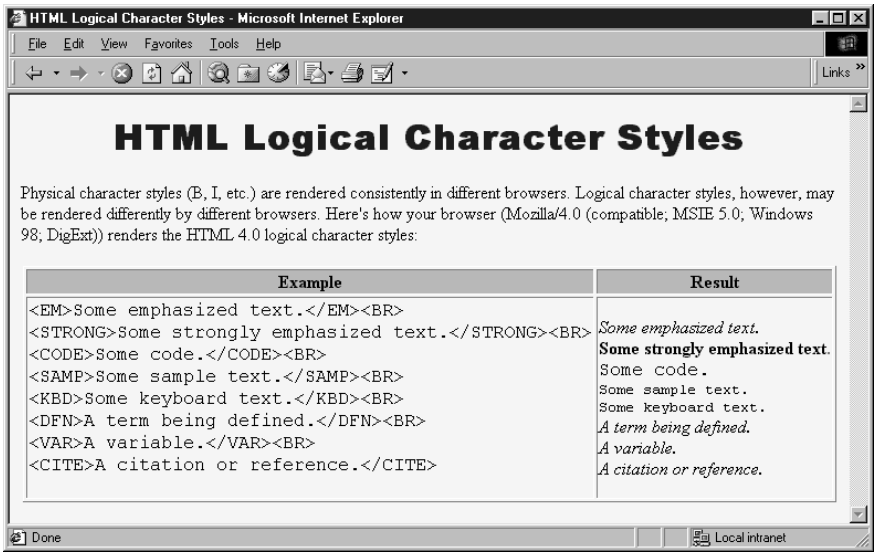

Figure 14–7 The `csajsp:filter` element lets you insert text without worrying about it containing special HTML characters.

14.7 Including or Manipulating the Tag Body Multiple Times

Rather than just including or processing the body of the tag a single time, you sometimes want to do so more than once. The ability to support multiple body inclusion lets you define a variety of iteration tags that repeat JSP fragments a variable number of times, repeat them until a certain condition occurs, and so forth. This section shows you how to build such tags.

The Tag Handler Class

Tags that process the body content multiple times should start by extending `BodyTagSupport` and implementing `doStartTag`, `doEndTag`, and, most importantly, `doAfterBody` as before. The difference lies in the return value of `doAfterBody`. If this method returns `EVAL_BODY_TAG`, the tag body is evaluated again, resulting in a new call to `doAfterBody`. This process continues until `doAfterBody` returns `SKIP_BODY`.

Listing 14.19 defines a tag that repeats the body content the number of times specified by the `reps` attribute. Since the body content can contain JSP (which gets made into servlet code at page translation time but invoked at request time), each repetition does not necessarily result in the same output to the client.

Listing 14.19 `RepeatTag.java`

```
package coreservlets.tags;

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

/** A tag that repeats the body the specified
 *  number of times.
 */

public class RepeatTag extends BodyTagSupport {
  private int reps;

  public void setReps(String repeats) {
    try {
      reps = Integer.parseInt(repeats);
    } catch(NumberFormatException nfe) {
      reps = 1;
    }
  }

  public int doAfterBody() {
    if (reps-- >= 1) {
      BodyContent body = getBodyContent();
      try {
        JspWriter out = body.getEnclosingWriter();
        out.println(body.getString());
        body.clearBody(); // Clear for next evaluation
      } catch(IOException ioe) {
        System.out.println("Error in RepeatTag: " + ioe);
      }
      return(EVAL_BODY_TAG);
    } else {
      return(SKIP_BODY);
    }
  }
}
```

The Tag Library Descriptor File

Listing 14.20 shows a TLD file that gives the name `csajsp:repeat` to the tag just defined. To accommodate request time values in the `reps` attribute, the file uses an `rtexprvalue` element (enclosing a value of `true`) within the `attribute` element.

Listing 14.20 `csajsp-taglib.tld`

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

<!-- a tag library descriptor -->

<taglib>
  <!-- after this the default space is
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    A tag library from Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- Other tags defined earlier... -->

  <tag>
    <name>repeat</name>
    <tagclass>coreservlets.tags.RepeatTag</tagclass>
    <info>Repeats body the specified number of times.</info>
    <bodycontent>JSP</bodycontent>
    <attribute>
      <name>reps</name>
      <required>true</required>
      <!-- rtexprvalue indicates whether attribute
           can be a JSP expression. -->
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>

</taglib>
```

The JSP File

Listing 14.21 shows a JSP document that creates a numbered list of prime numbers. The number of primes in the list is taken from the request time `repeats` parameter. Figure 14–8 shows one possible result.

Listing 14.21 `RepeatExample.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some 40-Digit Primes</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1>Some 40-Digit Primes</H1>
Each entry in the following list is the first prime number
higher than a randomly selected 40-digit number.

<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<OL>
<!-- Repeats N times. A null reps value means repeat once. -->
<csajsp:repeat reps='<%= request.getParameter("repeats") %>'>
  <LI><csajsp:prime length="40" />
</csajsp:repeat>
</OL>

</BODY>
</HTML>
```

14.8 Using Nested Tags

Although Listing 14.21 places the `csajsp:prime` element within the `csajsp:repeat` element, the two elements are independent of each other. The first generates a prime number regardless of where it is used, and the second repeats the enclosed content regardless of whether that content uses a `csajsp:prime` element.

Some tags, however, depend on a particular nesting. For example, in standard HTML, the `TD` and `TH` elements can only appear within `TR`, which in turn

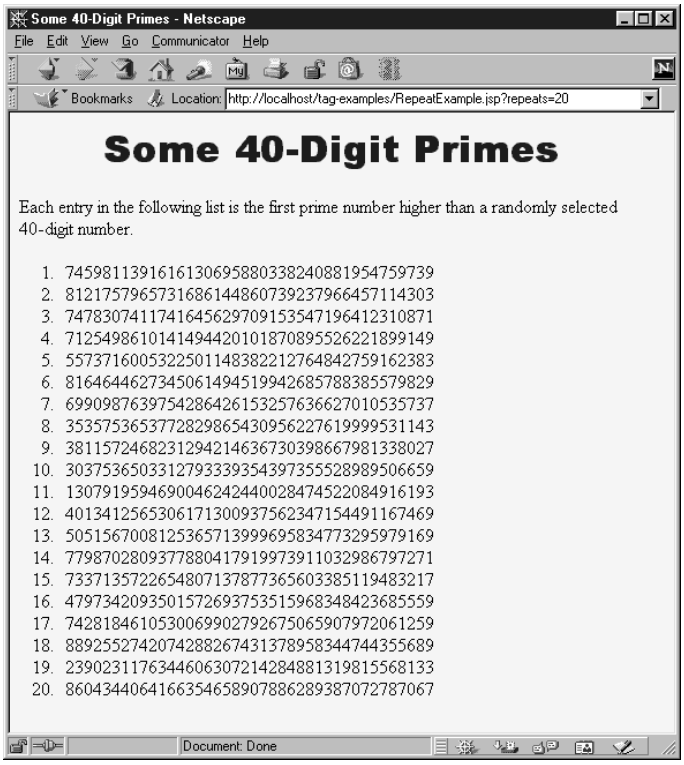

Figure 14–8 Result of `RepeatExample.jsp` when accessed with a `repeats` parameter of 20.

can only appear within `TABLE`. The color and alignment settings of `TABLE` are inherited by `TR`, and the values of `TR` affect how `TD` and `TH` behave. So, the nested elements cannot act in isolation even when nested properly. Similarly, the tag library descriptor file makes use of a number of elements like `taglib`, `tag`, `attribute` and `required` where a strict nesting hierarchy is imposed.

This section shows you how to define tags that depend on a particular nesting order and where the behavior of certain tags depends on values supplied by earlier ones.

The Tag Handler Classes

Class definitions for nested tags can extend *either* `TagSupport` or `BodyTagSupport`, depending on whether they need to manipulate their body content (these extend `BodyTagSupport`) or, more commonly, just ignore it or include it verbatim (these extend `TagSupport`).

There are two key new approaches for nested tags, however. First, nested tags can use `findAncestorWithClass` to find the tag in which they are nested. This method takes a reference to the current class (e.g., `this`) and the `Class` object of the enclosing class (e.g., `EnclosingTag.class`) as arguments. If no enclosing class is found, the method in the nested class can throw a `JspTagException` that reports the problem. Second, if one tag wants to store data that a later tag will use, it can place that data in the instance of the enclosing tag. The definition of the enclosing tag should provide methods for storing and accessing this data. Listing 14.22 outlines this approach.

Listing 14.22 Template for Nested Tags

```
public class OuterTag extends TagSupport {
  public void setSomeValue(SomeClass arg) { ... }
  public SomeClass getSomeValue() { ... }
}

public class FirstInnerTag extends BodyTagSupport {
  public int doStartTag() throws JspTagException {
    OuterTag parent =
      (OuterTag)findAncestorWithClass(this, OuterTag.class);
    if (parent == null) {
      throw new JspTagException("nesting error");
    } else {
      parent.setSomeValue(...);
    }
    return(EVAL_BODY_TAG);
  }
  ...
}

public class SecondInnerTag extends BodyTagSupport {
  public int doStartTag() throws JspTagException {
    OuterTag parent =
      (OuterTag)findAncestorWithClass(this, OuterTag.class);
    if (parent == null) {
      throw new JspTagException("nesting error");
    } else {
      SomeClass value = parent.getSomeValue();
      doSomethingWith(value);
    }
    return(EVAL_BODY_TAG);
  }
  ...
}
```

Now, suppose that we want to define a set of tags that would be used like this:

```
<csajsp:if>
  <csajsp:condition><%= someExpression %></csajsp:condition>
  <csajsp:then>JSP to include if condition is true</csajsp:then>
  <csajsp:else>JSP to include if condition is false</csajsp:else>
</csajsp:if>
```

To accomplish this task, the first step is to define an `IfTag` class to handle the `csajsp:if` tag. This handler should have methods to specify and check whether the condition is true or false (`setCondition` and `getCondition`) as well as methods to designate and check if the condition has ever been explicitly set (`setHasCondition` and `getHasCondition`), since we want to disallow `csajsp:if` tags that contain no `csajsp:condition` entry. Listing 14.23 shows the code for `IfTag`.

The second step is to define a tag handler for `csajsp:condition`. This class, called `IfConditionTag`, defines a `doStartTag` method that merely checks if the tag appears within `IfTag`. It returns `EVAL_BODY_TAG` if so and throws an exception if not. The handler's `doAfterBody` method looks up the body content (`getBodyContent`), converts it to a `String` (`getString`), and compares that to `"true"`. This approach means that an explicit value of `true` can be substituted for a JSP expression like `<%= expression %>` if, during initial page development, you want to temporarily designate that the `then` portion should always be used. Using a comparison to `"true"` also means that *any* other value will be considered "false." Once this comparison is performed, the result is stored in the enclosing tag by means of the `setCondition` method of `IfTag`. The code for `IfConditionTag` is shown in Listing 14.24.

The third step is to define a class to handle the `csajsp:then` tag. The `doStartTag` method of this class verifies that it is inside `IfTag` and also checks that an explicit condition has been set (i.e., that the `IfConditionTag` has already appeared within the `IfTag`). The `doAfterBody` method checks for the condition in the `IfTag` class, and, if it is true, looks up the body content and prints it. Listing 14.25 shows the code.

The final step in defining tag handlers is to define a class for `csajsp:else`. This class is very similar to the one to handle the `then` part of the tag, except that this handler only prints the tag body from `doAfterBody` if the condition from the surrounding `IfTag` is false. The code is shown in Listing 14.26.

Listing 14.23 `IfTag.java`

```
package coreservlets.tags;

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

/** A tag that acts like an if/then/else.
 */

public class IfTag extends TagSupport {
  private boolean condition;
  private boolean hasCondition = false;

  public void setCondition(boolean condition) {
    this.condition = condition;
    hasCondition = true;
  }

  public boolean getCondition() {
    return(condition);
  }

  public void setHasCondition(boolean flag) {
    this.hasCondition = flag;
  }

  /** Has the condition field been explicitly set? */

  public boolean hasCondition() {
    return(hasCondition);
  }

  public int doStartTag() {
    return(EVAL_BODY_INCLUDE);
  }
}
```

Listing 14.24 IfConditionTag.java

```
package coreservlets.tags;

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

/** The condition part of an if tag.
 */

public class IfConditionTag extends BodyTagSupport {
  public int doStartTag() throws JspTagException {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (parent == null) {
      throw new JspTagException("condition not inside if");
    }
    return(EVAL_BODY_TAG);
  }

  public int doAfterBody() {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    String bodyString = getBodyContent().getString();
    if (bodyString.trim().equals("true")) {
      parent.setCondition(true);
    } else {
      parent.setCondition(false);
    }
    return(SKIP_BODY);
  }
}
```

Listing 14.25 `IfThenTag.java`

```
package coreservlets.tags;

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

/** The then part of an if tag.
 */

public class IfThenTag extends BodyTagSupport {
  public int doStartTag() throws JspTagException {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (parent == null) {
      throw new JspTagException("then not inside if");
    } else if (!parent.hasCondition()) {
      String warning =
        "condition tag must come before then tag";
      throw new JspTagException(warning);
    }
    return(EVAL_BODY_TAG);
  }

  public int doAfterBody() {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (parent.getCondition()) {
      try {
        BodyContent body = getBodyContent();
        JspWriter out = body.getEnclosingWriter();
        out.print(body.getString());
      } catch(IOException ioe) {
        System.out.println("Error in IfThenTag: " + ioe);
      }
    }
    return(SKIP_BODY);
  }
}
```

Listing 14.26 IfElseTag.java

```
package coreservlets.tags;

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

/** The else part of an if tag.
 */

public class IfElseTag extends BodyTagSupport {
  public int doStartTag() throws JspTagException {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (parent == null) {
      throw new JspTagException("else not inside if");
    } else if (!parent.hasCondition()) {
      String warning =
        "condition tag must come before else tag";
      throw new JspTagException(warning);
    }
    return(EVAL_BODY_TAG);
  }

  public int doAfterBody() {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (!parent.getCondition()) {
      try {
        BodyContent body = getBodyContent();
        JspWriter out = body.getEnclosingWriter();
        out.print(body.getString());
      } catch(IOException ioe) {
        System.out.println("Error in IfElseTag: " + ioe);
      }
    }
    return(SKIP_BODY);
  }
}
```

The Tag Library Descriptor File

Even though there is an explicit required nesting structure for the tags just defined, the tags must be declared separately in the TLD file. This means that nesting validation is performed only at request time, not at page transla-

tion time. In principle, you could instruct the system to do some validation at page translation time by using a `TagExtraInfo` class. This class has a `getVariableInfo` method that you can use to check that attributes exist and where they are used. Once you have defined a subclass of `TagExtraInfo`, you associate it with your tag in the tag library descriptor file by means of the `teiclass` element, which is used just like `tagclass`. In practice, however, `TagExtraInfo` is poorly documented and cumbersome to use.

Listing 14.27 csajsp-taglib.tld

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

<!-- a tag library descriptor -->

<taglib>
  <!-- after this the default space is
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    A tag library from Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- Other tags defined earlier... -->

  <tag>
    <name>if</name>
    <tagclass>coreservlets.tags.IfTag</tagclass>
    <info>if/condition/then/else tag.</info>
    <bodycontent>JSP</bodycontent>
  </tag>

  <tag>
    <name>condition</name>
    <tagclass>coreservlets.tags.IfConditionTag</tagclass>
    <info>condition part of if/condition/then/else tag.</info>
    <bodycontent>JSP</bodycontent>
  </tag>
```

Listing 14.27 csajsp-taglib.tld (continued)

```
  <tag>
    <name>then</name>
    <tagclass>coreservlets.tags.IfThenTag</tagclass>
    <info>then part of if/condition/then/else tag.</info>
    <bodycontent>JSP</bodycontent>
  </tag>

  <tag>
    <name>else</name>
    <tagclass>coreservlets.tags.IfElseTag</tagclass>
    <info>else part of if/condition/then/else tag.</info>
    <bodycontent>JSP</bodycontent>
  </tag>

</taglib>
```

The JSP File

Listing 14.28 shows a page that uses the csajsp:if tag three different ways. In the first instance, a value of true is hardcoded for the condition. In the second instance, a parameter from the HTTP request is used for the condition, and in the third case, a random number is generated and compared to a fixed cutoff. Figure 14–9 shows a typical result.

Listing 14.28 IfExample.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>If Tag Example</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1>If Tag Example</H1>

<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<csajsp:if>
  <csajsp:condition>true</csajsp:condition>
  <csajsp:then>Condition was true</csajsp:then>
  <csajsp:else>Condition was false</csajsp:else>
</csajsp:if>
```

Listing 14.28 `IfExample.jsp` (continued)

```
<P>
<csajsp:if>
  <csajsp:condition><%= request.isSecure() %></csajsp:condition>
  <csajsp:then>Request is using SSL (https)</csajsp:then>
  <csajsp:else>Request is not using SSL</csajsp:else>
</csajsp:if>
<P>
Some coin tosses:<BR>
<csajsp:repeat reps="20">
  <csajsp:if>
    <csajsp:condition>
      <%= Math.random() > 0.5 %>
    </csajsp:condition>
    <csajsp:then><B>Heads</B><BR></csajsp:then>
    <csajsp:else><B>Tails</B><BR></csajsp:else>
  </csajsp:if>
</csajsp:repeat>

</BODY>
</HTML>
```

Figure 14–9 Result of `IfExample.jsp`.