

5

Controlling flow with loops

This chapter covers...

- Looping over collections with `<c:forEach>`
- Parsing strings with `<c:forTokens>`
- Iterating over subsets
- Determining current loop status

Looping involves repeatedly executing the same block of your JSTL page, over and over. It sounds mind-numbingly boring—and if you were a web server, you’d probably agree. But for us, loops are anything but boring. They’re a powerful feature that works as the cornerstone for many, if not most, dynamic web pages.

Looping is often called *iteration*. As the word suggests, iteration involves repetition. Figure 5.1 illustrates the way JSTL lets you take any valid JSP fragment—including tags, template text, or both—and cause it to be processed repeatedly. If your repeated JSP fragment depends on scoped variables that change during the looping process, then each repetition can produce slightly different text or HTML. During each loop (that is, during each single pass through the iteration), the static template text stays the same, but the tags each get another chance to run and produce new dynamic output. Because of this behavior, looping is very useful for building dynamic tables and lists.

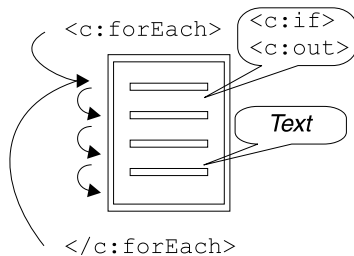


Figure 5.1
Text, other JSTL tags, and even arbitrary JSP can appear in the body of `<c:forEach>` tags.

In the core JSTL library, two tags handle looping: `<c:forEach>` and `<c:forEachTokens>`. These two tags have a lot in common; they mainly differ in the type of data they *loop over*—that is, the type of data they consider, item by item. In this chapter, we’ll look first at simple uses of `<c:forEach>` and `<c:forEachTokens>` separately, and then we’ll move to more complex iterations using tag attributes that are common to both `<c:forEach>` and `<c:forEachTokens>`.

5.1 General-purpose looping with `<c:forEach>`

The `<c:forEach>` tag is JSTL’s general-purpose looping tag. As you saw in chapter 4, the expression language can return a *collection* of items. The `<c:forEach>` tag lets you loop over nearly any sensible collection of items that the expression language returns. For instance, recall the picture of a shopping cart from figure 4.2 in chapter 4. The shopping cart contained three items: an apple, a TV, and a cup of coffee. If we looped over the shopping cart with `<c:forEach>`, then `<c:forEach>` would consider each of these items in turn.¹

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 `` 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.

`<c:forEach>` tag—in this case, the opening and closing `` tags—is included once each time the body of the tag is evaluated. The `<c:out>` tag prints a different value for each round of iteration because it refers to the `ailment` variable, which `<c:forEach>` sets to a new value for every loop. The `<c:forEach>` tag sets the `ailment` variable (and not some other variable with a different name) because that identifier appears in its `var` attribute.

The basic attributes of `<c:forEach>` are shown in table 5.1.

Table 5.1 `<c:forEach>` tag attributes for basic iteration



Attribute	Description	Required	Default
<code>items</code>	Collection over which to iterate	No	<i>None</i>
<code>var</code>	Name of the attribute to expose the current item	No	<i>None</i>

NOTE Even though the `items` attribute represents the core functionality of the `<c:forEach>` tag, you can use `<c:forEach>` without it. See section 5.3.2 for more information on when you might want to do this.

5.2 Iterating over strings with `<c:forTokens>`

Sometimes, data is not structured into a formal collection. If you are communicating with a so-called legacy application, accessing user input directly, or simply dealing with an application that has chosen to represent data as simple strings, you may need a tag that breaks a string into its constituent items. For instance, suppose you are writing an email-related web application, and the user has entered a list of email addresses in the following form:

```
shawn.bayern@yale.edu, david.davies@yale.edu, peter.peters@yale.edu
```

You might wish to analyze the string by breaking it into individual email addresses separated by commas and performing some action on each of them. (For instance, you might save them into your database of addresses so you can send unwanted junk mail to the entire world. Most web sites today seem to do this.) Analyzing a string is known as *parsing*.

When a string is broken into constituent items, these items are often called *tokens*. A token is a single, discrete unit within a larger string. The `<c:forTokens>` tag iterates over such tokens, which it parses from an input string. Table 5.2 shows the basic attributes that `<c:forTokens>` uses to retrieve tokens from within strings.

Table 5.2 `<c:forTokens>` tag attributes for basic iteration

Attribute	Description	Required	Default
items	Input string over which to iterate	Yes	None
delims	Delimiter characters that separate tokens	Yes	None
var	Name of the attribute to expose the current token	No	None

In short, `<c:forTokens>` uses the `items` and `delims` attributes to generate tokens, which it then exposes as the variable named by `var`. The `items` attribute refers to a string—either a simple, literal string typed directly into the tag, or an expression referring to a string. For instance, you could literally write

```
items="a,b,c"
```

or you could use the expression language

```
items="${emailAddresses}"
```

The `delims` attribute is a string that contains the characters you want to use to separate tokens inside the string. These separators are called *delimiters*. For instance, your string might be divided with a comma (`,`), but it could instead use another character, like a semicolon (`;`) or even the letter `q`. Each individual character in the `delims` attribute is treated, by itself, as a delimiter. Therefore, if `delims` is specified as

```
delims=".,;:"
```

then the four characters specified—period (`.`), comma (`,`), semicolon (`;`), and colon (`:`)—can separate tokens. For example, these `delims` separate the string

```
a,b.c;d:e:f.g
```

into the following tokens: `a`, `b`, `c`, `d`, `e`, `f`, and `g`.

Let's look at a few examples of `<c:forTokens>` in action. First, consider the case where you specify the `items` attribute's value directly inside the tag:

```
<c:forTokens items="a;b;c;d" delims=";" var="current">
  <li><c:out value="{current}" /></li>
</c:forTokens>
```

This example uses semicolons to separate the string `a;b;c;d` into four tokens: `a`, `b`, `c`, and `d`. It then prints the following output:

```
<li>a</li>
<li>b</li>
<li>c</li>
<li>d</li>
```

Now, suppose that instead of specifying a string directly in the `items` attribute, we use an expression. Imagine that a variable called `user` contains a `phone` property that stores a phone number. If `${user.phone}` returns the value `203-432-6687`, the following `<c:forTokens>` tag will break this string into three sets of numbers, each separated by a hyphen (-):

```
<c:forTokens items="${user.phone}" delims="-" var="current">
  <td><c:out value="${current}"/></td>
</c:forTokens>
```

This example prints the following output:

```
<td>203</td>
<td>432</td>
<td>6687</td>
```

Such output might be useful if you wanted to print a tabular list of numbers or separate the area code from the phone number.

Like `items`, the `delims` attribute accepts expressions. However, you'll use expressions in `items` much more commonly than in `delims`. (It's reasonable to assume that you'll just about always use an expression in `items` and rarely use one for `delims`.)

5.2.1 How JSTL parses strings

In an `items` string, delimiter characters that appear consecutively are treated as a single delimiter. Therefore, in a case like

```
<c:forTokens items="a,,b,,c" delims=",">
```

the `<c:forTokens>` tag finds three tokens: `a`, `b`, and `c`.

This rule applies even if there are different kinds of delimiters. For example, the following tag has the same effect as the last tag:

```
<c:forTokens items="a;;b;;c" delims=";">
```

Similarly, delimiters that appear at the beginning or at the end of `items` are ignored; they do not produce blank or empty tokens. Therefore, this tag has the same effect as the last two:

```
<c:forTokens items=";;a;;b;;c;;" delims=";">
```

The tag still finds just three tokens: `a`, `b`, and `c`.

Earlier, I mentioned that `<c:forEach>` can accept a simple string value for its own `items` attribute. In cases where this occurs, it is equivalent to `<c:forTokens>` with only the comma character as a delimiter (`delims=","`). This capability lets the `<c:forEach>` tag iterate over the tokens `a`, `b`, and `c` like this:

```
<c:forEach items="a,b,c">
```

Thus, think of `<c:forEach>` as supporting simple string tokenization and `<c:forTokens>` as providing more elaborate tokenization.

5.3 Advanced iteration with `<c:forEach>` and `<c:forTokens>`

So far, you've seen simple examples using `<c:forEach>` and `<c:forTokens>`. We've fed them collections of items and watched them run their bodies multiple times, producing different output each time.

But iteration in JSTL doesn't stop there. Using other tag attributes, you can customize the behavior of these tags. For instance, you can determine information about the current item's position within the overall loop: is it first, last, or somewhere in the middle? This ability is valuable in helping you construct visually appealing tables—for example, HTML tables whose rows alternate between two colors, or lists that treat the first item or last item specially. You can also use optional attributes of `<c:forEach>` and `<c:forTokens>` to decide to iterate over only part of the collection at hand: for instance, you might want to display the first 10 items only, leaving the rest for other pages (or a later portion of the same page). Or, you might want to print a table that uses only every second, or third, item in a collection (imagine, for instance, that a collection alternates between dates and times, and you want to print just dates).

We'll first consider JSTL's *subsetting* functionality: the ability of `<c:forEach>` and `<c:forTokens>` to use only part of the collection of items they have been given. Following that, we'll explore the use of *iteration status* (information about the current loop in the iteration) and how this status interacts with the features related to subsetting.

5.3.1 Looping over part of a collection

Sometimes, you simply have too much information, and you need to focus on just one part of it. For instance, consider again the `${user.medicalConditions}` expression we used in section 5.1. Imagine a situation where users are unfortunate enough to have developed so many ailments that the list won't manageably fit on a single page. (Of course, this example is somewhat outlandish, but it is not a difficult leap from this problem to one that most real-life search engines face.) In such situations, results are so numerous that they need to be spread over multiple pages. Therefore, an individual `<c:forEach>` or `<c:forTokens>` loop must deal with only part of the entire collection. Such a limited part is often called a *subset*.

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

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



Attribute	Description	Required	Default
<code>begin</code>	Item to start the loop (inclusive; 0=first item, 1=second item).	No	0
<code>end</code>	Item to end the loop (inclusive; 0=first item; 1=second item).	No	Last item
<code>step</code>	Iteration will process every <i>step</i> 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:forEachTokens>` 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:forEachTokens>` tag should begin with the first element and end with the fifth. Similarly, when a `<c:forEach>` or `<c:forEachToken>` 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.)

As an example of simple uses of `begin` and `end`, let's look at a `<c:forEachTokens>` tag that iterates over letters of the alphabet. Without a `begin` or `end` attribute, the following tag iterates over the letters from `a` through `f`:

```
<c:forEachTokens items="a,b,c,d,e,f" delims="," var="letter">
  <c:out value="${letter}"/>
</c:forEachTokens>
```

Suppose we add just the `begin` attribute:

```
<c:forEachTokens items="a,b,c,d,e,f" delims="," var="letter" begin="4">
  <c:out value="${letter}"/>
</c:forEachTokens>
```

This invocation of the tag will print out only the letters `e` and `f`. Similarly, we can use only `end`:

```
<c:forEachTokens items="a,b,c,d,e,f" delims="," var="letter" end="4">
  <c:out value="${letter}"/>
</c:forEachTokens>
```

This tag outputs `a`, `b`, `c`, `d`, and `e`, but not `f`, because the index of `e` in this collection is 4.

Of course, `begin` and `end` can be combined, as in tags like the following:

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

This loop outputs `c`, `d`, and `e`—the tokens with indexes 2, 3, and 4.

Only indexes that actually exist in the underlying collection can be included in the iteration. If `begin` has a value that is greater than any item's index—for instance, if you type `begin="20"` but there are only four items in your collection—then your `<c:forEach>` or `<c:forEachTokens>` tag won't iterate over anything. Its body will never be processed. If `end` is higher than the highest index—for instance, `end="50"` with a 47-element collection—then `end` has no effect.

The `step` attribute lets you filter the list further by skipping elements. The default behavior for `<c:forEach>` and `<c:forEachTokens>` is to include every element, but `step="2"` overrides this behavior and causes the tags to process only every second element. A value of 3 for `step` includes only every third element in the tag's iteration, and so on. If `step` does not line up evenly with `end`, or with the natural end of the collection, then some items at the end of the collection may be skipped. For instance, if 10 is the highest index in a collection, `begin` is 5, and `step` is 2, then only the indexes 5, 7, and 9 will be included.

Table 5.4 shows some examples of `begin`, `end`, and `step` operating together, assuming an underlying collection with 11 items (with indexes of 0 through 10). A hyphen indicates that the attribute was not specified.

Table 5.4 Examples of the effect of the `begin`, `end`, and `step` attributes in iteration tags. This table shows which values will be included in an iteration for various permutations of attributes.

Begin	End	Step	Included items (by index: 0=first item, 1=second item, and so forth)
-	-	-	0 1 2 3 4 5 6 7 8 9 10
3	-	-	3 4 5 6 7 8 9 10
-	3	-	0 1 2 3
-	-	3	0 3 6 9
3	3	-	3
3	3	3	3
0	9	2	0 2 4 6 8
0	9	3	0 3 6 9
0	9	4	0 4 8
0	9	5	0 5
0	9	6	0 6
0	9	20	0
20	30	1	<i>none (begin exceeds greatest index)</i>

5.3.2 Looping over numbers

I mentioned earlier that the `items` attribute is optional in `<c:forEach>`. If you don't specify `items`, then `begin` and `end` must be present. When this happens, a collection containing the numeric values specified with the `begin` and `end` attributes is the basis for the iteration. The following example

```
<c:forEach begin="1" end="5" var="current">
  <c:out value="\${current}"/>
</c:forEach>
```

outputs the following, ignoring white space:

```
1 2 3 4 5
```

In `<c:forEach>` tags that lack an `items` attribute, `step` is still permissible and has the expected effect. This example

```
<c:forEach begin="2" end="10" step="2" var="current">
  <c:out value="\${current}"/>
</c:forEach>
```

has the following output:

```
2 4 6 8 10
```

In `<c:forTokens>`, the `items` attribute is mandatory. If you want to iterate over numbers, just use `<c:forEach>`.

In the examples so far, we've typed the `begin`, `end`, and `step` attributes' values inside the tag. But like most other attributes in JSTL tags, these attributes accept expressions. If you have a scoped variable called `loopBoundary`, you could write

```
end="\${loopBoundary}"
```

TIP The primary use for iterating directly over numbers is to repeat the loop body a specific number of times. Sometimes, you just need to print a big list of numbers, but this is rare. Instead, looping over numbers is useful when you want to use `<c:forEach>` to repeat static content, but you want to decide how many times to print that content.

For example, many HTML pages manage white space manually using the ` `; HTML entity reference. Although there are often better approaches than using this entity to handle white space, it is more manageable to write

```
<c:forEach begin="1" end="50">&nbsp;</c:forEach>
```

than to type or paste ` `; 50 times. For example, you might find it easier to experiment with different numbers of spaces, or change the number quickly and in a well-defined manner, if you use the `<c:forEach>` tag. And, of course, you can vary the number dynamically. If `end="50"` in the previous example were instead `end="\${width}"`, where `\${width}` changed as appropriate, then the number of printed ` `; occurrences could grow or shrink as appropriate.

Although it feels like a dirty HTML hack, I have used a similar approach to produce table cells filled with dynamically generated ` `; entity references—for example, to represent data such as poll results or application progress graphically:

```
<table>
<tr>
  <td bgcolor="#00aa00">
    <c:forEach begin="1" end="\${width}">&nbsp;</c:forEach>
  </td>
```

```

    </tr>
</table>

```

This code prints a bar of variable width across the screen. You'll see this technique in practice in chapter 12.

5.3.3 Loop status

The `<c:forEach>` and `<c:forEachTokens>` tags also have a `varStatus` attribute that you can use to expose information about the iteration that is taking place. This information is useful when you want to exercise specialized control over a loop. Suppose, for instance, that you need to treat a collection's first or last item differently from the rest.

Like the `var` attribute in `<c:forEach>` and `<c:forEachTokens>`, `varStatus` lets you create a scoped variable that can be accessed inside the loop. If `var="current"`, the current item is exposed inside the loop as a page-scoped variable named `current`. Similarly, if `varStatus="s"`, then a variable that contains information about the current round of iteration is exposed as a variable named `s`. Table 5.5 shows the most useful properties of this status variable.

Table 5.5 The loop status variable (exposed by the `varStatus` attribute) includes properties for determining information about the current loop, such as whether it's the first or last iteration. You can also use it to determine the position of the current item in its original collection.

Property	Type	Description
<code>index</code>	number	The index of the current item in the underlying collection
<code>count</code>	number	The position of the current round of iteration, starting with 1
<code>first</code>	boolean	Whether the current round of iteration is the first
<code>last</code>	boolean	Whether the current round of iteration is the last

As you saw earlier, every item in a collection has an index number. These indexes start with 0 and increase by one for each element. The `index` property of `varStatus`'s variable holds the index of the current item in the loop. For instance, in a round of iteration for the first element in a collection, `index` will equal 0. For the second, `index` will equal 1. Whatever the value of the `step` attribute (the default 1 or a value greater than 1), `index` will jump by `step` for each round of iteration. As another example, if you use the attribute `begin="10"`, then for the first loop where `<c:forEach>` enters its body, `index` will equal 10.

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:forEachToken>` 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

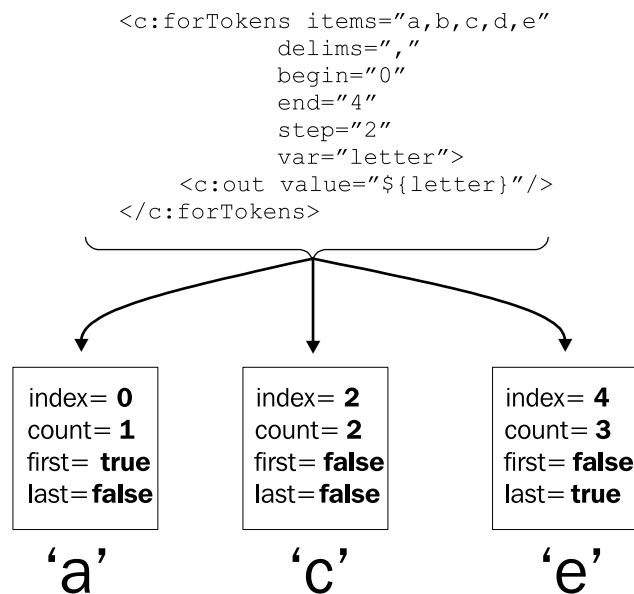


Figure 5.2
Values of the `varStatus` variable's properties during a sample iteration. The tag in this figure iterates three times, producing the letters `a`, `c`, and `e`. The boxes above each letter show the values of the `varStatus` variable for that letter's loop.

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.

Let's write a sample page that allows the user to scroll, or *page*, through information. The page's output should look like figure 5.3: the top prints ranges of data the user can click, and the bottom prints the data.

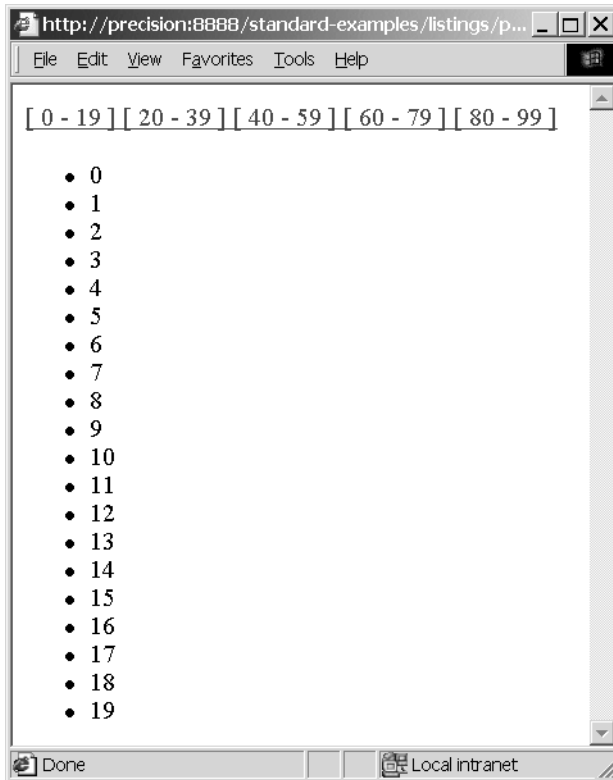


Figure 5.3
When a web page has too much data to fit on a single screen, it's useful to let the user jump around within the data. In the sample page we build in section 5.4, users can click the links at the top of the page to choose what data to view. You can produce pages with this feature using only the tags you've learned about so far.

We can accomplish this result with just the iteration and conditional tags you've seen so far, using the expression language and the `<c:out>` tag to assist us. Of course, for a page to be useful, it needs real data to display. Because we haven't yet looked at how to retrieve information from XML files and databases, we can't yet display real data. So, we'll use `<c:forEach>`'s ability to generate numbers for us automatically when we use the `begin` and `end` attributes without `items`. Thus, we'll be able to experiment with a simple page that lets us scroll over numbers. (As soon as you have real data, you can insert this data into `<c:forEach>`'s `items` attribute, and the page will then let you loop over interesting information, not just numbers.)

Without further ado, let's look at the code (see listing 5.1).

Listing 5.1 scroll.jsp: Using JSTL to let the user scroll through results

```

<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<c:set var="totalCount" scope="session" value="100"/>
<c:set var="perPage" scope="session" value="20"/>

<c:forEach
  var="boundaryStart"
  begin="0"
  end="${totalCount - 1}"
  step="${perPage}">

  <a href="?start=<c:out value="${boundaryStart}"/>">
    [
      <c:out value="${boundaryStart}"/>
      -
      <c:out value="${boundaryStart + perPage - 1}"/>
    ]
  </a>
</c:forEach>

<c:forEach
  var="current"
  varStatus="status"
  begin="${param.start}"
  end="${param.start + perPage - 1}">
<c:if test="${status.first}">
  <ul>
</c:if>
<li><c:out value="${current}"/></li>
<c:if test="${status.last}">
  </ul>
</c:if>
</c:forEach>

```

① Configuration

② Prints ranges

③ Prints data items

5.4.1 Understanding the example

This is our first real-world example of a full JSTL page, so let's go through it carefully. Overall, the example can be broken into three sections.

- ① We use two `<c:set>` tags to configure our page's behavior. The tags are here only for demonstrative purposes; the two values they set could easily come from anywhere else—for instance, back-end Java code, request parameters, and so on. The rest of the example depends on the two variables these tags create:
 - `totalCount`—The total number of items we'd like to let the user scroll through. For instance, if we retrieved 100 items from the database, we'd want `totalCount` to equal 100. That's the sample value we use here.

- `perPage`—The number of results we want to show the user on each page. Here, we set this value to 20, which seems like a reasonable number: it's not so small that it's irritating, but it's not so large that it overwhelms people.
- ② With these variables set, we can print out our page's top section (see figure 5.3). This section will look the same no matter which data the user has chosen. (As an exercise, think about how you could highlight or otherwise draw attention to the range the data is currently displaying.) This data depends only on the two variables that configure our page's behavior: `totalCount` and `perPage`.

The goal of this section is to produce links of the form

```
[ 0 - 19 ]
```

that let the user choose which results will be displayed.

To produce these links, we loop over our data from the first item (`begin="0"`) to the last one (`end="${totalCount - 1}"`). We set the `step` attribute to `perPage` so that we loop only once for each range we want to print. For instance, if we start with 0 and `perPage` is 20, we want the current item—which we call `boundaryStart`—to be 0, then 20, then 40, then 60, and so on. Within each loop, we print out the starting and ending numbers of the range. The starting number is simple: it's `boundaryStart` (the current item). To get the ending number of the range, we add `perPage` to `boundaryStart` using the expression language. Note that we subtract 1 from this sum, because we don't want this boundary to spill over into the next one. We want to print `[0 - 19]`, `[20 - 39]`, and so on, not `[0 - 20]` and then `[20 - 40]`. (We begin with 0 to demonstrate that our data starts with item 0, as is usually the case as far as `<c:forEach>` is concerned. However, if we were printing real data instead of numbers, we could add 1 to the expressions in both the `begin` and `end` attributes to make the numbers friendlier to the user: `[1 - 20]`, `[21 - 40]`, and so on.)

We use one trick that needs to be explained. The block of JSTL used to begin each link looks like this:

```
<a href="?start=<c:out value='${boundaryStart}'/">
```

We're using a `<c:out>` tag in the middle of an HTML `<a>` tag: no problem there! Our trick involves the `<a>` tag itself. Once the `<c:out>` placeholder gets filled in, the tag will look like this:

```
<a href="?start=20">
```

20 could just as easily have been any other number printed by the `<c:out>` tag. This `<a>` tag means, "Link back to the current page, sending a request parameter named `start` that equals 20." Thus, when our user chooses a specific range, a `start` parameter will indicate the start of the range the user asked for. We can access this parameter using the expression `${param.start}`.

TIP In general, you can send request parameters as part of a URL in the form `param1=value1¶m2=value2`. In chapter 6, we'll look at a much friendlier way to send parameters as part of an HTML `<a>` link. But it's useful to have seen this manual style in action at least once.

- ③ The final section of our page displays the selected results. If no results have been selected—that is, if the `start` parameter is empty—we begin with the number 0. We do so because `begin="${param.start}"` is the same as `begin="0"` if the `start` parameter doesn't exist. In the page's second loop, we begin at the start of the range and loop over `perPage` elements: `end="${param.start + perPage - 1}"`, which means, “End the iteration with the item `perPage` items away from `param.start`.” We subtract 1 because the `end` attribute causes the loop to include the final value, but we want to make sure our ranges don't overlap. Subtracting 1 from the `end` attribute is a fairly common pattern in JSTL pages when you work with ranges.

5.4.2 Using `varStatus` in the example

In the loop at the bottom of the page in our example, notice that we use `varStatus` to expose a variable named `status`. Instead of printing the beginning `` and the ending `` outside the `<c:forEach>` tag, we've moved those tags inside and placed `<c:if>` tags around them. This way, we can be sure the `` element will print only if there are `` items to fill it. That is, we print `` only when we encounter our first piece of data, and we print the closing `` tag for the last item. (Recall from the previous chapter that although empty lists—``—display fine in most browsers, they're technically an error.) The empty list is not a possibility in our example; because we're just iterating over numbers, they can't be missing! But a real-life collection might not be as big as we expect it to be, in which case these defensive checks are appropriate.

Because we expose this `status` variable, we can also use it to modify the loop's behavior in other ways. For instance, suppose we wanted to treat alternating rows differently. Normally, we might use different colors, but because colors don't show up well in a black-and-white book, we'll instead use different font sizes. Let's make every second row print in a small font:

```
<c:forEach
  var="current"
  varStatus="status"
  begin="${param.start}"
  end="${param.start + perPage - 1}">
<c:if test="${status.first}">
<ul>
```

```

</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 `` tag, and once to close it with ``.

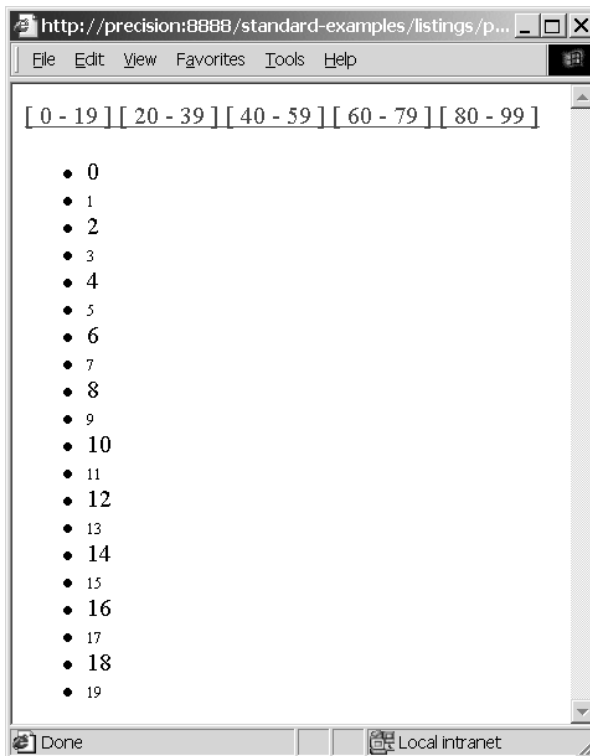


Figure 5.4
 Many web sites display alternating rows in different colors. Because colors don't show up well in a black-and-white book, our example of handling alternate rows uses font size instead of color. Here, every second row prints using small text.

5.5 Summary

When using JSTL's iteration (loop) tags, keep the following points in mind:

- The `<c:forEach>` tag is JSTL's general-purpose looping tag. It lets you loop over nearly any kind of collection.
- The `<c:forEachTokens>` tag breaks apart strings and loops over these string fragments, called *tokens*.
- When you iterate, you'll usually want to expose a variable using the iteration tag's `var` attribute. Doing so lets you access each individual item, one by one, in the body of your iteration tag.
- The iteration tags support a `varStatus` attribute that lets you recover information about where you are in the overall iteration. The tags also support `begin`, `end`, and `step` attributes to let you write underachieving `<c:forEach>` and `<c:forEachToken>` tags that iterate over only parts of a collection.