11 Common tasks

This chapter covers...

- Reading check boxes from HTML forms
- Reading dates from HTML forms
- Handling errors
- Validating user input

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.

Figure 11.1
A view of checkboxForm.html from listing 11.1 in a web browser. Check boxes typically show up as small boxes, either empty or checked depending on whether the user has selected them. Check boxes differ from radio buttons (see chapter 3) in that a user can choose multiple check boxes with the same name. For this reason, they're a little harder to handle than text fields or radio buttons.

11.1.1 The HTML form

Let's look at a soup-to-nuts example of how to handle checkbox parameters. Begin with listing 11.1, which shows an HTML form with several check boxes. The goal of this particular form is to let the user give feedback to a web site's customer-service department. Figure 11.1 shows what this form might look like in a browser.

Listing 11.1 checkboxForm.html: a form with checkbox parameters

```
<form method="post" action="checkbox.jsp">
  <p>Please check adjectives you would
  use to describe this web site's
  customer service:</p>

  <p>Atrocious
  <input type="checkbox" name="feedback" value="atrocious"/></p>

  <p>Loathsome
  <input type="checkbox" name="feedback" value="loathsome"/></p>

  <p>Flagitious
  <input type="checkbox" name="feedback" value="flagitious"/></p>

  <p>Satisfactory
  <input type="checkbox" name="feedback" value="satisfactory"/></p>

  <p><input type="submit" value="Submit" /></p>
</form>
```

Note how all the checkbox parameters have the same value for the `name` attribute: `feedback`. Users can choose as many check boxes as they feel are appropriate, and the expression `${paramValues.feedback}` will contain a collection of all the parameters.

11.1.2 A simple checkbox handler

Listing 11.2 shows how we can use this expression to loop over all the `feedback` parameters, one at a time.

Listing 11.2 checkbox.jsp: a page to handle checkbox parameters

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

<c:choose>
 <c:when test="${not empty paramValues.feedback}">
  You described our customer service as
  <ul>
  <c:forEach items="${paramValues.feedback}" var="adj">
    <li><c:out value="${adj}"/></li>
  </c:forEach>
  </ul>
 </c:when>
 <c:otherwise>
  You didn't choose any feedback checkboxes.
 </c:otherwise>
</c:choose>
```

Decides if there are any feedback params

As figure 11.2 shows, the checkbox.jsp page from listing 11.2 lists each adjective we've chosen in a bulleted list (`<ul>`). It does so by looping over `${paramValues.feedback}` with the `<c:forEach>` tag. The `<c:forEach>` tag exposes each element as a scoped variable named `adj`, which is printed out by a `<c:out>` tag.

Figure 11.2
The checkbox.jsp page figures out which check boxes the user has chosen and prints out the text corresponding to each. It does so with a `<c:forEach>` loop and a `<c:out>` tag.

Note how listing 11.2 prints a special message if the user didn't choose any of the check boxes. We do this by using a `<c:choose>` tag. The first condition—`<c:when>`—makes sure the user has chosen at least one checkbox. It does so using the expression

```
${not empty paramValues.feedback}
```

which is `true` if the user chose at least one `feedback` checkbox, and `false` otherwise.

I included this check to demonstrate a few things. Checks against nonexistent parameters will be useful to you later for form validation, so I wanted to show you a straightforward example of this technique. But in addition, I wanted to avoid looping over nonexistent values.

It's actually not a problem to loop over a collection that doesn't exist. If you say

```
<c:forEach items="${paramValues.nope}" var="item">
  Item!
</c:forEach>
```

and the expression `${paramValues.nope}` refers to a nonexistent parameter, then the `<c:forEach>` tag will simply do nothing. The rationale is that a missing collection is much like a collection with zero elements, so iterating zero times makes sense.

However, we don't rely on this behavior of `<c:forEach>` here, because we don't want to print the trappings of a list—the initial `<ul>` tag and the closing `</ul>` tag—when the list won't have any items. (HTML lists, like `<ul>` and `<ol>`, should all ideally have at least one `<li>` item.)

TIP There are other ways to ensure that `<ul>` and `</ul>` print only when the list has at least one item, other than wrapping the whole `<c:forEach>` loop with a `<c:if>` or `<c:when>` tag. You can use `<c:forEach>`'s `varStatus` attribute to determine whether to print `<ul>` before the list's first item and `</ul>` after the list's last item, from inside `<c:forEach>`'s body. For example, this set of tags will print out `<ul>` and `</ul>` only if `<c:forEach>` iterates in the first place (if `${paramValues.feedback}` contains at least one parameter):

```
<c:forEach var="adj" items=${paramValues.feedback}" varStatus="s">
  <c:if test="${s.first}">
    <ul>
  </c:if>
  <li><c:out value="${adj}"/></li>
  <c:if test="${s.last}">
    </ul>
  </c:if>
</c:forEach>
```

If you've used languages like XSLT, you might be surprised at this block of JSP code. Your first thought might be, "Wait! The `<ul>` tag doesn't line up with the `</ul>` tag. The tags are crossed: `<ul>` starts, and then another tag—`<c:if>`—is closed." These crossed tags aren't a problem for JSP pages.

11.1.3 Handling some check boxes specially

The page from listing 11.2 treats all `feedback` parameters the same; it prints them all out in an undifferentiated list. But within the `<c:forEach>` tag, it's easy to use expressions that refer to the current item—`adj`, in this case—to make decisions about how to handle each item individually. For instance, you might decide to save some data in a database, or to display an otherwise hidden part of your page—but only if a user has chosen a particular checkbox.

Typically, you handle individual checkbox parameters by including a `<c:choose>` tag directly within your `<c:forEach>` tag. Doing so lets you match, and take particular actions for, an individual parameter.

Listing 11.3 shows a somewhat frivolous example, but it should drive home the point. Instead of printing out the customer feedback that has been received, the checkbox2.jsp page from listing 11.3 glorifies more positive remarks by printing them in big letters and diminishes negative comments by making them small. You can see the result in figure 11.3.

Figure 11.3
The checkbox2.jsp page from listing 11.3 looks at individual parameters' values before printing them out. In this case, the page glorifies positive remarks by printing them in large letters; it similarly diminishes negative feedback by making it smaller.

Listing 11.3 checkbox2.jsp: a more interesting checkbox handler

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

<c:choose>
 <c:when test="${not empty paramValues.feedback}">
```

```
  You described our customer service as
  <ul>
  <c:forEach items="${paramValues.feedback}" var="adj">
    <c:choose>
      <c:when test="${adj == 'satisfactory'}">
         <font size="+2">
      </c:when>
      <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>
```

Checks to see if the current parameter equals a specific word

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:`

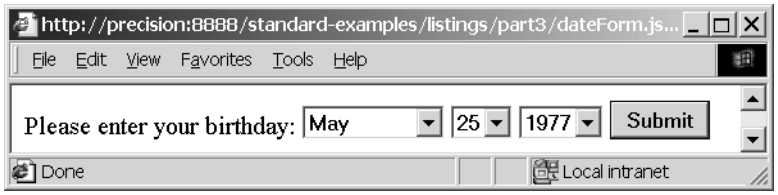

Figure 11.4 A convenient interface for entering a date. Users might appreciate this structured interface more than a simple text box that lets them type in a date—or lets them enter arbitrary text that might not be a date, which would require an irritating extra step for validating the input.

parseDate> tag discussed in chapter 10. But most users would think that such a generic interface is unfriendly. The whole point of <select> boxes and radio buttons is to guide the user toward sensible input, and this guidance is particularly useful when users have to enter structured information like dates. For instance, most users would find the interface shown in figure 11.4 convenient for entering a date. It uses three <select> boxes for the month, day, and year. In contrast with a free-text entry box, any user who uses this form is guaranteed to enter a structurally valid date. (Whether it's the right date is still, of course, up to the user.)

11.2.1 The HTML form

Listing 11.4 shows a JSP page that we could use to generate the form from figure 11.4.

Listing 11.4 dateForm.jsp: a JSP page that lets the user enter a date

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

<form method="post" action="dateHandler.jsp">

  Please enter your birthday:

  <select name="month">
    <option value="Jan">January</option>
    <option value="Feb">February</option>
    <option value="Mar">March</option>
    <option value="Apr">April</option>
    <option value="May">May</option>
    <option value="Jun">June</option>
    <option value="Jul">July</option>
    <option value="Aug">August</option>
    <option value="Sep">September</option>
    <option value="Oct">October</option>
    <option value="Nov">November</option>
    <option value="Dec">December</option>
  </select>
```

❶ **Simple, static form field**

```
  <select name="day">
    <c:forEach begin="1" end="31" var="day">
      <option><c:out value="${day}"/></option>
    </c:forEach>
  </select>

  <select name="year">
    <c:forEach begin="1930" end="2003" var="year">
      <option><c:out value="${year}"/></option>
    </c:forEach>
  </select>

  <input type="submit" value="Submit" />

</form>
```

❷ Dynamically generated fields

Note that although listing 11.4 generates a simple HTML input form, it's not a static HTML page. JSTL tags, of course, aren't limited to pages that handle forms; as you'll see in a moment, they can be useful to produce forms, too.

❶ The page starts with a simple static form field. This `month` field lets the user choose a month. Note how we map each month's full name (which we want the user to see) into an abbreviation (which is more convenient for us later).

❷ After this simple `month` field are two fields that we produce dynamically: `day` and `year`. These fields are unchanging lists of numbers, so we could write a static HTML page to produce them. However, when we're listing numbers like 1 to 31—and 1930 to 2003—it's substantially more convenient to produce these lists dynamically. The end result is the same as if we manually wrote out each `<option>` field, but we've avoided some busy-work.

Of course, you can include fields like these three—month, date, and year—in a form that lets the user enter other information. For now, we're focusing just on a single field because it demonstrates a worthwhile technique.

11.2.2 Handling the form and reading the date

Now that we've produced a form, let's look at how to retrieve the information users enter into the form. Note that in the dateForm.jsp page from listing 11.4, the form posts to a page called dateHandler.jsp. Listing 11.5 shows this page.

Listing 11.5 dateHandler.jsp: a page that reads a date from dateForm.jsp

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

<fmt:parseDate
  var="date"
  parseLocale="en_US"
  value="${param.month} ${param.day}, ${param.year}"/>
```

❶ Parses the date from the form

```
You were born
<fmt:formatDate
  value="${date}"
  dateStyle="full"/>.
```

❷ **Prints the date back out**

This page is deceptively simple. Let's look more closely at what's going on.

❶ The core of the page's function lies in the `<fmt:parseDate>` tag. Our strategy for handling the date the user entered is simple: because the date comes in three separate pieces—month, day, and year—we must combine them into a single string before parsing them. We perform this aggregation inside the `<fmt:parseDate>`'s value attribute, using three different expressions:

```
value="${param.month} ${param.day}, ${param.year}"
```

This expression causes the `month` parameter to be printed first, followed by a space, then the `day` parameter, then a comma, a space, and, finally, the `year` parameter. The result is a formatted string like `"May 25, 1997"` or `"Aug 24, 1981"`. As you might remember from chapter 10, such strings match the familiar U.S. English locale exactly. (Remember how, in dateForm.jsp, we were careful to map each month's name to an abbreviated value in English? This is why.) To force JSTL to parse this date using the U.S. English locale's rules, rather than the browser's current locale or one that the application was configured to use, we set the `<fmt:parseDate>` tag's `parseLocale` attribute equal to `"en_US"`. This way, the tag will parse the date correctly and save it in the scoped variable `date`, which we set using the `var` attribute.

❷ Now, with the user's chosen date saved in the `date` variable, we print out a full representation of it using the `<fmt:formatDate>` tag. (See figure 11.5.) Just to prove that we really parsed the date—and we aren't just spitting back some text the user sent us—we use the `dateStyle="full"` attribute, which causes the tag to print the date in its entirety, including weekday. Note that the system figured out the weekday

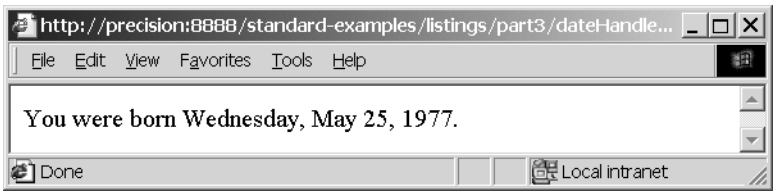

Figure 11.5 We parsed the user's date by printing the different form fields in a single, easy-to-parse format. Once we've done that, it's easy to treat the date as a date—which lets us compute information like the date's weekday, which the user never entered manually.

automatically, based on the date; the user never entered it. This is something we get for free when handling dates in JSTL.

Thus, our parsing strategy was to take all of the users' input and assemble it into an easy-to-parse string. From there, parsing with `<fmt:parseDate>` was straightforward, and printing the date with `<fmt:formatDate>` was simple.

Of course, we could do things with this date other than just printing it. We could save it to a database with tags like this:

```
<sql:update>
  UPDATE user SET birthdate=?
  <sql:param value="${date}"/>
</sql:update>
```

Or we could use the date in a custom tag. Because we parsed it, it's now a full-fledged date variable, not simply a collection of text.

11.3 Handling errors

Just as good automobile drivers can still have accidents, well-designed JSP pages can still run into errors or unexpected situations. Users might enter bizarre input, or a URL from which your page imports information might be unavailable. Any public, important page needs to take into account the possibility of errors. In some cases, it's enough to print a kind message to the user: "Something went wrong. Please try again, or call 203-432-6687 for assistance." In other cases, you might be able to recover automatically, without bothering the user. As an example, imagine that your page uses the `<c:import>` tag to fetch and display the weather in the upper-right corner of the browser's window. If the weather's URL was unavailable (perhaps as a result of a thunderstorm that brought down some power lines), you could simply print "Online weather unavailable; go look outside", or you could automatically switch to a different URL. The point is, you don't have to interrupt the entire page for an error that affects only a small piece of it.

When you use JSTL tags, there are basically three things you can do with errors:

- Avoid thinking about them. When you do this, any error that your page encounters causes the page to fail to load, usually with the JSP container providing information about the error in the page's place. This approach is great while you're debugging your pages, but it usually looks unprofessional to users.
- Use the `<c:catch>` tag to handle and even recover from errors within the same page. Doing so gives you flexibility, but it can clutter your pages, and it might not be appropriate if an error is meaningful and shouldn't be discarded.
- Use a facility that JSP gives you known as an *error page* (often written `errorPage`). A JSP `errorPage` is a page you can design and set up to handle

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

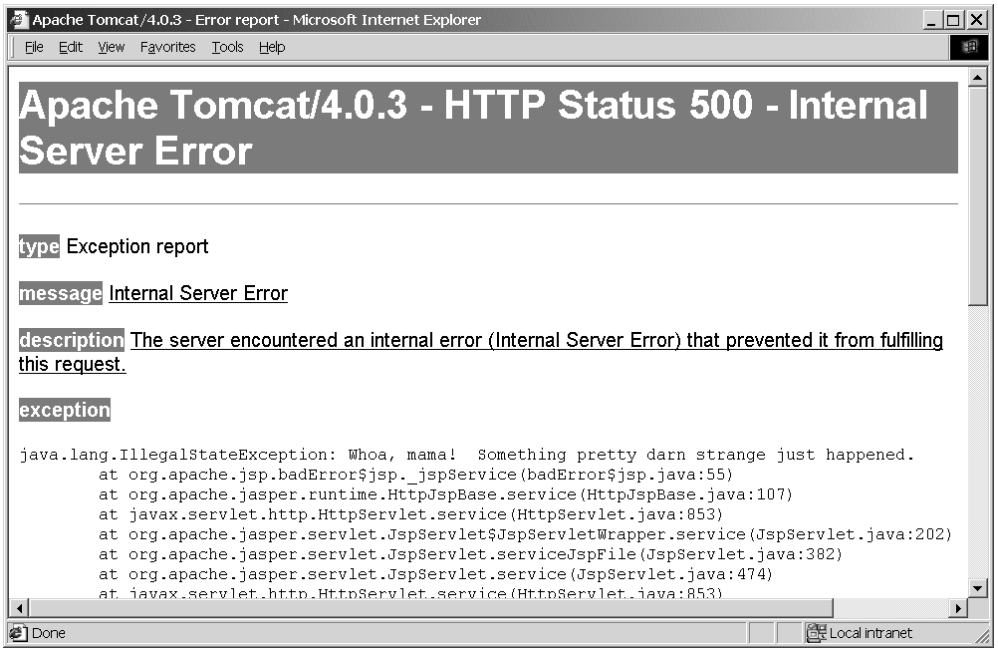

Figure 11.6 By default, errors in your JSP pages result in behavior that might look like this—or different, depending on what your JSP container decides to show. The point is, you probably don't want your important pages to produce such errors. Unless back-end Java programmers supporting your application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

figure 11.6 for an example of what a container's default error page might look like—in this case, Jakarta Tomcat's.)

Note that your page immediately aborts when it encounters its first unhandled error. That is, if line 2 of your page produces an error that you ignore, line 3 will never be executed. Instead, the user will see your application's default error behavior immediately.

11.3.2 Catching errors with <c:catch>

On the opposite side of the spectrum from simply forgetting about all errors is another alternative: you can sweep them all under the rug. The `<c:catch>` tag lets you capture errors and either discard them entirely or record information about them for later study. Table 11.1 shows its single attribute.

Table 11.1 `<c:catch>` tag attribute

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

Errors that occur inside the body of a `<c:catch>` tag do not cause your whole page to abort. Instead, they abort only the rest of the `<c:catch>` tag's body.

If you use `<c:catch>` without a `var` attribute, it ignores all errors that occur in its body and lets your page continue. This approach is useful if you want to try something speculatively but don't really care if it succeeds. For instance, remember the database-driven hit counter from chapter 9? If the database for the counter is down, that's probably not an error serious enough to warrant the failure of your entire page. And there's no need to tell the user about the error, because they couldn't do much. Instead, you could wrap all of the counter's logic in a `<c:catch>` … `</c:catch>` tag.

When you specify a `var` attribute, `<c:catch>` saves information about the error in the indicated scoped variable.

Let's look at an example of `<c:catch>` in action. Recall from chapter 10 that the `<fmt:parseNumber>` tag helps you read numbers that users enter into a form. In listing 10.1, you saw an example of a page that parses the number a user entered; the page reads the number, and then performs some simple arithmetic on it before printing it out.

Now, let's add some error handling. Listing 11.6 shows how to make the page from listing 10.1 more robust by allowing it to recover gracefully from bad input.

Listing 11.6 parseNumberCarefully.jsp: parsing and error recovery

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

<p>You entered "<c:out value="${param.favorite}"/>". </p>

<c:catch var="parsingError">                          <-- 1 Creates a parsingError variable for bad input
  <fmt:parseNumber var="fav" value="${param.favorite}"/>

  <p>As far as I can tell, this corresponds to the
  number <c:out value="${fav}"/>.</p>

  <p>If you multiply this number by 2 and add 1, you get
  <c:out value="${fav * 2 + 1}"/>.  I like that number
  better.</p>
</c:catch>

<c:if test="${not empty parsingError}">               <-- 2 Checks whether an error occurred
  Sorry, this doesn't look like a number to me.
  Perhaps you're in the wrong country?
</c:if>
```

❶ To add error handling to this page, we start by inserting a `<c:catch>` tag around it. This tag ensures that any error that occurs won't rise high enough to be noticed by the JSP container. Instead, it will be captured within the page, as suggested by figure 11.7.

Note that if an error occurs during the `<fmt:parseNumber>` tag, the rest of the `<c:catch>` tag's body never executes. This behavior is appropriate: if the number fails to parse, we don't want to start performing arithmetic on it or printing it out.

❷ The most useful thing you can usually do with a new scoped variable is to check if it's `empty`. In this case, because we used the attribute `var="parsingError"` in the earlier `<c:catch>` tag, our scoped variable will be named `parsingError`. If `parsingError` is `empty`, then no parsing error has occurred. Otherwise, we can be sure that some error occurred within the `<c:catch>` block.

We use this fact to make a decision in a `<c:if>` tag. If the `parsingError` variable is `not empty`, then we know an error has occurred, and we print out a custom error message. Thus, when an error occurs during the `<fmt:parseNumber>` tag, we display an error message and nothing else. (Remember, when `<fmt:parseNumber>` runs into an error, the rest of `<c:catch>`'s body—including all the template text beneath `<fmt:parseNumber>`—won't execute. The page picks up right after the closing `</c:catch>` tag.)

So, if we send this page a legitimate value like `500,000` for the parameter `favorite`, this page behaves exactly like listing 10.1; we get a response similar to the one

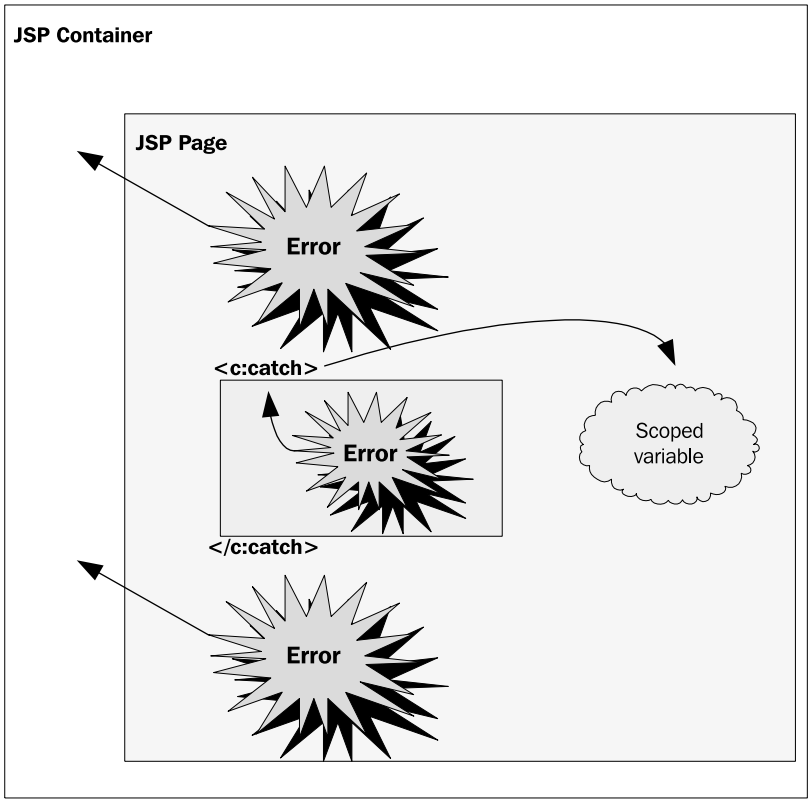

Figure 11.7 Normally, errors that occur in your JSP page are sent immediately to the JSP container. By default, they abort your page and cause the container to print out an error message. But the `<c:catch>` tag lets you capture and handle errors. When you catch an error with `<c:catch>`, you can save information about it in a scoped variable.

in figure 10.2. However, if we enter garbage like `!@#$%^`, we get the graceful error message shown in figure 11.8. Without the `<c:catch>` tag, we'd get a less friendly error message from the JSP container itself (like the one in figure 11.6).

Getting more information about errors

Variables created by `<c:catch>` have at least one useful property: `message`, which contains some information that describes the error that occurred. This property is useful if you want the user to have some sense of what went wrong. For instance, having this information might help the user describe the error to your organization's help desk.

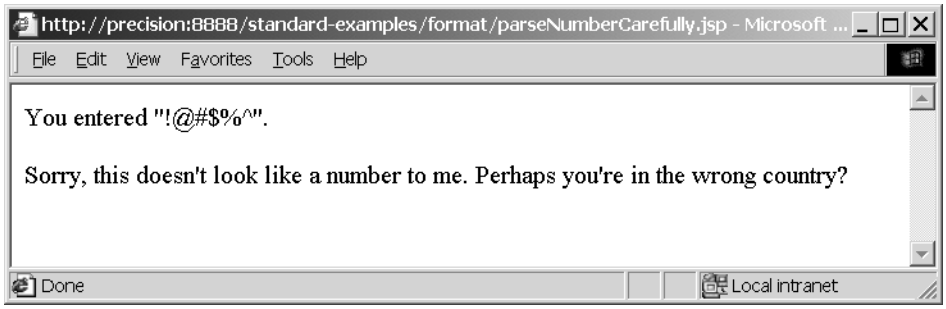

Figure 11.8 The `<c:catch>` tag lets us easily catch errors and print custom error messages like the one shown here.

To include information about the parsing error that might have occurred in listing 11.6, for example, we could use the following tag:

```
<c:out value="${parseError.message}"/>
```

We could also just write

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

Printing the scoped variable that stores the error itself—technically, it's a Java `Throwable` object—will include some more technical information about the error. (By default, it will include the name of the `Throwable`'s Java class.)

11.3.3 Passing errors to an error page

When an error reaches the JSP container (see figure 11.7), the container will, by default, display its own error message. However, you can change this behavior by using a JSP error page. When your page has an error page, the user is forwarded to this page whenever an error occurs (almost as if you had included a manual `<jsp:forward>` tag in your page).

To declare an error page, you use the `<%@ page %>` directive that JSP provides. Include the following, typically at the top of your page:

```
<%@ page errorPage="target" %>
```

In this directive, *`target`* is the name of your error page—for instance, myErrorPage.jsp.

This error page is just like any other JSP page; for instance, you can use JSTL tags within it as long as you use the correct `<%@ taglib %>` directives first. The only difference is that you should begin this page with the following line:

```
<%@ page isErrorPage="true" %>
```

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.

Figure 11.9
In contrast with figure 11.6, error pages let us control what is displayed when an error that's not handled by `<c:catch>` occurs in a page.

error page, we get an error like that shown in figure 11.9. Unlike the default error page, this error page is under our control; in this case, it displays the contents produced by listing 11.7.

11.4 Validating input

Whenever you accept information from the user, you should think about whether you need to validate it—or check to make sure it's sensible and correct—before processing it.

If you use JavaScript, you may already send pages to the browser with embedded JavaScript to validate web forms. For instance, your scripting code might pop open a window saying, "You must enter a username," if the user clicks on your Submit button without entering a username.

Although this *client-side validation* is convenient for the user, it's often not enough if you want to make sure your web application is secure and robust. The user's browser might not support JavaScript—or the user might even purposely be trying to wreak havoc with your web application.

Therefore, for important applications, it's best to think of client-side validation merely as a feature of the user interface. It's a convenience and a good first line of defense, but it's nothing more than that. You should always think about validating information on the server.

11.4.1 Different kinds of form validation

The first thing to realize about input validation is that, in a large web application, it might not be your job. JSP pages—with or without JSTL—might not be the best place to validate user input, especially in applications that use Struts or other frameworks that promote division of labor and implementation. In such applications, the job of your JSP pages might merely be to present some information and forms for the user to fill out; the back-end plumbing takes care of the rest.

JSTL 1.0 isn't intended for complex data validation. For instance, if you want to determine whether the user entered a reasonable phone number (based on the syntax of phone numbers, rules about which area codes your application accepts, and so on), you probably shouldn't use JSTL to do it. Instead, you should use a JavaBean or other object that's accessible as a scoped variable in your page. (If you're not a Java programmer, then tasks like this currently lie beyond your scope; they'll need to be handled by someone who writes Java.)

However, JSTL is useful for simple kinds of validation. This includes things like:

- Making sure the user entered something in a form field (versus leaving it blank)
- Checking to see that if one form field is filled in, another is too (or isn't)
- Comparing a number that a user entered to a limit or range (such as "is the user's age above 13?")

Not coincidentally, this is often the sort of validation that an application's *presentation tier* (its front-end JSP pages) conducts.

Handling these simple validations is straightforward; as you might expect, you can use the expression language, `<c:if>`, and `<c:choose>`, as appropriate. For example, consider the following tag:

```
<c:if test="${empty param.username}">
  <font color="red">You must enter a username!</font>
</c:if>
```

There's nothing complicated about this validation; we're simply checking to see if a form parameter is empty in order to determine whether the user entered something for it.

There are only two tricky aspects of input validation: precisely where in your pages to check the user's input, and what to do if it isn't valid. The example in this section shows one convenient approach for addressing these issues.

11.4.2 Tasks involved when validating a form

Because a good web application needs to make things easy for users, validating input isn't enough. When a user provides bad input, you need to make the user aware of the error so that he or she can correct it. Typically, web applications inform users of errors by printing the original form again, with some additional error messages included. For instance, near a text box where the user was supposed to enter a username, you might print, in bright red letters, "You must enter a username!" Feedback like this is convenient for users; it tells them exactly what they did wrong.

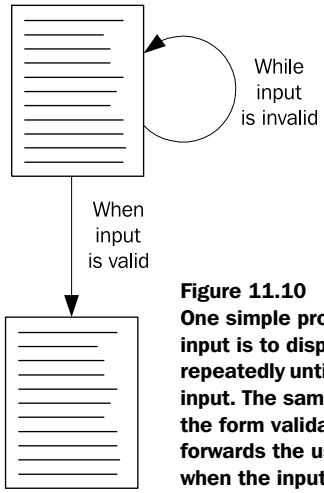

Figure 11.10
One simple process for validating input is to display the same page repeatedly until the user enters valid input. The same page that produces the form validates the input, and it forwards the user to the next page when the input is valid.

Fortunately, it's also easy to write pages that follow this pattern. Start by looking at figure 11.10. Our goal is to present a page, and then to keep presenting that page—over and over, if necessary—until the user gets the information right. Only then do we let the user proceed to the next page.

Under a model like this, the same page that produces the form also validates the input. That is, a single JSP page contains tags to display the form, to check the input, and to print error messages. The `<form>` tag in such a page directs the browser to send the form's data back to the same page. (You can do this by specifying an `action` attribute that points to the URL for the current page, although it's easier to simply leave out the `action` attribute to the HTML `<form>` tag. When no `action` is specified, the browser defaults to sending information to the current page.)

TIP Because a page that both displays and validates a form is complex, you can break such pages into smaller sections and include the different pieces with `<jsp:include>`. However, you might find it easier to keep everything in a single file.

Before we look at a page that both prints and validates, I should point out one subtlety. When you redisplay a form that has multiple fields, it's irritating to users if the form loses all the information they submitted. Unfortunately, this happens by default; when a browser loads the page a second time, the HTML forms will be empty. You'll see how to solve this problem when we examine our sample page.

Note that there are other models for validating input. Instead of cycling the same page repeatedly, you can cycle between two pages: one that displays the form, and another that validates input. I present a single-page cycle here because it demonstrates some JSTL features better than the alternatives.

11.4.3 A sample form validation

Listing 11.9 shows a page that displays our sample form, validation logic, and error messages, as appropriate.

Listing 11.9 formCycle.jsp: a page that prints and reprints a form until it's satisfied

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

<h1>Peter's Junk-Mail Service</h1>

<c:if test="${param.submitted}">
  <c:if test="${empty param.name}" var="noName" />
  <c:if test="${empty param.email}" var="noEmail" />
  <c:if test="${empty param.age}" var="noAge" />

  <c:catch var="ageError">
    <fmt:parseNumber var="parsedAge" value="${param.age}" />
    <c:if test="${parsedAge < 13}" var="youngAge" />
  </c:catch>
  <c:if test="${not empty ageError}" var="badAge" />

  <c:if
   test="${not (noName or noEmail or noAge or badAge or youngAge)}">
    <c:set value="${param.name}" var="name" scope="request"/>
    <c:set value="${param.email}" var="email" scope="request"/>
    <c:set value="${param.age}" var="age" scope="request"/>
   <jsp:forward page="spamFormHandler.jsp" />
  </c:if>
</c:if>

<form method="post">
  <p>
  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.
  </p>

  <input type="hidden" name="submitted" value="true" />

  <p>
  Enter your name:
  <input type="text" name="name"
    value="<c:out value="${param.name}"/>" />
  <br />
  <c:if test="${noName}">
```

❶ Validation logic

❷ Forwards to the next page on success

❸ If we didn't forward, then we display the form

❹ "Hidden" field sets param automatically

❺ Sets the default field value dynamically

```
 <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 pornographic
     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

form or (b) displaying the form for the first time. We'll explain later how the parameter named `submitted` gets set; for now, it's just important to realize that it will be set (that is, not `empty`) only when the page is responding to a form submission—not when it's being loaded for the first time.

❶ Inside the `<c:if>` tag that causes the code to run only if it's responding to a submitted form, we validate the form's parameters. Our overall goal is to make sure a few things happened:

- The user entered a name.
- The user entered an email address.
- The user entered an age greater than or equal to 13.

We begin this validation by using a series of `<c:if>` tags that don't have bodies; instead, they're designed only to set a scoped variable with the result of the check. For instance, the tag

```
<c:if test="${empty param.name}" var="noName" />
```

sets a scoped variable named `noName` to `true` if `param.name` doesn't have a meaningful value; if the value is not `empty`, then `noName` will be `false`. Thus, `noName` functions as an error flag: if it's `true`, we've got a problem.

When we check the user's age, we do something a little trickier. The `age` parameter can have a few different problems:

- The user could have left it blank.
- The user could have entered garbage (an unparseable number).
- The age might be too low for our purposes (less than 13).

The first case—a blank age—is easy to address; we treat it like a blank name or email address. But if the field isn't blank, we want to parse it in order to determine what numeric value the user entered. If the user didn't enter a number, this parse will fail. We use a `<c:catch>` tag to account for this possibility. The `<c:catch var="ageError">` tag sets the scoped variable named `ageError` if there was a parsing error while checking the age. Right after the `<c:catch>` tag, we determine (with another `<c:if>` tag) whether `ageError` has a value; if it does, we know that the user entered a bad value, and we set the scoped variable `badAge`. If, instead, the user entered a parseable date, then we compare it with 13 and set the `youngAge` flag to `true` if this check fails.

❷ After the validations are complete, we have an immediate use for their result. While still inside the `<c:if>` tag that makes sure the code runs only if it's responding to a form (and not printing it for the first time), we use a big `<c:if>` check to determine whether any error flags are set. This tag uses the following expression:

```
${not (noName or noEmail or noAge or badAge or youngAge)}
```

If this expression succeeds, we know the input is valid, so we `<jsp:forward>` to a new page. Before we forward, however, we do something interesting: we set a few request-scoped attributes. When we use `<jsp:forward>` to forward to a new page, that page will have access to our request parameters; it could say, for instance, `${param.name}`, and retrieve the name parameter the user entered. The problem with this approach, however, is that it gives the user a way to avoid validation. If we let the next page rely on request parameters, it would have to redo the validations to ensure its input makes sense. If the new page didn't check its input, then our checks would become meaningless; the user could contact the next page directly and send bad data. To get around this problem, we manually set the request-scoped variables that we know the next page will need. Request-scoped variables are never directly under the user's control, which makes this mechanism more secure and robust than simple request parameters.

This target page can do whatever it wants with the values. We don't show a sample target page here, although chapter 12 discusses typical things such a target page might do. For instance, if this were really a user-registration application, the page spamFormHandler.jsp would probably store the user's name and email address in a database.

❸ If we've come as far as the `<form>` tag without forwarding to a new page, then we know that one of two things is true: either this is the first time we're printing the form, or the form has errors. Either way, we must print the `<form>` and give the user a chance (perhaps *another* chance) to enter correct information. We can use the error flags to determine whether to display error messages as appropriate. We can also use `param.submitted` to differentiate between the first and subsequent requests to this page, just as we did earlier in the page. Doing so might be useful if we wanted to print a special message the first time the form loads, but not subsequent times. However, we don't bother with such details in this example.

❹ Earlier, I promised I'd explain how the parameter `submitted` is set. Recall that this parameter will be equal to `true` if we're responding to a form (instead of printing it the first time); otherwise, it won't be set. We create this distinction by making sure that every time the form is submitted, it sets a parameter named `submitted`. Of course, we can't rely on users to do this themselves, so we use a special type of HTML form field: `<input type="hidden">`. Hidden fields, true to their name, don't show up graphically in the form; the user never sees them. But behind the scenes, they force a parameter to be set with a particular value. We could achieve something similar by adding a `name` attribute to our `<input type="submit">` button; but hidden fields are more flexible, and I wanted to demonstrate one here.

❺ In section 11.4.2, I mentioned that we'd need to make sure we supplied the user's old values to a reprinted form. If we don't do so, then the form will be cleared each time

we represent it, and the user will need to reenter information. Users hate doing this, so we don't want to make them. Instead, we can force each form field to take a default value. The nature of the default value is simple: it's the parameter the user just entered.

For `<input type="text">` fields, as well as `<input type="password">`, we can use the `value` attribute to seed a value into the form. That's what we do in this listing:

```
<input type="text" name="name"
value="<c:out value="${param.name}"/>" />
```

This code sets the `value` field in the new form to the value of the `name` parameter in the old form.

For other types of input fields—selection boxes, radio buttons, and check boxes—it's trickier to add default values, but doing so is still more-or-less straightforward. For a `<select>` box, you can't simply specify a `value` attribute for the default value. Instead, you must add the attribute `selected="selected"` to the correct `<option>` tag. You can do so by comparing a parameter value with the value of the <option> tag you're about to print. For example:

```
<select name="milk">
  <option value="lowfat"
    <c:if test="${param.milk == 'lowfat'}">
      selected="selected"
    </c:if>
  >Low fat</option>
  <option value="skim"
    <c:if test="${param.milk == 'skim'}">
      selected="selected"
    </c:if>
  >Skim milk</option>
</select>
```

The `selected="selected"` attribute will print only for the correct value.

Radio buttons and check boxes work the same way, but instead of adding the attribute `selected="selected"`, you add the attribute `checked="checked"`. For `<textarea>` fields, simply insert your desired default value into the body of the `<textarea>`:

```
<textarea name="prose"><c:out
  value="${param.prose}"/></textarea>
```

TIP Note how I've avoided putting `<c:out>` on a new line within `<textarea>`, or otherwise using extra white space. It's good to be careful about doing this, because even the white space inside `<textarea>` is included in the text area's default value. The white space (and line break) *within* the `<c:out>` tag—before the `value` attribute—doesn't matter.

Figure 11.11
The first time our sample page from listing 11.9 loads, it displays a regular HTML form.

❻ At appropriate places in the form, we can use the error flags we created earlier to print out error messages. Remember, these error flags can't be set unless we're re-printing a form; this behavior is appropriate, because we wouldn't want to accuse users of crimes they didn't commit.

❼ We can also use `<c:choose>`, just as easily as `<c:if>`, as a way to make decisions about what error message to print. For instance, in the case of bad ages, we want to choose the most appropriate message to print. Our logic earlier in the page doesn't ensure that only one error flag related to age will be set; instead, we use `<c:choose>` to make sure that only one error based on these flags will print.

That's it. It's a big page, but overall, it works cleanly. The first time it's loaded, it displays the form shown in figure 11.11.

Now, suppose I enter bad input to this page. Let's say I enter a valid name, but I leave out the email address entirely, and I enter the age `6`. I'll end up with the form shown in figure 11.12. It's the same form, but it includes error messages. Note also that it includes all the information I entered earlier, saving me the trouble of having to type it again.

Finally, and only when the information is correct, we reach the target page—which, as I mentioned before, could save our information in a database. We don't show such a page; we'll go over thorough examples of pages that use databases in chapters 12 and 13.

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.

- If a user filled out a form incorrectly and you decide to redisplay it, it's important to seed that form with the values the user previously entered; otherwise, the user will have to retype the entire form. The process for setting default values varies for each HTML form element, but it's generally easy to do with either a `<c:out>` or `<c:if>` tag. See ❺ in listing 11.9.