Chapter 7

As discussed in Volume 1 (Section 11.2) of *Core Servlets and JavaServer Pages*, you have many options when it comes to generating dynamic content inside the JSP page. These options are as follows:

- Scripting elements calling servlet code directly
- Scripting elements calling servlet code indirectly (by means of utility classes)
- Beans
- Servlet/JSP combo (MVC)
- MVC with JSP expression language
- Custom tags

The options at the top of the list are much simpler to use and are just as legitimate as the options at the bottom of the list. However, industry has adopted a best practice to avoid placing Java code inside the JSP page. This best practice stems from it being much harder to debug and maintain Java code inside the JSP page. In addition, JSP pages should concentrate only on the presentation logic. Introducing Java code into the JSP page tends to divert its purpose and, inevitably, business logic starts to creep in. To enforce this best practice, version 2.4 of the servlet specification went so far as to provide a way to disable any type of JSP scripting for a group of JSP pages. We discuss how to disable scripting in Section 2.14 (Configuring JSP Pages).

That said, there are cases where the presentation logic itself is quite complex and using the non-Java code options in the JSP page to express that logic becomes either too clunky and unreadable or, sometimes, just impossible to achieve. This is where

logic through the familiar HTML-like structures.

This chapter discusses how to create and use custom tags utilizing the new `SimpleTag` API, which was introduced in version 2.4 of the servlet specification. As its name suggests, `SimpleTag` API is very easy to use in comparison to its predecessor, now known as the classic tag API.

Although the `SimpleTag` API completely replaces the classic tag API, you should keep in mind that it works only in containers compliant with servlet specification 2.4 and above. Because there are still a lot of applications running on servlet 2.3-compliant containers, you should consider avoiding the `SimpleTag` API if you are not sure what type of container your code will end up on.

7.1 Tag Library Components

To use custom JSP tags, you need to define three separate components:

- The tag handler class that defines the tag's behavior
- The TLD file that maps the XML element names to the tag implementations
- 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 styles of tags. Most people find that the first tag they write is the hardest—the difficulty being in knowing where each component should go, not in writing the components. So, we suggest that you start by just downloading the simplest of the examples of this chapter from http://volume2.coreservlets.com/ and getting those examples to work on your machine. After that, you can move on and try creating some of your own 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`SimpleTag` interface. In practice, you extend `SimpleTagSupport`, which implements the `SimpleTag` interface and supplies standard implementations for some of its methods. Both the `SimpleTag` interface and the `SimpleTagSupport` class reside in the `javax.servlet.jsp.tagext` package.

The very first action the container takes after loading the tag handler class is instantiating it with its no-arg constructor. This means that every tag handler must have a no-arg constructor or its instantiation will fail. Remember that the Java compiler provides one for you automatically unless you define a constructor with arguments. In that case, be sure to define a no-arg constructor yourself.

The code that does the actual work of the tag goes inside the `doTag` method. Usually, this code outputs content to the JSP page by invoking the `print` method of the `JspWriter` class. To obtain an instance of the `JstWriter` class you call `getJspContext().getOut()` inside the `doTag` method. The `doTag` method is called at request time. It's important to note that, unlike the classic tag model, the `SimpleTag` model never reuses tag handler instances. In fact, a new instance of the tag handler class is created for every tag occurrence on the page. This alleviates worries about race conditions and cached values even if you use instance variables in the tag handler class.

You place the compiled tag handler in the same location you would place a regular servlet, inside the `WEB-INF/classes` directory, keeping the package structure intact. For example, if your tag handler class belongs to the `mytags` package and its class name is `MyTag`, you would place the`MyTag.class` file inside the `WEB-INF/classes/mytags/` directory.

Listing 7.1 shows an example of a tag handler class.

Listing 7.1 Example Tag Handler Class

```
package somepackage;

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

public class ExampleTag extends SimpleTagSupport {
  public void doTag() throws JspException, IOException {
    JspWriter out = getJspContext().getOut();
    out.print("<b>Hello World!</b>");
  }
}
```

The Tag Library Descriptor File

Once you have defined a tag handler, your next task is to identify this class to the server and to associate it with a particular XML tag name. This task is accomplished by means of a TLD file in XML format. This file contains some fixed information (e.g., XML Schema instance declaration), an arbitrary short name for your library, a short description, and a series of tag descriptions. Listing 7.2 shows an example TLD file.

Listing 7.2 Example Tag Library Descriptor File

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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>csajsp-taglib</short-name>
  <tag>
    <description>Example tag</description>
    <name>example</name>
    <tag-class>package.TagHandlerClass</tag-class>
    <body-content>empty</body-content>
  </tag>
</taglib>
```

We describe the details of the contents of the TLD file in later sections. For now, just note that the `tag` element through the following subelements in their required order defines the custom tag.

- **`description`.** This optional element allows the tag developer to document the purpose of the custom tag.
- **`name`.** This required element defines the name of the tag as it will be referred to by the JSP page (really tag suffix, as will be seen shortly).
- **`tag-class`.** This required element identifies the fully qualified name of the implementing tag handler class.
- **`body-content`.** This required element tells the container how to treat the content between the beginning and ending occurrence of the tag, if any. The value that appears here can be either `empty`, `scriptless`, `tagdependent`, or `JSP`.

  The value of `empty` means that no content is allowed to appear in the body of the tag. This would mean that the declared tag can only appear in the form:

  ```
  <prefix:tag/>
  ```

  or

  ```
  <prefix:tag></prefix:tag>
  ```

  (without any spaces between the opening and closing tags). Placing any content inside the tag body would generate a page translation error.

The value of `scriptless` means that the tag body is allowed to have JSP content as long as it doesn't contain any scripting elements like `<% ... %>` or `<%= ... %>`. If present, the body of the tag would be processed just like any other JSP content.

The value of `tagdependent` means that the tag is allowed to have any type of content as its body. However, this content is not processed at all and completely ignored. It is up to the developer of the tag handler to get access to that content and do something with it. For example, if you wanted to develop a tag that would allow the JSP page developer to execute an SQL statement, providing the SQL in the body of the tag, you would use `tagdependent` as the value of the `body-content` element.

Finally, the value of `JSP` is provided for backward compatibility with the classic custom tag model. It is not a legal value when used with the `SimpleTag` API.

Note that there is no legal way of allowing any scripting elements to appear as the tag body under the new `SimpleTag` API model.

Core Warning

When using the `SimpleTag` API, it is illegal to include scripting elements in the body of the tag.

The TLD file must be placed inside the `WEB-INF` directory or any subdirectory thereof.

Core Note

The TLD file must be placed inside the `WEB-INF` directory or a subdirectory thereof.

We suggest that you don't try to retype the TLD every time you start a new tag library, but start with a template. You can download such a template from http://volume2.coreservlets.com/.

The JSP File

Once you have a tag handler implementation and a TLD, you are ready to write a JSP file that makes use of the tag. Listing 7.3 gives an example. Somewhere in the JSP page you need to place 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 TLD file like the one shown in Listing 7.2. For now, we will use a simple URL relative to the Web application's root directory. This makes it easy to refer to the same TLD file from multiple JSP pages in different directories. Remember that the TLD file must be placed somewhere inside the WEB-INF directory. Because this URL will be resolved on the server and not the client, it is allowed to refer to the WEB-INF directory, which is always protected from direct client access.

The required `prefix` attribute specifies a prefix to use in front of any tag name defined in the TLD of this `taglib` declaration. For example, if the TLD file defines a tag named `tag1` and the `prefix` attribute has a value of `test`, the JSP page would need to refer to the tag as `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 />
```

Listing 7.3 Example JSP File

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Example JSP page</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<%@ taglib uri="/WEB-INF/tlds/example.tld"
           prefix="test" %>
<test:example/>
<test:example></test:example>
</BODY></HTML>
```

7.2 Example: Simple Prime Tag

In this example we create a simple custom tag that would output a random 50-digit prime number to the JSP page (a real treat!). We accomplish this task with the help of the `Primes` class shown in Listing 7.4.

We define a tag handler class `SimplePrimeTag` that extends the `SimpleTagSupport` class. In its `doTag` method, we obtain a reference to the `JspWriter` by calling `getJspContext().getOut()`. Then, by using the static method `Primes.nextPrime` we generate our random 50-digit prime number. We output this number to the JSP page by invoking the `print` method on the `JspWriter` object reference. The code for SimplePrimeTag.java is shown in Listing 7.5.

Listing 7.4 Primes.java

```
package coreservlets;
import java.math.BigInteger;

/** A few utilities to generate a large random BigInteger,
 *  and find the next prime number above a given BigInteger.
 */
public class Primes {
  private static final BigInteger ZERO = BigInteger.ZERO;
  private static final BigInteger ONE = BigInteger.ONE;
  private static final BigInteger TWO = new BigInteger("2");

  // Likelihood of false prime is less than 1/2^ERR_VAL
  // Presumably BigInteger uses the Miller-Rabin test or
  // equivalent, and thus is NOT fooled by Carmichael numbers.
  // See section 33.8 of Cormen et al.'s Introduction to
  // Algorithms for details.
  private static final int ERR_VAL = 100;

  public static BigInteger nextPrime(BigInteger start) {
    if (isEven(start))
      start = start.add(ONE);
    else
      start = start.add(TWO);
    if (start.isProbablePrime(ERR_VAL))
      return(start);
    else
      return(nextPrime(start));
  }
```

Listing 7.4 Primes.java *(continued)*

```
private static boolean isEven(BigInteger n) {
  return(n.mod(TWO).equals(ZERO));
}

private static StringBuffer[] digits =
  { new StringBuffer("0"), new StringBuffer("1"),
    new StringBuffer("2"), new StringBuffer("3"),
    new StringBuffer("4"), new StringBuffer("5"),
    new StringBuffer("6"), new StringBuffer("7"),
    new StringBuffer("8"), new StringBuffer("9") };
private static StringBuffer randomDigit(boolean isZeroOK) {
  int index;
  if (isZeroOK) {
    index = (int)Math.floor(Math.random() * 10);
  } else {
    index = 1 + (int)Math.floor(Math.random() * 9);
  }
  return(digits[index]);
}

/** Create a random big integer where every digit is
 *  selected randomly (except that the first digit
 *  cannot be a zero).
 */
public static BigInteger random(int numDigits) {
  StringBuffer s = new StringBuffer("");
  for(int i=0; i<numDigits; i++) {
    if (i == 0) {
      // First digit must be non-zero.
      s.append(randomDigit(false));
    } else {
      s.append(randomDigit(true));
    }
  }
  return(new BigInteger(s.toString()));
}

/** Simple command-line program to test. Enter number
 *  of digits, and it picks a random number of that
 *  length and then prints the first 50 prime numbers
 *  above that.
 */

public static void main(String[] args) {
  int numDigits;
```

Listing 7.4 Primes.java *(continued)*

```
    try {
      numDigits = Integer.parseInt(args[0]);
    } catch (Exception e) { // No args or illegal arg.
      numDigits = 150;
    }
    BigInteger start = random(numDigits);
    for(int i=0; i<50; i++) {
      start = nextPrime(start);
      System.out.println("Prime " + i + " = " + start);
    }
  }
}
```

Listing 7.5 SimplePrimeTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.math.*;
import coreservlets.Primes;

/**
 *  SimplePrimeTag output a random 50-digit prime number
 *  to the JSP page.
 */
public class SimplePrimeTag extends SimpleTagSupport {
  protected int length = 50;

  public void doTag() throws JspException, IOException {
    JspWriter out = getJspContext().getOut();
    BigInteger prime = Primes.nextPrime(Primes.random(length));
    out.print(prime);
  }
}
```

Now that we have our tag handler class, we need to describe our tag to the container. We do this using the TLD csajsp-taglib.tld shown in Listing 7.6. Because all our tag does is output a prime number, we don't need to allow the tag to include a body, and so we specify `empty` as the value of the `body-content` element. We place the csajsp-taglib.tld file in the WEB-INF/tlds folder.

Listing 7.6 Excerpt from csajsp-taglib.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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>csajsp-taglib</short-name>

  <tag>
    <description>Outputs 50-digit primes</description>
    <name>simplePrime</name>
    <tag-class>coreservlets.tags.SimplePrimeTag</tag-class>
    <body-content>empty</body-content>
  </tag>
  ...
</taglib>
```

Listing 7.7 shows the simple-primes-1.jsp page, which uses the simple prime tag. We assign `csajsp` as the prefix for all tags (so far just `simplePrime`) in the `/WEB-INF/tlds/csajsp-taglib.tld` library. Also note that it is perfectly legal to use a closing tag with the `body-content` of `empty` as long as there is nothing, not even a space, between the opening tag and the closing tag, as shown by the last occurrence of the tag in the simple-primes-1.jsp page; that is, `<csajsp:simplePrime></csajsp:simplePrime>`. The resulting output is shown in Figure 7–1.

Listing 7.7 simple-primes-1.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="/WEB-INF/tlds/csajsp-taglib.tld"
           prefix="csajsp" %>
```

Listing 7.7 simple-primes-1.jsp *(continued)*

```
<UL>
  <LI><csajsp:simplePrime />
  <LI><csajsp:simplePrime />
  <LI><csajsp:simplePrime />
  <LI><csajsp:simplePrime></csajsp:simplePrime>
</UL>
</BODY></HTML>
```

Figure 7–1 Result of simple-primes-1.jsp.

7.3 Assigning Attributes to Tags

Allowing tags like

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

adds significant flexibility to your tag library because the attributes allow us to pass information to the tag. This section explains how to add attribute support to your tags.

Tag Attributes: 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 `SimpleTagSupport` (or that otherwise implements the `SimpleTag` interface). Consequently, adding support for an attribute named `attribute1` is merely a matter of implementing the following method in your tag handler class:

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

Note that an attribute with the name 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 for later use by the `doTag` method. For example, the following is a code snippet 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 is 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.

Tag Attributes: Tag Library Descriptor

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>`.

- **`name`.** This is a required element that defines the case-sensitive attribute name.
- **`required`.** This is an optional element that stipulates whether the attribute must always be supplied, `true`, or is optional, `false` (default). If `required` is `false` and the JSP page omits the attribute, no call is made to the `setAttributeName` method, so be sure to give default values to the fields that the method sets if the attribute is not declared as required. Omitting a tag attribute, which is declared with the `required` element equal to `true`, results in an error at page translation time.
- **`rtexprvalue`.** This is an optional element that indicates whether the attribute value can be either a JSP scripting expression like `<%= expression %>` or JSP EL like `${bean.value}` (`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. Note that even though it is never legal for the body of the tag to contain JSP scripting expressions like `<%= expression %>`, they are nevertheless legal as attribute values.

Tag Attributes: JSP File

As before, the JSP page has to declare the tag library using the `taglib` directive. This is done in the following form:

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

The usage of the tag is very similar, except now we are able to specify a custom attribute as well. Remember that just like tag names, the attribute names are case-sensitive and have to appear in the JSP page exactly as they were declared inside the TLD file. Because custom tags are based on XML syntax, the value of an attribute has to be enclosed by either single or double quotes. For example:

```
<some-prefix:tag1 attribute1="value" />
```

7.4 Example: Prime Tag with Variable Length

In this example, we modify the previous prime number example, shown in Section 7.2 (Example: Simple Prime Tag), to provide an attribute for specifying the length of the prime number. Listing 7.8 shows the `PrimeTag` class, a subclass of `SimplePrimeTag` that adds support for the `length` attribute. This change is achieved by supplying an additional method, `setLength`. When this method is called, it attempts to convert its `String` argument into an `int` and store it in an instance variable `length`. If it fails, the originally initialized value for the instance variable `length` is used.

The TLD, shown in Listing 7.9, declares the optional attribute `length`. It is this declaration that tells the container to call the `setLength` method if the attribute `length` appears in the tag when it's used in the JSP page.

The JSP page, shown in Listing 7.10, declares the tag library with the `taglib` directive as before. However, now we are able to specify how long our prime number should be. If we omit the `length` attribute, the prime tag defaults to `50`. Figure 7–2 shows the result of this page.

Listing 7.8 PrimeTag.java

```
package coreservlets.tags;

/** PrimeTag outputs a random prime number
 *  to the JSP page. The length of the prime number is
 *  specified by the length attribute supplied by the JSP
 *  page. If not supplied, it defaults to 50.
 */
public class PrimeTag extends SimplePrimeTag {
  public void setLength(String length) {
    try {
      this.length = Integer.parseInt(length);
    } catch(NumberFormatException nfe) {
      // Do nothing as length is already set to 50
    }
  }
}
```

Listing 7.9 Excerpt from csajsp-taglib.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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>csajsp-taglib</short-name>

  <tag>
    <description>Outputs an N-digit prime</description>
    <name>prime</name>
    <tag-class>coreservlets.tags.PrimeTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
      <description>N (prime number length)</description>
      <name>length</name>
      <required>false</required>
    </attribute>
  </tag>
</taglib>
```

Listing 7.10 primes-1.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="/WEB-INF/tlds/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 7–2 Result of primes-1.jsp.

7.5 Including Tag Body in the Tag Output

Up to this point, all of the custom tags you have seen did not allow a body and thus were always used as standalone tags of the following form:

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

Note that the second tag shown does not have any space between the opening and closing tags. The fact that these tags were not allowed to include a body was a direct result of supplying the element `body-content` with the value of `empty`.

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

```
<prefix:tagname>scriptless JSP content</prefix:tagname>
```

Tag Bodies: Tag Handler Class

Supporting tag bodies does not introduce any structural changes to the tag handler class. You still need to include setter methods for any attributes you are planning to declare and use. You still need to override the `doTag` method. To output the body content of the tag, inside the `doTag` method you need to acquire the `JspFragment` instance representing the body of the tag by calling the `getJspBody` method, then using its `invoke` method passing it `null` as its argument. Usually, this is done in a single step as follows:

```
getJspBody().invoke(null);
```

The container processes the JSP content found in the body of the tag just like any other JSP page content. If the `invoke` method is passed `null` as its argument, the resulting output of that JSP content is passed verbatim to the client. Therefore, the `doTag` method has no way of accessing the tag body output. All it can do is pass it along. We show how to access and modify the output of the tag body content before it's sent to the client in Section 8.1 (Manipulating Tag Body). It's important to stress, however, that it is the output resulting from the execution of the JSP code in the tag body, not the JSP code itself, that is passed to the client.

Core Note

When `getJspBody().invoke(null)` is called, it is the output resulting from the execution of the tag body's JSP content that gets passed to the client, not the JSP code itself.

In practice, you almost always output something before or after outputting the tag body as follows:

```
JspWriter out = getJspContext().getOut();
out.print("...");
getJspBody().invoke(null);
out.print("...");
```

Note that because sending the JSP content of the tag body boils down to a simple method invocation, it is very easy to create a tag that conditionally sends the JSP content to the client by surrounding the method call with an `if` statement. We show an example of this in Section 7.7 (Example: Debug Tag). It is also trivial to output the tag body content several times, as the method call can be placed inside a `for` loop and invoked many times. We show an example of this in Section 8.4 (Example: Simple Looping Tag).

Tag Bodies: Tag Library Descriptor

The change to the TLD is trivial. Instead of the value of `empty` for the required `body-content` element, we need to provide the value of `scriptless`.

Tag Bodies: JSP File

There are no changes to the JSP file. You still need to declare and assign a `prefix` to the TLD through the `taglib` directive. However, now we can use our tags with nonempty bodies.

Remember, however, that the `body-content` was declared as `scriptless`, and that `scriptless` means we are allowed to place JSP content into the body of the tag, but are not allowed to place JSP scriptlets there. So, the following is a legal usage of the tag:

```
<prefix:tagname>
  some content with ${bean.property}
</prefix:tagname>
```

The following would be illegal:

```
<prefix:tagname>
  some content with <%= bean.property %>
</prefix:tagname>
```

7.6 Example: Heading Tag

Listing 7.11 shows HeadingTag.java, which defines a tag for a heading element that is more flexible than the standard HTML H1 through H6 elements. (Yes, we know that the entire problem could be solved more elegantly with Cascading Style Sheets [CSS] and without the use of a custom tag, but this is for demonstration purposes only, so work with us.) 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 stylesheet attributes.

The doTag method first generates the <TABLE> and <SPAN> start tags, then invokes getJspBody().invoke(null) to instruct the system to include the tag body, and then generates the </SPAN> and </TABLE> tags. We use various setAttributeName methods to handle the attributes like bgColor and fontSize.

Listing 7.12 shows the excerpt from the csajsp-taglib.tld file that defines the heading tag. Listing 7.13 shows heading-1.jsp, which uses the heading tag. Figure 7–3 shows the resulting JSP page.

Listing 7.11 HeadingTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/** Heading tag allows the JSP developer to create
 *  a heading and specify alignment, background color,
 *  foreground color, font, etc. for that heading.
 */
public class HeadingTag extends SimpleTagSupport {
  private String align;
  private String bgColor;
  private String border;
  private String fgColor;
  private String font;
  private String size;

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

Listing 7.11 HeadingTag.java *(continued)*

```
  public void setBgColor(String bgColor) {
    this.bgColor = bgColor;
  }
  public void setBorder(String border) {
    this.border = border;
  }
  public void setFgColor(String fgColor) {
    this.fgColor = fgColor;
  }
  public void setFont(String font) {
    this.font = font;
  }
  public void setSize(String size) {
    this.size = size;
  }

 public void doTag() throws JspException, IOException {
    JspWriter out = getJspContext().getOut();
    out.print("<TABLE ALIGN=\"" + align + "\"\n" +
              "       BGCOLOR=\"" + bgColor + "\"\n" +
              "       BORDER=" + border + "\">\n");
    out.print("<TR><TH>");
    out.print("<SPAN STYLE=\"color: " + fgColor + ";\n" +
              "              font-family: " + font + ";\n" +
              "              font-size: " + size + "px; " +
              "\">\n");
    // Output content of the body
    getJspBody().invoke(null);
    out.println("</SPAN></TH></TR></TABLE>" +
                "<BR CLEAR=\"ALL\"><BR>");
  }
}
```

Listing 7.12 Excerpt from csajsp-taglib.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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>csajsp-taglib</short-name>
```

Listing 7.12 Excerpt from csajsp-taglib.tld *(continued)*

```
<tag>
    <description>Formats enclosed heading</description>
    <name>heading</name>
    <tag-class>coreservlets.tags.HeadingTag</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
      <name>align</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>bgColor</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>border</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>fgColor</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>font</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>size</name>
      <required>true</required>
    </attribute>
  </tag>
</taglib>
```

Listing 7.13 heading-1.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Headings</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<%@ taglib uri="/WEB-INF/tlds/csajsp-taglib.tld"
           prefix="csajsp" %>
```

Listing 7.13 heading-1.jsp *(continued)*

```
<csajsp:heading align="LEFT" bgColor="CYAN"
                border="10" fgColor="BLACK"
                font="Arial Black" size="78">
  First Heading
</csajsp:heading>

<csajsp:heading align="RIGHT" bgColor="RED"
                border="1" fgColor="YELLOW"
                font="Times New Roman" size="50">
  Second Heading
</csajsp:heading>

<csajsp:heading align="CENTER" bgColor="#C0C0C0"
                border="20" fgColor="BLUE"
                font="Arial Narrow" size="100">
  Third Heading
</csajsp:heading>
</BODY></HTML>
```

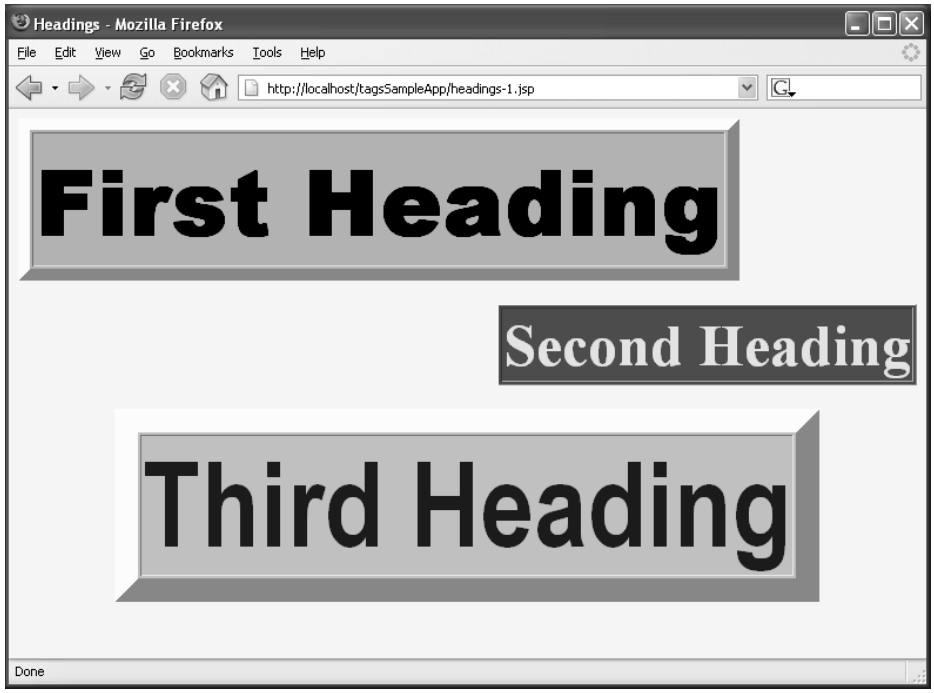

Figure 7–3 Result of heading-1.jsp.

7.7 Example: Debug Tag

In Section 7.5 (Including Tag Body in the Tag Output), we explained that to send the JSP content of the tag body to the client, one need only call the `getJspBody().invoke(null)` method inside the `doTag` method of the tag handler class. This simplicity allows us to easily create tags that output their bodies conditionally. This functionality can be achieved by simply surrounding the `getJspBody().invoke(null)` invocation within an `if` statement.

In this section, we present an example of a custom tag that conditionally outputs its tag body. It's quite often the case when the output of the JSP page is something other than what you expected. In such a case, it's useful to have the option of seeing some debugging information right on the page without having to resort to embedding `System.out.print` statements throughout the page. However, we do not want the user to see the debugging information in the production system. To solve this problem, we create a custom tag that conditionally outputs its body based on the presence of the `debug` request parameter. If the `debug` request parameter is present, it would signal to the JSP page to output the debugging information.

Listing 7.14 shows the DebugTag.java file. In its `doTag` method, we output the tag body if the `debug` request parameter is present and skip the body of the tag if it's not. Inside the JSP page, shown in Listing 7.16, we surround the debugging information with our `debug` tag. Listing 7.15 shows the excerpt from the csajsp-taglib.tld file declaring the `debug` tag to the container. Listing 7.16 shows the debug.jsp page that uses the `debug` tag. Figure 7–4 shows the result of the debug.jsp page when the `debug` request parameter is not present. Figure 7–5 shows the result of the debug.jsp page when the `debug` request parameter is supplied.

Listing 7.14 DebugTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import javax.servlet.http.*;

/**
 *  DebugTag outputs its body if the request parameter
 *  'debug' is present and skips it if it's not.
 */
```

Listing 7.14 DebugTag.java *(continued)*

```
public class DebugTag extends SimpleTagSupport {
  public void doTag() throws JspException, IOException {
    PageContext context = (PageContext) getJspContext();
    HttpServletRequest request =
      (HttpServletRequest) context.getRequest();
    // Output body of tag only if debug param is present.
    if (request.getParameter("debug") != null) {
      getJspBody().invoke(null);
    }
  }
}
```

Listing 7.15 Excerpt from csajsp-taglib.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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>csajsp-taglib</short-name>

  <tag>
    <description>Conditionally outputs enclosed body</description>
    <name>debug</name>
    <tag-class>coreservlets.tags.DebugTag</tag-class>
    <body-content>scriptless</body-content>
  </tag>
</taglib>
```

Listing 7.16 debug.jsp

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

Listing 7.16 debug.jsp *(continued)*

```
<BODY>
<H1>Some Hard-to-Debug Page</H1>
<%@ taglib uri="/WEB-INF/tlds/csajsp-taglib.tld"
           prefix="csajsp" %>
Top of regular page. Blah, blah, blah.
Yadda, yadda, yadda.
<csajsp:debug>
<H2>Debug Info:</H2>
**********<BR>
-Remote Host: ${pageContext.request.remoteHost}<BR>
-Session ID: ${pageContext.session.id}<BR>
-The foo parameter: ${param.foo}<BR>
**********<BR>
</csajsp:debug>
<P>
Bottom of regular page. Blah, blah, blah.
Yadda, yadda, yadda.
</BODY></HTML>
```

Figure 7–4 Result of debug.jsp page without supplying the `debug` request parameter.

Figure 7–5 Result of `debug.jsp` page when the `debug` request parameter is supplied.

7.8 Creating Tag Files

JSP specification version 2.0 introduced a JSP-based way to create custom tags using tag files. One of the key differences between what we talk about in the beginning of this chapter, Java-based custom tags, and tag files (or JSP-based custom tags) is that with Java-based tags the tag handler is a Java class, whereas with JSP-based tags the tag handler is a JSP page. Tag files are also a bit simpler to write because they don't require you to provide a TLD.

The guidelines for when to develop a JSP-based custom tag versus a Java-based custom tag are analogous to the guidelines for when to use a JSP page versus a servlet. When there is a lot of logic, use Java to create output. When there is a lot of HTML formatting, use tag files to create output. To review the general benefits of JSPs versus servlets, please see Section 10.2 of Volume 1.

There is one caveat that might force your choice between tag files and Java-based custom tags. Tag files run only in JSP 2.0, whereas Java-based custom tags have a "classic" version that does not rely on the new `SimpleTag` API. So, if the container you are targeting is only compliant with earlier versions of the specification, you have to use classic Java-based custom tag development. The bad news is that classic

Java-based custom tag development is quite more complicated than the SimpleTag API and we do not cover classic tags in this book. The good news is that almost all mainstream containers have been updated to be compliant with servlet specification 2.4 and JSP specification 2.0, so chances are you won't need to develop the classic Java-based custom tags.

In general, there are two steps to creating a JSP-based custom tag.

- **Create a JSP-based tag file.** This file is a fragment of a JSP page with some special directives and a .tag extension. It must be placed inside the WEB-INF/tags directory or a subdirectory thereof.
- **Create a JSP page that uses the tag file.** The JSP page points to the directory where the tag file resides. The name of the tag file (minus the .tag extension) becomes the name of the custom tag and therefore no TLD connecting the implementation of the tag with its name is needed.

In the next few sections, we reproduce the same custom tags we developed earlier in this chapter, but we use tag files to accomplish it.

7.9 Example: Simple Prime Tag Using Tag Files

Let's rewrite the simple prime custom tag example using tag files. Listing 7.17 shows simplePrime2.tag. It consists of just one line invoking the static method nextPrime of the Primes class. The Primes.java file is shown in Listing 7.4. We place the simplePrime2.tag file into the WEB-INF/tags directory. Listing 7.18 shows simple-primes-2.jsp, which uses our JSP-based custom tag. Note that the taglib directive no longer has a uri attribute, but uses a tagdir attribute instead. This attribute tells the container which directory contains the tag files. Figure 7–6 shows the result of simple-primes-2.jsp.

Listing 7.17 simplePrime2.tag

```
<%= coreservlets.Primes.nextPrime
      (coreservlets.Primes.random(50)) %>
```

Listing 7.18 simple-primes-2.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 tagdir="/WEB-INF/tags" prefix="csajsp" %>
<UL>
  <LI><csajsp:simplePrime2 />
  <LI><csajsp:simplePrime2 />
  <LI><csajsp:simplePrime2 />
  <LI><csajsp:simplePrime2 />
</UL>
</BODY></HTML>
```

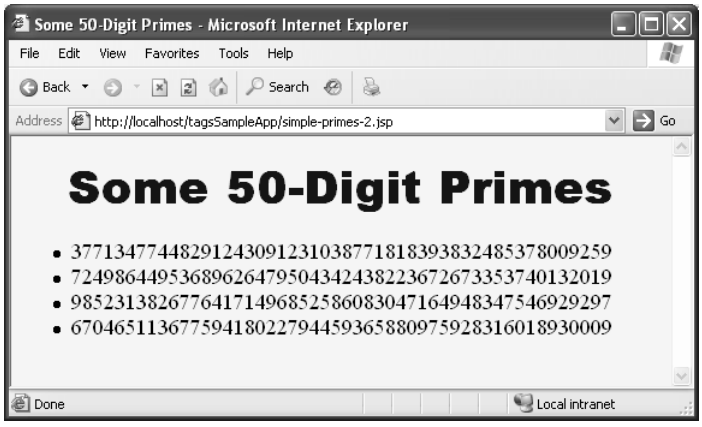

Figure 7–6 Result of simple-primes-2.jsp.

7.10 Example: Prime Tag with Variable Length Using Tag Files

In this section, we rewrite the example of Section 7.4 (Example: Prime Tag with Variable Length) with a JSP-based custom tag. To use attributes with a JSP-based custom tag, each attribute must be declared inside the tag file. This declaration is accomplished by the `attribute` directive. The `attribute` directive itself has attributes that provide the same information that the `attribute` subelements inside the TLD would provide. For example, you can specify whether an attribute is required or not by supplying a `required` attribute with a value of either `true` or `false`. When the value is passed through an attribute to the tag file, it is automatically stored into a scoped variable for access from the JSP EL and into a local variable for access from Java code (scriptlets and scripting expressions). Note once again that because the tag file has the ability to describe itself to the container, no TLD is required.

Listing 7.19 shows prime2.tag declaring an optional attribute called `length`. Note that we are able to refer to that attribute just like to any other local variable inside the Java code. The JSP page, primes-2.jsp, shown in Listing 7.20, uses our tag file to output prime numbers of different lengths. Figure 7–7 shows the result of primes-2.jsp.

Listing 7.19 prime2.tag

```
<%@ attribute name="length" required="false" %>
<%
int len = 50;
try {
  len = Integer.parseInt(length);
} catch(NumberFormatException nfe) {}
%>
<%= coreservlets.Primes.nextPrime
      (coreservlets.Primes.random(len)) %>
```

Listing 7.20 primes-2.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 tagdir="/WEB-INF/tags" prefix="csajsp" %>
<UL>
  <LI>20-digit: <csajsp:prime2 length="20" />
  <LI>40-digit: <csajsp:prime2 length="40" />
  <LI>80-digit: <csajsp:prime2 length="80" />
  <LI>Default (50-digit): <csajsp:prime2 />
</UL>
</BODY></HTML>
```

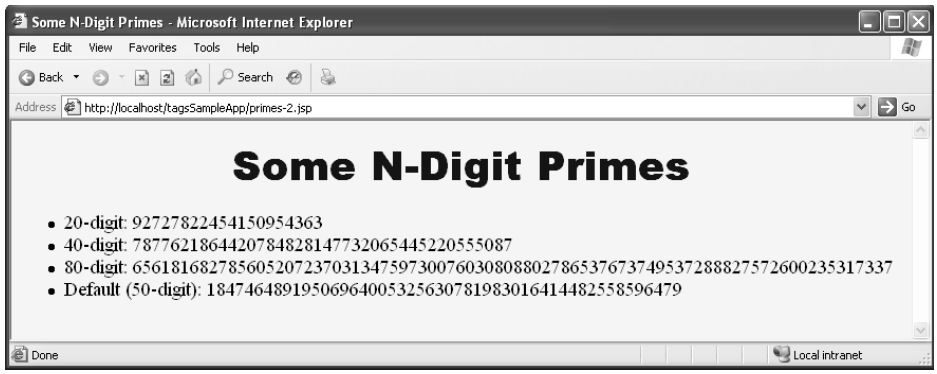

Figure 7–7 Result of primes-2.jsp.

7.11 Example: Heading Tag Using Tag Files

In this section, we rewrite the heading example of Section 7.6 (Example: Heading Tag) with a JSP-based custom tag. Outputting the tag body inside a tag file is as simple as providing a `<jsp:doBody/>` tag. That's it! No additional configurations, no TLD file, and the access to attributes is still the same simple process described in Section 7.10 (Example: Prime Tag with Variable Length Using Tag Files). Just place `<jsp:doBody/>` where you want the tag body to appear in the final output and you are done.

Listing 7.21 shows the heading2.tag file. It declares quite a number of required attributes and then proceeds to use them as regular scoped variables. We use `<jsp:doBody/>` to output the body of the tag to the client. Listing 7.22 shows the headings-2.jsp file, which uses the heading2.tag custom tag. Figure 7–8 shows the result of headings-2.jsp.

Listing 7.21 heading2.tag

```
<%@ attribute name="align" required="true" %>
<%@ attribute name="bgColor" required="true" %>
<%@ attribute name="border" required="true" %>
<%@ attribute name="fgColor" required="true" %>
<%@ attribute name="font" required="true" %>
<%@ attribute name="size" required="true" %>
<TABLE ALIGN="${align}"
       BGCOLOR="${bgColor}"
       BORDER="${border}">
  <TR><TH>
      <SPAN STYLE="color: ${fgColor};
                   font-family: ${font};
                   font-size: ${size}px;">
      <jsp:doBody/></SPAN>
</TABLE><BR CLEAR="ALL"><BR>
```

Listing 7.22 headings-2.jsp

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

Listing 7.22 headings-2.jsp *(continued)*

```
<BODY>
<%@ taglib tagdir="/WEB-INF/tags" prefix="csajsp" %>
<csajsp:heading2 align="LEFT" bgColor="CYAN"
                 border="10" fgColor="BLACK"
                 font="Arial Black" size="78">
  First Heading
</csajsp:heading2>
<csajsp:heading2 align="RIGHT" bgColor="RED"
                 border="1" fgColor="YELLOW"
                 font="Times New Roman" size="50">
  Second Heading
</csajsp:heading2>
<csajsp:heading2 align="CENTER" bgColor="#C0C0C0"
                 border="20" fgColor="BLUE"
                 font="Arial Narrow" size="100">
  Third Heading
</csajsp:heading2>
</BODY></HTML>
```

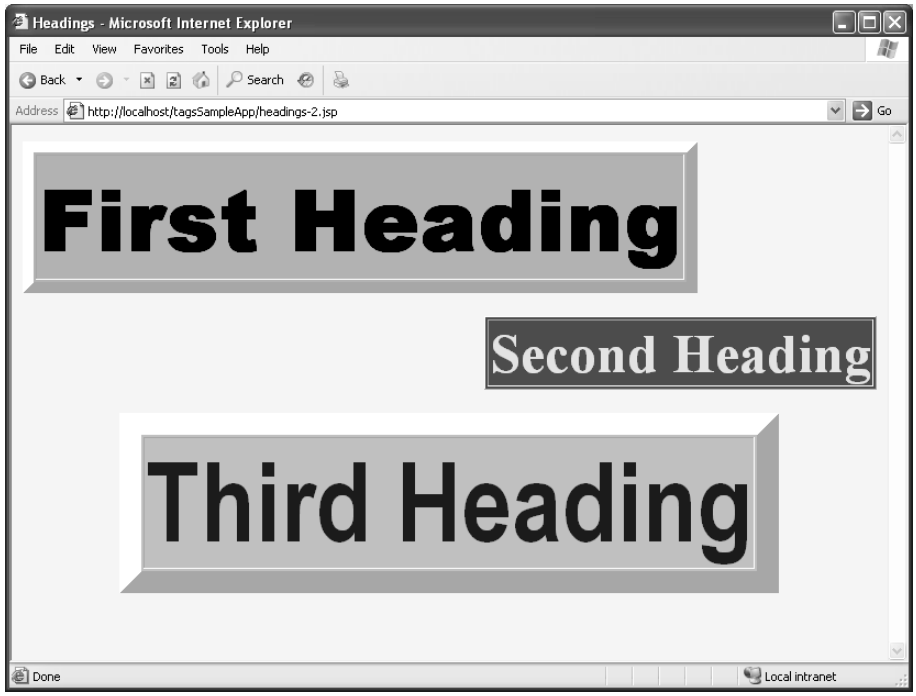

Figure 7–8 Result of headings-2.jsp.