A gentle introduction to the JSP Standard Tag Library

JSTL IN ACTION

Shawn Bayern

MANNING

JSTL in Action

JSTL in Action

SHAWN BAYERN

MANNING

Greenwich
(74° w. long.)

For my future wife and kids,
who, when I meet and conceive them,
respectively,
will likely be my love and my inspiration

This book will show you how to make the most of JSTL. It begins without assuming you know anything more than HTML, and it gently introduces you to all the principles you'll need to produce flexible, powerful web pages. The goal of this book isn't to satisfy my own ego by showing you how subtle and tricky technology can be, but instead to equip you to handle any JSTL-related issue that arises when you produce real-world, dynamic web sites. If you read an example in this book and think, "I didn't realize it could be so easy," then JSTL has done its job—and so have I.

names (for HTML, XML, or JSTL tags), tag attributes, scoped-variable names, and other words that normally appear within code.

- *Tables for tags*

  Just like HTML tags, JSTL tags have *attributes* that let you modify the tags' behavior. For instance, in the tag `<fmt:formatNumber type="currency"/>`, the text `type="currency"` is an attribute. I've listed tag attributes in tables that have a consistent format. Here's an example:

  `<c:spam>` tag attributes

Attribute	Description	Required	Default
`email`	Email address to send junk email to	Yes	*None*
`subject`	Subject of the junk-email message	No	`"Long distance service for less."`
`message`	Body of the junk-email message	No	*Body*

  This sample table shows a few things. First, tables for tag attributes have a "tag" icon to help you find them. Such tables have four columns describing the attribute name, a brief description of each attribute, information about whether the attribute must be specified for each use of the tag, and information about the default value of the attribute if you don't specify a value. If the Default column contains *None*, the attribute has no default. If this column contains *Body*, the default value comes from the tag's body. (See chapter 2 for more information about tags, attributes, and bodies. Note that `<c:spam>` is, of course, not a real JSTL tag—although given the number of applications that send out junk mail, there's clearly a need for it; perhaps we'll see it in JSTL 1.1.)

- *Highlighting*

  I highlight sections of code samples whenever I feel like it, usually to draw your attention to a part of the code sample that has changed. Highlighting isn't consistent; it's there only when I think it will be useful.

- *Code annotations*

  Some longer examples are annotated using bullets like this: ❽. These are often tied to paragraphs that follow and amplify the code.

- *Call-out boxes*

  Occasionally, I draw your attention to a Note, Tip, or Warning using a noticeable box in the middle of the page. To be honest, I do this just because other books do it; fortunately, I use these boxes sparingly.

Part 1

Background

Welcome to *JSTL in Action*, a guide to everything you'll need to know about JSTL. In the first part of this book, we explore what JSTL is and how it works. We start by discussing the simple ideas behind *dynamic content* on the Web.

After that, we look at some of the differences between HTML and XML. This topic is important because JSTL uses an XML-like syntax, so you'll need to be aware of its rules. Toward the end of part 1, we also discuss the basics of JavaServer Pages (JSP), the broader language that JSTL is based on.

Part 1 takes for granted only a basic knowledge of HTML. This book is designed to be a gentle but complete introduction to JSTL, and it doesn't assume you're familiar with any other programming or web-design languages. Part 1 lays a foundation so that you have all the tools you need to jump in and begin designing dynamic web pages.

1

Dynamic web sites

This chapter covers...

- Ideas behind dynamic web content
- What JSTL looks like
- Requirements for running JSTL
- JSTL's role in web applications

To conduct a mail merge and print a customized letter, you supply the information missing from this single master copy of the letter—perhaps at the prompting of your word processor, or as a preformatted, comma-separated text file. To be complete, each letter needs four pieces of information: `NAME`, `DOLLARS`, `PRESENT`, and `APPENDAGE`. Like the old *Mad-Libs* games, producing a customized letter simply involves filling in these placeholders. One set of legitimate values might be

```
Jack, 20, tuna sandwich, finger
```

Another might be

```
Leonard, 1200, television, arm
```

You'd use the mail merge in the first place because doing so is simpler than typing each letter manually—or even using a word processor to edit the letter yourself each time you need a new, customized copy.

Believe it or not, template languages for the Web work almost exactly the same way. Starting with a web-development language is no harder than using mail merge. The major difference is that instead of printing simple text letters or documents, the goal of a web-design language is usually to print HTML. For instance, here's what our sample mail-merge letter might look like in JSTL:

```
<html>
<head>
  <title>Nasty letter</title>
</head>
<body>
<h1>Dear <c:out value="${name}"/>:</h1>

<p>
  My records show that you owe me $<c:out value="${dollars}"/>.
  I need this money now to buy myself a big
  <c:out value="${present}"/>. If I don't get it,
  I will break your <c:out value="${appendage}"/>.
</p>
</body>
</html>
```

NOTE In this example, and throughout the rest of this book, I use **bold** type to highlight JSTL tags that occur within HTML text. This formatting makes it easier to differentiate the dynamic parts of a page from its static, template text.

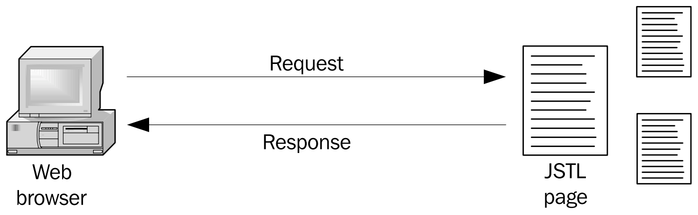


Figure 1.4 Small applications can be designed entirely using JSTL pages. Web browsers load the pages directly, and the pages know how to find all the information that they need to print.

design, the JSTL page does all the work. That is, it knows how to find all the data it needs to print, without any help from back-end Java code.

In contrast with figure 1.4's simple design, consider figure 1.5. The web browser makes a request for a web page, but this request is handled by a *servlet*, which is a web program written in the Java programming language. In order to handle this request, the servlet can interact with other Java code, as well as databases, directories, XML files, messaging systems, and nearly anything else. Finally, once the servlet has decided what it wants to display to the user, it *forwards*—that is, hands off—the request to a JSTL page, which decides how to print out the information.

One key principle of this model is that each JSTL page is designed to do a different thing. For example, one JSTL page might be written to print a shopping cart to cell phones using WML. Another would be designed to present a registration page for new users in HTML. The pages themselves don't decide what task to perform; they only decide what to display. The servlet takes care of all the behind-the-scenes action, which might include determining what kind of device the user's using (cell phone versus web browser) and what the user is asking for (shopping cart or registration page).

Organizing an application as shown in figure 1.5 has a number of benefits. Doing so supports division of labor in your organization, much like traditional division of labor in a factory assembly line. If you work for a large organization, you probably have a number of different kinds of colleagues: programmers, web-page authors, graphics designers, database administrators, and so on. Separating the pieces of your application into different blocks—a servlet, plain Java code, a database, JSTL pages—means that all the people in your organization can focus on what they do best.

This division of labor also makes a site more maintainable. Before template systems, it was common to include HTML in the middle of conventional programs, like this:

XML is an approach for using these tags to mark off information within a document.

XML, unlike HTML, does not describe a particular set of tags (`<p>`, `<b>`, and so forth) or relationships between such tags. Instead, it describes the rules for using tags in a document in the first place. To draw a loose analogy, XML is a general-purpose mechanism, like Arabic numerals—1, 2, 3, and so on. Receiving a group of Arabic numerals in isolation doesn't tell you much; for example, seeing "79" on a blank page doesn't convey any useful information without a context. However, you know that "79" is a valid string containing just Arabic numerals, and that "g", "49E", and "©" are not.

Similarly, the `<beef>` tag in the previous code snippet doesn't mean anything in isolation. In fact, neither does a tag like `<img src="ugly-man.jpg"/>`. This latter tag has a meaning when it appears in an HTML document, but alone, it is simply an arbitrary tag, just as "79" is an arbitrary string of digits. Nonetheless, it follows XML's rules, so it is a well-formed, recognizable XML tag, whereas

```
[am-I-an-XML-tag?]
```

is not. In a moment, we'll look more at XML's rules.

If you've browsed discussion groups online or exchanged email with enough people, you've probably seen informal uses of tags beyond HTML. For instance, I've often seen people mark off a particularly vibrant part of an email message with tags like `<rant>` and `</rant>`, or introduce a long, rambling section with a `<ramble>` tag. This pseudo-HTML markup, insofar as it technically adds structure to a document, represents the essential goal of XML: tags are used to mark a document in ways that help people and programs identify the purpose of each part of that document.

Jumping right in, we'll first look at some of the jargon used to describe XML tags and their relationships. Then we'll follow up with some syntactic rules of XML.

2.1.1 A dose of tag terminology

When we talk about JSTL, it's important to make sure we're on the same page (so to speak). To ensure this, one of the less glamorous things we need to do is cover some XML terminology. We'll also explain the terms and idioms that are used most commonly by JSP and JSTL users.

As you probably know from your experience with HTML, tags often come in pairs: one tag, which might look like `<p>`, starts a block; and a corresponding tag, such as `</p>`, ends it. Figure 2.1 shows an example of an XML *element*—a block of XML between, and including, corresponding start and end tags. The element begins with a *start tag*, optionally contains a *body* (some inner text, tags, or both), and wraps up with an *end tag*.

Table 2.1 Some relevant rules of XML syntax, with examples of violating and compliant markup *(continued)*

Rule	HTML example (violating rule)	XHTML example (following rule)
Empty elements must be closed	`<br>`	`<br/>`

A few straightforward rules

Most of these rules are self-explanatory. When writing plain HTML, you can be somewhat sloppy without causing any problems. When constructing a list, you can start a list item with `<li>` but neglect to end it with `</li>`. You can mix uppercase and lowercase freely. And, you can leave off quotation marks in tag attributes (modifiers within a tag) in most cases.

You can still do all these things when you use JSTL, as long as you're just trying to produce HTML pages and not strict XML pages. However, no matter how you use JSTL tags, you need to introduce them into your page following the rules in table 2.1. For instance, your document's `<a>` tags can be written as `<A>`, and you don't need to explicitly end all your HTML tags—but your JSTL tags must have their attributes quoted and must appear in the proper case.

Empty tags must be closed

The final rule in table 2.1 is one of the more confusing to HTML authors starting out with XML or JSTL. In well-formed XML, every tag that's meant to be empty must be closed immediately, using either the longhand form shown earlier (`<br></br>`) or the vastly more common shorthand (`<br/>`). Again, if you're producing loose HTML with your JSTL pages, you don't have to worry about your `<br>` tags. But if you introduce an empty JSTL tag—for instance, `<c:out>`—into your page, you need to close it or use the shorthand empty-tag syntax.

TIP If you are trying to produce well-formed XHTML pages, instead of loosely structured HTML documents, you might run into a problem. Some older browsers aren't smart enough to recognize empty tags like `<br/>` or `<hr/>`. They expect the loose form of HTML, where the tag is not necessarily closed. In such cases, you can use the expanded form (`<br></br>`). Often, to avoid this cumbersome syntax, you can simply insert a space between the `<br` and the `/>`; many browsers (even the older ones) can handle this correctly. Thus, tags end up looking like `<br />` or `<img src="uglier-man.jpg" />`—note the spaces before the `/>`.

Now, suppose b.jsp contains the following text:

```
Welcome to b.jsp.
```

Then a.jsp will output the following:

```
Welcome to a.jsp.
Now including b.jsp . . .
Welcome to b.jsp.
```

The contents of b.jsp have replaced the `<jsp:include>` tag in page a.jsp.

Note that, because the `<jsp:include>` tag (as used here) does not contain a body, it is closed by placing a forward slash before the closing angle bracket. As we described earlier, JSP tags—which follow XML syntax—need to be closed in this fashion if they are empty.

The `<jsp:include>` tag can only include local files—files from the same JSP engine servicing the page in which `<jsp:include>` appears. Either static or dynamic files can be included. That is, the tag can include a simple text file, another JSP page, or even a servlet or other arbitrary resource on the local server.

WARNING If you are an experienced designer of web applications, you might have used the HTML `<base>` tag. This tag allows you to specify a location that all tags like `<a>` and `<img>` will use as their base. That is, if you specify a new base with

```
<base href="http://www.jstlbook.com/"/>
```

then a tag like `<img src="image.jpg"/>` will cause the browser to try to load http://www.jstlbook.com/image.jpg, not the local image.jpg file in the same directory as the web page.

The `<base>` tag, however, does not affect the way that JSP tags like `<jsp:include>` operate. To a JSP engine, the `<base>` tag is arbitrary HTML. `<base>` has its effect because the browser interprets it and uses it to modify the way the rest of the page loads. But JSP engines do not interpret HTML tags; they simply pass them through to the browser. Therefore, although it makes sense to think of `<jsp:include>` as finding files in a manner similar to `<a>` and `<img>`, the analogy is not perfect. `<jsp:include>` always looks for files on the local server.

A typical pattern is to use `<jsp:include>` to include header and footer text in multiple pages. For instance:

```
<jsp:include page="header.jsp"/>
Page contents
<jsp:include page="footer.jsp"/>
```

Table 2.3 Before you can use a tag library, you need to import it. You can use the following lines to import each JSTL library into your page. For each page, you only need to import the libraries you actually use, although there's no harm in importing all of them.

JSTL tag library	`<%@ taglib %>` directive
Core	`<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>`
XML	`<%@ taglib prefix="x" uri="http://java.sun.com/jstl/xml" %>`
Formatting	`<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>`
Database	`<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql" %>`

In this book, I won't always show the `<%@ taglib %>` directive every time I give you a short example of a JSTL tag. However, you'll need to include these directives if you plan to run the tag. (All source code available from the Manning web site includes the appropriate directives, as do this book's longer examples.)

2.2.4 Other JSP directives

In addition to `<%@ taglib %>`, JSP has two other directives that are worth looking at quickly. As you just saw, directives are pseudo-tags that have special meaning to the container; they are not passed through to the browser but, instead, are processed by the JSP engine. This section is, by necessity, somewhat technical; you will not miss much if you skip it and come back to it later.

The <%@ include %> directive

Earlier in this chapter, you saw how to include other pages using the `<jsp:include>` tag. JSP also has a directive that lets you include other files: `<%@ include %>`. It takes a `file` attribute corresponding to a relative path, similar to the `<jsp:include>` tag. For instance, to include b.jsp from a.jsp, you could use a directive like

```
<%@ include file="b.jsp" %>
```

Why have two mechanisms to include data? The difference between the two is somewhat subtle and technical, but it boils down to this: the `<%@ include %>` directive works by finding the target file and inserting it into your JSP page, just as if you had cut and pasted it using a text editor. By contrast, `<jsp:include>` locates the target page while your JSP page is executing. This difference in operation implies the following differences in behavior:

- If a file included with `<%@ include %>` changes, its changes will not be noticed until the page containing the `<%@ include %>` directive also changes. Recall from chapter 1 that the JSP engine notices when files are changed and processes them automatically. However, the container doesn't keep track of

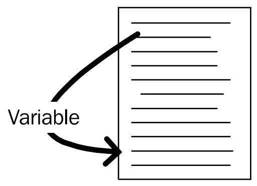


Figure 2.8
Page scope lets one part of a page share data with another part.

Request scope

Earlier in this chapter, we looked at the `<jsp:include>` and `<jsp:forward>` tags. These tags have something in common: they tie together multiple pages.

To access web pages, a web browser makes a *request* for data from a web server. If this request hits a JSP page that uses a `<jsp:include>` or `<jsp:forward>` tag, then multiple JSP pages can be used to service a single request. All of the pages that are used to respond to a single request have access to a common *request scope.* For instance, as suggested by figure 2.9, a page that uses `<jsp:include>` can use the request scope to transfer data to or from the pages it includes. Request scope is useful if the target page needs to act differently depending on an event that occurred in the page that includes it. Or perhaps the target page wants to set a variable that the page using `<jsp:include>` needs to access. Either way, request scope—which is broader than page scope—can be appropriate.

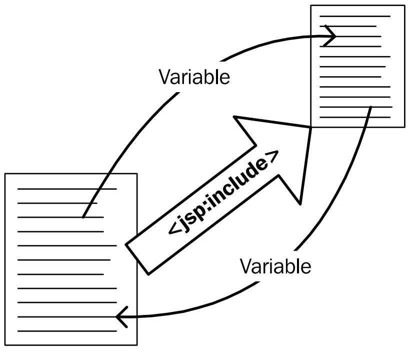


Figure 2.9
Request scope lets pages linked by `<jsp:include>` or `<jsp:forward>` communicate among themselves.

Application scope

In the world of server-side Java programming, the term *web application* has a specific meaning. A web application is a collection of JSP pages and other resources, like servlets and HTML pages. Typically, a web application is located under a common directory on the web server, and it represents a cohesive unit of functionality. For instance, an entire online store or auction site is a good candidate for a web application.

Part 2

Learning JSTL

Now that you've seen the tip of the iceberg, it's time to focus on the details and principles of JSTL. We've discussed what JSTL's supposed to do; now you get to see how it works.

Part 2 introduces and demonstrates nearly every JSTL tag. (We'll leave two tags until later.) We start with the most fundamental ones: those that handle simple decisions and loops in your pages. Then, we explore all the features JSTL has to offer, from databases to powerful XML support.

Although part 2 is designed as a tutorial and reference, we do (when appropriate) take a step back and look at useful examples of JSTL in action. For more in-depth examples, see part 3.

3 The expression language

This chapter covers…

- JSTL's expression language syntax
- Printing dynamic content
- Storing and retrieving scoped variables
- Producing and reading HTML forms

times, instead of printing nothing, you want to print an error message, placeholder, or other default value. For cases like these, `<c:out>` takes a parameter called `default`. If `value`'s expression fails for any reason, `default` runs instead. For instance, look at this tag:

```
<c:out value="${username}" default="Nobody"/>
```

This tag works just like the first `<c:out>` tag we presented; but if `${username}` doesn't produce a sensible value, then the tag simply prints out the static text `Nobody`. The `<c:out>` tag can also accept a body, which you can use as another way of specifying a default value. Thus, the following tag is equivalent to the last one:

```
<c:out value="${username}">
  Nothing
</c:out>
```

This tag can be useful if your default value is too long to fit conveniently inside an attribute. Or, you can stick other JSTL tags in the body, and they'll be used as the default if `${username}` doesn't produce a sensible value.

3.1.4 *Special characters and <c:out>*

You should know one more useful thing about `<c:out>`. By default, it makes sure that any characters with special meaning to HTML or XML are escaped using the *entity references* we discussed briefly in chapter 2. This feature lets you use `<c:out>` without worrying that your data will get in the way of the HTML or XML output you're producing.

Imagine that a scoped variable contains the text `AT&T`, or `<o>`, or another string that has one or more characters with special meaning to XML. (The following characters are special to XML: `&`, `<`, `>`, `'`, and `"`.) By default, if you print such a variable with `<c:out>`, any special characters that it contains will be escaped as `&amp;`, `&lt;`, and so forth. This escaping causes HTML browsers to display the characters to the user instead of treating them as part of HTML or XML tags. For example, if the variable `eye` contains the text `<o>`, then

```
<c:out value="${eye}"/>
```

will output

```
&lt;o&gt;
```

where `&lt;` stands in for `<` and `&gt;` stands in for `>`. Thus, an HTML browser will display the text `<o>`—the original value of `${eye}`—to the user. If `<c:out>` were instead to output `<o>` unescaped, then the browser would see an unrecognized HTML tag, and the user wouldn't see the information at all.

matical way of expressing statements like, “If I took my wristwatch off, it must be on the nightstand. But it isn’t on the nightstand, so I must not have taken it off. Or maybe I’m just growing senile.”)

A boolean variable has two possible values: `true` and `false`. These values can also be interpreted as “yes” and “no.” Whereas strings and numbers can take on virtually unlimited values, a boolean variable can store only these two values.

This limitation makes boolean variables particularly useful for yes-or-no questions. For example, the `escapeXml` attribute for `<c:out>` that we discussed in section 3.1 is a boolean attribute: it needs a boolean variable. Our earlier example showed `escapeXml` being used as follows:

```
<c:out value="${username}" escapeXml="false"/>
```

In this case, `escapeXml` was given the static value `false`. But just as `<c:out>`’s `value` attribute can accept expressions, so can `escapeXml`. If the scoped variable `status` has a boolean value, you can write this:

```
<c:out value="${username}" escapeXml="${status}"/>
```

This `<c:out>` tag will decide whether to escape special characters depending on the value of the scoped attribute `status`. In chapter 4, you’ll see how to set scoped boolean variables.

If you print a boolean value using `<c:out>`, it will be printed as `"true"` or `"false"`, as appropriate.

NOTE Java has two different boolean data types: `boolean` and `Boolean`. (Java is case-sensitive, so these represent different types.) For our purposes, they are nearly identical, so you don’t have to worry about the differences between them.

Collections

When a scoped variable is a string, number, or boolean, it stores exactly one thing: a piece of text, a number, or a truth value. Sometimes, however, a single scoped variable can store an entire collection of objects. The most obvious example, in our mercenary world, is a shopping cart. An application might make a shopping-cart variable accessible as

```
${sessionScope.shoppingCart}
```

Such a variable refers to an entire collection of objects, organized under a single name: `shoppingCart`.

```
          <option value="summer">Summer</option>
          <option value="fall">Fall</option>
       </select>
   </p>

   <p>Languages you can read:
     English
       <input type="checkbox" name="language" value="english" />
     Spanish
       <input type="checkbox" name="language" value="spanish" />
     French
       <input type="checkbox" name="language" value="french" />
   </p>

   <p>Ontological speculations:<br />
     <textarea rows="5" columns="40" name="philosophy" />
   </p>

   <input type="submit" value="Sign up!" />

  </form>
```

It should be easy to see how the individual tags in listing 3.1 line up with the various parts of the form shown in figure 3.4. Let's look at each piece of the form in turn.

The <form> tag

An HTML form begins with `<form>` and ends with `</form>`. Between these two tags come tags for the various form elements, such as `<input>`, `<select>`, and `<textarea>`. We'll look at these individual tags in a moment; for now, I want to draw your attention to the start tag for `<form>`:

```
<form method="post" action="formHandler.jsp">
```

This tag has two attributes, `method` and `action`. The `action` attribute is more important for us. It functions basically like `href` in `<a>` or `src` in `<img>`—that is, it lets you enter a link. For our purposes, this link will typically be a relative URL and point to a JSP file in the current directory. The `action` attribute means, "When the user submits this form, what page should I load, and where should I send the input?" For example, the `<form>` tag we just looked at causes a page named formHandler.jsp to run and receive the form's input when the user submits the form. (You'll see in a moment how the user submits a form.)

The value of the `method` parameter doesn't matter much for now, but you can think of it this way: by default, or if `method="get"`, all of the form's input will show up encoded into the URL. (You'll see more about the way this data is structured in chapters 5 and 6.) By contrast, when `method="post"`, this data is hidden from the casual observer and is instead sent to the target page using a different behind-thetle. The entire block of text the user types into a `<textarea>` comes back to your JSP page as a single parameter. For example, the entire box created by

```
<textarea rows="5" columns="40" name="philosophy" />
```

comes back as a single parameter: `${param.philosophy}`.

Submitting a form

To add a submission button to a form, you add an `<input>` tag with the attribute `type="submit"`:

```
<input type="submit" value="Sign up!" />
```

This tag adds a button to the form (labeled with whatever's inside the `value` attribute); when the user clicks it, the form is sent to the page named in the `action` attribute of the original `<form>` tag.

TIP Submission buttons created with `<input type="submit">` can also have `name` attributes. That is, they can also create request parameters. For instance, a button like

```
<input type="submit" value="Register" name="choice" />
```

will set a request parameter `${param.choice}` equal to the string `Register`. This functionality is particularly useful if you want your form to have multiple submission buttons, and you want to figure out which button the user clicked to submit the form.

3.3.2 *A page that reads request parameters*

We've spent quite a bit of time discussing HTML forms and individual request parameters. Let's look, at last, at a dynamic page that reads some parameters. As an example, we'll write a page called formHandler.jsp that handles the form in figure 3.4 (and listing 3.1). To get this page to work, simply add it to the same directory as the page that produced the form. Listing 3.2 shows an example of such a page.

Listing 3.2 formHandler.jsp: a page that prints out the results of a form

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<p>Wow, I know a lot about you...</p>
<p>Your name is <c:out value="${param.username}"/>.</p>
<p>Your password (sssssh!) is '<c:out value="${param.pw}"/>'.</p>
<p>You are <c:out value="${param.gender}"/>.</p>
```

Table 3.5 JSTL expressions involving `pageContext`. These somewhat involved expressions are useful if you need detailed information about the current page's environment. Normally, you won't need to use such expressions, but they come in handy on occasion. *(continued)*

Expression	Description	Sample value
`${pageContext.request.queryString}`	Your page's entire query string	`p1=value1&p2=value2`
`${pageContext.request.requestURL}`	The URL used to access your page	`http://server/app/page.jsp`
`${pageContext.session.new}`	`true` if the session is new; `false` otherwise	`true`
`${pageContext.servletContext.serverInfo}`	Information about your JSP container	`Apache Tomcat/5.0.0`
`${pageContext.exception.message}`	For a page marked as an `errorPage`, a description of the error that occurred	`"Something very, very bad happened"`

3.4.3 Comparisons

You can use the expression language to produce boolean values even when your inputs aren't boolean. For instance, the expression `${2 == 2}` results in `true`. Note the use of two equal signs (`==`) as a way of comparing two values. Many programming languages, including Java and JavaScript, use similar syntax, so it might look familiar.

Table 3.6 lists the JSTL expression language's comparison and equality operators.

Table 3.6 JSTL supports these comparison and equality operators in expressions. You can use these operators to write expressions like `${2 == 2}` or `${user.weight gt user.IQ}`. Every comparison operator has a symbolic version (`==`) and a textual one (`eq`).

Operator	Description	Sample expression	Result

`==`
`eq`
	Equals	`${5 == 5}`	`true`

`!=`
`ne`
	Not equals	`${5 != 5}`	`false`

`<`
`lt`
	Less than	`${5 < 7}`	`true`

`>`
`gt`
	Greater than	`${5 > 7}`	`false`

`<=`
`le`
	Less than or equal to	`${5 le 5}`	`true`

`>=`
`ge`
	Greater than or equal to	`${5 ge 6}`	`false`

take the result of this attribute—which may, of course, contain expressions—and save it to the variable indicated by `var` and `scope`.

For instance, consider the following tag:

```
<c:set var="four" value="${3 + 1}"/>
```

This tag stores the value `4` in a scoped variable named `four`. The scoped variable named `four` is given page scope. If you wanted to store it in the session, you'd instead write

```
<c:set var="four" scope="session" value="${3 + 1}"/>
```

The `<c:set>` tag can take any kind of JSTL expression in the `value` attribute; it can result in a string, number, boolean, collection, or anything else.

Using the tag's body

If you write a `<c:set>` tag without a `value` attribute, then `<c:set>` will take whatever appears in its body and save it to the scoped variable indicated by `var` and `scope`. It's important to realize that if other tags appear within `<c:set>`'s body, these tags will be evaluated; like the browser itself, `<c:set>` only sees their output.

This process might seem unusual, and it's the first time we've encountered a concept that will keep coming up in JSTL. JSP lets every tag have access to the output of its body. Normally, everything in your page simply gets printed to the browser; either it's template text and gets printed directly, or it's a tag and can produce dynamic output (see figure 3.6). But when template text and tags appear inside another tag, the inner text and tags don't get a chance to send their output directly to the browser. Instead, the parent tag collects the output from its body and then decides what to do with it. It can decide to send it on to the browser, to save it to a scope variable, or to ignore it completely; it's the tag's choice (see figure 3.7). In

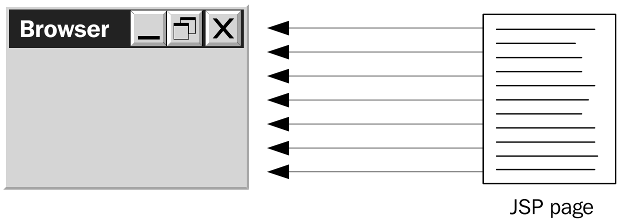


Figure 3.6 Normally, all template text and JSTL tags in your page get the opportunity to output directly to a web browser. The template text goes right through (as itself), and the JSTL tags (like `<c:out>`) have a chance to produce dynamic output that, by default, gets sent to a web browser.

4 Controlling flow with conditions

This chapter covers…

- Simple conditions
- Mutually exclusive conditions
- Nesting condition tags
- Syntactic rules for JSTL conditions

4.2.2 Using <c:if> within HTML tags

Because JSP doesn't draw any distinction between plain text and HTML tags, you can use `<c:if>` tags anywhere in your page—even in the middle of an HTML tag. For instance, consider this use of a `<c:if>` tag:

```
<font size="2"
  <c:if test="${user.education == 'doctorate'}">
    color="red"
  </c:if>
>
    <c:out value="${user.name}"/>
</font>
```

This code prints the user's name in red if the user is a doctor. The code checks the `user.education` property and, if it is equal to `doctorate`, outputs the following (ignoring white space) where *`name`* is the user's name, as output by the `<c:out>` tag:

```
<font size="2" color="red">
    name
</font>
```

If `user.education` is different from `doctorate`, we instead get

```
<font size="2">
    name
</font>
```

In the first case, the HTML `<font>` tag explicitly sets the text color to red, whereas the latter case uses the default color. Recall from chapter 1 that the browser doesn't care how an HTML markup tag was generated—whether it was template text, the output of a JSTL tag, or both. Thus, JSTL tags, like `<c:if>`, can easily be used to produce HTML tags or parts of them.

4.2.3 Multiple <c:if> tags

When `<c:if>` tags appear next to one another, they act independently:

```
<ul>
<c:if test="${error1}">
   <li>Error 1 has occurred.</li>
</c:if>
<c:if test="${error2}">
   <li>Error 2 has occurred.</li>
</c:if>
</ul>
```

This example assumes that when certain errors have occurred, your page (or back-end Java code) has stored the value `true` in page-scoped boolean variables called `error1`, `error2`, and so on. (You'll see how to create such variables in section 4.2.5.)

```
<c:if test="${user.education=='doctorate'}">
 Dr.
</c:if>
```

then the page might appear inconsistent to the user. After a certain point in the page, the `Dr.` title will appear; but at the beginning, it won't. This difference might not matter; but if consistency is important, we should record the value—whatever it happened to be at a particular point—and then use it for the rest of the page. The `var` attribute allows us to do this.

Consider the following example:

```
<c:if test="${sessionScope.flags.errors.serious.error1}"
    var="error1">     <— Saves variable
  A serious error has occurred.
</c:if>

[… large page body …]

<c:if test="${error1}">     <— Uses variable
  Since a serious error occurred, your data has not been saved.
</c:if>
```

In this example, when the first tag is reached, the expression in its `test` attribute is evaluated, and the result is saved into a page-scoped variable called `error1`. From this point forward, even if the value of `flags.errors.serious.error1` in the session scope changes, the local `error1` variable will stay the same. Thus, even if the session-scoped `flags.errors.serious.error1` flag changes for any reason, the user will be given a message at the bottom of the page that is consistent with the one displayed at the top. (Note also that in the second `<c:if>` tag, we save some typing by using our own shorter variable name.)

Although most `<c:if>` tags have a body, JSTL's don't require them to. So, you can use `<c:if>` to write a tag whose only purpose is to expose a scoped variable. For example, the following empty tag exposes a boolean variable named `error1`:

```
<c:if test="${sessionScope.flags.errors.serious.error1}"
  var="error1"/>
```

This tag isn't used to make a decision during execution of the page; later tags on the page, however, can use the `error1` variable that this tag creates.

```
    <li>Error 3 has occurred.</li>
  </c:when>
  <c:otherwise>
    <li>Everything is fine.</li>
  </c:otherwise>
</c:choose>
```

Addition of <c:otherwise> to display the default message

The example now prints out a reassuring message—by way of the optional `<c:otherwise>` tag—if `${error1}`, `${error2}`, and `${error3}` are all *not* `true`.

Example 3

Let's consider a slightly more involved example. In the first example in this chapter, we discussed a `<c:if>` tag used to print the text `Dr.` if `user.education` indicated that the user had a doctorate. In that example, `Dr.` would either appear or it wouldn't; there was no third choice. Instead of this simple yes-or-no choice, let's look at an example that prints one of three choices—`Dr.`, `Ms.`, or `Mr.`—as appropriate. To do this, we have our tags check both a `user.education` property and another property, `user.gender`:

```
<c:choose>
  <c:when test="${user.education=='doctorate'}">
    Dr.
  </c:when>
  <c:when test="${user.gender=='female'}">
    Ms.
  </c:when>
  <c:when test="${user.gender=='male'}">
    Mr.
  </c:when>
</c:choose>
<c:out value="${user.name}"/>
```

We use two different properties of the `user` variable, but all our tests are grouped under a single `<c:choose>` tag. The result is that an appropriate title (`Dr.`, `Ms.`, or `Mr.`) is displayed in all cases. Note that the check for `Dr.` appears first because it transcends gender. If we checked for a particular gender first, we would miss all the members of that gender who were also doctors. Instead, we want to check `gender` only if the user is not a doctor.

This example demonstrates that JSTL strictly adheres to the order of your `<c:when>` tags. If the first `<c:when>` tag succeeds, then the second (and remaining) tags won't be evaluated; if the second succeeds, then the third (and remaining) tags won't be evaluated; and so on.

The `<c:when>` tag does not accept a `var` attribute, but it can use boolean variables exposed by earlier `<c:if>` tags.

The basic function of <c:forEach> is to consider every item in the collection specified by its items attribute. For each item in the collection, the body of the <c:forEach> tag will be processed once, with the current item being exposed as a page-scoped variable whose name is specified by <c:forEach>'s var attribute. Because this variable takes a different value for each loop, the body of the <c:forEach> tag can print different text each time it is evaluated.

Let's make this behavior concrete. Consider the following use of <c:forEach>:

```
<c:forEach items="${user.medicalConditions}" var="ailment">
    <c:out value="${ailment}"/>
</c:forEach>
```

This <c:forEach> tag loops over every item in the medicalConditions property of the user variable. If this property contains a list of medical conditions, like gingivitis, myopia, and dehydration, then the example will print a string for each of these items.

You can also include static template text inside a <c:forEach> tag's body, in which case it will appear unchanged for each loop that <c:forEach> makes. For example:

```
<p>Sorry, you are afflicted with the following
minor medical conditions:</p>
<ul>
<c:forEach items="${user.medicalConditions}" var="ailment">
    <li><c:out value="${ailment}"/></li>
</c:forEach>
</ul>
```

If ${user.medicalConditions} contains the three conditions I mentioned earlier, this fragment will output the following HTML (ignoring white space):

```
<p>Sorry, you are afflicted with the following
minor medical conditions:</p>
<ul>
   <li>gingivitis</li>
   <li>myopia</li>
   <li>dehydration</li>
</ul>
```

The template text outside the <c:forEach> tag is, of course, included only once. For instance, this example prints only one <ul> tag. But text within the

[1] In case you encounter specific Java types when talking with Java programmers—or in case you're a developer yourself—you might be interested to know the names of the data types <c:forEach> accepts. They include arrays, Collection variables (including Lists and Sets), Maps, Iterators, and Enumerations. As you'll see in section 5.2, it can also accept simple strings.

Both `<c:forEach>` and `<c:forTokens>` accept three optional attributes in support of subsetting, as shown in table 5.3.

Table 5.3 Subsetting attributes for `<c:forEach>` and `<c:forTokens>`

Attribute	Description	Required	Default
`begin`	Item to start the loop (inclusive; `0`=first item, `1`=second item).	No	`0`
`end`	Item to end the loop (inclusive; `0`=first item; `1`=second item).	No	Last item
`step`	Iteration will process every *step*th element (`1`=every element, `2`=every second element).	No	`1`

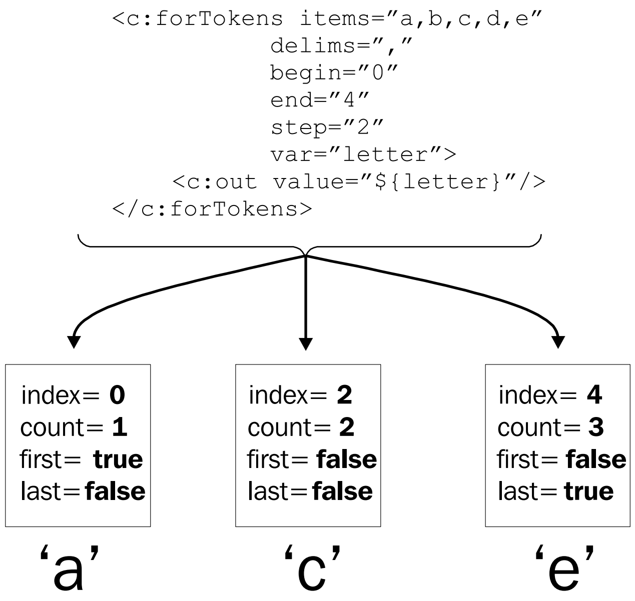
JSTL assigns an *index* to every item in a collection; this index represents the item's place in the overall collection. For each collection, the index begins with 0, which—interestingly enough—corresponds to the first item. Each successive element takes the next index number: the second element has an index of 1, the third element has an index of 2, and so on.

The `begin` and `end` attributes accept numbers corresponding to these indexes. By default, `<c:forEach>` and `<c:forTokens>` process the entire collection available to them; like dutiful cogs in a machine, they start at the beginning and finish at the end. The `begin` and `end` attributes override this default behavior by identifying particular start and end indexes. The `begin` attribute directs the tag to start with the item at a particular index, and `end` causes iteration to end with a particular index. For example, `begin="0"` and `end="4"` together instruct that a `<c:forEach>` or `<c:forTokens>` tag should begin with the first element and end with the fifth. Similarly, when a `<c:forEach>` or `<c:forToken>` tag is given the attributes `begin="5"` and `end="9"`, only the indexes 5, 6, 7, 8, and 9 will be included (that is, the sixth through tenth elements).

WARNING Be careful! Because 0 represents the first element, `end="4"` will cause iteration to proceed through the fifth element. Zero-based indexes can be confusing, but many programming languages adhere to them for consistency. If you have worked with JavaScript or Java before, you probably are familiar with zero-based indexes. (Zero-based indexes are not limited to programming languages. Not too far from where I live, a highway mile marker labeled 0 indicates the beginning of the highway. As a programmer, it warms my heart.)

The `count` property, on the other hand, starts with `1` and reflects the current loop's position among the items for which `<c:forEach>` runs its body. No matter what, `count` always increases by one for each loop. For any `<c:forEach>` or `<c:forToken>` tag, `count` will be `1` the first time the body is processed, `2` the second time, and so on. The `first` and `last` properties are boolean properties indicating whether the current loop is the tag's first or last, respectively. (The `first` property is just a convenient way of checking to see whether `count` currently equals `1`.) The `count` attribute's behavior isn't affected by the `begin`, `end`, or `step` attribute.

Figure 5.2 shows the values of these properties for a sample iteration.[2]



```
<c:forTokens items="a,b,c,d,e"
             delims=","
             begin="0"
             end="4"
             step="2"
             var="letter">
    <c:out value="${letter}"/>
</c:forTokens>
```

Figure 5.2 Values of the `varStatus` variable's properties during a sample iteration. The tag in this figure iterates three times, producing the letters *a*, *c*, and *e*. The boxes above each letter show the values of the `varStatus` variable for that letter's loop.

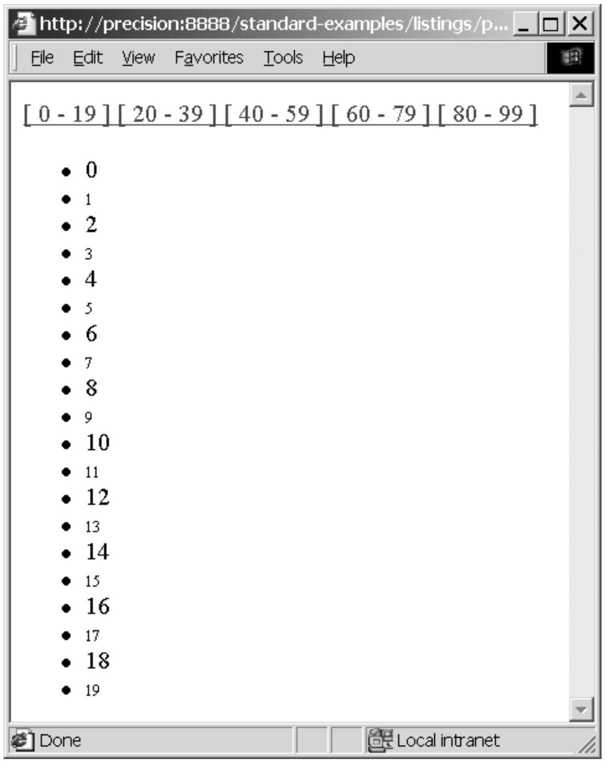
5.4 Loop example: scrolling through results

Earlier, we discussed how you can use the `begin` and `end` attributes to display only part of a collection, in cases where the collection is too big to fit reasonably on a single screen. Many applications, when they have too much information for a single page, let users pick the information to view. For instance, the user can decide whether to display results 0 through 19, 20 through 39, and so on.

[2] The variable that `varStatus` creates has some other properties, but they are intended more for developers of custom tags than for page authors. If you're a Java developer and are interested in these extra properties, see the `LoopTagStatus` interface in appendix B.

```
  </c:if>
  <c:if test="${status.count % 2 == 0}">
    <font size="-2">
  </c:if>
  <li><c:out value="${current}"/></li>
  <c:if test="${status.count % 2 == 0}">
    </font>
  </c:if>
  <c:if test="${status.last}">
    </ul>
  </c:if>
</c:forEach>
```

To pick out every second row, we use the expression `${status.count % 2 == 0}`. Recall from chapter 3 that `%` in JSTL's expression language is a *remainder* operator. Thus, `status.count % 2` means, "Divide `status.count` by 2 and take the remainder." This remainder will be 0 only for the even rows. Thus, only these rows print in a smaller font in figure 5.4. Note that we use the same condition twice: once to open a `<font>` tag, and once to close it with `</font>`.


http://precision:8888/standard-examples/listings/p...
File Edit View Favorites Tools Help
Done
Local intranet

[0 - 19] [20 - 39] [40 - 59] [60 - 79] [80 - 99]

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19

Figure 5.4
Many web sites display alternating rows in different colors. Because colors don't show up well in a black-and-white book, our example of handling alternate rows uses font size instead of color. Here, every second row prints using small text.

Another important task you'll need to handle when you write dynamic web pages is managing Uniform Resource Locators (URLs). You need to use URLs when you import content with `<c:import>`, but URLs show up in other places as well. For example, every time your pages display hyperlinks (HTML `<a>` tags) to other pages, they use URLs.

In this chapter, we look at `<c:import>` and other tags that help you manage and use URLs. We'll also show how you can communicate with the pages you include, in order to customize their output.

6.1 Including text with the <c:import> tag

To retrieve content from a local JSP page or from another server, you can use the `<c:import>` tag. Sometimes you'll just want to print the information that you retrieve, but `<c:import>` also lets you store the retrieved text in a scoped variable instead of printing it.

Table 6.1 shows the `<c:import>` tag's attributes.

Table 6.1 Basic `<c:import>` tag attributes

Attribute	Description	Required	Default
`url`	URL to retrieve and import into the page	Yes	*None*
`context`	/ followed by the name of a local web application	No	*Current context*
`var`	Name of the attribute to expose the `String` contents of the URL	No	*None*
`scope`	Scope of the attribute to expose the `String` contents of the URL	No	`page`

The crucial attribute is `url`, which specifies the URL of the content to retrieve. The other attributes let you modify the way the tag handles its URL.[1]

Often, a page that uses `<c:import>` is called a *source page*, and the page whose contents are included with `<c:import>` is called a *target page*.

6.1.1 Absolute and relative URLs

You're probably familiar with the basics of URLs simply from browsing the Web. A URL, which is often called a *web address* by the sort of person who's captivated by

[1] The `<c:import>` tag has a few advanced attributes that you'll need only if you're performing relatively sophisticated text imports. See chapter 14 for more information about advanced `<c:import>` techniques.

The `context` attribute names another web application on the same server as the page you're writing. This name needs to start with a forward slash (/). For instance, consider the following tag:

```
<c:import context="/other" url="/directory/target.jsp"/>
```

This tag imports the page /directory/target.jsp from the web application named "other" in the same JSP container as our source page. Thus, the URL that appears in the `url` attribute is treated as if it is relative to the root of this other web application (that is, the other *context*).

Using expressions

Of course, you're not limited to using URLs that you type literally into the `<c:import>` tag's `url` attribute. The `<c:import>` tag supports the full range of JSTL expressions. For instance, the target URL can come from an expression, as follows:

```
<c:import url="${applicationScope.target}"/>
```

This tag looks up the `target` attribute in the application scope, treats it as a URL, and retrieves information from this URL. The `context` attribute can also come from an expression:

```
<c:import url="${applicationScope.target}"
  context="${applicationScope.targetContext}"/>
```

6.1.3 *Saving information for later*

By default, `<c:import>` retrieves information from a URL and then immediately prints it to your page. This is exactly what `<jsp:include>` does, and in most cases, it's also what you want.

However, suppose you don't want to immediately print the data you retrieve. Sometimes, for instance, you want to import a page and then include its text multiple times in your page. (As an example, imagine a file that contains nothing but some HTML formatting to produce a stylized, horizontal line.) Or you might want to retrieve some text every time the user logs in, and then store this text in the user's session scope for use during the user's session. Saving data from `<c:import>` lets you avoid having to retrieve the contents of a URL multiple times, which can sometimes take a long time and slow your pages.

To save the result of `<c:import>` instead of printing it out, you can use the `var` attribute. Specifying a `var` attribute to the `<c:import>` tag causes the tag to not output anything. Instead, the tag will simply retrieve text and save it to a scoped variable. As with other JSTL tags, you can also use a `scope` attribute to set the scope of the variable you create. (As usual, when you use `var` and `scope`, you need to specify the name and scope manually; you can't use expressions in these two attributes.)

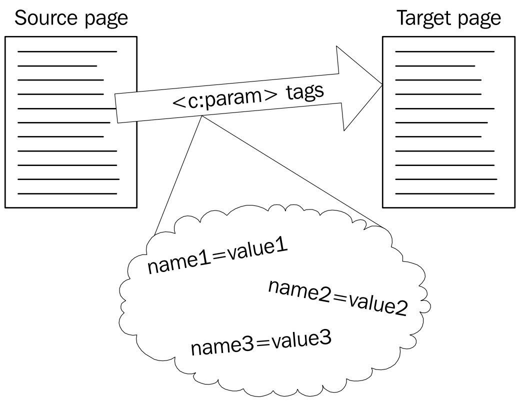


Figure 6.3
`<c:param>` lets you communicate simple request parameters, which take the form `name=value`. Request parameters are flexible, but they can only consist of simple text strings, and they only support one-way communication (from the source page to the target).

web site, you might appreciate the simplicity of `<c:param>` and the fact that, when you use `<c:param>` within `<c:import>`, you can immediately see what data two pages share. If you want to understand two pages that use scoped variables, you may need to spend more time looking at the source code for both the source and the target.

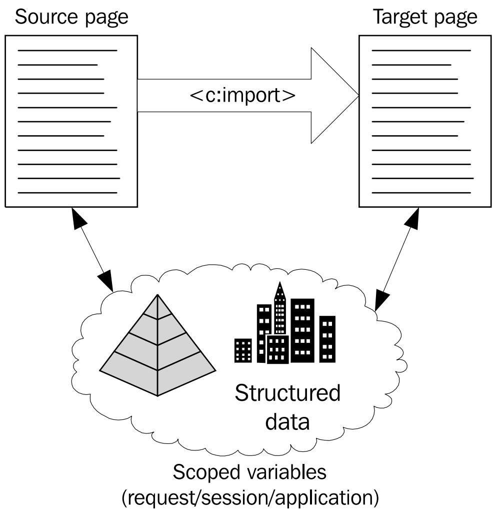


Figure 6.4
In contrast with request parameters, scoped variables—which can be accessed by both the source and target pages of a `<c:import>` tag—can include arbitrarily structured data. They support two-way communication as well. However, they only work for target pages within your web application.

6.1.5 Import example: a customized header

Let's look at a concrete example of pages communicating with one another using `<c:import>`. Many web applications need to standardize the appearance of a header throughout the application. We'll throw in a twist, however: in our example, the header will display a customized title that the source page (the one using

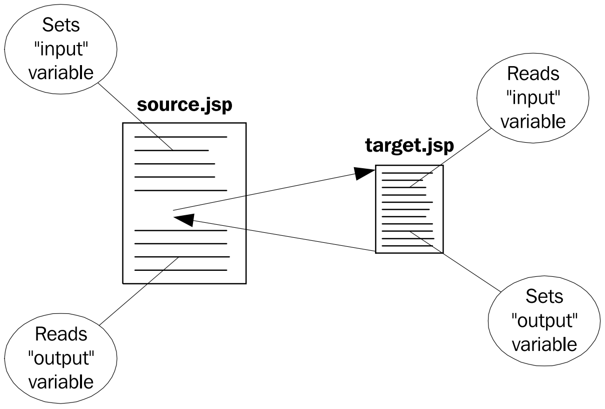
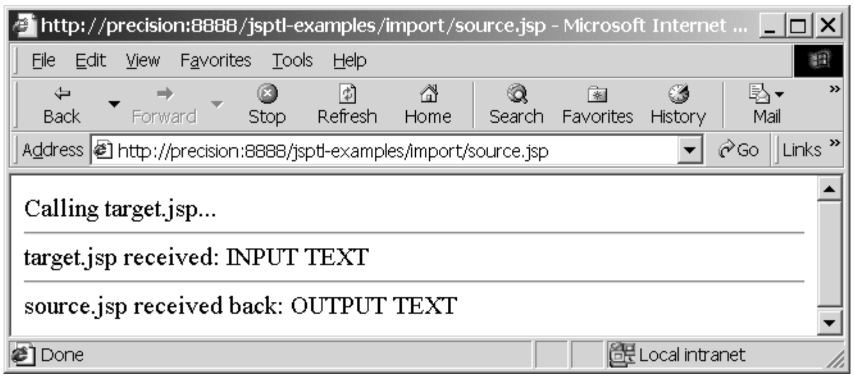


Figure 6.8 source.jsp sets a variable and then imports target.jsp, which reads the variable. Before target.jsp finishes, it sets its own variable, which source.jsp later reads.


http://precision:8888/jsptl-examples/import/source.jsp - Microsoft Internet ...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address http://precision:8888/jsptl-examples/import/source.jsp Go Links
Done
Local intranet

```
Calling target.jsp...
target.jsp received: INPUT TEXT
source.jsp received back: OUTPUT TEXT
```

Figure 6.9 source.jsp displays output that looks like this when loaded by a web browser.

6.2 Redirecting with <c:redirect>

In some situations, your web pages need to act like seasoned bureaucrats and refer you elsewhere. Fortunately, web browsers tend to have more patience than most people do. Normally, when a browser sends a request for a web page, it receives back an HTML file, image, or other content in response. Sometimes, however, it gets *redirected* to another page. Essentially, the server says, "I don't have what you want; go look *here* instead," where *here* is a particular URL the browser needs to follow. The browser then loads this URL and displays its content—or perhaps it's redirected to yet another URL.

Here, the `<c:url>` tag is embedded within the `<img>` tag's `src` attribute; it causes the URL to be transformed appropriately so that the user's browser can understand it.

You can use `<c:url>`'s `context` attribute to create a URL to a page in another web application in your JSP container.

The `<c:url>` tag is also useful if you want to save a URL (using the `var` and `scope` attributes) and use it multiple times in your application.

6.4 Summary

In this chapter, we looked at tags that support text retrieval, redirection, and URL management. Key points to remember include the following:

- `<c:import>` works like `<jsp:include>`, but it lets you retrieve data from absolute URLs, as well as pages from different web applications on the same JSP server. It also lets you save data instead of printing it out immediately.
- If the source and target pages are in the same web application, then they can share variables in request, session, and application scope. Doing so allows two-way transmission of whatever data you'd like (including, of course, simple strings).
- `<c:redirect>` lets you bounce the user to a new page, using either an absolute or relative URL.
- Whenever you write out a relative URL to a page, you should use `<c:url>` instead of printing the URL directly. Doing so makes sure sessions work even in browsers that don't support cookies, and it also simplifies use of context-relative URLs (those that begin with /).
- `<c:param>` lets you pass simple text strings from the source page to the target page. It works with `<c:import>`, `<c:redirect>`, and `<c:url>`.

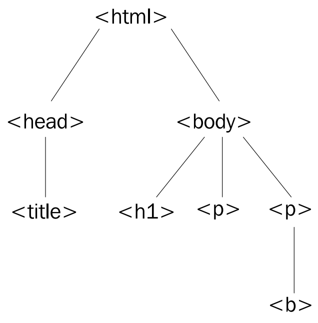


Figure 7.1
The tree structure of a sample HTML document. When an element like `<h1>` occurs inside `<body>`, you can think of it as a child of that `<body>` element.

7.2 XPath's basic syntax

XPath operates on documents using the type of tree structure you just saw. As you probably know, trees are commonly seen on computers. On nearly all modern operating systems, for example, a disk is organized into *directories* (or *folders*), each of which can contain other directories. This kind of organization naturally arranges itself into a tree, and we often speak of *child directories* or *subdirectories* when we discuss disks.

XPath takes advantage of our familiarity with traditional filenames, applying a similar syntax to the tree representing an XML document. If you have three directories on your disk—a, b, and c—and you are running Windows, you can refer to these directories as follows:

```
c:\a\b\c
```

Note how the backslash character (\) is used to separate the directory names. Unix systems use the regular slash (/) character in a similar capacity:

```
/a/b/c
```

XPath adopts this Unix convention, using the slash character to separate the name of one XML element from another. For example, in the tree from figure 7.1, the `<b>` element could be described by the following path:

```
/html/body/p/b
```

This XPath expression matches the highlighted part of our sample document:

```
<html>
 <head>
  <title>Poem</title>
 </head>
 <body>
  <h1>Poem</h1>
```

Therefore, `//p` is `true` if it is applied to a document that has at least one `<p>` element, and it's `false` if the document has no `<p>` elements.

NOTE XPath expressions are more flexible than I've shown here. For instance, they can also call functions that directly return numeric values, boolean values, and so on. XPath also provides general rules for converting between numbers, booleans, and other types of data. Details about XPath data types are beyond the scope of this book because they're not needed to use JSTL; see appendix D for references to more information.

7.3 XPath variables and JSTL

Like many languages, XPath supports variables. Just as in JavaScript, Java, and other languages, XPath variables are evaluated and replaced with actual values, which might be different every time an XPath expression executes.

JSTL depends on XPath variables in a somewhat novel way: it maps them to dynamic scopes that resemble JSTL's expression language. Therefore, XPath variables can refer to things that are similar to those the familiar expression language can refer to (see chapter 3).

Broadly speaking, an XPath variable is simply a qualified name (see chapter 2) introduced with a dollar sign (`$`). That is, it's a dollar sign followed by either a name without a colon, like `stomach`, or a name with a colon, like `large:intestine`.

The XPath expressions we've presented until now haven't used variables; they simply contained text, as in

```
/a/b/c
```

You can introduce a variable into this static XPath expression. This variable can have a different value each time an XPath expression is evaluated. Variables can refer to data from a variety of sources. For instance, the expression

```
$pageScope:document/b/c/d
```

contains the variable `$pageScope:document`. Recall from chapter 2 that in the name `pageScope:document`, `pageScope` is a namespace prefix, and `document` is a specific, local name. JSTL recognizes the namespace prefixes listed in table 7.1.

These prefixes have the same behavior as the *implicit objects* described in chapter 3 for the general-purpose expression language. Furthermore, just as in JSTL's language, the default behavior when searching for a variable (the behavior when no namespace prefix is specified) is to search first in the page scope, and then

7.6 Summary

In this chapter we explored XPath's basic syntax, in order to let you use XPath with JSTL. Keep in mind the following points:

- JSTL's support for XML manipulation depends on XPath.
- XPath (the XML Path Language) can be used to select parts of XML documents.
- XPath treats XML documents as trees and accesses individual nodes in the document in a similar manner to the way you access files on a disk.
- You can use XPath to filter documents based on node names, attribute values, and even the order in which nodes appear. But be careful if your documents use namespaces.
- XPath includes many more features than we've discussed here. Appendix D lists resources that will help you learn XPath in more depth, if you want to do so.

```
//table
```

we'd write

```
$doc//table
```

This expression tells JSTL to find the `doc` variable, and then find all `<table>` tags within the document it represents.

It's easy to confuse a variable that points to a document with the root element of that document. For instance, consider the following `<x:parse>` tag:

```
<x:parse var="orders">
  <orders>
    <order item="4"/>
  </orders>
<x:parse>
```

To refer to the inner `<order>` element, you could write `$orders/orders/order`, but *not* `$orders/order`. The inner `<order>` element is not a direct child of the document; it is a child of the `<orders>` element.

8.2.2 The <x:out> tag

The `<x:out>` tag evaluates and prints out the string value of an XPath expression; the starting node is often retrieved from an XPath variable. (For more information about string values, see section 7.2.) The `<x:out>` tag is one of the most basic ways of introducing an XPath expression into your JSP page. The tag takes the attributes listed in table 8.2.

Table 8.2 `<x:out>` tag attributes

Attribute	Description	Required	Default
`select`	XPath expression	Yes	*None*
`escapeXml`	Whether to print characters like `&` as `&amp;`	No	`true`

Let's look at `<x:out>` in action. Suppose our page contains the following `<x:parse>` tag:

```
<x:parse var="simple">
  <a>
    <b>
      <c>C</c>
    </b>
    <d>
      <e>E</e>
    </d>
  </a>
</x:parse>
```

match any elements. If `customerId` equals `525`, however, then the expression will match the two order records for Jim Heinz.

So, if Jim Heinz is the current customer (the one whose number is stored in `customerId`), we'll match two nodes. For Roberto, we won't match any. Note that we're not interested in *what* the nodes are. For example, we couldn't care less if the order number is 20005. Because we simply want to differentiate customers who have placed orders from those who haven't, the mere presence of `<order>` elements for Jim Heinz (and their absence for Roberto del Monte) is decisive. Recall XPath's boolean conversion rules from section 7.2: an XPath expression that matches one or more nodes is `true`, and one that doesn't match any nodes is `false`. Therefore, our sample XPath expression is `true` for Jim Heinz because he has placed orders, and it's `false` for Roberto del Monte because he hasn't. Jim will therefore receive the special message intended for repeat customers, and Roberto won't. Problem solved!

Storing a boolean result

The `<x:if>` tag, just like `<c:if>`, lets you save the result of a condition to a boolean variable using the `var` and `scope` attributes. As before, this tag has a number of uses:

- To avoid wasteful re-evaluation of a condition.
- To "lock in" a condition if you're afraid it will change.
- To use the result of a condition in a `<c:when>` or `<x:when>` tag that appears later in the page. (We'll look at `<x:when>` in a moment.)

8.3.2 Compound conditions with <x:choose>

Just as the core JSTL library provides `<c:choose>`, `<c:when>`, and `<c:otherwise>` for complex, mutually exclusive conditionals, the XML library offers `<x:choose>`, `<x:when>`, and `<x:otherwise>` for compound XML-based conditions. Their use is identical to the core library's, except that each `<x:when>` tag uses an XPath expression. Table 8.5 shows the attribute for `<x:when>`. (As with the core library, the other mutually exclusive conditional tags don't take attributes.)

Table 8.5 `<x:when>` tag attribute

Attribute	Description	Required	Default
`select`	XPath expression to evaluate. If `true`, process the tag's body; if `false`, ignore the body.	Yes	*None*

As a simple example of `<x:choose>`, `<x:when>`, and `<x:otherwise>`, consider the following small document, which you first saw in chapter 7:

```
    <font
       <x:choose>
         <x:when select="@status='preferred'">
           color="#000000"
         </x:when
         <x:otherwise>
           color="#888888"
         </x:otherwise>
       </x:choose>
    >
     <x:out select="name"/>
    </font>
  </p>
</x:forEach>
```

In this case, preferred customers are printed in a deep black (`color="#000000"`), and regular customers are printed in a lighter gray (`color="#888888"`). Ignoring white space, the example outputs the following HTML text:

```
<p>
  <font color="#888888">Jim Heinz</font>
</p>
<p>
  <font color="#000000">Roberto del Monte</font>
</p>
<p>
  <font color="#000000">Richard Hunt</font>
</p>
```

Note how we use XPath's `@` syntax to refer to attributes of the context node. Read the expression "`@status='preferred'`" as "Does the current node's `status` attribute equal `preferred`?"

Nested iteration

If a `<x:forEach>` tag appears inside another `<x:forEach>`, it inherits the outer tag's context node.

Consider the following sample document:

```
<customers>
  <customer id="555">
    <order id="1310">
      <item id="30"/>
      <item id="84"/>
    </order>
    <order id="1340">
      <item id="46"/>
      <item id="84"/>
    </order>
  </customer>
</customers>
```

The JSP page in listing 8.1 is an archetypal example of how to use XSLT from JSTL. First, a `<c:set>` tag sets the `xml` variable with body content that appears directly in the page. (Of course, this inline content could easily be replaced with a `<c:import>` tag to fetch a document from elsewhere). Next, another `<c:set>` tag sets the `xsl` variable using a simple, typed-in stylesheet. (Again, this stylesheet could reside elsewhere, and the page could retrieve it using `<c:import>`.) Finally, the `<x:transform>` applies the XSLT stylesheet to the XML document and outputs the result. The page therefore outputs the following, ignoring white space:[3]

```
<?xml version="1.0" encoding="UTF-8"?>
<p>
     This document uses <b>unusual</b> markup,
     which we want to replace with <b>HTML</b>.
</p>
```

For information on how the XSLT stylesheet works, see the references listed in appendix D.

8.4.2 Using the var attribute

If you specify a `var` attribute for `<x:transform>`, the document that results from the `<x:transform>` tag's transformation is saved in a variable instead of being output to the page. This result can be useful in a number of situations. For instance, once you've stored the output of a transformation using `var`, the output can be used as input to another `<x:transform>` tag. Or, you can select portions from the resulting document using XPath and `<x:out>`.

For example, the final line of listing 8.1 is a simple `<x:transform>` tag that outputs its result to the page:

```
<x:transform xml="${xml}" xslt="${xsl}"/>
```

Suppose we replaced this tag with one that stores the document in a variable and uses it in an `<x:out>` tag, as follows:

```
<x:transform var="doc2" xml="${xml}" xslt="${xsl}"/>
<x:out select="$doc2//b[2]"/>
```

If these two lines replace the final line in listing 8.1, the listing's JSP page then outputs simply `HTML`—the string value of the second `<b>` tag in the resulting document.

Instead of passing `$doc2` to the `<x:out>` tag, we could have passed it to another `<x:transform>` tag. Chaining XSLT transformations—applying them successively, using the output of one transformation as input to another—is a flexible technique

[3] In this example, the XML declaration (beginning `<?xml`) is added by the XSLT processor that's used behind the scenes to perform the transformation.

```
    </a>
  </li>
</x:forEach>
</ul>
```

This surprisingly short example is all we need to handle simple RSS files. We start by loading and parsing the RSS file from a URL specified by one of our request parameters, `rssUrl`. To pass the simpleRss.jsp page this parameter, we might use an HTML form like this:

```
<form method="post" action="simpleRss.jsp">
  Enter the URL for an RSS feed:
  <input type="text" name="rssUrl" />
  <input type="submit" />
</form>
```

Once simpleRss.jsp has retrieved its RSS file over the Web, it loops over each `<item>` tag in the RSS file and prints out its `<link>` and `<title>` children. We insert the contents of the `<link>` item into an `<a>` tag's `href` attribute, and we print the headline (`<title>`) as the body of the hyperlink. A sample result is shown in figure 8.5. (This example uses a news feed from CNet, which was available at the following URL at the time this chapter was written: http://export.cnet.com/export/feeds/news/rss/1,11176,,00.xml. See appendix D for more examples of RSS feeds.)

Dealing with namespaces

The simpleRss.jsp example is short and sweet, and it works for many RSS files, but it has a problem: it doesn't work for newer types of RSS files that use XML namespaces. This limitation arises because, as you saw in chapter 7, XPath expressions like `//item` and `link` don't match elements that use namespaces. To match these items in all RSS files, you need to use a slightly different syntax. Instead of writing

```
//item
```

to match all `<item>` tags, we'll need to use an XPath expression like this:

```
//*[name()='item']
```

This expression matches all tags whose name is equal to `item`, regardless of the RSS document's use of namespaces. Listing 8.3 shows a more general page that parses and prints out RSS documents.

Listing 8.3 rss.jsp: converts an RSS channel (with namespaces) into a list of hyperlinks

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jstl/xml" %>
```

When you need a user's information to last for an entire session, you can store it in JSP's session scope. However, some information needs to last longer than the session scope allows.

For instance, you probably don't want to make your visitors enter their preferences each time they come to your site. Most users would prefer to enter their information once and have your site remember it. Some information—like a customer's full name, address, and phone number—might not even have anything to do with the user's session or web experience; you might simply need to gather this information for use offline, after the user has left, to process orders or conduct other business operations.

To store data for long periods of time, you can use a software product called a *relational database management system*—abbreviated RDBMS but often, these days, described by the more general term *database*. Database packages include Oracle, Microsoft SQL Server, PostgreSQL, MySQL.

Of course, simple files on disk can also store information for a long time. You might wonder why you should use a database when you can store data in straightforward text files.

The answer is that using databases is safer, and in many cases more convenient, than managing arbitrary files on a disk. Databases are designed to store structured information. When you write to files, you must devise a way to represent your data manually. For instance, you can separate names and phone numbers with commas, and then store each user's record on a different line in the file. But this process is as error-prone as it is tedious, and it makes your file idiosyncratic. A missing comma might cause you to greet a user as "Dear Mr. 203-432-6687." If other people or applications need to read your data, they must learn the format you personally devised and implemented. By contrast, databases provide standard interfaces to your data, and they help you organize it.

Databases also help keep your data safe and consistent. A database can be set up to ensure that every entry for a customer comes with a phone number and birthdate, so you don't accidentally end up with partial data. When databases guarantee the consistency—or *integrity*—of data, they let you focus on other considerations. You can set up a database once (or have a database administrator set one up) and then read and write data to it, confident in its ability to handle the data quickly and accurately.

All the tags we introduce in this chapter come from JSTL's `sql` tag library. (See chapter 2 for more information on JSTL's various tag libraries.) To use any of the examples in this chapter, you'll need to use a directive like the following at the top of your pages:

```
<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql" %>
```

To prepare connections to this database, we'd use the following tag:

```
<sql:setDataSource
  driver="org.hsqldb.jdbcDriver"
  url="jdbc:hsqldb:."
  user="sa"
  password="donkey"/>
```

Because this `<sql:setDataSource>` tag doesn't have a `var` or a `scope` attribute, it will replace the page's default database. That is, any other database tags that appear later in the same page will use the database identified by this tag's attributes. Suppose we add just a `scope` attribute, as follows:

```
<sql:setDataSource
  driver="org.hsqldb.jdbcDriver"
  url="jdbc:hsqldb:."
  user="sa"
  password="donkey"
  scope="session" />
```

With this new attribute, the `<sql:setDataSource>` tag will set up a new default database for the user's session. We could also specify `scope="request"` or `scope="application"` if we wanted to set a default for the request or application scope.

Setting a default is useful when your application has only—or primarily—one database to use. For instance, you can put an `<sql:setDataSource>` tag in a common header file included with `<c:import>` into your page. If such an `<sql:setDataSource>` tag has a `scope="application"` attribute, then it sets an application-wide default, and you may never have to think about `<sql:setDataSource>` again until you start working on a new application.

When different default databases exist for the page, request, session, and application scopes, then JSTL's database tags use page first, followed by request, session, or application. This sequence lets you set a default for a specific scope without destroying the defaults for more general scopes. For instance, you can use `<sql:setDataSource>` in a single page but rely on a session-scoped default database for other pages.

If your application works with multiple databases, then instead of using `<sql:setDataSource>` to set a default connection, you might instead use it to expose a scoped variable that represents a database. You can do this by adding a `var` attribute:

```
<sql:setDataSource
  driver="org.hsqldb.jdbcDriver"
  url="jdbc:hsqldb:."
  user="sa"
```

```
<sql:query var="result">
  SELECT NAME, IQ FROM USERS WHERE IQ > 120
</sql:query>
```

The SQL query in this tag produces a result with exactly two columns: `NAME` and `IQ`. The number of rows depends on the data itself—in this case, on the number of people in the `USERS` table who have IQs above 120.

Table 9.3 shows a sample result for this SQL query. The user named Richard has an IQ of 132, Jonathan weighs in at a less-impressive 121, and so on.

Table 9.3 A sample result from a database, with two columns and five rows

NAME	IQ
Richard	132
Jonathan	121
Liz	140
Michael	162
Rachel	149

The job of `<sql:query>` is to retrieve a result—just like that in table 9.3—and expose it as a scoped variable. Such a scoped variable isn't as simple as a string or number; instead, it's divided into a number of properties. These properties let you access two things about a database result:

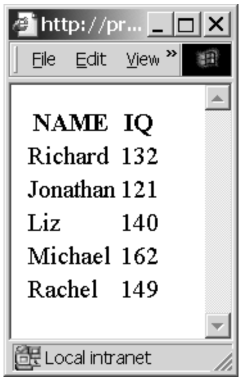
- The data in the table
- Information about the data (often called *metadata*)

Figure 9.3 shows all the properties of the variable that each `<sql:query>` exposes. The first two, `rows` and `rowsByIndex`, are for accessing data. The remaining properties—`columnNames`, `rowCount`, and `limitedByMaxRows`—just help describe the data.

Accessing metadata

Let's begin by looking at the metadata. Suppose you've used an `<sql:query>` tag to create a variable called `result`. The simplest property of this `result` variable is `rowCount`. The expression `${result.rowCount}` lets you retrieve the number of rows in the result. For instance, for table 9.3, `rowCount` would be 5, because five pairs of `NAME` and `IQ` values are listed.

You can also use the `result` variable to retrieve the names of the columns in the result. The `columnNames` property is a list of column names. Recall from chapter 3 that you can access the items in an ordered list using square brackets (`[]`) and index



NAME	IQ
Richard	132
Jonathan	121
Liz	140
Michael	162
Rachel	149

Figure 9.5
Sample output from printQuery.jsp, using the data shown in table 9.3. The generic printQuery.jsp page accepts any result from `<sql:query>` and formats it as a simple HTML table.

9.3.3 *Limiting the size of a query's result*

We use databases because they're good at storing large amounts of data. If all applications managed only a small amount of data, a general-purpose, relational database would probably be overkill. The size of databases, though, can lead to a problem: it becomes easy, with a simple query, to retrieve a set of results that is unmanageably large. For example, the documentation for PostgreSQL, a free high-quality database, says that some PostgreSQL installations have databases 60GB in size. (That's more than 64 *billion* characters.)

Imagine that your application has a large database, and you perform a query based on user input. You have a page that prints data for all customers who match the user's keyword. Now, suppose the user enters an uninspired keyword like "Bob" that matches 50,000 rows. JSTL lets you prevent the query from going out of control by using two attributes of the `<sql:query>` tag: `maxRows` and `startRow`.

The maxRows attribute

The `maxRows` attribute is straightforward. When it appears in an `<sql:query>` tag, it ensures that no more than a specific number of rows will be stored by the scoped variable that `<sql:query>` stores. For example, the following tag might produce a very large result named `customers`:

```
<sql:query var="customers">
  SELECT * FROM CUSTOMERS
</sql:query>
```

However, this tag will never store more than 20 rows in `customers`:

```
<sql:query var="customers" maxRows="20">
  SELECT * FROM CUSTOMERS
</sql:query>
```

9.5.1 Template queries

One way to use queries like this is to customize them with simple JSP, just like you customize an HTML page. After all, JSP is great for adding dynamic content to otherwise static text. For example, we can use JSTL's `<c:out>` tag (see chapter 3) to fill in part of an SQL query:

```
<sql:query var="result">
  SELECT * FROM TABLE
  WHERE CUSTOMER_NUMBER=<c:out value="${customerNumber}"/>
</sql:query>
```

This is a simple way to use JSTL to modify a query. It effectively plugs the value of a scoped variable into an SQL statement. However, this technique is more problematic than it might seem at first. You're not always working with numbers; sometimes you'll use strings. In SQL, strings must be quoted with single quotes. So far, that doesn't sound like a problem; we could just insert the quotes manually, like this:

```
WHERE CUSTOMER_NAME='<c:out value="${customerName}"/>'
```

However, if the customer's name contains a quotation mark, like `David O'Davies`, the result will be the following unfortunate text:

```
WHERE CUSTOMER_NAME='David O&#039;Davies'
```

Because `<c:out>` escapes the quotation mark by default, it yields an incorrect value; SQL does not understand XML escaping.

There's even a security risk in building up queries manually. If you decide to get around `<c:out>`'s escaping problem by using the attribute `escapeXml="false"`, a malicious user could purposely corrupt the query to retrieve private information or even alter your database. For example, suppose the user, instead of a name like `David O'Davies`, enters the following unexpected text:

```
David' OR CUSTOMER_NAME <> 'David
```

In this case, the end of the query becomes

```
WHERE CUSTOMER_NAME='David' OR CUSTOMER_NAME <> 'David'
```

Because every customer name is either equal or not equal to `'David'`, this query will match every row in the table! Therefore, it's not usually a good idea to use `<c:out>` to build up an SQL statement yourself.

9.5.2 Safe, convenient parameters with <sql:param>

JSTL lets you avoid these problems by using a special syntax borrowed from JDBC, the Java package that supports database connectivity. Using this syntax, you can five operations must be treated as a single unit. Operations that need to succeed or fail as a single unit are known as *transactions*.

WARNING Although most well-engineered database systems support transactions, not every software product does. Before using the tags in this section, check with your software's documentation or your database administrator to ensure your database supports transactions.

9.6.1 The <sql:transaction> tag

In JSTL, transactions let you treat a series of `<sql:query>` and `<sql:update>` tags as part of a unified whole. All query and update tags within a transaction succeed or fail together; there is no middle ground. If the end of a transaction doesn't complete successfully, the beginning is stricken from the record: the database pretends it never happened. This sort of pretending is formally called *rolling back*, and it involves restoring the database's state to a prior one—specifically, to the way things were before the first `<sql:update>` in the transaction executed.

JSTL supports transactions with a tag called `<sql:transaction>`. This tag acts as a parent tag for `<sql:update>` and `<sql:query>` tags. Each `<sql:transaction>` tag

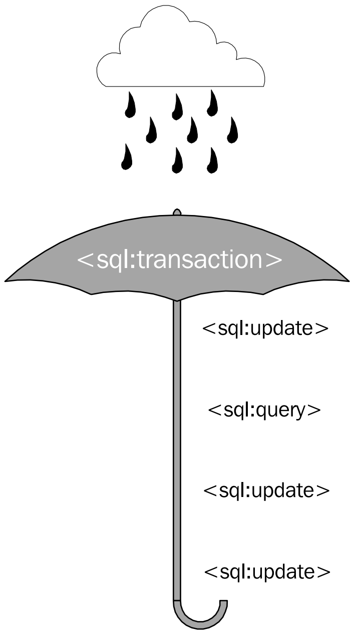


Figure 9.8
The `<sql:transaction>` tag protects its `<sql:update>` and `<sql:query>` children. It does so by ensuring that these children succeed or fail as a unit. If any of the individual steps under an `<sql:transaction>` fails, the database will be rolled back to a prior state, as if the transaction had never begun.

number. We can create a suitable table, which we'll call `counter`, using the following SQL command:

```
create table counter (
    counter integer
)
```

NOTE You'll need to type this command into your database's text interface. The instructions for doing so vary from database to database, so you'll need to check with your database's manual or administrator to determine how to send it commands manually. (My hsqldb tutorial at Manning Publication's web site describes the procedure for hsqldb. See appendix D for its URL.)

If you have trouble sending commands to your database manually, you can enter the command into an `<sql:update>` tag and run the tag manually by loading its page. This technique is somewhat clumsy, but it's a decent alternative. For instance, the following tag will create the `counter` table in the default database:

```
<sql:update>
  create table counter (
    counter integer
  )
</sql:update>
```

The `counter` table has a single column, also called `counter`, which stores an integer. Our table will contain a single row, and this row's value for the `counter` column will represent the current tally of web-page hits. Before we use the counter, we'll need to create this row manually. To do so, we can run the following SQL command:

```
insert into counter(counter) values(0)
```

This line initializes our database and sets the counter's starting value to `0`.

Now that we've set up the `counter` table, we're ready to look at a page that uses it. Listing 9.2 shows such a page.[4]

Listing 9.2 counter.jsp: a simple hit counter

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql" %>

<sql:transaction>
```

[4] Remember, this chapter's examples, including listing 9.2, assume you have a default database set up. If you don't, you'll need to use the `<sql:setDataSource>` tag and the `dataSource` attribute for `<sql:transaction>`. See section 9.2.

will output

```
500000.01
```

The `<c:out>` tag prints the number in a simple, default form. Integers, for instance, are presented as a sequence of digits. Floating-point numbers are displayed similarly, but with a decimal point (.) separating some digits from the other digits. This simple format might be okay for many of your pages, but if a page prints out a lot of numbers, or if presenting numbers is a page's main job, then you'll probably want more control over how numbers are printed. That's what `<fmt:formatNumber>` is for.

10.1.1 Basic usage of <fmt:formatNumber>

In its simplest form, you can use the `<fmt:formatNumber>` just like `<c:out>`. For example, we can write

```
<fmt:formatNumber value="${netWorth}"/>
```

This usage is similar to `<c:out>`: the tag has an attribute, `value`, that points to the number we want to print out. However, even in this simple form, the `<fmt:formatNumber>` tag does something more interesting than `<c:out>`: it prints the number using its best guess about what format the user wants to see. Web browsers can convey information about their *locale*—essentially, their location and preferred formats for numbers, dates, and other data. The `<fmt:formatNumber>` tag can automatically sense this locale and customize its output. So, if `${netWorth}` equals `500000.01`, the simple `<fmt:formatNumber>` we just presented will output the following values for these countries:

Country	Sample numeric format
United States	`500,000.01`
France	`500 000,01`
Germany	`500.000,01`
Switzerland	`500'000.01`

As this table shows, the format is different for the United States, France, Germany, and (as you might know if you have a Swiss bank account) Switzerland.

TIP If you're using Windows and Internet Explorer, you can experiment with different locales by going to the Start menu and choosing Settings, then Control Panel, and finally Regional Options. From there, the General tab lets you pick your locale. (These instructions may vary slightly if you use

WARNING By default, the `currencyCode` attribute works only on JDK 1.4 and later versions. Check with your system administrator if you're not sure what version of the JDK your JSP container runs on. If you use the `currencyCode` attribute on a system that has an older version of Java, the code you use will be printed as a currency symbol. I wouldn't recommend this approach; use the `currencySymbol` attribute instead.

A separate attribute, `currencySymbol`, lets you set a specific currency symbol to use. For instance, you might write `currencySymbol="$"` to indicate the dollar.

10.1.5 Grouping digits together … or not

By default, `<fmt:formatNumber>` arranges digits into groups that are appropriate for the browser's locale. For example, as you saw earlier, the number `500000.01` is printed as `500,000.01` in English. This formatting is used because of the locale's customary rules: groups of three digits are separated by a comma (`,`). In Switzerland, the style uses groups of three digits separated by an apostrophe (`'`).

You can use the `groupingUsed` attribute to explicitly shut off this grouping, which will cause the number to be printed without any group separator. Figure 10.1 shows an example.

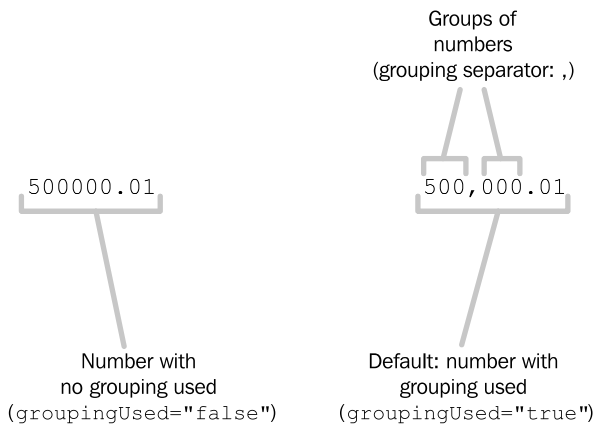

Groups of numbers (grouping separator: ,)
500000.01
500,000.01
Number with no grouping used (groupingUsed="false")
Default: number with grouping used (groupingUsed="true")

Figure 10.1 By default, the `<fmt:formatNumber>` tag arranges numbers into groups of digits, using a locale-specific group separator. You can shut off this behavior with the `groupingUsed` attribute.

The following two tags are equivalent because `groupingUsed="true"` is the default:

```
<fmt:formatNumber value="500000.01" />
<fmt:formatNumber value="500000.01" groupingUsed="true" />
```

For the English locale, these tags both print

```
500,000.01
```

For more information on how patterns work, you can read the Javadoc page for the `DecimalFormat` class, which (for the version of Java that was current at the time this chapter was written) should be available at http://java.sun.com/j2se/1.4/docs/api/java/text/DecimalFormat.html.

10.2 Printing dates with <fmt:formatDate>

Just as JSTL provides support for formatting numbers with `<fmt:formatNumber>`, it gives you `<fmt:formatDate>` to help print out dates and times. Table 10.4 lists its attributes.

Table 10.4 `<fmt:formatDate>` tag attributes

Attribute	Description	Required	Default
`value`	Date to print	Yes	*None*
`type`	Whether to print dates, times, or both	No	`date`
`dateStyle`	Preformatted style to use for the date	No	`default`
`timeStyle`	Preformatted style to use for the time	No	`default`
`timeZone`	Time zone to use when formatting the date	No	*See section 10.5*
`pattern`	Explicit formatting pattern to use	No	*None*
`var`	Variable to expose the formatted date (as a string)	No	*None*
`scope`	Scope in which to expose the formatted date	No	`page`

10.2.1 Differences from <fmt:formatNumber>

Besides the obvious difference that `<fmt:formatDate>` is for printing dates and `<fmt:formatNumber>` is for printing numbers, a few syntactic differences exist between the two tags. First, `<fmt:formatDate>` always takes a `value` attribute; this attribute is required. In addition, it cannot accept data from its body.

The `value` attribute for `<fmt:formatDate>` must point to a date variable; it can't simply point to a string that represents a date, like `"Jan 1, 2001"`. There's no good, unambiguous way for `<fmt:formatDate>` to accept and interpret strings as dates. That job is given to another JSTL tag, `<fmt:parseDate>`, which we'll encounter later.

You can get real date variables a few ways. You might retrieve one from a database or receive one from back-end Java code. Or, you might use the `<fmt:parseDate>` tag we just mentioned, which we'll describe in section 10.4, to produce a date variable. You can also produce a date using an advanced tag called `<jsp:useBean>`. We'll leave this tag's inner workings as magic for now; we'll mention it again in

Tag	Output
`<fmt:formatDate value="${d}" pattern="d MMM yyyy"/>`	`5 Apr 2002`
`<fmt:formatDate value="${d}" pattern="m 'after' h"/>`	`4 after 9`
`<fmt:formatDate value="${d}" pattern="MMMM ''yy"/>`	`April '02`
`<fmt:formatDate value="${d}" pattern="EEEE 'at' h:mm a"/>`	`Friday at 9:04 PM`

10.3 Reading numbers with <fmt:parseNumber>

So far in this chapter, we've only discussed outputting data—formatting dates and numbers and then (usually) printing them or (less frequently) saving them to scoped variables. JSTL has two tags that help you handle input: `<fmt:parseNumber>` to help you read numbers, and `<fmt:parseDate>` to help you read dates.

In many cases, you don't need these tags. As you saw in chapter 4, JSTL lets you treat simple numbers as strings, and vice versa. For example, if the request parameter named `boundary` equals the number `50` because that's what the user entered in an HTML form, we can say

```
<c:forEach … end="${param.boundary}">
```

and the `<c:forEach>` tag will know to stop its iteration after the fifty-first element.

The `<fmt:parseNumber>` tag is specifically for cases in which you need to *parse*—or interpret—more complicated numbers. If the user enters `50,000` (including the comma), or if you read values that contain commas or spaces from an XML file or database, you can't treat these values as numbers; you need to parse them first.

Table 10.7 lists the attributes that `<fmt:parseNumber>` accepts.

Table 10.7 `<fmt:parseNumber>` tag attributes

Attribute	Description	Required	Default
`value`	The string to parse into a number	No	*Body*
`type`	How to parse the number (`number`, `currency`, or `percent`)	No	`number`

would work for the English locale, but

```
<fmt:parseDate value="Aug 24 1981"/>
```

would lead to an error because it lacks a comma. This behavior makes the default case almost useless for processing input from users, because it's usually inappropriate to force users to be so specific in the values they enter. However, this use of `<fmt:parseDate>` is appropriate in a few situations:

- It's useful if you know you're getting data that was printed with `<fmt:formatDate>`.
- You can also use this simple form of `<fmt:parseDate>` if you're generating a string based on individual fields of user input—for instance, a pull-down menu for month, followed by another one for date, and so on. See chapter 11 for an example of this technique.

If given a `var` attribute, `<fmt:parseDate>` stores a scoped variable that holds a date and time (the time is always midnight in this simple case). Otherwise, it prints the date in a somewhat ugly, unlocalized format:

```
Sat Aug 24 00:00:00 EDT 2002
```

You therefore almost always want to use a `var` attribute with `<fmt:parseDate>` (except, perhaps, if you're just testing your page).

10.4.2 Changing how <fmt:parseDate> parses dates

The `<fmt:parseDate>` tag comes with four attributes that let you change how it parses dates. The first is simple: you can use the `type` attribute to let the tag parse times as well as dates. Just as with `<fmt:formatDate>`, the `type` attribute has three possible values: `date`, `time`, and `both`; `date` is the default. If you specify `type="time"`, then `<fmt:parseDate>` tries to read and parse a time in the locale's default representation (for example, `"07:45:02 PM"`). For `type="both"`, the tag expects a default date/time combination, like

```
Aug 24, 2002 08:52:00 PM
```

The `type` attribute is somewhat limited when used alone. It can be useful when used in conjunction with two more powerful attributes, `timeStyle` and `dateStyle`. They let the `<fmt:parseDate>` tag accept the sorts of values shown in table 10.5, earlier in this chapter.

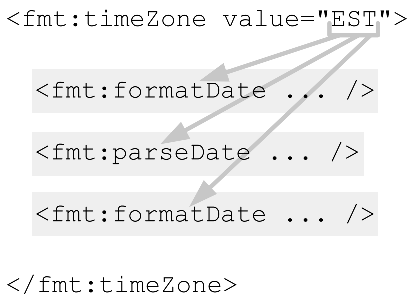


Figure 10.3
When the `<fmt:timeZone>` tag surrounds one or more `<fmt:formatDate>` or `<fmt:parseDate>` tags, the time zone from `<fmt:timeZone>` automatically applies to each of these child tags.

10.6 Overriding locales with <fmt:setLocale>

Throughout this chapter, I've mentioned that tags use the user's web browser's preferred locale by default. But JSTL page authors and back-end Java programmers can also influence the locale used for the `<fmt:format…>` and `<fmt:parse…>` tags. Doing so can be useful if you want to give users a choice of locale instead of letting the browser automatically speak for them.

Just as with time zones, back-end programmers have control over what locales are used; they can explicitly choose to override the browser's locale. See chapter 14 for information (geared to programmers) about how to do this.

JSTL also lets you control the locale using a tag: `<fmt:setLocale>`. Table 10.11 lists this tag's attributes.

Table 10.11 `<fmt:setTimeZone>` tag attributes

Attribute	Description	Required	Default
`value`	Name of a locale to use (see section 10.6.1)	Yes	*None*
`variant`	Specific variety of the chosen local to use (see section 10.6.1)	No	*None*
`scope`	Scope for which to override the locale	No	`page`

As table 10.11 shows, `value` is always required. Using `value`, you can specify the name of a locale. This locale will become the new default for the scope identified by the `scope` attribute—or for the current page by default, if you don't specify a `scope` attribute. (The `variant` attribute is beyond the scope of this book.)

10.7.2 *Loading a bundle family with <fmt:bundle> and <fmt:setBundle>*

If no back-end Java code manages message bundles for your pages, or if you want to override the bundle, you can use the `<fmt:bundle>` and `<fmt:setBundle>` tags.

Table 10.15 lists the attributes for `<fmt:bundle>`.

Table 10.15 `<fmt:bundle>` tag attributes

Attribute	Description	Required	Default
`basename`	Name of the resource-bundle family to use	Yes	*None*
`prefix`	String to prepend to each key (for long key names)	No	*None*

Table 10.16 lists the attributes for `<fmt:setBundle>`.

Table 10.16 `<fmt:setBundle>` tag attributes

Attribute	Description	Required	Default
`basename`	Name of the resource-bundle family to use	Yes	*None*
`var`	Variable to expose the bundle	No	*None*
`scope`	Scope in which to expose the bundle	No	`page`

The difference between `<fmt:bundle>` and `<fmt:setBundle>` is the same as the difference between `<fmt:timeZone>` and `<fmt:setTimeZone>`. The tags with `set` in their names change the defaults for an entire scope, whereas the tags without `set` apply only to their child tags.

JSTL's two bundle-related tags let you describe a group of related bundles using the `basename` attribute. You'll know the *base name* of a bundle if you've internationalized an application yourself; if you're using someone else's bundle, then whoever internationalized the application should tell you the base name.

The `<fmt:bundle>` tag changes the bundle for all the `<fmt:message>` tags in its body. For instance:

```
<fmt:bundle basename="my.bundle">
  <fmt:message key="my.key.Welcome"/>
  <fmt:message key="my.key.Error"/>
</fmt:bundle>
```

When you use the `<fmt:bundle>` tag like this, you can give it a `prefix` attribute. This attribute is a string that is added before every key in each `<fmt:message>` tag

Part 3

JSTL in action

So far, we've looked at what JSTL is and how it works. You've seen a few examples of JSTL in action, but now we'll examine more closely how to handle practical tasks using JSTL. In chapter 11, we'll show how you can use JSTL to address some common but small-scale needs. In the chapters after that, we'll discuss more in-depth examples of web development with JSTL.

A minor warning is in order. In some cases, you won't be using JSTL as a stand-alone technology. You might use it with Jakarta Struts, for example, or with tag libraries developed specifically for your site. We can't cover all the technologies that JSTL might interact with in this book; therefore, although most of the material in part 3 is core, nuts-and-bolts stuff that you can use immediately, some of the examples push JSTL to its limits. This is intentional; I think the best way to learn a technology is by trying to use it creatively. So don't be surprised if some of the examples use JSTL for tasks that you might otherwise solve with a custom, local library or with Struts. The examples here aren't designed to demonstrate principles of web-application architecture; books like *Web Development with JavaServer Pages*[1] already address that topic quite well. Instead, my goal is to show you as many uses of JSTL as you could possibly want to see.

My hope is that these "stretches" will serve as a good reference as your knowledge of JSTL progresses. You just might find that JSTL can handle more than you'd expect!

---

1 Duane Fields, Mark Kolb, and Shawn Bayern, 2nd ed. (Manning Publications, 2001).

Some tasks never go away. If I had a dime for every time I had to write a JSP page that signed up new users for a web application, I'd probably have more than $1.50 by now.

However, JSTL makes lots of common tasks easier. In this chapter, we look at how to use JSTL to address some common, specific issues like reading a date from a user, accepting `<input type="checkbox">` parameters, and handling errors. These are all practical, but somewhat isolated, examples.

They're meant to help you generalize about JSTL. For instance, if you ever need to read a date from a user of your web page, section 11.2 is a cookbook-like solution. But even if you don't need to read dates frequently, understanding the examples in section 11.2 will be useful to solidify your knowledge of the `<fmt:parseDate>` and `<fmt:formatDate>` tags. Similarly, the discussion of `paramValues` applies equally well to `headerValues` and other collections; `paramValues` is just more common and practical.

Before leaving this chapter, we get as far as a basic HTML-form handler that validates its input and prepares to register a new user. Chapters 12 and 13 show more complete, application-like examples.

11.1 Handling checkbox parameters

When we originally discussed JSTL's expression language in chapter 3, you saw how to use the expression language to handle HTML forms. For instance, an HTML form parameter from a tag like

```
<input type="text" name="username" />
```

shows up to your JSTL tags as the expression `${param.username}`.

In chapter 3, however, I mentioned that checkbox parameters are special because the same name can map to multiple values. Suppose we have an HTML form with the following tags:

```
<input type="checkbox" name="language" value="english" />
<input type="checkbox" name="language" value="spanish" />
<input type="checkbox" name="language" value="french" />
```

If the user checks all three boxes, then the `language` parameter will have three values: `english`, `spanish`, and `french`.

You can access a collection that contains all these values by using the expression `${paramValues.`*`name`*`}`, where *name* is the name of the parameter you're looking for. You can use the `<c:forEach>` tag to loop over the individual parameters in this collection and handle them one at a time.

```
  You described our customer service as
  <ul>
  <c:forEach items="${paramValues.feedback}" var="adj">
    <c:choose>
      <c:when test="${adj == 'satisfactory'}">      <-- Checks to see if the
         <font size="+2">                                current parameter
      </c:when>                                          equals a specific word
      <c:otherwise>
         <font size="-2">
      </c:otherwise>
    </c:choose>
    <li><c:out value="${adj}"/></li>
    </font>
  </c:forEach>
 </c:when>
 <c:otherwise>
  You didn't choose any feedback checkboxes.
 </c:otherwise>
</c:choose>
```

Again, this listing's demonstration is somewhat silly, but it demonstrates an important technique: using a `<c:choose>` tag directly below a `<c:forEach>` tag in order to take special action depending on the parameter's value. Instead of the trivial

```
<font size="+2">
```

it's easy to see how you could do something more useful, like

```
<sql:update>
  UPDATE feedback SET satisfactory = satisfactory + 1
</sql:update>
```

This code would update a database when the page was loaded, but only if the box for `satisfactory` customer service had been checked.

11.2 Accepting dates

Although HTML forms are flexible, they don't do a lot of user-interface work for you. HTML lets you choose from a few basic types of input fields—text boxes, selection boxes, radio buttons, and so on—but it doesn't make it easy to accept special kinds of formatted input from the user. For instance, if you need your user to enter a date or time, HTML doesn't give you any tools to handle this input automatically. Instead, you have to construct and interpret individual form fields.

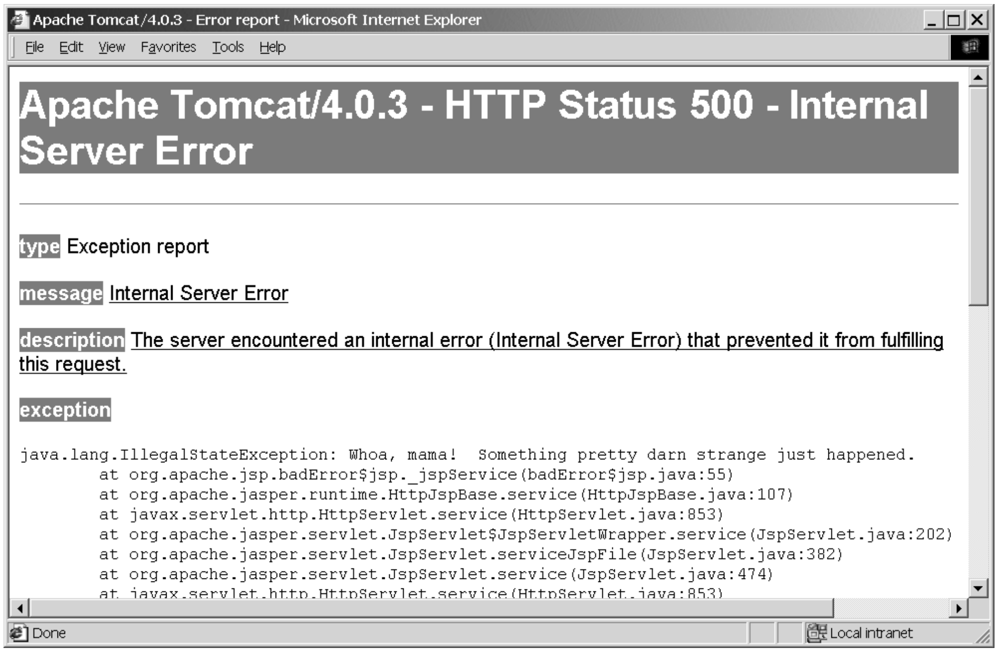
Using JSTL, it's easy to write an HTML form that asks the user for a date or time. Of course, you could always prompt the user for a date by displaying a text box and asking them to type one in. You could then parse the date with the `<fmt:`

your errors from multiple pages. The advantage of such a page is that it lets you easily apply the same behavior to a group of pages; simply point all of them at the same error page, and then figure out what to do in that page. It's an ideal place to say, "Something went wrong. Please try again."

11.3.1 *Ignoring the issue*

JSP and JSTL don't force you to think about errors. Some web applications' JSP pages don't have to worry about errors, because the application could have been deployed with error handling already set up. Just as back-end Java programmers and application deployers can manage things like default locales, time zones, and databases, they can also manage default error handling.

Separately, if you're reasonably confident that your pages won't encounter any unexpected errors—or if you're happy with the look and feel of your JSP container's default error message—then you can forget about them and move on. (See


Apache Tomcat/4.0.3 - Error report - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Done
Local intranet

Apache Tomcat/4.0.3 - HTTP Status 500 - Internal Server Error

type Exception report

message Internal Server Error

description The server encountered an internal error (Internal Server Error) that prevented it from fulfilling this request.

exception

```
java.lang.IllegalStateException: Whoa, mama!  Something pretty darn strange just happened.
        at org.apache.jsp.badError$jsp._jspService(badError$jsp.java:55)
        at org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:107)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
        at org.apache.jasper.servlet.JspServlet$JspServletWrapper.service(JspServlet.java:202)
        at org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:382)
        at org.apache.jasper.servlet.JspServlet.service(JspServlet.java:474)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
```

Figure 11.6 By default, errors in your JSP pages result in behavior that might look like this—or different, depending on what your JSP container decides to show. The point is, you probably don't want your important pages to produce such errors. Unless back-end Java programmers supporting your application have promised to take care of errors, you should catch and handle them yourself—or use a JSP `errorPage`.

This line tells the JSP container that it can use this page as an error page.

Note that you can use an error page and the `<c:catch>` tag from the same page. The error page applies only if an error occurs but isn't captured by a `<c:catch>` tag.

Error pages are particularly useful when you want to provide a single, easily changeable way to handle all your application's errors. If you design an error page that looks like the rest of your site—for instance, with the same headers, footers, fonts, and color scheme—then your site's error handling will look much more professional.

Creating an error page

Listing 11.7 shows a simple error page.

Listing 11.7 errorPage.jsp: a sample JSP error page

```
<%@ page isErrorPage="true" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>

<h4>Error!</h4>

<p>Something bad happened in one of your pages:</p>

<p><c:out value="${pageContext.exception.message}"/></p>
```

Note how we use the expression `${pageContext.exception.message}` to print out information about the error that occurred. The `pageContext.exception` variable is just like the scoped variable that `<c:catch>` stores: you can use its `message` property to get information about the error that occurred, or you can print out the error itself, which usually includes more technical information.[1]

Listing 11.8 shows how to use an error page.

Listing 11.8 useErrorPage.jsp: a page that uses a JSP error page

```
<%@ page errorPage="errorPage.jsp" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>

<fmt:parseDate value="A midsummer night"/>    <— Always an error
```

If we load the page in listing 11.8, we'll always get an error; the string `"A midsummer night"` is not a valid date. Without the first line—`<%@ page errorPage="errorPage.jsp" %>`—we'd get an error much like the one in figure 11.6. But with the

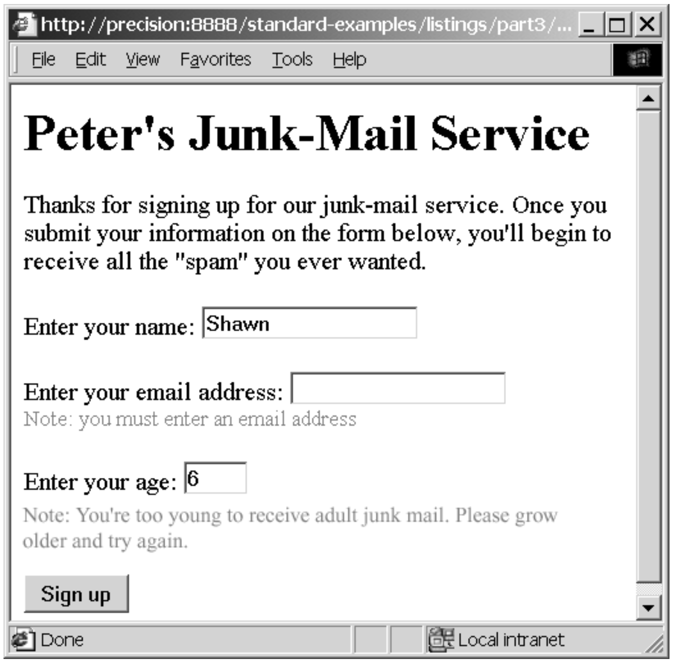
[1] Note that the error message's details can vary from one implementation of JSTL to another.

```
   <small><font color="red">
     Note: you must enter a name
   </font></small>
  </c:if>
  </p>

  <p>
  Enter your email address:
  <input type="text" name="email"
    value="<c:out value="${param.email}"/>" />
  <br />
  <c:if test="${noEmail}">
   <small><font color="red">
     Note: you must enter an email address
   </font></small>
  </c:if>
  </p>

  <p>
  Enter your age:
  <input type="text" name="age" size="3"
    value="<c:out value="${param.age}"/>" />
  <br />
  <c:choose>
    <c:when test="${noAge}">
     <small><font color="red">
       Note: you must enter your age
     </font></small>
    </c:when>
    <c:when test="${badAge}">
     <small><font color="red">
       Note: I couldn't decipher the age you typed in
     </font></small>
    </c:when>
    <c:when test="${youngAge}">
     <small><font color="red">
       Note: You're too young to receive adult
       junk mail.  Please grow older and try again.
     </font></small>
    </c:when>
  </c:choose>
  </p>

  <input type="submit" value="Sign up" />

</form>
```

❻ Prints an error message if appropriate

❼ More complicated error-handling logic

The page begins by looking more like a page that *handles* a form than a page that *displays* one. This page does both, but we start with the validation logic. First, we use a `<c:if>` tag to determine whether this page is (a) responding to a submitted


http://precision:8888/standard-examples/listings/part3/...
File Edit View Favorites Tools Help
Sign up
Done
Local intranet

Peter's Junk-Mail Service

Thanks for signing up for our junk-mail service. Once you submit your information on the form below, you'll begin to receive all the "spam" you ever wanted.

Enter your name: Shawn

Enter your email address:
Note: you must enter an email address

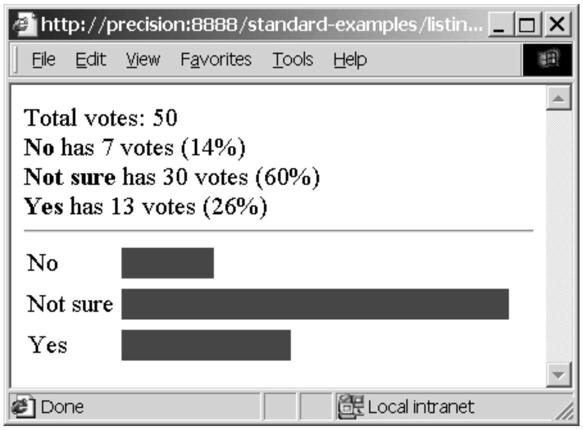
Enter your age: 6
Note: You're too young to receive adult junk mail. Please grow older and try again.

Figure 11.12
When the page from listing 11.9 is fed invalid input, it reprints its form with appropriate error messages interspersed. It also explicitly fills in the form with the values the user entered, making it easier for the user to correct them.

11.5 Summary

In this chapter, we discussed a few demonstrations of JSTL in action. Take the following pointers from these examples:

- Because checkbox parameters and cookies come in lists, you often need to loop over them with `<c:forEach>`. You can find checkbox parameters with the expression `${paramValues.`*`name`*`}`, where *name* is the name of the parameter you're looking for.
- If you need to let the user enter dates, you can use multiple expressions in the same attribute value to assemble different request parameters (different form fields) into a date that's parseable by `<fmt:parseDate>`.
- The `<c:catch>` tag lets you handle errors within your page.
- JSP's `errorPage` mechanism helps you control errors that aren't handled in your page.
- You can use `<c:if>` tags, `<c:choose>` tags, and the expression language to validate form input. One common strategy for validating input is to present a page that cycles (see figure 11.10) until it receives correct input. Listing 11.9 shows an example of such a page.


http://precision:8888/standard-examples/listin...
File Edit View Favorites Tools Help
Done
Local intranet

Total votes: 50
No has 7 votes (14%)
Not sure has 30 votes (60%)
Yes has 13 votes (26%)

Figure 12.2 Our survey can respond by printing detailed information about the user's choices, or even a graph of the results.

The `survey_results` table has two columns (see table 12.1).

Table 12.1 Our survey application's database table (`survey_results`) has two columns: one that identifies the survey and another that stores the choices users make.

Column name	Type	Purpose
`survey_id`	`INTEGER`	Stores a number that distinguishes each survey question from the others
`choice`	`VARCHAR(30)`	Contains an individual user's choice for the corresponding `survey_id`

The first column in this table, `survey_id`, contains a number that identifies an individual survey question. For instance, survey `2` could be, "Which of your internal organs do you find most appealing?" and survey `3` could be, "Are you a Democrat or a Republican?" The questions don't need to have anything to do with each other; all questions that your site uses can live side-by-side in the `survey_results` table.

Every row in this table stores an individual user's choice. For instance, if a user submits the value `pancreas` for survey `2`, we add a row that looks like `(2, 'pancreas')`. The next user might vote for the choice `liver`, which would lead us to create a new row: `(2, 'liver')`. The table thus allows duplicate rows, which are fine in a relational database.

TIP We could have structured this table differently. Instead of storing one row for each response, `survey_results` could contain one row for each different choice and could keep a counter that is incremented, much like the counter we used in chapter 9. I decided to use the approach outlined in table 12.1 for two reasons:

```
          </tr>
         </table>
        </td>
       </tr>

       <c:if test="${s.last}">
         </table>
       </c:if>
     </c:forEach>

   </c:otherwise>
 </c:choose>
```

❶ The survey.jsp page starts with an error check. If we don't receive the two parameters we need (`surveyId` and `choice`), then we don't bother going on. We put the error message in a `<c:when>` block and the rest of the page in `<c:otherwise>`. Thus, the page won't try to update or access the database unless it's given the necessary parameters.

❷ The first `<sql:update>` tag saves the user's vote in the database. Note that this tag—and all the database tags in survey.jsp—doesn't use a `dataSource` attribute. As I mentioned at the beginning of this chapter, I assume the page runs in an environment where the default database has been set up correctly. If this is not the case, you'll need to add a `dataSource` attribute to every `<sql:query>` and `<sql:update>` tag in the page. For information on how to do this properly, see chapter 9.

The SQL command that we use to save the user's choice is simple. Remember that the `survey_results` table is supposed to have a row for every individual survey response. Thus, all we need to do is add a row for the current user's vote. To do this, we use the base query

```
insert into survey_results(survey_id, choice)
  values(?, ?)
```

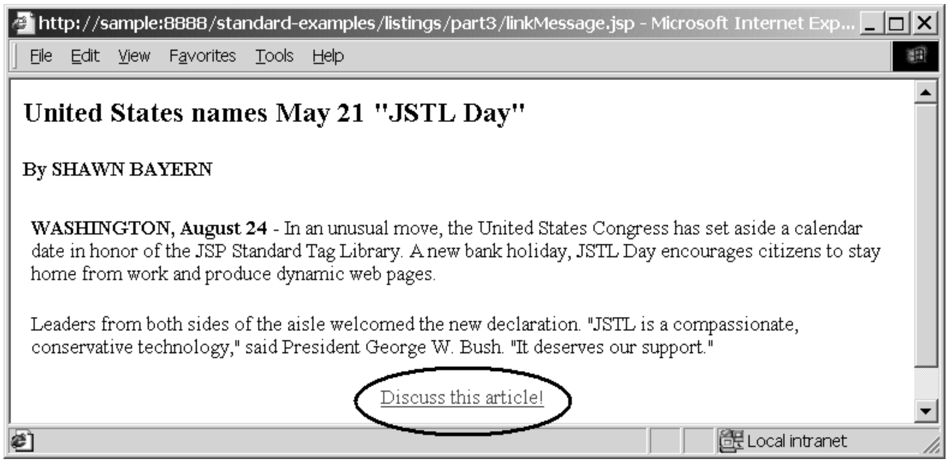
and send it our two parameters using `<sql:param>` tags.

❸ Now, after saving the new result, we want to retrieve all results for the requested survey. We do this with an `<sql:query>` tag that contains an `<sql:param>` tag to pass the `survey_id` number we're interested in. We use a bit of advanced SQL here, so let's go over it carefully.

Our SQL query retrieves two things from the `survey_results` table. It begins as follows:

```
select choice, count(choice) from survey_results
  where survey_id = ?
```

The first thing we're selecting—`choice`—is simple; it's the value of a column. It's clear that we're only interested in rows that have a particular `survey_id`.


http://sample:8888/standard-examples/listings/part3/linkMessage.jsp - Microsoft Internet Exp...
File Edit View Favorites Tools Help
Local intranet

United States names May 21 "JSTL Day"

By SHAWN BAYERN

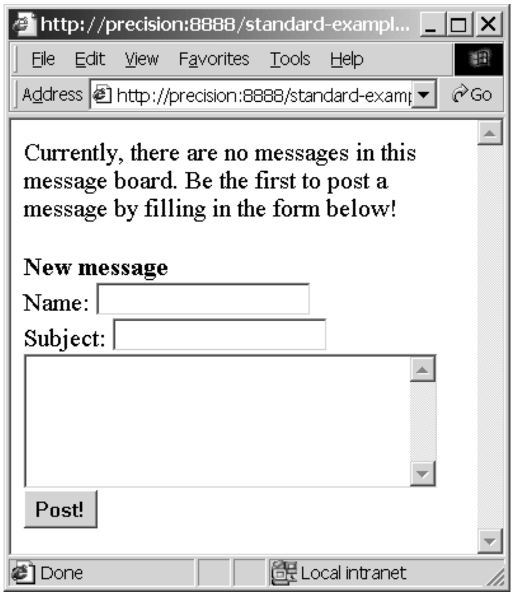
WASHINGTON, August 24 - In an unusual move, the United States Congress has set aside a calendar date in honor of the JSP Standard Tag Library. A new bank holiday, JSTL Day encourages citizens to stay home from work and produce dynamic web pages.

Leaders from both sides of the aisle welcomed the new declaration. "JSTL is a compassionate, conservative technology," said President George W. Bush. "It deserves our support."

Discuss this article!

Figure 12.4 It's easy to add support for our message board to any web page. Simply add a link where it's appropriate. You'll see what form this URL takes in section 12.2.3.

up. If this board doesn't have any messages yet, then users are invited to be the first to post a message (figure 12.5). Otherwise, users see prior messages and can add to the discussion (figure 12.6).


http://precision:8888/standard-exampl...
File Edit View Favorites Tools Help
Address http://precision:8888/standard-examp
Go
Done
Local intranet

Currently, there are no messages in this message board. Be the first to post a message by filling in the form below!

New message
Name:
Subject:
Post!

Figure 12.5 Users who ask for an empty forum are told that they can be the first user to post a message in the forum.

```
    <c:choose>
      <c:when test="${result.rowCount == 0}">        ❷ Checks whether the board
        <p>                                             has any messages
          Currently, there are no messages in this message board.
          Be the first to post a message by filling in the form
          below!
        </p>
      </c:when>
      <c:otherwise>
        <c:forEach items="${result.rows}" var="row">
          <p>
          From: <c:out value="${row.AUTHOR}" /> <br />
          Date: <c:out value="${row.SENT_DATE}" /> <br />      ❸ Prints each
          Subject: <c:out value="${row.SUBJECT}" /> <br />       message
          <blockquote>
             <tt><c:out value="${row.BODY}" /></tt>
          </blockquote>
          <hr />
        </c:forEach>
      </c:otherwise>
    </c:choose>

    <form method="post" action="postMessage.jsp">        ❹ Lets a user enter
      <p>                                                   a new message
      <b>New message</b> <br />
      Name: <input type="text" name="name" /> <br />
      Subject: <input type="text" name="subject" /> <br />
      <textarea cols="30" rows="5" name="body"></textarea> <br />
      <input type="hidden" name="messageBoard"
        value="<c:out value="${param.messageBoard}" />" />    ❺ Passes the
      <input type="submit" value="Post!" />                      current board
      </p>                                                       number
    </form>

  </c:otherwise>
</c:choose>
```

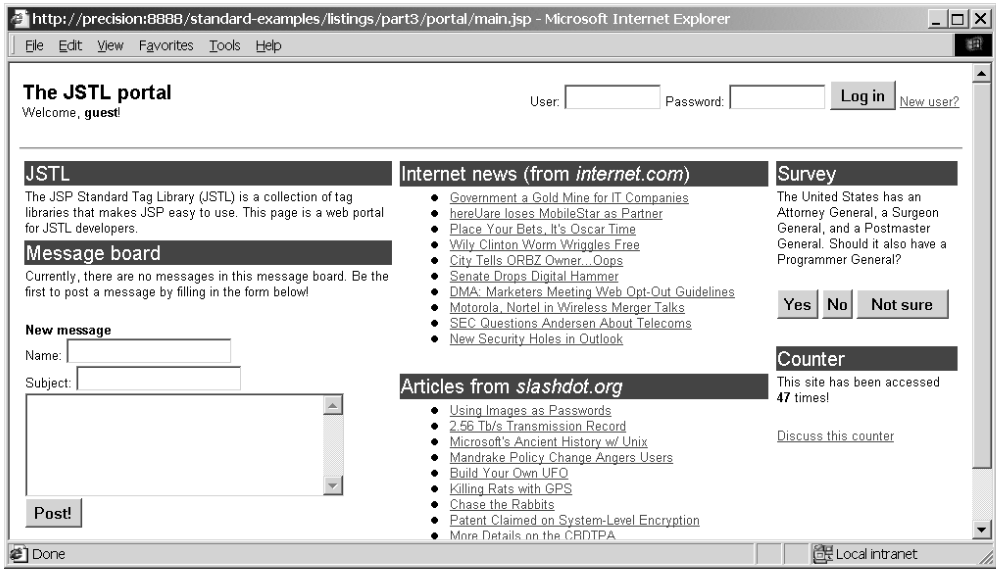
❶ After confirming our parameters using a technique that should look familiar by now, we perform a simple SQL query against our database, finding all messages associated with our `messageBoard` parameter. We use the SQL clause `SELECT *`, which retrieves all columns from the database. Unlike in the survey application from section 12.1, we'll refer to each row's columns by name (for instance `${row.name}`) instead of by number (as in `${row[1]}`).

Note that the SQL query controls the order in which messages display. In listing 12.2, the messages will be displayed in chronological order, starting with the

So far, you've seen how to use JSTL to solve specific problems and to write individual applications. Now, let's look at how to tie it all together.

In this chapter, we'll build a simple web portal, like the one shown in figure 13.1. You've probably run into portals before, such as my.yahoo.com or my.netscape.com. To be honest, I'm not an enthusiastic user of portals. I often keep 12 different browser windows open at once, and I know almost all the URLs I use by heart—so I don't need a single site to tie things together for me. But apparently, lots of users do. They feel more comfortable with a central, customizable site that becomes their home on the web.

Whether you use portals or not, writing one in JSTL will be a good way to tie our separate applications into a single web site. We'll essentially use JSTL to create a primitive content-management system that lets us plug in new channels to our master web site. We'll also see how to register users, let them log in, and personalize the site for them.


http://precision:8888/standard-examples/listings/part3/portal/main.jsp - Microsoft Internet Explorer
File Edit View Favorites Tools Help
User:
Password:
Log in
New user?
Post!
Yes
No
Not sure
Done
Local intranet

The JSTL portal
Welcome, **guest**!

JSTL
The JSP Standard Tag Library (JSTL) is a collection of tag libraries that makes JSP easy to use. This page is a web portal for JSTL developers.

Message board
Currently, there are no messages in this message board. Be the first to post a message by filling in the form below!

New message
Name:
Subject:

Internet news (from *internet.com*)
- Government a Gold Mine for IT Companies
- hereUare loses MobileStar as Partner
- Place Your Bets, It's Oscar Time
- Wily Clinton Worm Wriggles Free
- City Tells ORBZ Owner...Oops
- Senate Drops Digital Hammer
- DMA: Marketers Meeting Web Opt-Out Guidelines
- Motorola, Nortel in Wireless Merger Talks
- SEC Questions Andersen About Telecoms
- New Security Holes in Outlook

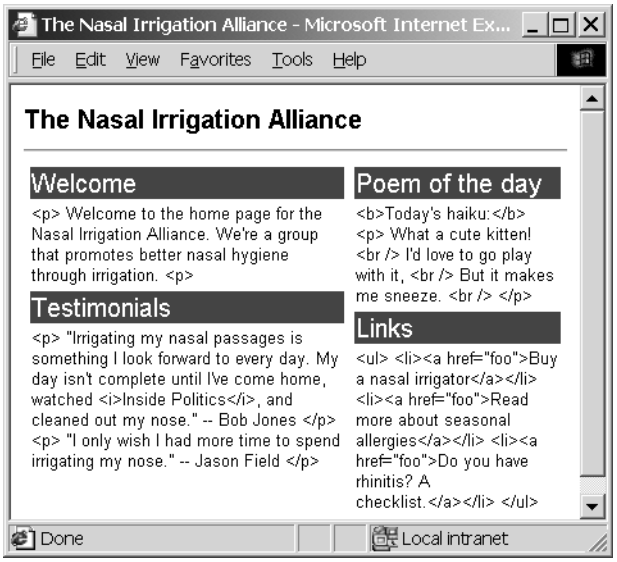
Articles from *slashdot.org*
- Using Images as Passwords
- 2.56 Tb/s Transmission Record
- Microsoft's Ancient History w/ Unix
- Mandrake Policy Change Angers Users
- Build Your Own UFO
- Killing Rats with GPS
- Chase the Rabbits
- Patent Claimed on System-Level Encryption
- More Details on the CBDTPA

Survey
The United States has an Attorney General, a Surgeon General, and a Postmaster General. Should it also have a Programmer General?

Counter
This site has been accessed **47** times!

Discuss this counter

Figure 13.1 In this chapter, we design a simple web portal that combines some features we've written into a single web page. This portal uses JSTL to manage the layout and lets you insert pluggable *channels* as you see fit.


The Nasal Irrigation Alliance - Microsoft Internet Ex...
File Edit View Favorites Tools Help
Done
Local intranet

The Nasal Irrigation Alliance

Welcome

<p> Welcome to the home page for the Nasal Irrigation Alliance. We're a group that promotes better nasal hygiene through irrigation. <p>

Testimonials

<p> "Irrigating my nasal passages is something I look forward to every day. My day isn't complete until I've come home, watched <i>Inside Politics</i>, and cleaned out my nose." -- Bob Jones </p> <p> "I only wish I had more time to spend irrigating my nose." -- Jason Field </p>

Poem of the day

<b>Today's haiku:</b> <p> What a cute kitten! <br /> I'd love to go play with it, <br /> But it makes me sneeze. <br /> </p>

Links

<ul> <li><a href="foo">Buy a nasal irrigator</a></li> <li><a href="foo">Read more about seasonal allergies</a></li> <li><a href="foo">Do you have rhinitis? A checklist.</a></li> </ul>

Figure 13.3
If we remove the `escapeXml="false"` attribute from the `<c:out>` tag that produces each channel's body, we see the raw HTML formatting that was used to produce each channel.

markup; headings will do so less frequently, but we don't want to prevent ourselves from formatting a headline with, for instance, `<i>` or `<u>` tags.)

As I mentioned before, the channel.jsp page knows what information to print by checking its request parameters. It imports the page specified by the request parameter `page` and prints the headline from the parameter `headline`. Note that we could have used the `<c:import>` tag directly in the body of the table (without saving the page we imported in the `body` variable), but the way it's currently arranged is more instructive. If you experiment with channel.jsp, try removing the `escapeXml="false"` attribute from the second `<c:out>` tag; you'll get output that looks like figure 13.3.

Individual channels

Our top-down view of simplePortal.jsp doesn't end with channel.jsp. As I mentioned, channel.jsp doesn't display any content of its own, other than a headline and the HTML formatting for a channel. The final content for our simple portal, as figure 13.4 demonstrates, comes from individual, target pages.

The simplePortal.jsp page decides what pages should ultimately be used as channels. For instance, in listing 13.1, our page created four channels. The first channel's content comes from welcome.html, the second from quotes.html, and so on. In this example, these are just local files in the same directory as the portal, but they could be anywhere else; these files could be loaded from a different directory on the same web server, or even from a completely different web server. Instead of

Because message boards in chapter 12's message system can grow without bound, you normally wouldn't include an entire board in a single portal channel; but you could if you wanted to. (Instead, you'd probably link to a forum, the way the whimsical "Discuss this counter" link in figure 13.1 does.)

Our survey application from chapter 12 integrates cleanly, as well. Simply ask the survey question in a channel that includes an appropriate HTML `<form>`, and have the form open in new window—a technique you saw how to handle in chapter 12.

In figure 13.1, we also included chapter 9's counter in a channel. Let's look more closely at how to do this; it's a good end-to-end example of including dynamic content in the portal.

To begin with, we modify the counter example from chapter 9 to print the count, not simply to store the value as a scoped variable. The result is the counter.jsp page from listing 13.4.

Listing 13.4 counter.jsp: a channel that adds a counter to the portal

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql" %>

<sql:transaction>

  <sql:update>
    update counter set counter = counter + 1
  </sql:update>
  <sql:query var="result">
    select * from counter
  </sql:query>
  <c:set var="count" value="${result.rows[0].counter}" />

</sql:transaction>

<p>
  This site has been accessed
  <b><c:out value="${count}" /></b>
  times!
</p>

<p>
 <a target="_blank"
   href="<c:url value="viewMessages.jsp">
           <c:param name="messageBoard" value="2"/>
         </c:url>">
  Discuss this counter
 </a>
</p>
```

```
</form>

</body>
</html>
```

Most of this page should be familiar from our prior experience in chapter 11. The only thing that's different about register.jsp is that it refers to a request-scoped attribute called `takenName`. It does this twice. First, in its initial validation, it refuses to pass the user on to doRegister.jsp if `takenName` is `true`. Then, it prints out a special error message if `takenName` is `true`.

From the special error message that prints, you can probably figure out what the `takenName` variable represents: it's `true` if the user has entered a username that has already been taken by another user. But what sets this scoped variable, and how does it do so?

The doRegister.jsp page sets the scoped variable. When the user enters values for all three form fields, we forward the user to doRegister.jsp. Normally, doRegister.jsp just adds the new user to a database and bounces the user back to the main

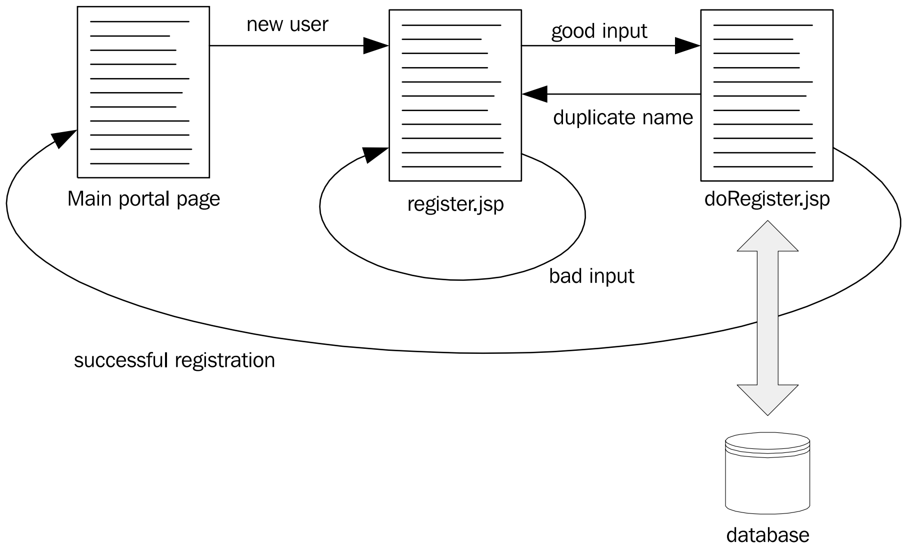


Figure 13.8 New users who try to register with the portal need to fill out the information in the form presented by register.jsp. If the user enters valid information, then register.jsp forwards to doRegister.jsp. Normally, doRegister.jsp adds the user to the database and returns the user to the main portal page. However, if the user enters a duplicate username, doRegister.jsp forwards the user back to register.jsp and gives him a chance to choose a new name.

able to retrieve the name of the current user simply by using an expression like the following:

```
${pageContext.request.remoteUser}
```

Alternatively, back-end Java logic may manage authentication and set some information in the session scope for you to use.

One principle for authentication is important to keep in mind: don't reinforce your front door but leave your back door wide open. If you handle authentication at a single page in your application, consider what might happen if a user tried to access one of your other page's URLs directly. Using session-scoped variables is a good idea because users can't set scoped variables directly. Similarly, using a parameter to pass a secure username from one page to another is a very bad idea, because users *can* set parameters.

A final note: in some environments, network security is important. Sending a cleartext password to a web site might not be acceptable in some environments. You may have noticed that some web pages have URLs that start with *https* instead of *http*. These URLs use the Secure Sockets Layer (SSL), which is a mechanism that can provide both encryption and authentication for the Web.

Overall, be careful when handling the authentication of users. In other words, don't use the system that we built here to protect your valuable assets until you've thought long and hard about computer security!

13.5 Personalizing the site

The portal and any channel can use the session-scoped `user` variable to determine who (if anyone) is logged in and react accordingly. Channels and the portal can also use other session-scoped variables to configure themselves.

You've already seen one simple example of this kind of personalization. The header for the main portal page that we added in section 13.3 will print a customized greeting for the user. For example, instead of saying, "Welcome, guest!" it will say, "Welcome back, Shawn!" if I log in.

Let's look at a few other examples of personalization.

13.5.1 Filling in a form automatically

Look again at figure 13.1. The first column contains a message board from chapter 12's messaging system. Normally, users need to enter a username when posting a message. But because the portal might know who's logged in, let's let the messaging system take advantage of that fact.

Part 4

JSTL for programmers

Parts 1 through 3 of this book have covered everything that web-page authors need to know about JSTL. But if you're a Java programmer, JSTL offers you a few special features.

In part 4, we present more advanced material. None of this material is necessary to use JSTL, but you might find it useful if you're a programmer who wants to get the most out of JSTL. First, we discuss some more advanced uses of JSTL than you saw in parts 1 through 3. Then, we examine ways to configure JSTL tags and otherwise assist the page authors you work with. For instance, you can use Java code to manage locales, time zones, and databases so that your page authors don't have to. Part 4 shows you how.

Finally, we explore how JSTL makes it easier to write custom JSP tags. If you've been intimidated by the JSP Tag Extension API, then you will probably appreciate JSTL's more convenient APIs for iteration and conditional tags.

If you're not a programmer, don't despair. You won't need to know any of the material in these chapters. However, I certainly encourage you to be ambitious: Java isn't that hard to learn, and JSTL is designed to make things easier—for programmers, too. If you don't know Java, I suggest you start with a good introductory book on Java, like Peter van der Linden's *Just Java*.[1] Then, feel free to wade into part 4's material. My hope is that you'll find it more interesting and helpful than you expected.

[1] Prentice Hall, 2001.

14

Control and performance

This chapter covers...

- Mixing Java code and JSTL
- Exposing data for JSTL tags
- Advanced features of JSTL tags
- Configuring JSTL

14.1.2 JSTL's dual libraries

To work better with scripting expressions, every JSTL tag library has a twin that uses scripting expressions (`<%= … %>`) instead of the JSTL expression language. Recall from chapter 2 that JSTL offers four tag libraries:

- The core library
- The XML-processing library
- The text-formatting and internationalization library
- The database library

Each of these libraries has a counterpart that's identical, except that every attribute that accepts a JSTL expression in the familiar library accepts, instead, a scripting expression in the twin library. Formally, when a tag accepts a scripting expression for an attribute, that tag is said to accept an *rtexprvalue*, or *request-time expression value*. For instance, the following tag uses an rtexprvalue:

```
<fmt_rt:formatNumber value="<%= netWorth %>"/>
```

Table 14.1 lists the four rtexprvalue-oriented JSTL libraries.

Table 14.1 For each JSTL tag library that we discussed earlier in this book, JSTL supports a twin rtexprvalue library that accepts Java expressions instead of JSTL expressions. This table lists the URIs and suggested prefixes of the four JSTL rtexprvalue libraries.

JSTL tag library	Suggested prefix	URI	Example tag
Core library (iteration, conditions, and so on)	`c_rt`	http://java.sun.com/jstl/core_rt	`<c_rt:forEach>`
XML processing library	`x_rt`	http://java.sun.com/jstl/xml_rt	`<x_rt:forEach>`
Internationalization (i18n) and formatting	`fmt_rt`	http://java.sun.com/jstl/fmt_rt	`<fmt_rt:formatDate>`
Database (SQL) access	`sql_rt`	http://java.sun.com/jstl/sql_rt	`<sql_rt:query>`

Importing the rtexprvalue libraries is as simple as importing the familiar libraries you've already seen. For instance, chapter 2 showed how to import the core library:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
```

To import the rtexprvalue version of the core library, you'd instead write the following:

```
<%@ taglib prefix="c_rt" uri="http://java.sun.com/jstl/core_rt" %>
```

Then, if you wanted to use the `forEach` tag in the core library, you'd write

```
<c_rt:forEach ...>
```

One final note of caution: `varReader` may not be effective when you import relative URLs. It works fine, but under most implementations of JSTL (including the reference implementation), it won't be any faster than using a simple string.

14.3.2 Character encoding

Let's look at another advanced feature: `<c:import>` gives you control over what character set to use if you import from a URL that offers binary data. If you've programmed in Java, you might be familiar with the difference between an `InputStream` and a `Reader`. Specifically, both classes let you read data, but `InputStream` returns binary data, whereas `Reader` returns text characters. If you're retrieving data from a URL and this data begins with the character "S", then a `Reader` object simply provides the "S" to you. (Think of this "S" almost like a high-level object; you can treat it as if it represents some real-world entity without worrying about how it's stored internally by the computer.) `InputStream`, however, returns a simple byte, like 01010011—or, because it's usually convenient to interpret bytes as numbers—83. But 83 isn't a character; it's still just a number. To convert it to a character, you need to use a *character encoding*, otherwise known as a *character set*. (In the character set that's most widely used, 83 represents the character "S".)

Some resources can return characters to you directly. In particular, if you import a JSP page with `<c:import>`, and the page resides in the same JSP container as the one you're writing, then the two pages communicate using characters, and no character encoding is necessary. The target page simply sends characters like "S" and "T", and you don't need to interpret them; you can immediately use them as characters.

But when you import files over the network—for example, every time you use an absolute URL—the data is transferred over a binary medium, and you must use a character encoding to figure out how to interpret the data. Picture a URL as returning a series of numbers to you: 87, 72, 89, and so on. You need a character encoding to figure out what these numbers mean.

By default, `<c:import>` usually does a pretty good job of interpreting these numbers. When you load an absolute URL from a web server, this absolute URL has a chance to declare its character encoding. Picture it responding by saying something like this: "Here are some bytes, encoded using the ISO-8859-4 character set: 87, 72, 89, …." The `<c:import>` tag receives this message and normally can decipher the bytes.

However, in some situations you want to specify a character encoding yourself. In particular, sometimes a URL doesn't declare its character encoding appropriately. In this case, `<c:import>` falls back to a decoding that works *most* of the time. This encoding is called ISO-8859-1, and it represents a character encoding used document based on a subset of XPath's syntax—corresponding roughly to the portion of XPath we discussed chapter 7. (This isn't a coincidence; it's the subset of XPath that I think is most useful in most situations.)

You can experiment with the SPath filter by copying the spath.tld file from the src/org/apache/taglibs/standard/extra/spath directory of the reference implementation's source code distribution, available from http://jakarta.apache.org/taglibs, to your web application's WEB-INF directory. Then, you can import SPath's small tag library into your page using the `spath` prefix with the following directive:

```
<%@ taglib prefix="spath" uri="/WEB-INF/spath.tld" %>
```

After this, you're free to use a tag called `<spath:filter>` in your page. The `<spath:filter>` tag takes two attributes: `select`, which lets you specify an expression in the small SPath language, and `var`, which lets you expose a filter. You can use the tag as follows:

```
<spath:filter select='//customer[@id="525"]' var="spath"/>
<x:parse xml="${bigDocument}" var="filtered" filter="${spath}"/>
```

These two lines have a very similar effect to the following:

```
<x:parse xml="${bigDocument}" var="unfiltered"/>
<x:set select='$unfiltered//customer[@id="525"]' var="unfiltered"/>
```

However, for large documents, the former example should run much faster than the latter; it applies an `XMLFilter` before the document is ever exposed, instead of simply applying an XPath expression to pare down an already large document.

NOTE I said the two examples have a "very similar effect"—not an identical effect. The reason for the difference is somewhat technical. The first example exposes an entire document (in a variable called `filtered`), whereas the second exposes the root XML element of a document (in a variable called `unfiltered`). This might not seem like a big difference, but XPath draws a distinction between the *root node* of a document and the same document's root element. If the variable `filtered` points to the root node of a document, you can expose the document's root element as a variable called `doc` by using the following tag:

```
<x:set select="$filtered/node()" var="doc" />
```

The XPath expression in this tag works because in XPath, the root element is a child of the root node. If this concept still seems inscrutable, don't worry; you can get along fine without understanding the details.

Some programmers and style guides discourage this sort of bulk import, but for core classes like those in Java's standard `java.util` package, there isn't much harm in importing everything.

Once you expose your list using the previous code, a page author could access it using expressions like `${strings[0]}`, `${strings[1]}`, and so on.

In some ways, `java.util.Map` objects are more flexible than `java.util.List` objects. They let you store pairs of items, where each item has a name and a value. Maps are therefore useful when you have a collection of objects to expose and you want to give each a name. For instance, we might use a map to associate ZIP codes with city names. We can create a map as follows:

```
Map m = new HashMap();
m.put("11791", "Syosset, NY");
m.put("06510", "New Haven, CT");
m.put("33767", "Clearwater, FL");
```

NOTE As before, you'll need to make sure that `java.util.Map` and `java.util.HashMap` are imported into your Java source.

Exposing this map in the session scope lets page authors access the pairs of ZIP codes and place names. For instance, if we store the map as a session-scoped variable named `zips`, then the following tag will print out `"New Haven, CT"`:

```
<c:out value='${sessionScope.zips["06510"]}'/>
```

14.6.3 Writing JavaBeans

Sometimes, you want to expose one of your own Java classes to a page author. Doing so can be more efficient and convenient than creating maps and lists for all of your application's data. You can expose any Java object as a scoped variable, but simply exposing an object doesn't mean that JSTL will be able to do anything useful with it. If you want your object's data to be easily readable by page authors, you'll need to follow a few conventions when designing your classes.

Fortunately, these conventions are straightforward. The rules are laid out by the JavaBeans specification (this might sound intimidating, but it's not). You don't have to jump through any complicated hoops to write a JavaBean; in fact, whether a class is a JavaBean or not is something of a blurry distinction these days. For JSTL's purposes, you'll just need to follow a few simple rules.

Think of your class as a collection of *properties* you want to expose. For instance, if you're writing a class to represent a customer, your `Customer` class might have

```
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
  <description>
    My web application.
  </description>
  <context-param>
    <param-name>my.initialization.parameter</param-name>
    <param-value>my.parameter.value</param-value>
  </context-param>
</web-app>
```

The tags that create an initialization parameter are highlighted. They appear within the `<web-app>` element. The outer tag, `<context-param>`, declares a single context-initialization parameter. This tag has two children: `<param-name>`, which specifies the name of the parameter, and `<param-value>`, which specifies the value of the parameter. Our sample web.xml file sets the parameter named `my.initialization.parameter` to a value of `my.parameter.value`.

The constants declared in the `Config` class aren't relevant when setting configuration variables using context-initialization parameters.

JSTL expressions can access context-initialization parameters directly using the `initParam` implicit object, as in `${initParam["my.initialization.parameter"]}`. However, page authors don't need to access configuration variables manually; their defaults take effect automatically.

14.7.2 Managing database access

Now that we've looked at how to set configuration variables, let's see what specific variables JSTL looks for. JSTL's database tags support the variables listed in table 14.4.

Table 14.4 JSTL's database tags support configuration variables to help you set up default databases for your pages and to help prevent against runaway queries.

Variable constant	Variable name	Purpose
`Config.SQL_DATA_SOURCE`	`javax.servlet.jsp.jstl.sql.dataSource`	Default `DataSource` object or path
`Config.SQL_MAX_ROWS`	`javax.servlet.jsp.jstl.sql.maxRows`	Default value for `<sql:query>`'s `maxRows` attribute

Default DataSource

The `javax.servlet.jsp.jstl.sql.dataSource` variable (`Config.SQL_DATA_SOURCE`) lets you install an object that represents your pages' default database. This variable

```
  </validator>
  <tag>
    <name>noop</name>
    <tag-class>javax.servlet.jsp.tagext.TagSupport</tag-class>
    <body-content>empty</body-content>
  </tag>
</taglib>
```

Most of this document is boilerplate. However, it has four interesting sections, marked by the tags `<init-param>` and `</init-param>`. These sections let you configure how strict you'd like to be in monitoring against scripting elements. You can decide to limit

1. declarations (`<%! … %>`)
2. scriptlets (`<% … %>`)
3. scripting expressions (`<%= … %>`)
4. rtexprvalues (a scripting expression within a JSP tag attribute)

In the file I've shown, all of these scripting elements are prohibited; you can selectively allow them by replacing the word `false` with `true` inside the corresponding `<param-value>` tags.

If you save this file as /WEB-INF/scriptfree.tld in your web application, then scripting expressions will be prohibited from any page that uses the following directive:

```
<%@ taglib uri="/WEB-INF/scriptfree.tld" prefix="scriptfree" %>
```

As I mentioned, this limitation requires buy-in from any JSP page author whose behavior you're trying to control; a page author can always choose not to include this directive.

If you know how to write and package tag libraries (a topic we'll introduce in chapter 15) then you can include the `<validator>` element from scriptfree.tld in your own TLD files, thus requiring that anyone who uses your taglibs also *not* use scripting elements.

14.8.2 Enumerating legal tag libraries

JSTL's second validator lets you list the tag libraries that are valid for a particular page. Listing 14.2 shows an example that ensures no tag libraries other than JSTL's non-rtepxrvalue libraries are used in a page.

As you've seen, JSTL gives page authors the tools they need to access databases, format text and XML, internationalize applications, and perform many other common tasks. In many cases, authors of JSP pages don't need to look beyond the flexible set of tags that JSTL offers.

However, JSTL's tags aren't meant to solve every potential problem a page author might run into. When page authors have a need that JSTL doesn't address, they depend on back-end Java programmers to fill in the gaps. For example, JSTL 1.0 doesn't offer a way to send email, read from online directories using the Java Naming and Directory Interface (JNDI), send messages using the Java Message Service (JMS), and so on. If page authors need to accomplish these tasks, they need to be helped along by back-end Java programmers in their organization (or third-party tag libraries they download or purchase).

In this chapter, we look at how JSTL makes it easier to develop custom tag libraries. At this point, I assume you have some knowledge of the Java programming language. As you'll see, JSTL lets you develop some kinds of tags without making you learn the details of JSP's complex tag-related APIs. However, under JSP 1.2, you still need to know Java to develop custom tags.

NOTE At the time I wrote this chapter, the JSP 1.3 expert group was considering how to provide a way for non-programmers to produce custom tags using JSP instead of Java. So, under JSP 1.3, developing tags might become even easier. For the moment, though, JSTL's support for tag developers is a useful step in the right direction.

15.1 Developing and installing tag libraries

Tag libraries are written in Java using JSP's *tag extension API*. This API lets you develop *tag handlers*, which are Java classes that implement custom JSP tags. For instance, we might write a Java class named

```
com.jstlbook.examples.MyIfTag
```

whose code runs every time the tag

```
<book:if>
```

appears in our site's JSP pages. For such a class to be a tag handler, it must implement the `javax.servlet.jsp.tagext.Tag` interface, which is defined by the JSP specification.

Although this expression language is useful in many situations, some pages require more specific, focused conditional logic. The expression language lets you compare two values, for example, but it doesn't let you ask all the conditional questions that Java lets you ask. That's what custom tag handlers are for.

15.2.1 A simple conditional tag

For our first example of custom conditional tags, suppose a page author for our application needs to display different data depending on whether it's the weekend or weekday. Imagine the following requirement: when a page is loaded any time between Monday and Friday, the page must print, "Our operators are standing by at this very moment." Otherwise, it should not print this message: no use inviting telephone calls when nobody's around to answer them.

We might be able to implement this functionality using the `<fmt:formatDate>` tag from chapter 10 and some clever applications of the `pattern` attribute. But although this strategy would probably be fun to implement, it would lead to an awkward, hard-to-maintain page. Instead, we'd like to create simple logic that differentiates weekdays from weekends and expose this logic to page authors who don't necessarily know how to program. That is, we want page authors to be able to write something like this:

```
<book:ifWeekday>
  Our operators are standing by at this very moment.
</book:ifWeekday>
```

The new tag, `<book:ifWeekday>`, should let its body be processed only if the current day is a weekday. Thus, on Monday through Friday, this tag will cause its body to be printed; on the weekends, it will prevent its body from printing. With this simple syntax, pages using the tag will be easy to maintain.

Before we create the `<book:ifWeekday>` tag, we need to figure out how to write code to differentiate weekends from weekdays. Ideally, we'd like to write a simple `isWeekday()` method that returns `true` on weekdays and `false` on weekends. Listing 15.1 shows one way to write such a method, spelled out in detail to make sure it's clear.

Listing 15.1 weekday.java: a class with a method that detects weekends

```
package com.jstlbook.examples;

import java.util.*;

public class Weekday {

    public boolean isWeekday() {       ❶ Returns a boolean
```

```
  private int day = -1;
  private int after = -1;
  private int before = -1;

  public void setDay(String s) {
    if (s.equals("sunday"))
      day = Calendar.SUNDAY;
    else if (s.equals("monday"))
      day = Calendar.MONDAY;
    else if (s.equals("tuesday"))
      day = Calendar.TUESDAY;
    else if (s.equals("wednesday"))
      day = Calendar.WEDNESDAY;
    else if (s.equals("thursday"))
      day = Calendar.THURSDAY;
    else if (s.equals("friday"))
      day = Calendar.FRIDAY;
    else if (s.equals("saturday"))
      day = Calendar.SATURDAY;
    else throw new IllegalArgumentException("bad weekday: " + s);
  }

  public void setBefore(int i) {
    if (i < 0 || i > 23)
      throw new IllegalArgumentException("bad hour: " + i);
    before = i;
  }

  public void setAfter(int i) {
    if (i < 0 || i > 23)
      throw new IllegalArgumentException("bad hour: " + i);
    after = i;
  }

  protected boolean condition() {
    Calendar now = Calendar.getInstance();
    int currentDay = now.get(Calendar.DAY_OF_WEEK);
    int currentHour = now.get(Calendar.HOUR_OF_DAY);

    if (day != -1 && currentDay != day)
      return false;

    if (before != -1 && currentHour >= before)
      return false;

    if (after != -1 && currentHour < after)
      return false;

    return true;
  }
}
```

1 Variables for attributes

2 Accepts the day attribute

3 Retrieves the current date and time

4 Ensures the date and time meet requirements

integrate your tags into `<c:when>` blocks, and it makes it easy for you to write tags that behave like JSTL's tags.

To demonstrate this functionality, let's add a few lines to the `<tag>` element for `<book:ifTime>` in our TLD. The new lines are highlighted:

```
<tag>
  <name>ifTime</name>
  <tag-class>com.jstlbook.examples.TimeTag</tag-class>
  <attribute>
   <name>day</name>
  </attribute>
  <attribute>
    <name>before</name>
  </attribute>
  <attribute>
    <name>after</name>
  </attribute>
  <attribute>
    <name>var</name>
  </attribute>
</tag>
```

This is all we need to add. The base `ConditionalTagSupport` class provides the setter method for var (`setVar()`) and exposes the variable automatically when the page author specifies a `var` attribute. `ConditionalTagSupport` similarly supports a `scope` attribute if you'd like to add it to your tag.

15.2.4 Using the expression language

JSTL 1.0 doesn't provide a standard way to use the expression language in your own tags; there is no standard JSTL API for invoking the expression language to interpret expressions within your own tags. Although JSP 1.3 wasn't yet released at the time this chapter was written, the plan is for JSP 1.3 to take care of resolving expressions for you. Thus, if JSTL 1.0 provided a standard expression API, it would be useful only under JSP 1.2, and the Java Community Process typically avoids intentionally providing legacy interfaces.

However, the lack of a standard API doesn't mean you can't accept JSTL expressions in your own tags. It just makes it hard to describe an exact procedure in this book! To use the expression language in your own tags under JSP 1.2, you'll need to use an API specific to an individual implementation of JSTL—one that decides to expose an expression-language API for you to use. The JSTL reference implementation, which is available from the Jakarta Taglibs web site at http://jakarta.apache.org/taglibs, provides such an interface.

At the time this was written, accessing the expression language using the JSTL reference implementation was simple: make sure the file standard.jar from the JSTL

```
  protected void prepare() throws JspTagException {
    try {
      if (input != null)
        input.close();
      input = new BufferedReader(new FileReader(filename));
    } catch (IOException ex) {
      throw new JspTagException(ex.toString());
    }
  }

  public void release() {
    try {
      if (input != null)
        input.close();
    } catch (IOException ex) {
      // ignore
    }
  }

  protected boolean hasNext() throws JspTagException {
    try {
      return input.ready();
    } catch (IOException ex) {
      throw new JspTagException(ex.toString());
    }
  }

  protected Object next() throws JspTagException {
    try {
      return input.readLine();
    } catch (IOException ex) {
      throw new JspTagException(ex.toString());
    }
  }

}
```

❸ Prepares for an iteration

❹ Cleans up when we're done

❺ Determines if there's more data

❻ Returns the next line from the file

Understanding the code

Listing 15.4 shows the code for a custom iteration tag that extends JSTL's `LoopTag-Support` base class. We store the value of the `filename` attribute in our own variable named `filename` (❶), which is set from the `setFilename()` accessor (❷). Before looping over the data, we prepare for the iteration by opening a file (❸).

We use the `release()` method to call the `close()` method (❹) for the stream we've opened. Using the simple interface that `LoopTagSupport` provides, we can't easily close the stream as soon as we're done with it;[3] but we want to make sure that

[3] If we only closed it immediately before returning `false` from `hasNext()`, we would leave it open in cases where it wasn't fully consumed.

```
  this.beginSpecified = true;
  validateBegin();
}
```

Our `setBegin()` method declares `JspTagException` because `validateBegin()` may throw this exception if it decides the new value for `begin` is invalid.

You could write `setEnd()` and `setStep()` methods in the same style. Then, simply add your attributes to the TLD, and you can support iteration with subsetting, just like the core JSTL tags.

15.4 Summary

In this chapter, we looked at how JSTL makes it easier to develop condition and iteration tags for JSP pages. When developing tags using JSTL, keep in mind the following points:

- JSP comes with a tag-extension API for writing custom tags.
- The tag library descriptor (TLD) document maps tag handler classes to tag names. In a JSP page, the `<%@ taglib %>` directive imports a tag library by referring to its TLD.
- JSTL simplifies the process of writing tags by providing base classes that do some of the heavy lifting for you.
- JSTL's `ConditionalTagSupport` class lets you write conditional tags by supplying a `condition()` method that causes the tag to either include or skip its body.
- JSTL's `LoopTagSupport` class lets you write iteration tags by supplying items to iterate over.
- Tag handlers that extend `ConditionalTagSupport` and `LoopTagSupport` must provide accessor (`set`*`Xxx`*`()`) methods for their own attributes, but the JSTL base classes provide `setVar()` automatically. Thus, they expose variables without your needing to do any of the work.

Boolean logic

The comparison operators produce boolean expressions, and JSTL expressions can also access boolean primitives. To combine boolean subexpressions, JSTL provides the following operators:

Operator	Description
`&&` `and`	`True` only if both sides are true
`||` `or`	`True` if either or both sides are true
`!` `not`	`True` only if the expression following it is false

JSTL supports two boolean literals: `true` and `false`.

Parentheses

JSTL expressions can use parentheses to group subexpressions. For example, `${(1 + 2) * 3}` equals `9`, but `${1 + (2 * 3)}` equals `7`.

A.2 Core tag library

JSTL's core tag library supports output, management of variables, conditional logic, loops, text imports, and URL manipulation. JSP pages can import the core tag library with the following directive:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
```

A.2.1 General-purpose tags

JSTL provides `<c:out>` for writing data, `<c:set>` for saving data to memory, `<c:remove>` for deleting data, and `<c:catch>` for handling errors.

Examples

```
Thanks for logging in, <c:out value="${name}"/>.
```

```
<c:set var="loggedIn" scope="session" value="${true}"/>
```

```
<c:remove var="loggedOut" scope="session"/>
```

Tag attributes

The `<c:catch>` tag's attribute is as follows:

Attribute	Description	Required	Default
`var`	Variable to expose information about error	No	*None*

A.3 XML tag library

JSTL's XML-processing tag library supports parsing of XML documents, selection of XML fragments, flow control based on XML, and XSLT transformations. JSP pages can import the XML-processing tag library with the following directive:

```
<%@ taglib prefix="x" uri="http://java.sun.com/jstl/xml" %>
```

A.3.1 Parsing and general manipulation

Before you work with an XML document, it must be parsed with `<x:parse>` or back-end Java code. The `<x:out>` and `<x:set>` tags can retrieve fragments of parsed documents, whether these documents are DOM objects or a JSTL implementation's own choice of data type.

Examples

```
<c:set var="textDocument">
  <orders>
    <order>
      762 cans of low-fat yogurt
    </order>
    <order>
      6 spoons
    </order>
  </orders>
</c:set>
<x:parse xml="${textDocument}" var="xml"/>
<x:out select="$xml/orders/order[1]"/>
```

```
<x:set var="fragment" select="$xml//order"/>
```

Tag attributes

The `<x:parse>` tag's attributes are as follows:

Attribute	Description	Required	Default
`xml`	Text of the document to parse (`String` or `Reader`)	No	*Body*
`systemId`	URI of the original document (for entity resolution)	No	*None*
`filter`	`XMLFilter` object to filter the document	No	*None*
`var`	Name of the variable to expose the parsed document	No	*None*
`scope`	Scoped of the variable to expose the parsed document	No	*None*
`varDom`	Name of the variable to expose the parsed DOM	No	*None*
`scopeDom`	Scoped of the variable to expose the parsed DOM	No	*None*

Attribute	Description	Required	Default
`user`	Database username	No	*None*
`password`	Database password	No	*None*
`dataSource`	Database prepared in advance (`String` or `javax.sql.DataSource`)	No	*None*
`var`	Name of the variable to represent the database	No	*Set default*
`scope`	Scope of the variable to represent the database	No	`page`

A.4.2 Queries and updates

JSTL can read from databases with `<sql:query>` and write to them with `<sql:update>`. These tags support SQL commands with `?` placeholders, which `<sql:param>` and `<sql:dateParam>` can fill in.

Examples

```
<sql:query var="result">
  SELECT ORDER
  FROM ORDERS
  WHERE CUSTOMER_ID='52'
    AND PRODUCT_NAME='Oat Bran'
</sql:query>
<c:forEach items="${result.rows}" var="row">
  <c:out value="${row.product_name}"/>
</c:forEach>
```

```
<sql:update var="count">
  UPDATE CONVICTS
    SET ARRESTS=ARRESTS+1
    WHERE CONVICT_ID=?
  <sql:param value="${currentConvict}"/>
</sql:update>
```

Tag attributes

The `<sql:query>` tag's attributes are as follows:

Attribute	Description	Required	Default
`sql`	SQL command to execute (should return a `ResultSet`)	No	*Body*
`dataSource`	Database connection to use (overrides the default)	No	*Default database*
`maxRows`	Maximum number of results to store in the variable	No	*Unlimited*
`startRow`	Number of the row in the result at which to start recording	No	`0`

Attribute	Description	Required	Default
`timeZone`	Time zone of the parsed date	No	*Default time zone*
`var`	Name of the variable to store the parsed date (as a `java.util.Date`)	No	*Print to page*
`scope`	Scope of the variable to store the parsed date	No	`page`

The `<fmt:timeZone>` tag's attribute is as follows:

Attribute	Description	Required	Default
`value`	Time zone to apply to the body (`string` or `java.util.TimeZone`)	Yes	*None*

The `<fmt:setTimeZone>` tag's attributes are as follows:

Attribute	Description	Required	Default
`value`	Time zone to expose as a scoped or configuration variable	Yes	*None*
`var`	Name of the variable to store the new time zone	No	*Replace default*
`scope`	Scope of the variable to store the new time zone	No	`page`

A.5.3 *Other internationalization*

To assist with customized internationalization of applications, JSTL offers the following tags: `<fmt:setLocale>` to specify a new default locale, `<fmt:bundle>` and `<fmt:setBundle>` to prepare resource bundles for use, and `<fmt:message>` and `<fmt:param>` to output localized messages.

Examples

```
<fmt:setLocale value="en_US"/>
<fmt:setBundle basename="vulgarInsults"/>

<fmt:bundle basename="org.apache.bookies">
  <fmt:message key="threat" >
    <fmt:param value="${address}"/>
    <fmt:param value="${numberOfChildren}"/>
    <fmt:param value="${nameOfSpouse}"/>
  </fmt:message>
</fmt:bundle>
```

`REQUEST_SCOPE`, `PageContext.SESSION_SCOPE`, or `PageContext.APPLICATION_SCOPE` from `javax.servlet.jsp.PageContext`). It returns `null` if no such configuration variable is found.

```
static Object get(javax.servlet.ServletRequest request,
                  String name)
```

This method retrieves the configuration variable `name` from the request scope represented by `request`. It returns `null` if no such configuration variable is found.

```
static Object get(javax.servlet.http.HttpSession session,
                  String name)
```

This method retrieves the configuration variable `name` from the session scope represented by `session`. It returns `null` if no such configuration variable is found.

```
static Object get(javax.servlet.ServletContext application,
                  String name)
```

This method retrieves the configuration variable *name* from the application scope represented by *application*. It returns `null` if no such configuration variable is found.

Methods for removing configuration variables

```
static void remove(javax.servlet.jsp.PageContext pageContext,
                   String name,
                   int scope)
```

This method removes the scoped variable `name` from `pageContext`—specifically, from the given `scope` (one of `PageContext.PAGE_SCOPE`, `PageContext.REQUEST_SCOPE`, `PageContext.SESSION_SCOPE`, or `PageContext.APPLICATION_SCOPE` from `javax.servlet.jsp.PageContext`).

```
static void remove(javax.servlet.ServletRequest request,
                   String name)
```

This method removes the scoped variable `name` from the request scope represented by `request`.

```
static void remove(javax.servlet.http.HttpSession session,
                   String name)
```

This method removes the scoped variable `name` from the session scope represented by `session`.

```
static void remove(javax.servlet.ServletContext application,
                   String name)
```

This method removes the scoped variable `name` from the application scope represented by `application`.

This method returns a `Result` object based on the given `ResultSet`. Note that the `ResultSet` is consumed; it must be reset before further use (and if it is a one-way `ResultSet`, it will no longer be usable).

```
static Result toResult(ResultSet rs, int maxRows)
```

This method returns a `Result` object based on the given `ResultSet`, limiting it to `maxRows` rows if necessary. Recall that the `Result` objects returned by `ResultSupport` methods cache data. The `maxRows` parameter lets you avoid consuming too much memory as the result of a runaway query (for instance, a negligence to join two tables in a query, producing an unanticipated, unfiltered cross-product of two relations).

B.3.3 The javax.servlet.jsp.jstl.sql.SQLExecutionTag interface

JSTL provides two tags for setting `PreparedStatement` parameters: `<sql:param>` and `<sql:dateParam>`. However, SQL supports many data types; applications and databases may need more support. To let you plug in your own parameter tags, JSTL provides the `SQLExecutionTag` interface.

To write a custom parameter tag designed to set a `?`-style parameter in a `PreparedStatement`, simply have your tag find its nearest `SQLExecutionTag` ancestor and call the following method for this ancestor:

```
public void addSQLParameter(Object value)
```

This method adds a `PreparedStatement` parameter to the SQL execution tag (typically `<sql:query>` or `<sql:update>`). The SQL tag will accept this parameter among those sent by other child tags, such as `<sql:param>`.

For instance, a custom child tag might contain the following code:

```
SQLExecutionTag t =
  (SQLExecutionTag)
    findAncestorWithClass(this, SQLExecutionTag.class);
t.addSQLParameter(myParameter);
```

B.4 Using JSTL's localization algorithms

To help you internationalize your applications, JSTL provides two classes related to formatting and globalization.

B.4.1 The javax.servlet.jsp.jstl.fmt.LocaleSupport class

JSTL uses a detailed algorithm to select which locale to use when internationalizing applications. This algorithm is designed to choose the best locale when the set of available locales to satisfy any requested operation, such as a keyed lookup of an internationalized message, does not contain the precise locale the user would prefer.

The database tags from chapter 9 and the examples in part 3 use the Structured Query Language (SQL) to access data. This appendix shows how you can use SQL in your `<sql:query>` and `<sql:update>` tags. For more information about `<sql:query>` and `<sql:update>`, see chapter 9. If you're not familiar with SQL, this appendix will help you follow the book's examples.

NOTE This is not meant to be a complete guide to SQL—just a crash course to help you understand this book's examples if you haven't used SQL before. For a more complete introduction to SQL, see the resources in appendix D.

C.1 SQL and <sql:update>

SQL has two categories of commands. Commands in its *Data Definition Language* (DDL) let you alter the structure of a database—for instance, add or remove a table. By contrast, commands in SQL's *Data Manipulation Language* (DML) let you work with data—add, modify, or remove rows, for example.

The `<sql:update>` tag supports both kinds of commands. You can therefore use `<sql:update>` not only to add data but also to change the structure of your database.

C.1.1 Managing tables

In a relational database, a table is a collection of data that's organized into *rows* and *columns*. Think of a row of data as a single record or entry, and a column as a field, or a placeholder filled in by each row.

For example, a table of users might have three columns: `ID`, `IQ`, and `BLOOD_TYPE`. Table C.1 shows an example, with some sample data.

Table 15.1 A simple table that lists users' ID numbers, IQs, and blood types. Relational databases store information in tables that work similarly to printed tables in books: they are divided into rows and columns.

NAME	IQ	BLOOD_TYPE
1	106	O
2	82	A-
3	164	B+
4	143	
5	128	AB+

ment benefits might apply only if the employee's age and years of service total 70 or greater. We could express this condition with the following expression:

```
AGE + SERVICE > 70
```

C.1.4 Removing data

Of course, data stored in a relational database isn't permanent. You can delete it with the `DELETE` command, whose syntax is straightforward:

```
DELETE FROM table
[ WHERE condition ]
```

If a `DELETE` command appears without a `WHERE` clause, it deletes every row from the table. Be very careful when using `DELETE`, because you can easily remove all data from a table with a command like this:

```
DELETE FROM CUSTOMERS
```

This command deletes all data from the `CUSTOMERS` table, which probably isn't a good idea unless you're going out of business.

As with `UPDATE`, you can use the keyword `WHERE`, followed by a conditional SQL expression, to narrow the set of rows to delete. For instance, if we needed to delete all users under the age of 18, we could use the command

```
DELETE FROM PEOPLE
WHERE AGE < 18
```

C.2 SQL and <sql:query>

To retrieve data from a database, use the SQL `SELECT` command. Remember that, as chapter 9 showed, `<sql:query>`'s only job is to acquire data, not to print it out. `SELECT` is the SQL statement you use to tell a database what information you want to receive.

C.2.1 Basic SELECT syntax

The simplest form of the `SELECT` statement is

```
SELECT * FROM table
```

This simple statement retrieves all data from the table named *`table`*. That is, every column of every row is retrieved. To print the entire contents of a table, you can use this `SELECT` statement with the printQuery.jsp page from listing 9.1 in chapter 9.

Suppose, however, that you're not interested in every row. For instance, we might be a publisher releasing a new philosophical tract that we think will only interest people with an IQ over 130. (As with the other examples in this chapter,

- *XSL Transformations (XSLT), Version 1.0*

  http://www.w3.org/TR/xslt. JSTL includes support for XSLT (see chapter 8), a language that's primarily useful for converting XML documents from one form to another.
- *World Wide Web Consortium (W3C)*

  http://www.w3.org/. The web site for the W3C offers technical standards for XML-related technologies. In addition to the specific URLs listed earlier, you can read more about the Document Object Model (DOM) at this site.

D.5 Miscellaneous references

- *HTML and CSS reference*

  http://www.blooberry.com. The HTML and CSS guides at Blooberry are well organized and thorough.
- *Just Java*

  By Peter van der Linden (Prentice Hall, 2001); ISBN 0130320722. This book is an excellent, readable introduction to the Java programming language.