A JSTL primer: Getting down to the core

Flow control and URL management through custom tags

Level: Intermediate

Mark A. Kolb (mak@taglib.com)
Software Engineer
March 18, 2003

The JSP Standard Tag Library (JSTL) `core` library, as its name suggests, provides custom tags for basic functionality, such as managing scoped variables and interacting with URLs, as well as for fundamental operations like iteration and conditionalization. Not only can these tags be leveraged directly by page authors, but they also provide a foundation for more complex presentation logic in combination with the other JSTL libraries. Mark Kolb continues his exploration into JSTL and the `core` library with a look at tags to assist with flow control and URL management.

In the initial article of this series, you got your first look at JSTL. We described the use of its *expression language* (EL) to access data and operate on it. As you learned, the EL is used to assign dynamic values to the attributes of JSTL custom tags, and thus plays the same role as JSP expressions for specifying request-time attribute values for the built-in actions and other custom tag libraries.

To demonstrate the use of the EL, we introduced three tags from the `core` library: `<c:set>`, `<c:remove>`, and `<c:out>`. `<c:set>` and `<c:remove>` are used for managing scoped variables; `<c:out>` is for displaying data, particularly values computed using the EL. Based on this groundwork, then, we will focus our attention in this article on the remaining tags in the `core` library, which can be broadly grouped into two major categories: flow control and URL management.

Example application

To demonstrate the JSTL tags, we'll use examples from a working application for the remaining articles in this series. Because of their growing popularity and familiarity, we'll use a simple Java-based Weblog for this purpose; see Resources to download the JSP pages and source code for this application. A Weblog (also known as a blog) is a Web-based journal of short commentaries on topics of interest to the Weblog's author, typically with links to related articles or discussions elsewhere on the Web. A screenshot of the running application is shown in Figure 1.

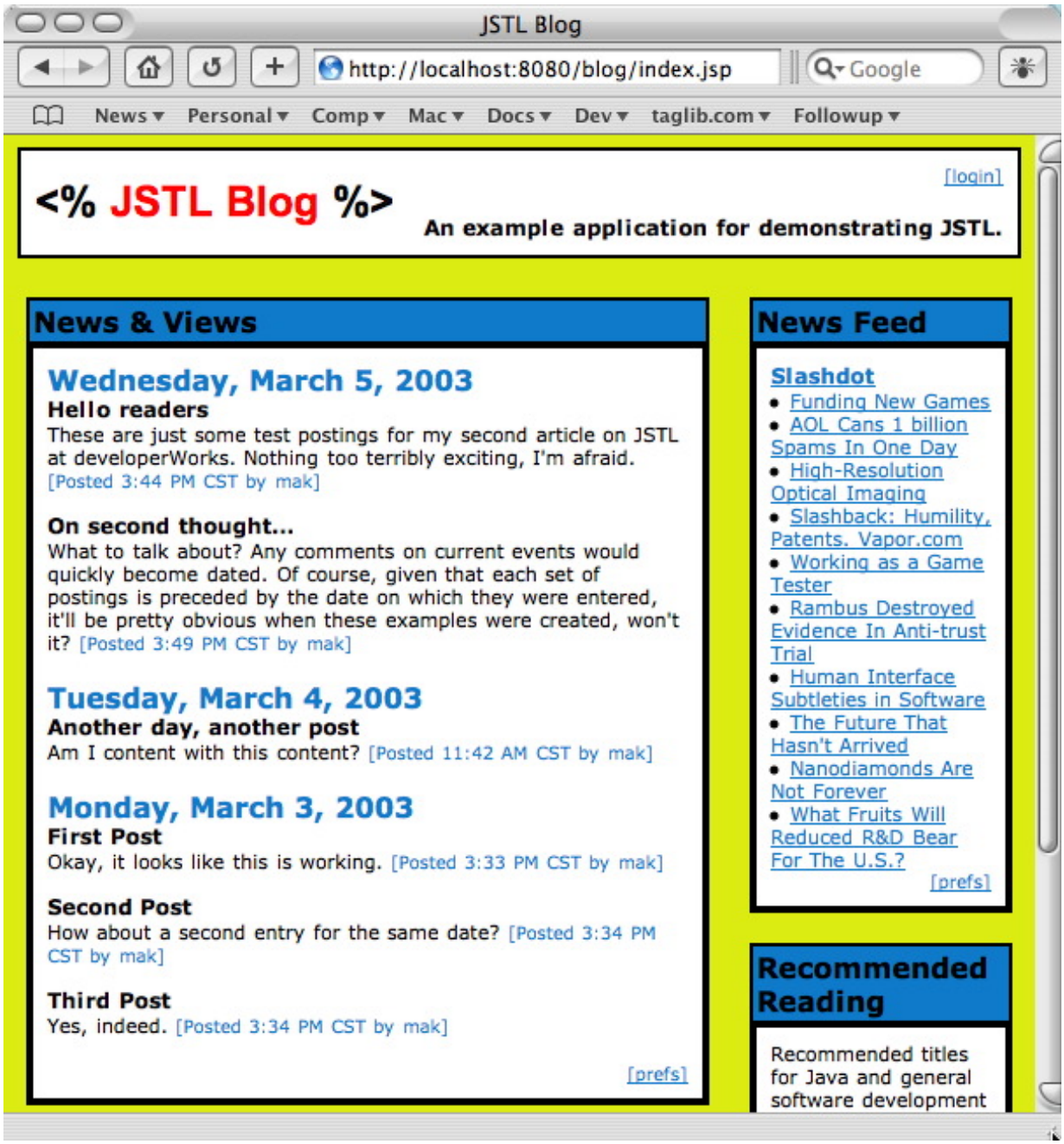
Figure 1. The Weblog application

Related content:

A JSTL primer: The expression language
Using JSPs and custom tags within VisualAge for Java and WebSphere Studio
Take control of your JSP pages with custom tags
JSP taglibs: Better usability by design
Subscribe to the developerWorks newsletter

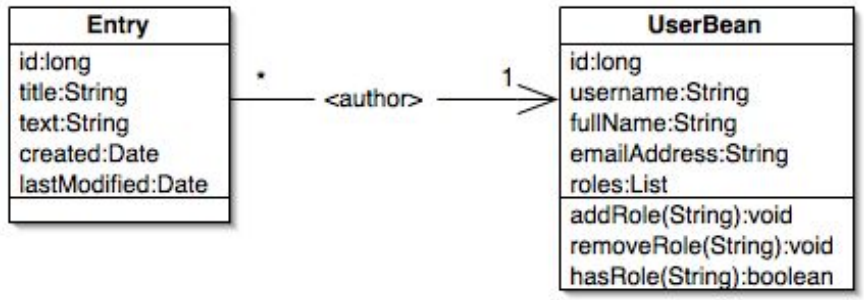
Also in the Java zone:

Tutorials
Tools and products
Code and components
Articles



Although a couple dozen Java classes are required for the full implementation, only two of the Weblog application's classes, `Entry` and `UserBean`, are referenced in the presentation layer. To understand the JSTL examples, then, only these two classes are important. A class diagram for `Entry` and `UserBean` is shown in Figure 2.

Figure 2. Class diagram for the Weblog application



The `Entry` class represents a dated entry within a Weblog. Its `id` attribute is used to store and retrieve the entry in a database, while the `title` and `text` attributes represent the entry's actual content. Two instances of the Java language's `Date` class are referenced by the `created` and `lastModified` attributes, representing when the entry was first created and last edited. The `author` attribute references a `UserBean` instance identifying the person who created the entry.

The `UserBean` class stores information about the application's authenticated users, such as user name, full name, and e-mail address. This class also includes an `id` attribute for interacting with an associated database. Its final attribute, `roles`, references a list of `String` values identifying the application-specific roles associated with the corresponding user. For the Weblog application, the relevant roles are "User" (the default role common to all application users) and "Author" (the role that designates users who are allowed to create and edit Weblog entries).

Flow control

Because the EL can be used in place of JSP expressions to specify dynamic attribute values, it reduces the need for page authors to use scripting elements. Because scripting elements can be a significant source of maintenance problems in JSP pages, providing simple (and standard) alternatives to their use is a major advantage of JSTL.

The EL retrieves data from the JSP container, traverses object hierarchies, and performs simple operations on the results. In addition to accessing and manipulating data, however, another common use of JSP scripting elements is flow control. In particular, it is fairly common for page authors to resort to scriptlets to implement iterative or conditional content. However, because such operations are beyond the capabilities of the EL, the `core` library instead provides several custom actions to manage flow control in the form of iteration, conditionalization, and exception handling.

Iteration

In the context of Web applications, iteration is primarily used to fetch and display collections of data, typically in the form of a list or sequence of rows in a table. The primary JSTL action for implementing iterative content is the `<c:forEach>` custom tag. This tag supports two different styles of iteration: iteration over an integer range (like the Java language's `for` statement) and iteration over a collection (like the Java language's `Iterator` and `Enumeration` classes).

To iterate over a range of integers, the syntax of the `<c:forEach>` tag shown in Listing 1 is used. The `begin` and `end` attributes should be either static integer values or expressions evaluating to integer values. They specify the initial value of the index for the iteration and the index value at which iteration should cease, respectively. When iterating over a range of integers using `<c:forEach>`, these two attributes are required and all others are optional.

Listing 1. Syntax for numerical iteration through the <c:forEach> action

```
<c:forEach var="name" varStatus="name"
    begin="expression" end="expression" step="expression">
  body content
</c:forEach>
```

The `step` attribute, when present, must also have an integer value. It specifies the amount to be added to

the index after each iteration. The index of the iteration thus starts at the value of the `begin` attribute, is incremented by the value of the `step` attribute, and halts iteration when it exceeds the value of the `end` attribute. Note that if the `step` attribute is omitted, the step size defaults to 1.

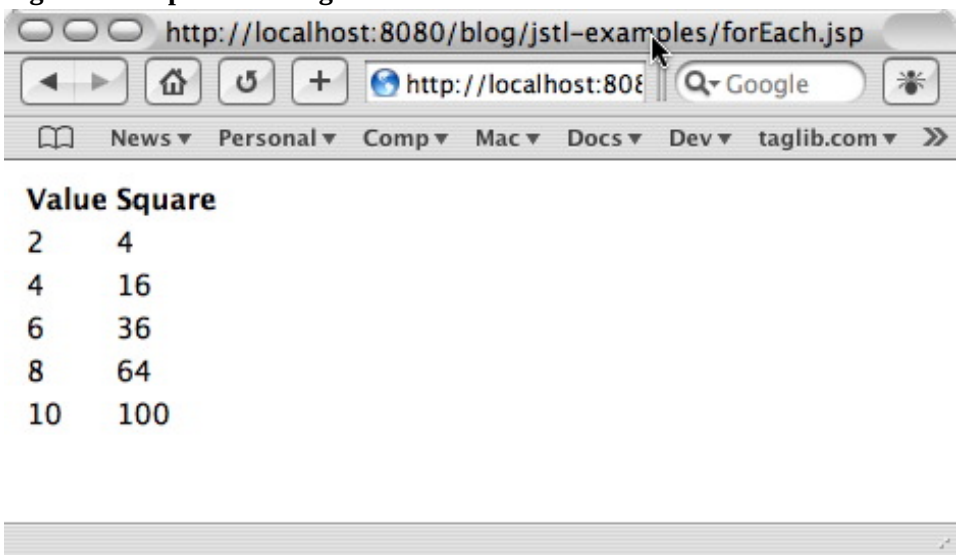
If the `var` attribute is specified, then a scoped variable with the indicated name will be created and assigned the current value of the index for each pass through the iteration. This scoped variable has nested visibility -- it can only be accessed within the body of the `<c:forEach>` tag. (We'll discuss the use of the optional `varStatus` attribute shortly.) Listing 2 shows an example of the `<c:forEach>` action for iterating over a fixed set of integer values.

Listing 2. Using the <c:forEach> tag to generate tabular data corresponding to a range of numeric values

```
<table>
<tr><th>Value</th>
    <th>Square</th></tr>
<c:forEach var="x" begin="0" end="10" step="2">
  <tr><td><c:out value="${x}"/></td>
      <td><c:out value="${x * x}"/></td></tr>
</c:forEach>
</table>
```

This example code generates a table of the squares of the first five even numbers, as shown in Figure 3. This is accomplished by specifying a value of two for both the `begin` and `step` attributes, and a value of ten for the `end` attribute. In addition, the `var` attribute is used to create a scoped variable for storing the index value, which is referenced within the body of the `<c:forEach>` tag. Specifically, a pair of `<c:out>` actions are used to display the index and its square, the latter of which is computed using a simple expression.

Figure 3. Output of Listing 2



When iterating over the members of a collection, one additional attribute of the `<c:forEach>` tag is used: the `items` attribute, which is shown in Listing 3. When you use this form of the `<c:forEach>` tag, the `items` attribute is the only required attribute. The value of the `items` attribute should be the collection over whose members the iteration is to occur, and is typically specified using an EL expression. If a variable name is specified through the `<c:forEach>` tag's `item` attribute, then the named variable will be bound to successive elements of the collection for each iteration pass.

Listing 3. Syntax for iterating through a collection through the <c:forEach> action

```
<c:forEach var="name" items="expression" varStatus="name"
    begin="expression" end="expression" step="expression">
  body content
</c:forEach>
```

All of the standard collection types provided by the Java platform are supported by the `<c:forEach>` tag. In addition, you can use this action to iterate through the elements of an array, including arrays of primitives. Table 1 contains a complete list of the values supported by the `items` attribute. As the final row of the table indicates, JSTL defines its own interface, `javax.servlet.jsp.jstl.sql.Result`, for iterating through the result of an SQL query. (We'll present further details on this capability in a later article in this series.)

Table 1. Collections supported by the `items` attribute of the <c:forEach> tag

Value for `items`	**Resulting `item` values**
`java.util.Collection`	Elements from call to `iterator()`
`java.util.Map`	Instances of `java.util.Map.Entry`
`java.util.Iterator`	Iterator elements
`java.util.Enumeration`	Enumeration elements
Array of `Object` instances	Array elements
Array of primitive values	Wrapped array elements
Comma-delimited `String`	Substrings
`javax.servlet.jsp.jstl.sql.Result`	Rows from an SQL query

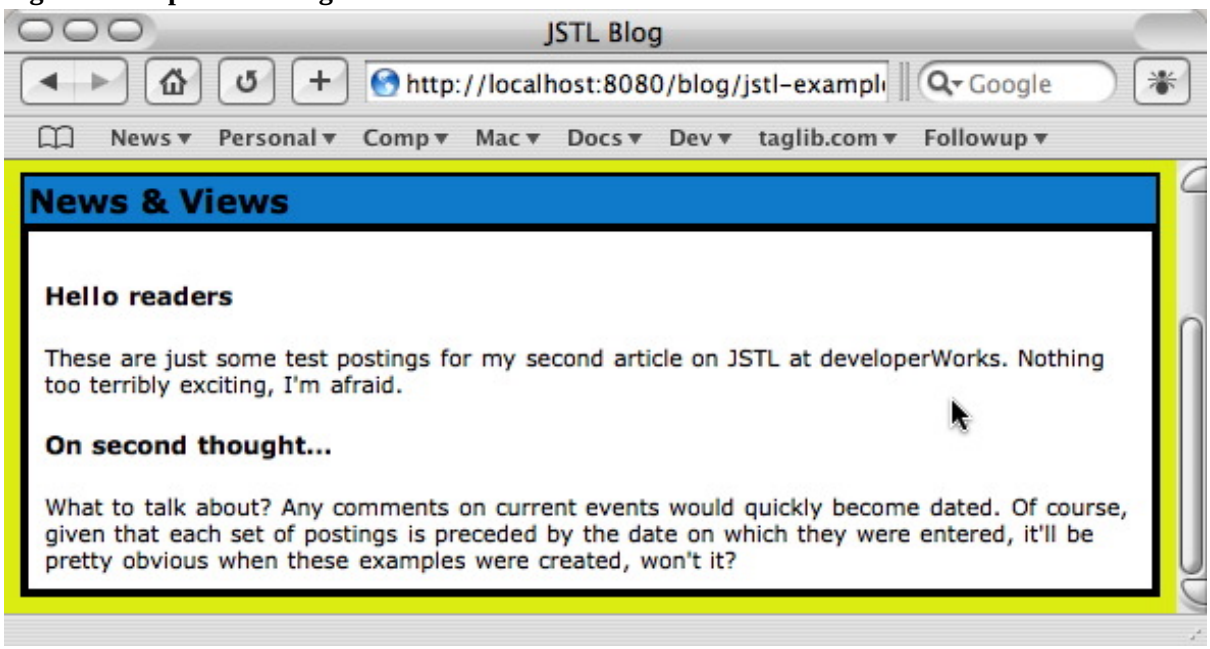
You can use the `begin`, `end`, and `step` attributes to restrict which elements of the collection are included in the iteration. As was the case for numerical iteration through `<c:forEach>`, an iteration index is also maintained when iterating through the elements of a collection. Only those elements that correspond to index values matching the specified `begin`, `end`, and `step` values will actually be processed by the `<c:forEach>` tag.

Listing 4 shows the `<c:forEach>` tag being used to iterate through a collection. For this JSP fragment, a scoped variable named `entryList` has been set to a list (specifically, an `ArrayList`) of `Entry` objects. The `<c:forEach>` tag processes each element of this list in turn, assigning it to a scoped variable named `blogEntry`, and generating two table rows -- one for the Weblog entry's `title` and a second for its `text`. These properties are retrieved from the `blogEntry` variable through a pair of `<c:out>` actions with corresponding EL expressions. Note that, because both the title and text of a Weblog entry might contain HTML markup, the `escapeXml` attribute of both `<c:out>` tags is set to false. Figure 4 shows the result.

Listing 4. Displaying the Weblog entries for a given date using the <c:forEach> tag

```
<table>
  <c:forEach items="${entryList}" var="blogEntry">
    <tr><td align="left" class="blogTitle">
      <c:out value="${blogEntry.title}" escapeXml="false"/>
    </td></tr>
    <tr><td align="left" class="blogText">
      <c:out value="${blogEntry.text}" escapeXml="false"/>
    </td></tr>
  </c:forEach>
</table>
```

Figure 4. Output of Listing 4



The remaining `<c:forEach>` attribute, `varStatus`, plays the same role whether iterating over integers or collections. Like the `var` attribute, `varStatus` is used to create a scoped variable. Instead of storing the current index value or the current element, however, the variable named by the `varStatus` attribute is assigned an instance of the `javax.servlet.jsp.jstl.core.LoopTagStatus` class. This class defines a set of properties, listed in Table 2, that describe the current state of an iteration.

Table 2. Properties of the LoopTagStatus object

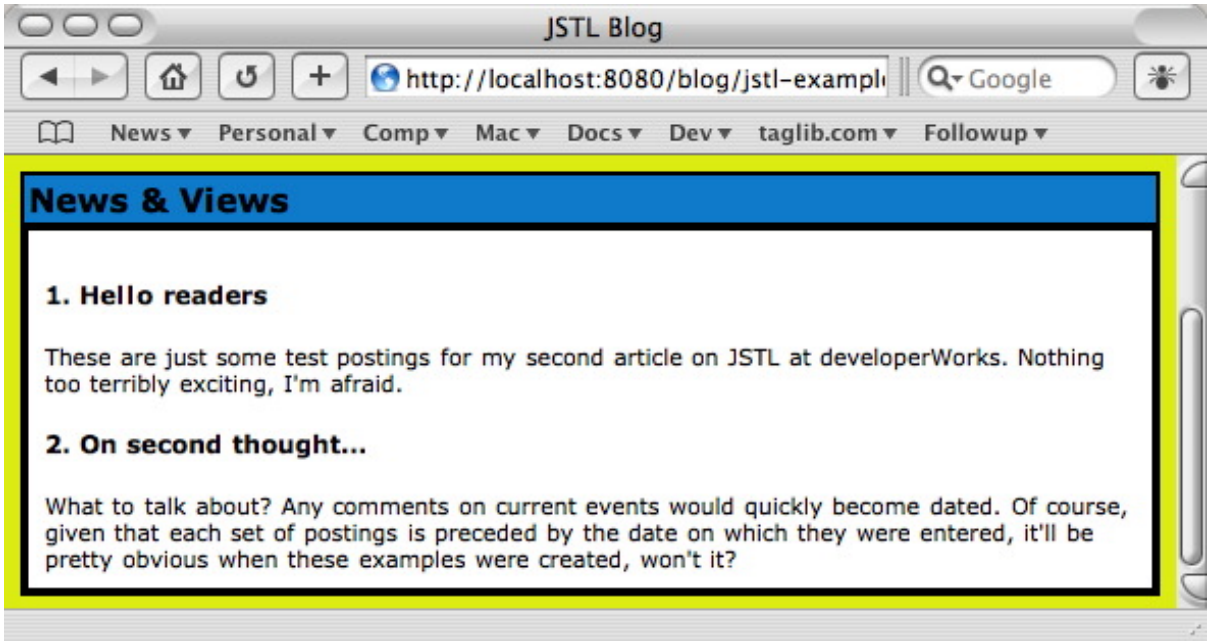
Property	Getter	Description
current	`getCurrent()`	The item (from the collection) for the current round of iteration
index	`getIndex()`	The zero-based index for the current round of iteration
count	`getCount()`	The one-based count for the current round of iteration
first	`isFirst()`	Flag indicating whether the current round is the first pass through the iteration
last	`isLast()`	Flag indicating whether the current round is the last pass through the iteration
begin	`getBegin()`	The value of the `begin` attribute
end	`getEnd()`	The value of the `end` attribute
step	`getStep()`	The value of the `step` attribute

Listing 5 shows an example of how the `varStatus` attribute is used. It modifies the code in Listing 4 to add numbering of the Weblog entries to the table rows displaying their titles. This is done by specifying a value for the `varStatus` attribute and then accessing the `count` property of the resulting scoped variable. The results appear in Figure 5.

Listing 5. Using the varStatus attribute to display a count of Weblog entries

```
<table>
  <c:forEach items=
    "${entryList}" var="blogEntry" varStatus="status">
    <tr><td align="left" class="blogTitle">
      <c:out value="${status.count}"/>.
      <c:out value="${blogEntry.title}" escapeXml="false"/>
    </td></tr>
    <tr><td align="left" class="blogText">
      <c:out value="${blogEntry.text}" escapeXml="false"/>
    </td></tr>
  </c:forEach>
</table>
```

Figure 5. Output of Listing 5



In addition to `<c:forEach>`, the `core` library provides a second iteration tag: `<c:forTokens>`. This custom action is the JSTL analog of the Java language's `StringTokenizer` class. The `<c:forTokens>` tag, shown in Listing 6, has the same set of attributes as the collection-oriented version of `<c:forEach>`, with one addition. For `<c:forTokens>`, the string to be tokenized is specified through the `items` attribute, while the set of delimiters used to generate the tokens is provided through the `delims` attribute. As was the case for `<c:forEach>`, you can use the `begin`, `end`, and `step` attributes to restrict the tokens to be processed to those matching the corresponding index values.

Listing 6. Syntax for iterating through a string's tokens with the <c:forTokens> action

```
<c:forTokens var="name" items="expression"
    delims="expression" varStatus="name"
    begin="expression" end="expression" step="expression">
  body content
</c:forTokens>
```

Conditionalization

For Web pages containing dynamic content, you might want different categories of users to see different forms of content. In our Weblog, for instance, visitors should be able to read entries and perhaps submit

feedback, but only authorized users should be able to post new entries or edit existing content.

Both usability and software maintenance are often improved by implementing such features within the same JSP page and then using conditional logic to control what gets displayed on a per-request basis. The `core` library provides two different conditionalization tags -- `<c:if>` and `<c:choose>` -- to implement these features.

The more straightforward of these two actions, `<c:if>`, simply evaluates a single test expression and then processes its body content only if that expression evaluates to `true`. If not, the tag's body content is ignored. As Listing 7 shows, `<c:if>` can optionally assign the result of the test to a scoped variable through its `var` and `scope` attributes (which play the same role here as they do for `<c:set>`). This capability is particularly useful if the test is expensive: the result can be cached in a scoped variable and retrieved in subsequent calls to `<c:if>` or other JSTL tags.

Listing 7. Syntax for the <c:if> conditional action

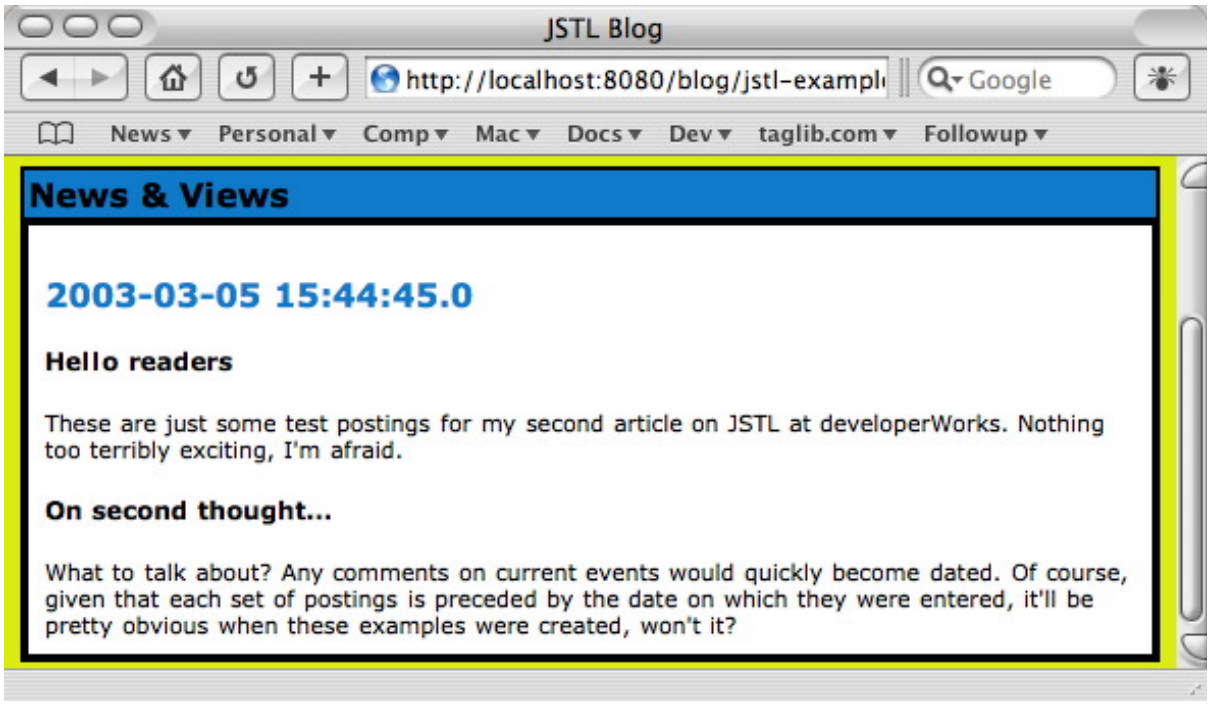
```
<c:if test="expression" var="name" scope="scope">
  body content
</c:if>
```

Listing 8 shows `<c:if>` used with the `first` property of a `<c:forEach>` tag's `LoopTagStatus` object. In this case, as shown in Figure 6, the creation date for a set of Weblog entries is displayed just above the first entry for that date, but is not repeated before any of the other entries.

Listing 8. Using <c:if> to display the date for Weblog entries

```
<table>
  <c:forEach items=
    "${entryList}" var="blogEntry" varStatus="status">
    <c:if test="${status.first}">
      <tr><td align="left" class="blogDate">
            <c:out value="${blogEntry.created}"/>
      </td></tr>
    </c:if>
    <tr><td align="left" class="blogTitle">
      <c:out value="${blogEntry.title}" escapeXml="false"/>
    </td></tr>
    <tr><td align="left" class="blogText">
      <c:out value="${blogEntry.text}" escapeXml="false"/>
    </td></tr>
  </c:forEach>
</table>
```

Figure 6. Output of Listing 8



As Listing 8 shows, the `<c:if>` tag provides a very compact notation for simple cases of conditionalized content. For cases in which mutually exclusive tests are required to determine what content should be displayed, the JSTL `core` library also provides the `<c:choose>` action. The syntax for `<c:choose>` is shown in Listing 9.

Listing 9. Syntax for the <c:choose> action

```
<c:choose>
  <c:when test="expression">
    body content
  </c:when>
  ...
  <c:otherwise>
    body content
  </c:otherwise>
</c:choose>
```

Each condition to be tested is represented by a corresponding `<c:when>` tag, of which there must be at least one. Only the body content of the first `<c:when>` tag whose `test` evaluates to `true` will be processed. If none of the `<c:when>` tests return `true`, then the body content of the `<c:otherwise>` tag will be processed. Note, though, that the `<c:otherwise>` tag is optional; a `<c:choose>` tag can have at most one nested `<c:otherwise>` tag. If all `<c:when>` tests are `false` and no `<c:otherwise>` action is present, then no `<c:choose>` body content will be processed.

Listing 10 shows an example of the `<c:choose>` tag in action. Here, protocol information is retrieved from the request object (by means of the EL's `pageContext` implicit object) and tested using a simple string comparison. Based on the results of these tests, a corresponding text message is displayed.

Listing 10. Content conditionalization using <c:choose>

```
<c:choose>
  <c:when test="${pageContext.request.scheme eq 'http'}">
    This is an insecure Web session.
  </c:when>
  <c:when test="${pageContext.request.scheme eq 'https'}">
    This is a secure Web session.
  </c:when>
  <c:otherwise>
    You are using an unrecognized Web protocol. How did this happen?!
  </c:otherwise>
</c:choose>
```

Exception handling

The final flow-control tag is `<c:catch>`, which allows for rudimentary exception handling within a JSP page. More specifically, any exceptions raised within the body content of this tag will be caught and ignored (that is, the standard JSP error-handling mechanism will not be invoked). However, if an exception is raised and the `<c:catch>` tag's optional `var` attribute has been specified, the exception will be assigned to the specified variable (with page scope), enabling custom error handling within the page itself. Listing 11 shows the syntax of `<c:catch>` (an example appears later in Listing 18).

Listing 11. Syntax for the <c:catch> action

```
<c:catch var="name">
  body content
</c:catch>
```

URL actions

The remaining tags in the JSTL `core` library focus on URLs. The first of these, the aptly named `<c:url>` tag, is used to generate URLs. In particular, `<c:url>` provides three elements of functionality that are particularly important when constructing URLs for J2EE Web applications:

- Prepending the name of the current servlet context
- URL re-writing for session management
- URL encoding of request-parameter names and values

Listing 12 shows the syntax for the `<c:url>` tag. The `value` attribute is used to specify a base URL, which the tag then transforms as necessary. If this base URL starts with a forward slash, then a servlet context name will be prepended. An explicit context name can be provided using the `context` attribute. If this attribute is omitted, then the name of the current servlet context will be used. This is particularly useful because servlet context names are decided during deployment, rather than during development. (If the base URL does not start with a forward slash, then it is assumed to be a relative URL, in which case the addition of a context name is unnecessary.)

Listing 12. Syntax for the <c:url> action

```
<c:url value="expression" context="expression"
    var="name" scope="scope">
  <c:param name="expression" value="expression"/>
  ...
</c:url>
```

URL rewriting is automatically performed by the `<c:url>` action. If the JSP container detects a cookie storing the user's current session ID, no rewriting is necessary. If no such cookie is present, however, all URLs generated by `<c:url>` will be rewritten to encode the session ID. Note that if an appropriate

cookie is present in subsequent requests, `<c:url>` will stop rewriting URLs to include this ID.

If a value is supplied for the `var` attribute (optionally accompanied by a corresponding value for the `scope` attribute), the generated URL will be assigned as the value of the specified scoped variable. Otherwise, the resulting URL will be output using the current `JspWriter`. This ability to directly output its result allows the `<c:url>` tag to appear as the value, for example, of the `href` attribute of an HTML `<a>` tag, as shown in Listing 13.

Listing 13. Generating a URL as the attribute value for an HTML tag

```
<a href="<c:url value='/content/sitemap.jsp'/>">View sitemap</a>
```

Finally, if any request parameters are specified through nested `<c:param>` tags, then their names and values will be appended to the generated URL using the standard notation for HTTP GET requests. In addition, URL encoding is performed: any characters present in either the names or values of these parameters that must be transformed in order to yield a valid URL will be translated appropriately. Listing 14 illustrates the behavior of `<c:url>`.

Listing 14. Generating a URL with request parameters

```
<c:url value="/content/search.jsp">
  <c:param name="keyword" value="${searchTerm}"/>
  <c:param name="month" value="02/2003"/>
</c:url>
```

The JSP code in Listing 14 has been deployed to a servlet context named `blog`, and the value of the scoped variable `searchTerm` has been set to `"core library"`. If a session cookie has been detected, then the URL generated by Listing 14 will be like the one in Listing 15. Note that the context name has been prepended, and the request parameters have been appended. In addition, the space in the value of the `keyword` parameter and the forward slash in the value of the `month` parameter have been encoded as required for HTTP GET parameters (specifically, the space has been translated into a `+` and the slash has been translated into the sequence `%2F`).

Listing 15. URL generated in the presence of a session cookie

```
/blog/content/search.jsp?keyword=foo+bar&month=02%2F2003
```

When no session cookie is present, the URL in Listing 16 is the result. Again, the servlet context has been prepended and the URL-encoded request parameters have been appended. In addition, however, the base URL has been rewritten to include specification of a session ID. When a browser sends a request for a URL that has been rewritten in this manner, the JSP container automatically extracts the session ID and associates the request with the corresponding session. In this way, a J2EE application that requires session management doesn't need to rely on cookies being enabled by users of the application.

Listing 16. URL generated in the absence of a session cookie

```
/blog/content/search.jsp;jsessionid=233379C7CD2D0ED2E9F3963906DB4290
  ?keyword=foo+bar&month=02%2F2003
```

Importing content

JSP has two built-in mechanisms to incorporate content from a different URL into a JSP page: the `include` directive and the `<jsp:include>` action. In both cases, however, the content to be included must be part of the same Web application (or servlet context) as the page itself. The major distinction between these two tags is that the `include` directive incorporates the included content during page compilation, while the `<jsp:include>` action operates during request-time processing of JSP pages.

The core library's <c:import> action is essentially a more generic, more powerful version of <jsp:include> (sort of a <jsp:include> on steroids). Like <jsp:include>, <c:import> is a request-time action, and its basic task is to insert the content of some other Web resource into a JSP page. Its syntax is very similar to that of <c:url>, as shown in Listing 17.

Listing 17. Syntax for the <c:import> action

```
<c:import url="expression" context="expression"
    charEncoding="expression" var="name" scope="scope">
  <c:param name="expression" value="expression"/>
  ...
</c:import>
```

The URL for the content to be imported is specified through the url attribute, which is <c:import>'s only required attribute. Relative URLs are permitted and are resolved against the URL of the current page. If the value of the url attribute starts with a forward slash, however, it is interpreted as an absolute URL within the local JSP container. Without a value for the context attribute, such an absolute URL is assumed to reference a resource in the current servlet context. If an explicit context is specified through the context attribute, then the absolute (local) URL is resolved against the named servlet context.

The <c:import> action is not limited to accessing local content, however. Complete URIs, including protocol and host names, can also be specified as the value of the url attribute. In fact, the protocol is not even restricted to HTTP. Any protocol supported by the java.net.URL class may be used in the value for the url attribute of <c:import>. This capability is shown in Listing 18.

Here, the <c:import> action is used to include the content of a document accessed through the FTP protocol. In addition, the <c:catch> action is employed to locally handle any errors that might occur during the FTP file transfer. This is accomplished by specifying a scoped variable for the exception using <c:catch>'s var attribute, and then checking its value using <c:if>. If an exception was raised, then assignment to the scoped variable will occur: as the EL expression in Listing 18 suggests, its value will *not* be empty. Since retrieval of the FTP document will have failed, an error message to that effect is displayed.

Listing 18. Example combining <c:import> and <c:catch>

```
<c:catch var="exception">
  <c:import url="ftp://ftp.example.com/package/README"/>
</c:catch>
<c:if test="${not empty exception}">
  Sorry, the remote content is not currently available.
</c:if>
```

The final two (optional) attributes of the <c:import> action are var and scope. The var attribute causes the content fetched from the specified URL to be stored (as a String value) in a scoped variable, rather than included in the current JSP page. The scope attribute controls the scoping of this variable, and defaults to page scope. As we will see in a later article, this ability of <c:import> to store an entire document in a scoped variable is leveraged by the tags in the JSTL xml library.

Note also that (optional) nested <c:param> tags may be used to specify request parameters for the URL being imported. As was the case for <c:param> tags nested with <c:url>, parameter names and values are URL encoded as necessary.

Request redirection

The final core library tag is <c:redirect>. This action is used to send an HTTP redirect response to a user's browser, and is the JSTL equivalent of the sendRedirect() method of javax.servlet.http.HttpServletResponse. The behavior of this tag's url and context

attributes, shown in Listing 19, is identical to the behavior of `<c:import>`'s `url` and `context` attributes, as is the effect of any nested `<c:param>` tags.

Listing 19. Syntax for the <c:redirect>action

```
<c:redirect url="expression" context="expression">
  <c:param name="expression" value="expression"/>
  ...
</c:redirect>
```

Listing 20 shows the `<c:redirect>` action, which replaces the error message in Listing 18 with a redirect to a designated error page. In this example, the `<c:redirect>` tag is used in a similar way as the standard `<jsp:forward>` action. Recall, however, that forwarding through a request dispatcher is implemented on the server side, while redirects are performed by the browser. From the developer's perspective, forwarding is more efficient than redirecting, but the `<c:redirect>` action is a bit more flexible because `<jsp:forward>` can only dispatch to other JSP pages within the current servlet context.

Listing 20. Redirecting in response to an exception

```
<c:catch var="exception">
  <c:import url="ftp://ftp.example.com/package/README"/>
</c:catch>
<c:if test="${not empty exception}">
  <c:redirect url="/errors/remote.jsp"/>
</c:if>
```

The main difference from the user's perspective is that a redirect will update the URL displayed by the browser and will therefore affect the setting of bookmarks. Forwarding, on the other hand, is transparent to the end user. The choice between `<c:redirect>` and `<jsp:forward>`, then, also depends upon the desired user experience.

Summary

The JSTL `core` library contains a variety of general-purpose custom tags that should be of use to a wide spectrum of JSP developers. The URL and exception-handling tags, for example, nicely complement existing JSP functionality, such as the `<jsp:include>` and `<jsp:forward>` actions, the `include` directive, and the `errorpage` attribute of the `page` directive. The iteration and conditional actions enable complex presentation logic to be implemented without the need for scripting elements, particularly in combination with the variable tags (`<c:set>` and `<c:remove>`) and the EL.

Resources

- Part 1 in this series, A JSTL primer: The expression language (*developerWorks*, February 2003), introduces JSTL and details the expression language and several of the tags in `core` library.
- Download the source code for the Weblog example application.
- Sun's product page for the JSP Standard Tag Library is a good starting point to learn more about JSTL.
- The JSTL 1.0 Specification is the final authority on the EL and the four JSTL tag libraries.
- The Jakarta Taglibs project is home to the reference implementation for JSTL 1.0.
- *JSTL in Action* by Shawn Bayern (Manning, 2002) provides excellent coverage of all JSTL features,

having been written by the reference implementation lead.

- Popular Java programming author David Geary has also written a book on JSTL, entitled *Core JSTL*.
- JSPTags.com is a directory of JSP resources, focusing particularly on custom tag libraries.
- Coverage of JSTL is included as part of Sun's Java Web Services Tutorial.
- "Using JSPs and custom tags within VisualAge for Java and WebSphere Studio" (WebSphere Developer Domain) is a WBOnline hands-on workshop demonstrating the use of servlets, JSP pages, and custom tag libraries.
- Learn all about custom tag libraries with Jeff Wilson's excellent article, "Take control of your JSP pages with custom tags" (*developerWorks*, January 2002).
- Noel Bergman's article, "JSP taglibs: Better usability by design" (*developerWorks*, December 2001), shows you how declarative tags will help improve the usability of your JSP pages.
- Find hundreds more Java technology resources on the *developerWorks* Java technology zone.

About the author
Mark Kolb is a Software Engineer working in Austin, Texas. He is a frequent industry speaker on server-side Java topics and the co-author of *Web Development with JavaServer Pages, 2nd Edition*. You can contact Mark at mak@taglib.com.