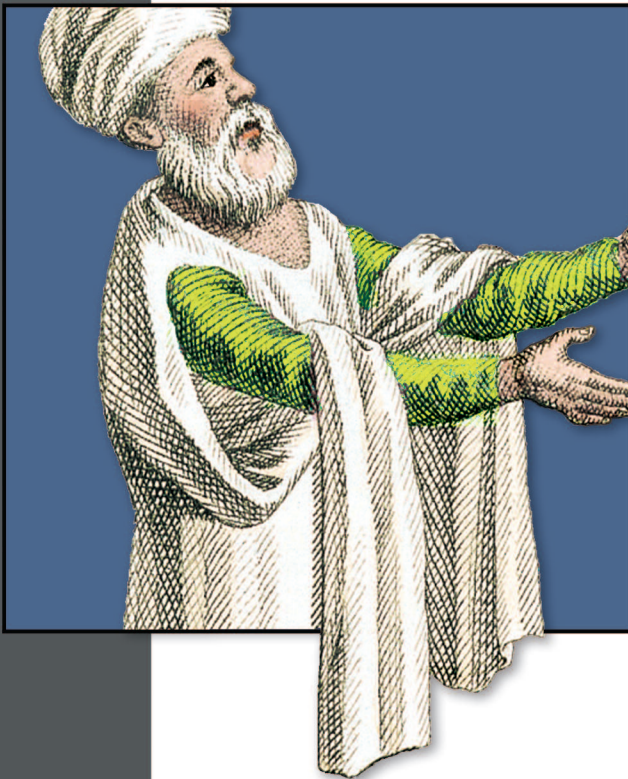


SCWCD

SCE 310-081

EXAM STUDY KIT

SECOND EDITION



JAVA WEB COMPONENT DEVELOPER CERTIFICATION

**Hanumant Deshmukh
Jignesh Malavia
Matthew Scarpino**

 **MANNING**



SCWCD Study Kit Second Edition

by Matthew Scarpino (Second Edition author)

Hanumant Deshmukh

Jignesh Malavia

with Jacquelyn Carter

Sample Chapter 17

Copyright 2005 Manning Publications

brief contents

Part 1 Getting started 1

- 1 Understanding Java servlets 3*
- 2 Understanding JavaServer Pages 14*
- 3 Web application and HTTP basics 21*

Part 2 Servlets 29

- 4 The servlet model 31*
- 5 Structure and deployment 67*
- 6 The servlet container model 83*
- 7 Using filters 97*
- 8 Session management 119*
- 9 Developing secure web applications 139*

Part 3 JavaServer Pages and design patterns 163

- 10 The JSP technology model—the basics 165*
- 11 The JSP technology model—advanced topics 188*
- 12 Reusable web components 219*

13	<i>Creating JSPs with the Expression Language (EL)</i>	236
14	<i>Using JavaBeans</i>	251
15	<i>Using custom tags</i>	285
16	<i>Developing “Classic” custom tag libraries</i>	309
17	<i>Developing “Simple” custom tag libraries</i>	352
18	<i>Design patterns</i>	376

Appendices

A	<i>Installing Tomcat 5.0.25</i>	403
B	<i>A sample web.xml file</i>	408
C	<i>Review Q & A</i>	412
D	<i>Exam Quick Prep</i>	475



CHAPTER 17

Developing “Simple” custom tag libraries

- 17.1 Understanding SimpleTags 353
- 17.2 Incorporating SimpleTags in JSPs 357
- 17.3 Creating Java-free libraries with tag files 364
- 17.4 Summary 371
- 17.5 Review questions 372

EXAM OBJECTIVES

- 10.4** Describe the semantics of the “Simple” custom tag event model when the event method (doTag) is executed; write a tag handler class; and explain the constraints on the JSP content within the tag.
(Sections 17.1 and 17.2)
- 10.5** Describe the semantics of the Tag File model; describe the web application structure for tag files; write a tag file; and explain the constraints on the JSP content in the body of the tag.
(Section 17.3)

So far, you've seen the interfaces for classic tag development (`Tag`, `IterationTag`, and `BodyTag`) and their associated event methods (`doStartTag()`, `doAfterBody()`, and `doEndTag()`). These constructs give you a great deal of flexibility in building custom tag libraries, but coding can be time-consuming and complex. The JSP 2.0 standard reduces the burden by providing an alternate means of creating your tag libraries: the simple tag model.

With this new methodology, you only need to keep track of one interface, `SimpleTag`, and a single event method, `doTag()`. This way, you can concentrate on Java code instead of directing the web container's operation. JSP 2.0 also gives you different options for processing body content and tag attributes.

In addition, the new specification provides a new means of library development with *tag files*. Tag file processing is similar to regular JSP tag processing, but doesn't use tag library descriptors or tag handlers. Instead, tag files are coded with regular JSP syntax, and can include script elements. These building blocks simplify the process of library creation and make it more modular.

In practical web development, you can choose whatever tag library development method you prefer. But for the SCWCD exam, you need to become familiar with each. So, having discussed the classic way of building custom tag libraries, let's explore the simple method.

17.1 UNDERSTANDING SIMPLETAGS

Building custom tag libraries with `SimpleTags` is similar to the process we described in chapter 16. You still need to create a Java tag handler, reference the class in a tag library descriptor (TLD), and include the TLD in your JSP.

The differences between classic and simple development concern the processing needed in a Java-based tag handler. `SimpleTag` classes use fewer methods, interface differently with body content, and have different implicit objects available. This section covers the principles behind `SimpleTags` and how they reduce the difficulty of building tag libraries.

17.1.1 A brief example

Before we get into the theory of `SimpleTags`, you can appreciate how easy they are to use by looking at example code. Listing 17.1 shows a Java tag handler that sends a message to the JSP output.

Listing 17.1 `SimpleTagExample.java`

```
package myTags;

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

public class SimpleTagExample extends SimpleTagSupport
{
    public void doTag() throws JspException, IOException
```

```

{
    getJspContext().getOut().print(
        "I can't believe it's so simple!"
    );
}
}

```

That's it! There are no concerns with `SKIP_BODY`, `EVAL_BODY`, `EVAL_PAGE`, or any of the return values associated with classic tags. There's no need to consider different interfaces depending on body content or iterations. Instead, there's just one method, `doTag()`, and a single line of code to send output to the JSP.

The web container can access this class through a tag library descriptor in the same way as a `Tag` or `BodyTag` class. For example, the snippet below matches the `SimpleTagExample` class with a tag name called "message":

```

<taglib>
..
<tag>
  <name>message</name>
  <tag-class>myTags.SimpleTagExample</tag-class>
  <body-content>empty</body-content>
  <description>Sends a message to the JSP</description>
</tag>
..
</taglib>

```

Then, the "message" tag can be inserted into a JSP page just as with the classic model.

The superclass of `SimpleTagExample`, `SimpleTagSupport`, makes all of this possible. To see why this is the case, you need to understand both it and its interface, `SimpleTag`. In particular, we'll present the methods contained in `SimpleTag` and its life cycle.

17.1.2 Exploring SimpleTag and SimpleTagSupport

In the classic method of building custom tag libraries, Java classes implement the `BodyTag` interface if body content needs processing, the `IterationTag` interface if multiple operations are required, or the `Tag` interface if neither is necessary. With the simple model, the `SimpleTag` interface can be used in all three cases. The relationship between these interfaces is shown in figure 17.1.

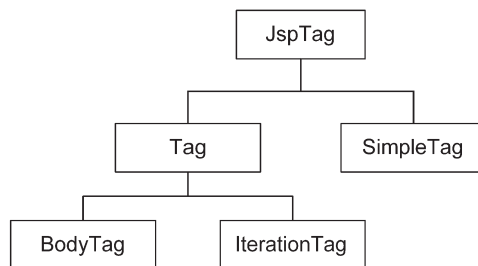


Figure 17.1
JSP tag
interfaces

The JSP 2.0 specification also provides a new adapter class for tag classes. Just as classic tag handlers extend `BodyTagSupport` or `TagSupport`, classes in the simple model can extend `SimpleTagSupport`. This class contains all of the methods needed to implement `SimpleTag`, and provides additional methods for extracting information from the web container.

Using the SimpleTag interface—methods and life cycle

Like those of `Tag` and `BodyTag`, the methods in the `SimpleTag` interface serve two purposes. First, they allow you to transfer information between your Java class and the JSP. Second, they are invoked by the web container to initialize `SimpleTag` operation. Table 17.1 lists these methods with descriptions.

Table 17.1 Methods of the `SimpleTag` interface

Name	Description
<code>setJspContext()</code>	Makes the <code>JspContext</code> available for tag processing
<code>setParent()</code>	Called by the web container to make the parent tag available
<code>setJspBody()</code>	Makes the body content available for tag processing
<code>doTag()</code>	Called by the container to begin <code>SimpleTag</code> operation
<code>getParent()</code>	Called by the Java class to obtain its parent <code>JspTag</code>

These methods are listed in the order that they are invoked in the `SimpleTag` life cycle, which has three main steps.

Step 1 Initialize the information associated with the SimpleTag

After the web container creates an instantiation of the `SimpleTag` class, it calls the `setJspContext()` method. This method returns an instance of the `JspContext` class, which is the superclass of `PageContext`—the object returned by the `setPageContext()` method in the `Tag` or `BodyTag` interface. Like the `PageContext`, the `JspContext` allows your Java class to access scoped attributes and implicit variables.

Most of the `JspContext` methods are similar to those in the `PageContext` class, but there are a few different methods that can be very useful. First, as shown in the example, the `getOut()` method returns a `JspWriter` that you can use to send information to the JSP output stream. Also, there are two methods, `getExpressionEvaluator()` and `getVariableResolver()`, that allow you to access the Expression Language handling capability of the container. An important thing to keep in mind is that, while the `PageContext` relies on J2EE servlet processing, the `JspContext` class is meant to be technology-neutral and able to interface with different packages or languages.

After the `JspContext` has been initialized for the `SimpleTag`, the web container calls `setParent()`. This method is invoked only if the `SimpleTag` is

surrounded by another set of tags. Because `setParent()` returns a `JspTag` object, the returned parent tag can implement the `Tag`, `BodyTag`, `IterationTag`, or the `SimpleTag` interface.

Step 2 Make body content available for `SimpleTag` processing

If there is any JSP code inside the tags, the web container invokes `setBody()` to make it available for the Java class. The method's return type is `JspFragment`. This class will be fully discussed later in this chapter, but for now, it is important to understand that a `JspFragment` contains regular JSP code (HTML, XML, tags, text) *without scripts*. So you can't include JSP declarations, expressions, or scriptlets inside `SimpleTags`. But EL terms can be added to the `JspFragment`.

Step 3 Invoke `doTag()`

The `doTag()` method of the `SimpleTag` interface combines the functions of the `Tag`'s `doStartTag()`, `doAfterBody()`, and `doEndTag()` methods. It doesn't return any values, and when it finishes, the web container returns to its previous processing tasks. Instead of calling special methods, you can control all of the iteration and body processing with regular Java commands.

It is important to understand why the `SimpleTag` interface is able to streamline the development of custom tag libraries. The reason has to do with JSP scripts. A great deal of the extra processing performed by a classic tag occurs because of the need to keep track of JSP scripts in the page. For example, if the tag body relies on a JSP variable declaration, then the tag processing needs to be able to access that variable.

With `SimpleTags`, this isn't an option. When the web container processes `SimpleTags`, it doesn't take JSP scripts into account. This makes for simpler coding and faster operation, but you need to keep this constraint in mind—both for the exam and your own web development.

Using `SimpleTagSupport`

The `SimpleTagSupport` class allows you to implement the `SimpleTag` interface without having to code each of its methods by yourself. Instead, it provides for each of the methods mentioned above, and three others, which are listed in table 17.2.

Table 17.2 Additional methods provided by the `SimpleTagSupport` class

Name	Description
<code>getJspContext()</code>	Returns the <code>JspContext</code> for processing in the tag
<code>getJspBody()</code>	Returns the <code>JspFragment</code> object for body processing in the tag
<code>findAncestorWithClass()</code>	Returns the ancestor tag with the specified class

These methods are similar to those in the `TagSupport` and `BodyTagSupport` classes, with two exceptions. First, the first two methods return a `JspContext` and a `JspFragment` instead of a `PageContext` and a `BodyContent` object. Second,

SimpleTagSupport leaves out many of the methods in TagSupport and BodyTagSupport that deal with tag processing, such as `release()`.

Now that you've seen how the SimpleTag interface and the SimpleTagSupport class functions, you can appreciate why Sun included them in the new JSP specification. In the next section, we will use these data structures to build practical JSPs.

Quizlet

Q: Which of the following methods aren't immediately available for a subclass of SimpleTagSupport?

- a** `getJspBody()` ;
- b** `getJspContext().getAttribute("name")` ;
- c** `getParent()` ;
- d** `getBodyContent()` ;

A: The answer is option d. The `getBodyContent()` method is provided by the BodyTagSupport class, and returns a BodyContent object. Instead, SimpleTagSupport invokes the `getJspBody()` method, which returns a JspFragment.

17.2 INCORPORATING SIMPLETAGS IN JSPs

The process of building a SimpleTag library and using its JSP tags is similar to that for classic tag libraries, but there are important differences between the two. In particular, SimpleTags process tag attributes and body content differently than Tag, IterationTag, or BodyTag classes. In this section, we'll make these characteristics apparent by building a tag library and JSP for calculating square roots.

Each SimpleTag class performs its main processing inside the `doTag()` method, but the structure of the class also depends on attribute tags and body content. To present these classes, we'll proceed from the simple to complex. This means starting with an empty SimpleTag.

17.2.1 Coding empty SimpleTags

Empty SimpleTag classes are used to send static information to the JSP. In this case, we'll start with a short class that sends a simple mathematical expression to a JspWriter. This may seem trivial, but we'll add more as we explore the SimpleTag interface in greater depth.

Listing 17.2 presents `MathTag.java`, located in the `myTags` package.

Listing 17.2 MathTag.java

```
package myTags;

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

```

public class MathTag extends SimpleTagSupport
{
    int x = 289;

    public void doTag() throws JspException, IOException
    {
        getJspContext().getOut().print(
            "The square root of " + x +
            " is " + Math.sqrt(x) + "."
        );
    }
}

```

After the web container creates an instance of `MathTag`, it will make the `JspContext` available. Since there are no nested tags or body content, it will then invoke the `doTag()` method directly.

A tag library descriptor is needed to tell the web container how to match the `MathTag` class with its JSP tag. Listing 17.3 shows `MathTag.tld`, which also informs the web container that the `MathTag` class has no attributes and doesn't process body content.

Listing 17.3 `MathTag.tld`

```

<!DOCTYPE taglib PUBLIC
    "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
    "http://java.sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
    <uri>www.manning.com/scwcd/math</uri>
    <tlib-version>1.0</tlib-version>
    <jsp-version>2.0</jsp-version>
    <tag>
        <name>sqrt</name>
        <tag-class>myTags.MathTag</tag-class>
        <body-content>empty</body-content>
        <description>
            Sends a math expression to the JSP
        </description>
    </tag>
</taglib>

```

In this example, we'll use the URI specified in the TLD instead of updating the deployment descriptor. We lose the capability of centralized library referencing, but the code is simpler and we can create the JSP directly. The JSP itself, shown in listing 17.4, tells the web container the library's URI, and then uses the empty tag to display the math statement.

Listing 17.4 math.jsp

```
<%@ taglib prefix="math" uri="www.manning.com/scwcd/math" %>
<html><body>
  <math:sqrt />
</body></html>
```

The result, shown in figure 17.2, shows that the JSP works as desired.

The square root of 289 is 17.0.

Now that you've seen how to build a basic SimpleTag-based JSP, we can add more powerful features. Next, we'll add dynamic attributes to the SimpleTag.

Figure 17.2 Static output from an empty SimpleTag instance

17.2.2 Adding dynamic attributes to SimpleTags

In the previous chapter, we showed how tags implementing Tag, IterationTag, and BodyTag process attributes with JSP 1.x. By adding a setter method for the given attribute (setXYZ () for the XYZ attribute), you can incorporate its value into your Java class. Then, you need to update the TLD to tell the web container what attributes it should accept.

This process remains the same using the simple model of tag library creation, but what if you don't know the name of the tag's attributes? What if you don't know how many there are? With JSP 1.x, you face serious problems. But JSP 2.0 provides the DynamicValues interface, which allows you to process multiple, unspecified attributes with a single method, setDynamicAttribute().

To show how this works, we're going to add static and dynamic attributes to the MathTag example. This time, the JSP will display a table of math functions whose entries are determined by the tag's attributes. Listing 17.5 updates MathTag.tld to tell the web container what kind of attributes to expect in the JSP.

Listing 17.5 MathTag.tld (Updated)

```
<!DOCTYPE taglib PUBLIC
  "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
  "http://java.sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
  <uri>www.manning.com/scwcd/math</uri>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <tag>
    <name>functions</name>
    <tag-class>myTags.MathTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
      <name>num</name>
```

```

        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    <dynamic-attributes>
        true
    </dynamic-attributes>
    <description>
        Sends a math expression to the JSP
    </description>
</tag>
</taglib>

```

The body content remains empty, but there are now two elements for attributes. The first, `<attribute>`, tells the web container about a static attribute called `num`, which is required and can be dynamically calculated at runtime. The second, `<dynamic-attributes>`, tells the web container that the tag may contain other attributes besides `num`, and it should create a `Map` to hold their names and values.

Listing 17.6 updates the `MathTag.java` code to reflect the new attribute processing. This class creates a `String` called `output` that is updated by the `setDynamicAttribute()` method. The web container calls this method each time it encounters an attribute not mentioned in the TLD. Once it finishes reading the attributes, it invokes `doTag()`, which sends the `String` to the JSP for display.

Listing 17.6 MathTag.java (Updated)

```

package myTags;

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

public class MathTag extends SimpleTagSupport
    implements DynamicAttributes
{
    double num = 0;
    String output = "";

    public void setNum(double num)
    {
        this.num = num;
    }

    public void setDynamicAttribute(String uri, String localName,
        Object value ) throws JspException
    {
        double val = Double.parseDouble((String)value);
        if (localName == "min")
        {
            output = output + "<tr><td>The minimum of "+num+" and "+
                val + "</td><td>" + Math.min(num, val) + "</td></tr>";
        }
    }
}

```

```

    }
    else if (localName == "max")
    {
        output = output + "<tr><td>The maximum of "+num+" and "+
            val + "</td><td>" + Math.max(num, val) + "</td></tr>";
    }
    else if (localName == "pow")
    {
        output = output + "<tr><td>"+num+" raised to the "+val+
            " power"+"</td><td>"+Math.pow(num, val)+"</td></tr>";
    }
}

public void doTag() throws JspException, IOException
{
    getJspContext().getOut().print(output);
}
}

```

After the web container initializes the `JspContext` and parent tag (if necessary), it processes the tag's attributes. If the attribute is static, such as `num`, it calls the setter method with the name of the attribute—`setNum()` in our example. If the attribute isn't mentioned in the TLD, then it is dynamic, and the container invokes `setDynamicAttribute()`.

Since this method does most of the work in the example, it's important to learn how it functions. It provides three items of information: a `URI String` representing the attribute's namespace, a `String` containing its name, and the `Object` containing its value. After converting the value into a double, `MathTag` continues processing according to the attribute's name. If the name is `min`, `max`, or `pow`, then `output` is updated with a new table row.

Listing 17.7 shows how this tag is coded in the JSP. Note that the static attribute, `num`, needs to be included *first*. This way, its value will be available when the rest of the attributes are processed.

Listing 17.7 `MathTag.jsp` (Updated)

```

<%@ taglib prefix="math" uri="http://www.manning.com/scwcd/math" %>
<html><body>
    Math Functions:<p>
    <table border="1">
    <math:functions num="{3*2}" pow="2" min="4" max="8"/>
    </table>
</body></html>

```

The use of HTML tables and EL may seem unnecessary. But in the exam, Sun will make the JSP code as complicated as possible. So make sure you have a solid grasp of both topics.

We can specify the `num` attribute with EL because the TLD sets its `<rtexprvalue>` tag to `true`. But the dynamic attributes don't have this option. If you try to use EL to set the values of `pow`, `min`, and `max`, you'll get an error.

Figure 17.3 shows the JSP's output.

Although the `DynamicAttributes` interface is new, you can still extend its usage to the classic `Tag`, `IterationTag`, and `BodyTag` classes. But as we've shown, building `SimpleTags` requires less code and complexity. Let's finish our discussion on this topic by looking at how `SimpleTags` process body content.

Math Functions:	
6.0 raised to the 2.0 power	36.0
The minimum of 6.0 and 4.0	4.0
The maximum of 6.0 and 8.0	8.0

Figure 17.3 Dynamic output from a `SimpleTag` implementing `DynamicAttributes`

17.2.3 Processing body content inside `SimpleTags`

In the classic model, `BodyTag` classes acquire the text and code inside their JSP tags by invoking `getBodyContent()`. This returns a `BodyContent` object that can be converted into a `String` or a `Reader`. This means that you can parse through the body and alter it if needed.

These options aren't available with `SimpleTags`. If you want to access the body, the `getJspBody()` method will return a `JspFragment`. This object only has two methods. The first, `getJspContext()`, returns the `JspContext` associated with the fragment. The second, `invoke()`, executes the JSP code and directs its output to the `JspWriter`. Neither method allows you to access and manipulate the body's contents as you can in the classic model.

Further, a `SimpleTag`'s body must not contain scripts—no declarations, expressions, or scriptlets. So it is invalid for a `SimpleTag`'s tag library descriptor to specify its `<body-content>` as `JSP`. Therefore, if you want to process a `SimpleTag`'s body, you need to set its `<body-content>` to `tagdependent` or `scriptless`. This is an important constraint to remember.

Because `SimpleTag` development doesn't add any new capabilities for processing body content, we will present code snippets instead of a new example. The code below shows how the `doTag()` method acquires the `SimpleTag`'s body and directs it to the JSP for display:

```
public void doTag() throws JspException, IOException
{
    getJspContext().getOut().print(output);
    getJspBody().invoke(null);
}
```

Note that the `invoke()` method requires an argument specifying the `JspWriter` that will receive the `JspFragment`'s output. In this case, the null argument directs the output to the `JspWriter` returned by `getJspContext().getOut()`.

As shown here, the only change required in the TLD is the `<body-content>`. Since JSP is invalid and empty is erroneous, we'll set the value to `tagdependent`:

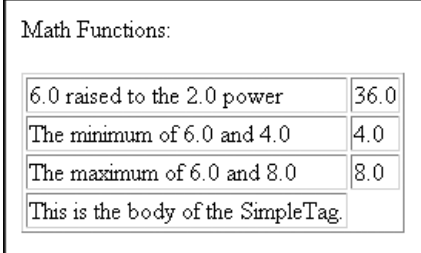
```
<tag>
  <name>functions</name>
  <tag-class>myTags.MathTag</tag-class>
  <body-content>tagdependent</body-content>
  ...
</tag>
```

For the JSP, we'll add a new row to the table by including it as body content. This is shown in the code that follows. Because the `SimpleTag` executes the `JspFragment` last, this will be the last row of the table.

```
<math:functions num="{3*2}" pow="2" min="4" max="8">
  <td>This is the body of the SimpleTag.</td>
</math:functions>
```

Figure 17.4 shows the output of the new JSP.

Although the `SimpleTag` class reduces the amount of Java needed to create custom tag libraries, it still requires building and compiling Java classes. To the creators of the JSP 2.0 specification, this is still too much work. So they came up with an even simpler way of building tag libraries. With *tag files*, you don't need TLDs or Java at all! In the next section, we'll see how this new method works.



Math Functions:	
6.0 raised to the 2.0 power	36.0
The minimum of 6.0 and 4.0	4.0
The maximum of 6.0 and 8.0	8.0
This is the body of the SimpleTag.	

Figure 17.4 Output updated with `SimpleTag` body content

Quizlet

- Q:** What is the main difference between a TLD for `SimpleTags` and a TLD for a classic `Tag`?
- A:** `SimpleTag` TLDs cannot set their `<body-value>` elements equal to JSP. This is because a `SimpleTag` cannot process script elements in body content.

17.3 CREATING JAVA-FREE LIBRARIES WITH TAG FILES

JSTL and EL reduce the amount of Java in a JSP and `SimpleTags` reduce the amount of Java in a tag handler. But tag files remove the need for Java programming altogether. As long as you understand the JSP syntax, you can now build custom tags for your pages.

We'll begin our discussion of tag files with a simple example. Next, we'll cover the directives that enable you to communicate information to the web container. Finally, we'll look at fragments and how they are processed with tag file actions.

17.3.1 Introducing tag files

At its simplest, a tag file is a file made up of JSP code that has a `.tag` or `.tagx` extension. It can include EL expressions, directives, and custom and standard tags. Unlike `SimpleTag` JSPs, tag files can also contain script elements. In fact, the only JSP elements that can't be used in tag files are page attributes.

To see how tag files work, let's start with a simple example. Listing 17.8 presents `example.tag`, which displays a sequence of six numbers.

Listing 17.8 `example.tag`

```
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<c:forTokens items="0 1 1 2 3 5" delims=" " var="fibNum">
  <c:out value="${fibNum}"/>
</c:forTokens>
```

Listing 17.8 contains just regular JSP code using the JSTL `forTokens` action and an EL expression. The JSP presented in listing 17.9 accesses this tag and displays its output.

Listing 17.9 `example.jsp`

```
<%@ taglib prefix="ex" tagdir="/WEB-INF/tags" %>
<html><body>
  The first six numbers in the Fibonacci sequence are:
  <ex:example/>
</body></html>
```

The code may look trivial, but this new capability is important. The key is simplicity. You don't need a background in Java to create custom tags with tag files. You don't have to compile Java classes and keep track of their packages. You don't even need tag library descriptors. Thanks to the JSP 2.0 specification, any developer of presentation logic can now create a custom tag library.

The new specification also makes it simple to integrate tag files within a JSP. There are only two steps:

- 1 Add a `taglib` directive to the JSP with a `prefix` attribute and the `tagdir` attribute equal to `/WEB-INF/tags`.
- 2 Place a tag containing the prefix and the name of the tag file (without the extension) wherever you want the file's JSP code invoked.

This brings up an important question. If Java-based tags need TLDs to locate their classes, how do these tags locate their tag files? To answer this, we need to look at how the web container accesses and processes tag files.

17.3.2 Tag files and TLDs

In the example JSP above, the `tagdir` attribute is set to `/WEB-INF/tags`. This is necessary since the web container automatically looks there for tag files. Then, the container builds an implicit tag library and TLD for this directory and each subdirectory beneath it. The good news is that you don't have to create TLDs for tag files. The bad news is that your tag files *must* be in `/WEB-INF/tags/` or a subdirectory.

But if you deploy your tag files inside a JAR, the situation changes. In this case, you need to create a tag library descriptor for your files. This TLD is similar to regular TLDs, but instead of matching tags to tag handlers, it matches names of tag files to their paths.

To make this possible, tag file TLDs use `<tag-file>` elements in place of `<tag>` elements. The definition of a `<tag-file>` element is as follows:

```
<!ELEMENT tag-file (description?, display-name?,  
                    icon?, name, path, example?, tag-extension?) >
```

The only necessary subelements are `<name>`, which specifies the tag file name without its suffix, and `<path>`, which specifies the file's path from the archive's root. Therefore, `<path>` must begin with `/META-INF/tags`. Here is an example TLD for an archived tag file:

```
<taglib>  
...  
<uri>www.manning.com/scwcd/example</uri>  
<tag-file>  
    <name>example</name>  
    <path>/META-INF/tags/example.tag</path>  
</tag-file>  
</taglib>
```

This TLD must be located in the `META-INF` directory and the tag file(s) must be placed in `META-INF/tags` or a subdirectory. An example directory structure is shown here:

```
META-INF/  
example.tld  
tags/  
example.tag
```

Since the tag file isn't located in or under `/WEB-INF/tags`, you can't use the `tagdir` attribute in the `taglib` directive. Instead, you need to specify the TLD's URI (`www.manning.com/scwcd/example`) using the `URI` attribute. For this example, the following JSP directive will tell the web container where to find the tag file's TLD:

```
<%@ taglib prefix="ex" uri="www.manning.com/scwcd/example" %>
```

Other important differences between tag file TLDs and tag TLDs concern the `<attribute>` and `<body-content>` elements. Tags furnish this information in their TLDs, but tag files can't. Instead, tag files use a special set of directives. They tell the web container how to process the tag file, and it is important to understand how they work.

17.3.3 Controlling tag processing with tag file directives

JSPs contain three different kinds of directives: `page`, `taglib`, and `include`. Tag files remove the `page` directive and add three more. The first, `variable`, creates and initializes a variable for use in tag processing. The second, `tag`, tells the web container how to process the tag file. The third, `attribute`, describes the attributes that can be used in the tag. We'll investigate each of these, and provide snippets of example code.

Creating JSP variables with the variable directive

In the previous chapter, we showed how to declare JSP variables in tag library descriptors by adding `<variable>` elements. Then, you can assign and display the variable with JSTL actions and EL expressions.

Tag files provide a similar capability with the `variable` directive. The attributes of this directive are the same as the `<variable>` subelements, using `scope` to define the variable's visibility, and `name-given` and `name-from-attribute` to provide the variable's name. The only difference is the `alias` attribute, which provides a local name for the variable when its real name is determined by an attribute value (using `name-from-attribute`).

As an example, if the tag file contains the directive

```
<%@ variable name-given="x" %>
```

then the JSP can set the variable's value with the JSTL action

```
<c:set var="x">
  Hooray!
</c:set>
```

and display this value inside the JSP with `${x}`.

The important point about the `variable` directive is that you don't need to rely on script declarations to declare variables in a JSP. But this is a minor function. The `tag` directive accomplishes much more.

Using the tag directive in tag files

The first new directive, `tag`, works like the `page` directive in a JSP. It provides the web container with settings that apply to the entire file. Table 17.3 describes the attributes that can be specified within a tag file's `tag` directive.

Table 17.3 Tag file attributes within the `tag` directive

Name	Description
<code>body-content</code>	Similar to the TLD subelement—can be empty, tagdependent, or scriptless. Set to <code>scriptless</code> by default.
<code>description</code>	Optional String statement describing the tag file.
<code>display-name</code>	String used by XML tools. Set to the name of the tag file (without the extension by default.
<code>dynamic-attributes</code>	Tells the container to create a named Map to hold unspecified attributes and their values.
<code>example</code>	String providing an instance of the tag's usage.
<code>import</code>	Adds a class, interface, or package to the tag processing.
<code>isELIgnored</code>	Specifies whether EL constructs will be ignored.
<code>language</code>	Sets the programming language used in the tag file. "Java" by default.
<code>large-icon</code>	Path to the large image representing the tag.
<code>page-encoding</code>	Specifies the character encoding of the tag file.
<code>small-icon</code>	Path to the small image representing the tag.

One attribute that has no JSP counterpart is `dynamic-attributes`. This works like the TLD `<dynamic-attributes>` subelement, but instead of directing attributes to a Java method, the web container updates a local variable specified by the directive. For example, the tag file that follows uses the `tag` directive to send dynamic attribute data to `attrib`. This data is then displayed using the JSTL `forEach` action.

```
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@ tag dynamic-attributes="attrib" %>
<c:forEach items="${attrib}" var="att">
    ${att.value}<br>
</c:forEach>
```

The JSP code shown next accesses this tag file (`dynatt.tag`) and sets the names and values of the tag's attributes. When the JSP is invoked, it will display a list of these values.

```
<%@ taglib prefix="dyn" tagdir="/WEB-INF/tags" %>
<html><body>
    <dyn:dynatt first="first" second="second" third="third"/>
</body></html>
```

Now that you've learned how to specify dynamic attributes in tag files, it's important to understand how to add static attributes. This requires the `attribute` directive.

Adding static attributes with the `attribute` directive

Dynamic attributes provide flexibility, but if you already know your tag's attributes, you can inform the web container in advance with static attributes. Traditional tags have `<attribute>` subelements in TLDs for this purpose. But to set attributes in tag files, you need `attribute` directives.

The attributes associated with the `attribute` directive are similar to the subelements of the TLD's `<tag>` element. The `name` attribute provides identification, `required` informs the web container whether the attribute must be present, and `rtexprvalue` tells the container that the attribute's value can be determined at runtime.

A brief example will show how `attribute` directives are used in tag files. The following tag file snippet sends output to the JSP according to the value of `x`.

```
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@ attribute name="x" required="true" %>
<c:choose>
  <c:when test='${x == "yes"}'>
    Yippee!
  </c:when>
  <c:otherwise>
    Rats!
  </c:otherwise>
</c:choose>
```

Then, the following JSP snippet uses the tag file and sets its attribute:

```
<%@ taglib prefix="attr" tagdir="/WEB-INF/tags" %>
<html><body>
  <attr:statatt x="yes" />
</body></html>
```

The `attribute` directive also allows you to insert JSP code into static attributes by setting its `fragment` attribute equal to `true`. However, to process this fragment, you need to look outside tag file directives, and concern yourself with standard actions.

17.3.4 Processing fragments and body content with tag file actions

JSPs provide a set of standard actions to direct the web container's processing of the page. Tag files can use all of these, and provide two more. The first, `jsp:invoke`, makes use of the fragment declared in the `attribute` directive. The second, `jsp:doBody`, processes the tag's body content.

Manipulating fragments with the `jsp:invoke` action

`SimpleTag` classes retrieve body content by calling `getJspBody()`, which returns a `JspFragment`. Then, to direct the fragment's output to a `JspWriter`, the tag handler calls the fragment's `invoke()` method. This method's argument determines which `JspWriter` object will receive the fragment's output.

The `jsp:invoke` action performs essentially the same function as `invoke()`, but is used for attributes declared as fragments, not for body content. Also, this action can do more with the `JspFragment` than just directing it to a `JspWriter`. It can convert the fragment to a `String` or a `Reader` object. However, just as with `SimpleTags`, tag files *cannot* process script elements (declarations, expressions, scriptlets) inside body content.

Table 17.4 lists and describes the attributes needed to configure this action in tag files.

Table 17.4 Tag file attributes within the tag directive

Name	Description
<code>fragment</code>	Identifies the <code>JspFragment</code> for processing.
<code>var</code>	Name of the <code>String</code> used to contain the <code>JspFragment</code> . Cannot be used with <code>varReader</code> .
<code>varReader</code>	Name of the <code>Reader</code> used to contain the <code>JspFragment</code> . Cannot be used with <code>var</code> .
<code>scope</code>	Scope of the stored <code>JspFragment</code> . Must be <code>page</code> , <code>request</code> , <code>session</code> , or <code>application</code> .

Of these attributes, only `fragment` is required in the `jsp:invoke` action. If neither `var` nor `varReader` is set, then the `JspFragment` will be directed to the default `JspWriter`. If one of `var` or `varReader` is set, but `scope` isn't, then the fragment's scope will be set to `page`.

Listing 17.10 presents an example tag file that uses the `jsp:invoke` action. First, it specifies a required attribute named `frag`, whose value will be contained in a `JspFragment`. Then, depending to the value of `proc`, it returns the fragment to the JSP as a `String` variable.

Listing 17.10 `invokeaction.tag`

```
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@ attribute name="frag" required="true" fragment="true"%>
<%@ attribute name="proc" required="true" %>
<c:if test='${proc} == "yes"'>
    <jsp:invoke fragment="frag"/>
</c:if>
```

Listing 17.11 presents the JSP needed to test this tag file. First, it incorporates the tag and sets its `proc` attribute to `yes`. Then, using the `<jsp:attribute>` action, it specifies a line of JSP code to serve as the value of the `frag` attribute.

Listing 17.11 `invokeaction.jsp`

```
<%@ taglib prefix="inv" tagdir="/WEB-INF/tags" %>
<html><body>
  <inv:invokeaction proc="yes">
    <jsp:attribute name="frag">
      Two + two = ${2+2}
    </jsp:attribute>
  </inv:invokeaction >
</body></html>
```

So far, you've seen all there is to know about setting and processing tag file attributes. Now, let's see how tag files make use of the information between the tags. To enable you to process this body content, tag files provide the `<jsp:doBody>` action.

Processing body content with the `jsp:doBody` action

The `jsp:doBody` action works like `jsp:invoke`, but it receives the tag's body instead of a fragment attribute. It contains the same attributes as `jsp:invoke`, except `fragment`. So, when a tag file receives body content, it can manipulate it in three ways: display it with the default `JspWriter`, send it to a variable with the `var` attribute, or store it as a `Reader` object with the `varReader` attribute.

The tag file in listing 17.12 processes the tag's body content according to the `att` attribute. When `att` equals `"var"`, it will be stored within a variable, and when `att` equals `"reader"`, it will be stored in a `Reader` object. If `att` isn't specified, the default `JspWriter` will display its output.

Listing 17.12 `bodyaction.tag`

```
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@ attribute name="att" required="true" %>
<c:choose>
  <c:when test='${att == "var"}'>
    <jsp:doBody var="bodyvar" scope="application"/>
  </c:when>
  <c:when test='${att == "reader"}'>
    <jsp:doBody varReader="bodyReader" />
  </c:when>
  <c:otherwise >
    <jsp:doBody />
  </c:otherwise>
</c:choose>
```

The JSP in listing 17.13 performs two tasks. First, it accesses the tag file and sets the `att` attribute to “var.” Then, using EL, it displays the variable containing the tag’s body content.

Listing 17.13 `bodyaction.jsp`

```
<%@ taglib prefix="bod" tagdir="/WEB-INF/tags" %>
<html><body>
  <bod:bodyaction att="var">
    This is the tag body.
  </bod:bodyaction >
  ${bodyvar}
</body></html>
```

In an earlier chapter, we showed how JSPs (*.jsp) can be converted into JSP documents (*.jspx) by using well-formed XML. The process of creating tag file documents (*.tagx) is very similar. The main task involves replacing tag file directives, such as `<%@ attribute ... %>`, with XML statements, such as `<jsp:directive.attribute ... />`.

This discussion ends our treatment of JSPs in general and tag library development in particular. As you can see, there are many different methods of creating tag libraries. If you are familiar with Java, you may want to use the simple model, but if you need to incorporate scripts, the classic method may be best. But if you prefer building tags with JSP code, you can’t do better than to use the tag files described here.

Quizlet

- Q:** In what directory should you place unarchived tag files? What directory for archived tag files?
- A:** All unarchived tag files should be placed in `/WEB-INF/tags` or a subdirectory underneath. All archived tag files should be placed in `/META-INF/tags` or a subdirectory.

17.4 SUMMARY

One of Sun’s primary goals in releasing JSP 2.0 was to simplify JSP development. To reduce the amount of Java in JSPs, they created Expression Language. To reduce the amount of code in tag handlers, they came up with the `SimpleTag` interface. Finally, to remove the need for tag handlers and TLDs altogether, they introduced tag files.

The advantages of `SimpleTags` over `Tags`, `BodyTags`, and `IterationTags` stem from their less-complex life cycles. With `SimpleTags`, there are no elaborate flowcharts or multiple event-based tag handler methods. After the web container performs its initialization, it only invokes one method, `doBody()`, which performs all of the `SimpleTag`’s processing. JSP 2.0 also provides the `SimpleTagSupport` adapter class for additional capabilities.

The drawback to `SimpleTag` operation involves its processing of body content. The body content is encapsulated in a `JspFragment` object, which cannot contain script elements such as declarations, expressions, or scriptlets. Further, `JspFragments` have no built-in mechanisms for converting body content into `Strings` or `Readers`.

Tag files are a fascinating addition to the traditional methods of tag library development. By specifying a precise directory for tag file location, the new JSP specification removes the need for tag library descriptors. Tag files still provide all of the information normally contained in TLDs, but they use directives and actions instead. The tag directive resembles the JSP's `page` directive, and `attribute` and `variable` resemble their corresponding TLD elements. Finally, the `jsp:doBody` and `jsp:invoke` actions allow you to process JSP code in the tag body and tag attributes, respectively.

17.5 REVIEW QUESTIONS

1. What method should you use in a `SimpleTag` tag handler to access dynamic variables?
 - a `doTag()`
 - b `setDynamicAttribute()`
 - c `getParent()`
 - d `setDynamicParameter()`
2. Which object does a `SimpleTag` tag handler use to access implicit variables?
 - a `PageContext`
 - b `BodyContent`
 - c `JspContext`
 - d `SimpleTagSupport`
3. Consider the following TLD excerpt:

```
<body-content>
  empty
</body-content>
<attribute>
  <name>color</name>
  <rtexprvalue>true</rtexprvalue>
</attribute>
<dynamic-attributes>
  true
</dynamic-attributes>
```

If the name of the tag is `tagname` and its prefix is `pre`, which of the following JSP statements is valid?

- a `<pre:tagname color="yellow" size=${sizenum} />`
- b `<pre:tagname size="18" color="red"> </pre:tagname>`

```

c <pre:tagname color="{colorname}" size="22" font="verdana"></pre:tagname>
d <pre:tagname color="green" size="30">font="Times New Roman"</pre:tagname>
e <pre:tagname color="{colorname}" size="18"></pre>

```

4. If placed inside the body of a simple tag, which of the following statements won't produce "9"? (Select one)

```

a ${3 + 3 + 3}
b "9"
c <c:out value="9">
d <%= 27/3 %>

```

5. Which of the following methods need to be invoked in a SimpleTag to provide iterative processing? (Select one)

```

a setDynamicAttribute()
b getParent()
c getJspBody()
d doTag()
e getJspContext()

```

6. Which of the following values is invalid inside a SimpleTag's <body-content> subelement? (Select one)

```

a JSP
b scriptless
c tagdependent
d empty

```

7. Which of the following is a valid return value for the SimpleTag's doTag() method? (Select one)

```

a EVAL_BODY_INCLUDE
b SKIP_BODY
c void
d EVAL_PAGE
e SKIP_PAGE

```

8. Which tag file directive makes it possible to process dynamic attributes?

```

a taglib
b page
c tag
d attribute

```

9. Which of the following statements can't be used to access a tag file from a JSP? (Select one)
- a** `<%@ taglib prefix="pre" uri="www.mysite.com/dir/" %>`
 - b** `<%@ taglib prefix="pre" tagdir="/WEB-INF/tags" %>`
 - c** `<%@ taglib prefix="pre" tagdir="/WEB-INF/tagfiles" %>`
 - d** `<%@ taglib prefix="pre" tagdir="/WEB-INF/tags/myDirectory" %>`
10. Which tag file action processes JspFragments in tag attributes?
- a** `taglib`
 - b** `jsp:invoke`
 - c** `tag`
 - d** `jsp:doBody`
 - e** `attribute`
11. Which JspFragment method is used to process body content in a SimpleTag? (Select one)
- a** `invoke()`
 - b** `getOut()`
 - c** `getJspContext()`
 - d** `getBodyContent()`
12. Which class provides an implementation of the `doTag()` method? (Select one)
- a** `TagSupport`
 - b** `BodyTagSupport`
 - c** `SimpleTagSupport`
 - d** `IterationTagSupport`
 - e** `JspTagSupport`
13. In what directory shouldn't you place tag files? (Select one)
- a** `/META-INF/tags/tagfiles`
 - b** `/WEB-INF/`
 - c** `/WEB-INF/tags/tagfiles/tagdir/taglocation`
 - d** `/META-INF/tags/`
14. Which type of object is returned by `JspContext.getOut()`? (Select one)
- a** `ServletOutputStream`
 - b** `HttpServletOutputStream`
 - c** `JspWriter`
 - d** `BodyContent`

15. Which of the following methods does the web container call first to initiate a SimpleTag's life cycle?
- a** `setJspContext()`
 - b** `setParent()`
 - c** `getJspContext()`
 - d** `getJspBody()`
 - e** `getParent()`

SCWCD Exam Study Kit **SECOND EDITION**

Java Web Component Developer Certification

H. Deshmukh • J. Malavia • M. Scarpino

With the tremendous penetration of J2EE in the enterprise, passing the Sun Certified Web Component Developer exam has become an important qualification for Java and J2EE developers. To pass the SCWCD exam (Number: 310-081) you need to answer 69 questions in 135 minutes and get 62% of them right. You also need \$150 and this (completely updated and newly revised) book.

In its first edition, the *SCWCD Exam Study Kit* was the most popular book used to pass this most desirable web development certification exam. The new edition will help you learn the concepts—large and small—that you need to know. It covers the newest version of the exam and not a single topic is missed.

The SCWCD exam is for Sun Certified Java Programmers who have a certain amount of experience with Servlets and JSPs, but for those who do not, the book starts with three introductory chapters on these topics. Although the *SCWCD Exam Study Kit* has one purpose, to help you get certified, you will find yourself returning to it as a reference after passing the exam.

What's Inside

- Expression Language
- JSP Standard Tag Library (JSTL 1.1)
- Custom tags—'Classic' and 'Simple'
- Session management
- Security
- Design patterns
- Filters
- Example code and the Tomcat servlet container
- All exam objectives, carefully explained
- Review questions and quizlets
- Quick Prep section for last-minute cramming



The authors, *Deshmukh*, *Malavia*, and *Scarpino*, are Sun Certified Web Component Developers who have written a focused and practical book thanks to their extensive background in Java/J2EE design and development. They live, respectively, in Iselin, New Jersey, Ardsley, New York, and Austin, Texas.



Ask the Authors



Ebook edition

www.manning.com/deshmukh2



9 781932 394382



54995

ISBN 1-932394-38-9