A JSTL primer: The expression language

PDF e-mail it!

Simplify software maintenance for JSP applications by avoiding scripting elements
Level: Intermediate

Mark A. Kolb (mak@taglib.com)
Software Engineer
February 2003

The JSP Standard Tag Library (JSTL) is a collection of custom tag libraries that implement general-purpose functionality common to Web applications, including iteration and conditionalization, data management formatting, manipulation of XML, and database access. In this first installment of his new series on developerWorks, software engineer Mark Kolb shows you how to use JSTL tags to avoid using scripting elements in your JSP pages. You'll also learn how to simplify software maintenance by removing source code from the presentation layer. Finally, you'll learn about JSTL's simplified expression language, which allows dynamic attribute values to be specified for JSTL actions without having to use a full-blown programming language.

Contents:

Related content:
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

JavaServer Pages (JSP) technology is the standard presentation-layer technology for the J2EE platform. JSP technology provides both scripting elements and actions for performing computations intended to generate page content dynamically. Scripting elements allow program source code to be included in a JSP page for execution when the page is rendered in response to a user request. Actions encapsulate computational operations into tags that more closely resemble the HTML or XML markup that typically comprises the template text of a JSP page. There are only a handful of actions defined as standard by the JSP specification, but starting with JSP 1.1, developers have been able to create their own actions in the form of custom tag libraries.

The JSP Standard Tag Library (JSTL) is a collection of JSP 1.2 custom tag libraries that implement basic functionality common to a wide range of server-side Java applications. By providing standard implementations for typical presentation-layer tasks such as data formatting and iterative or conditional content, JSTL allows JSP authors to focus on application-specific development needs, rather than "reinventing the wheel" for these generic operations.

Of course, you could implement such tasks using the JSP scripting elements: scriptlets, expressions, and declarations. Conditional content, for example, can be implemented using three scriptlets, highlighted in Listing 1. Because they rely on embedding program source code (typically Java code) within the page, though, scripting elements tend to complicate the software maintenance task significantly for JSP pages that use them. The scriptlet example in Listing 1, for instance, is critically dependent upon proper matching of braces. Nesting additional scriptlets within the conditionalized content can wreak havoc if a syntax error is inadvertently introduced, and it can be quite a challenge to make sense of the resulting error message when the page is compiled by the JSP container.

Listing 1. Implementing conditional content through scriptlets

```
<% if (user.getRole() == "member")) { %>
    <p>Welcome, member!</p>
<% } else { %>
    <p>Welcome, guest!</p>
<% } %>
```

Fixing such problems typically requires a fair bit of programming experience. Whereas the markup in a JSP page might typically be developed and maintained by a designer well-versed in page layout and graphic design, the scripting elements in that same page require the intervention of a programmer when problems arise. This shared responsibility for the code within a single file makes developing, debugging, and enhancing such JSP pages a cumbersome task. By packaging common functionality into a standardized set of custom tag libraries, JSTL allows JSP authors to reduce or eliminate the need for scripting elements and avoid the associated maintenance costs.

JSTL 1.0

Released in June 2002, JSTL 1.0 consists of four custom tag libraries (`core`, `format`, `xml`, and `sql`) and a pair of general-purpose tag library validators (`ScriptFreeTLV` and `PermittedTaglibsTLV`). The `core` tag library provides custom actions to manage data through scoped variables, as well as to perform iteration and conditionalization of page content. It also provides tags to generate and operate on URLs. The `format` tag library, as its name suggests, defines actions to format data, specifically numbers and dates. It also provides support for internationalizing JSP pages using localized resource bundles. The `xml` library includes tags to manipulate data represented through XML, while the `sql` library defines actions to query relational databases.

The two JSTL tag library validators allow developers to enforce coding standards within their JSP applications. You can configure the `ScriptFreeTLV` validator to prohibit the use of the various types of JSP scripting elements -- scriptlets, expressions, and declarations -- within a JSP page. Similarly, the `PermittedTaglibsTLV` validator can be used to restrict the set of custom tag libraries (including the JSTL tag libraries) that may be accessed by an application's JSP pages.

While JSTL will eventually be a required component of the J2EE platform, only a small number of application servers include it today. The reference implementation for JSTL 1.0 is available as part of the Apache Software Foundation's Jakarta Taglibs project (see Resources). The custom tag libraries in the reference implementation can be incorporated into any application server supporting the JSP 1.2 and Servlet 2.3 specifications in order to add JSTL support.

Expression language

In JSP 1.2, the attributes of JSP actions are specified using either static character strings or, where permitted, expressions. In Listing 2, for example, static values are specified for the `name` and `property` attributes of this `<jsp:setProperty>` action, while an expression is used to specify its `value` attribute. This action has the effect of assigning the current value of a request parameter to the named bean property. Expressions used in this fashion are called *request-time attribute values* and are the only mechanism built into the JSP specification for specifying attribute values dynamically.

Listing 2. A JSP action incorporating a request-time attribute value

```
<jsp:setProperty name="user" property="timezonePref"
                 value='<%= request.getParameter("timezone") %>'/>
```

Because request-time attribute values are specified using expressions, they are prone to the same software maintenance issues as other scripting elements. For this reason, the JSTL custom tags support an alternate mechanism for specifying dynamic attribute values. Rather than using full-blown JSP expressions, attribute values for JSTL actions can be specified using a simplified *expression language* (EL). The EL provides identifiers, accessors, and operators for retrieving and manipulating data resident in the JSP container. The EL is loosely based on EcmaScript (see Resources) and the XML Path Language (XPath), so its syntax should be familiar to both page designers and programmers. The EL is geared toward looking up objects and their properties, and performing simple operations on them; it is not a programming language, or even a scripting language. When combined with the JSTL tags, however, it enables complex behavior to be represented using a simple and convenient notation. EL expressions are delimited using a leading dollar sign ($) and both leading and trailing braces ({}), as highlighted in Listing 3.

Listing 3. A JSTL action illustrating the EL expression delimiters

```
<c:out value="${user.firstName}"/>
```

In addition, you can combine multiple expressions with static text to construct a dynamic attribute value through string concatenation, as highlighted in Listing 4. Individual expressions are comprised of identifiers, accessors, literals, and operators. Identifiers are used to reference data objects stored in the data center. The EL has 11 reserved identifiers, corresponding to 11 EL implicit objects. All other identifiers are assumed to refer to *scoped variables*. Accessors are used to retrieve the properties of an object or the elements of a collection. Literals represent fixed values -- numbers, character strings, booleans, or nulls. Operators allow data and literals to be combined and compared.

Listing 4. Combining static text and multiple EL expressions to specify a dynamic attribute value

```
<c:out value="Hello ${user.firstName} ${user.lastName}"/>
```

Scoped variables

The JSP API, through the `<jsp:useBean>` action, allows data to be stored and retrieved from four different scopes within the JSP container. JSTL extends this capability by providing additional actions for assigning and removing objects within these scopes. Furthermore, the EL provides built-in support for retrieving these objects as scoped variables. In particular, any identifier appearing in an EL expression that does not correspond to one of the EL's implicit objects is automatically assumed to reference an object stored in one of the four JSP scopes:

- Page scope
- Request scope
- Session scope
- Application scope

As you may recall, objects stored in page scope can only be retrieved during the processing of that page for a specific request. Objects stored in request scope can be retrieved during the processing of all pages taking part in the processing of a request (such as if the processing of a request encounters one or more `<jsp:include>` or `<jsp:forward>` actions). If an object is stored in session scope, it can be retrieved by any pages accessed by a user during a single interactive session with the Web application (that is, until the `HttpSession` object associated with that user's interaction is invalidated). An object stored in application scope is accessible from all pages and for all users, until the Web application itself is unloaded (typically as a result of the JSP container being shut down).

An object is stored in a scope by mapping a character string to the object within the desired scope. You can then retrieve the object from the scope by providing the same character string. The string is looked up in the scope's mapping, and the mapped object is returned. Within the Servlet API, such objects are referred to as *attributes* of the corresponding scope. In the context of the EL, however, the character string associated with an attribute can also be thought of as the name of a variable, which is bound to a particular value by means of the attribute mappings.

In the EL, identifiers not associated with implicit objects are assumed to name objects stored in the four JSP scopes. Any such identifier is first checked against page scope, then request scope, then session scope, and finally application scope, successively testing whether the name of the identifier matches the name of an object stored in that scope. The first such match is returned as the value of the EL identifier. It is in this way that EL identifiers can be thought of as referencing scoped variables.

In more technical terms, identifiers that do not map to implicit objects are evaluated using the `findAttribute()` method of the `PageContext` instance representing the processing of the page on which the expression occurs for the request currently being handled. The name of the identifier is passed as the argument to this method, which searches each of the four scopes in turn for an attribute with the same name. The first match found is returned as the value of the `findAttribute()` method. If no such attribute is located in any of the four scopes, `null` is returned.

Ultimately, then, scoped variables are attributes of the four JSP scopes that have names that can be used as EL identifiers. As long as they are assigned alphanumeric names, scoped variables can be created by any of the mechanisms present in JSP for setting attributes. This includes the built-in `<jsp:useBean>` action, as well as the `setAttribute()` method defined by several of the classes in the Servlet API. In addition, many of the custom tags defined in the four JSTL libraries are themselves capable of setting attribute values for use as scoped variables.

Implicit objects
The identifiers for the 11 EL implicit objects are listed in Table 1. Don't confuse these with the JSP implicit objects (of which there are only nine), as only one object is common to both.

Table 1. The EL implicit objects

Category	**Identifier**	**Description**
JSP	`pageContext`	The `PageContext` instance corresponding to the processing of the current page
Scopes	`pageScope`	A `Map` associating the names and values of page-scoped attributes
	`requestScope`	A `Map` associating the names and values of request-scoped attributes

	`sessionScope`	A `Map` associating the names and values of session-scoped attributes
	`applicationScope`	A `Map` associating the names and values of application-scoped attributes
Request parameters	`param`	A `Map` storing the primary values of the request parameters by name
	`paramValues`	A `Map` storing all values of the request parameters as `String` arrays
Request headers	`header`	A `Map` storing the primary values of the request headers by name
	`headerValues`	A `Map` storing all values of the request headers as `String` arrays
Cookies	`cookie`	A `Map` storing the cookies accompanying the request by name
Initialization parameters	`initParam`	A `Map` storing the context initialization parameters of the Web application by name

While JSP and EL implicit objects have only one object in common (`pageContext`), other JSP implicit objects are still accessible from the EL. The reason is that `pageContext` has properties for accessing all of the other eight JSP implicit objects. Indeed, this is the primary reason for including it among the EL implicit objects.

All of the remaining EL implicit objects are maps, which may be used to look up objects corresponding to a name. The first four maps represent the various attribute scopes discussed previously. They can be used to look up identifiers in specific scopes, rather than relying on the sequential lookup process that the EL uses by default.

The next four maps are for fetching the values of request parameters and headers. Since the HTTP protocol allows both request parameters and headers to be multi-valued, there is a pair of maps for each. The first map in each pair simply returns the primary value for the request parameter or header, typically whichever value happens to have been specified first in the actual request. The second map in each pair allows all of a parameter's or header's values to be retrieved. The keys in these maps are the names of the parameters or headers, while the values are arrays of `String` objects, each element of which is a single parameter or header value.

The cookie implicit object provides access to the cookies set by a request. This object maps the names of all the cookies associated with a request to `Cookie` objects representing the properties of those cookies.

The final EL implicit object, `initParam`, is a map storing the names and values of any context initialization parameters associated with the Web application. Initialization parameters are specified through the `web.xml` deployment descriptor file that appears in the application's `WEB-INF` directory.

Accessors

Since EL identifiers are resolved either as implicit objects or as scoped variables (which are implemented through attributes), they will by necessity evaluate to Java objects. The EL can automatically wrap and unwrap primitives in their corresponding Java classes (for instance, `int` can be coerced into an `Integer` class behind the scenes, and vice versa), but identifiers for the most part will be pointers to full-blown Java objects.

As a result, it's often desirable to access the properties of these objects or, in the case of arrays and collections, their elements. The EL provides two different accessors for just this purpose -- the dot operator (`.`) and the bracket operator (`[]`) -- enabling properties and elements to be operated upon through the EL, as well.

The dot operator is typically used for accessing the properties of an object. In the expression `${user.firstName}`, for example, the dot operator is used to access the property named `firstName` of the object referenced by the `user` identifier. The EL accesses object properties using the Java beans conventions, so a getter for this property (typically a method named `getFirstName()`) must be defined in order for this expression to evaluate correctly. When the property being accessed is itself an object, the dot operator can be applied recursively. For instance, if our hypothetical `user` object has an `address` property that is implemented as a Java object, then the dot operator can also be used to access the properties of this object. The expression `${user.address.city}`, for example, will return the nested `city` property of this address object.

The bracket operator is used to retrieve elements of arrays and collections. In the case of arrays and ordered collections (that is, collections implementing the `java.util.List` interface), the index of the element to be retrieved appears inside the brackets. For example, the expression `${urls[3]}` returns the fourth element of the array or collection referenced by the `urls` identifier (indices are zero-based in the EL, just as in the Java language and JavaScript).

For collections implementing the `java.util.Map` interface, the bracket operator looks up a value stored in the map using the associated key. The key is specified inside the brackets, and the corresponding value is returned as the value of the expression. For example, the expression `${commands["dir"]}` returns the value associated with the `"dir"` key in the `Map` referenced by the `commands` identifier.

In either case, it is permissible for an expression to appear inside the brackets. The result of evaluating the nested expression will serve as the index or key for retrieving the appropriate element of the collection or array. As was true of the dot operator, the bracket operator can be applied recursively. This allows the EL to retrieve elements from multi-dimensional arrays, nested collections, or any combination of the two. Furthermore, the dot operator and the bracket operator are interoperable. For example, if the elements of an array are themselves objects, the bracket operator can be used to retrieve an element of the array and be combined with the dot operator to retrieve one of the element's properties (for instance, `${urls[3].protocol}`).

Given the EL's role as a simplified language for specifying dynamic attribute values, one interesting feature of the EL accessors is that, unlike the Java language's accessors, they do not throw exceptions when applied to `null`. If the object to which an EL accessor is applied (for instance, the `foo` identifier in both `${foo.bar}` and `${foo["bar"]}`) is `null`, then the result of applying the accessor will also be `null`. This turns out to be rather helpful behavior under most circumstances, as you'll see shortly.

Finally, the dot operator and the bracket operator are somewhat interchangeable. For example, `${user["firstName"]}` could also be used to retrieve the `firstName` property of the `user` object, just as `${commands.dir}` could be used to fetch the value associated with the `"dir"` key in the `commands` map.

Operators

Using identifiers and accessors, then, the EL is able to traverse object hierarchies containing either application data (exposed through scoped variables) or information about the environment (through the EL implicit objects). Simply accessing such data, however, is often inadequate for implementing the presentation logic needed by many JSP applications.

To this end, the EL also includes several operators to manipulate and compare data accessed by EL expressions. These operators are summarized in Table 2.

Table 2. The EL operators

Category	Operators
Arithmetic	`+`, `-`, `*`, `/` (or `div`), `%` (or `mod`)
Relational	`==` (or `eq`), `!=` (or `ne`), `<` (or `lt`), `>` (or `gt`), `<=` (or `le`), `>=` (or `ge`)
Logical	`&&` (or `and`), `||` (or `or`), `!` (or `not`)
Validation	`empty`

The arithmetic operators support addition, subtraction, multiplication, and division of numeric values. A remainder operator is also provided. Note that the division and remainder operators have alternate, non-symbolic names (in order to be consistent with XPath). An example expression demonstrating the use of the arithmetic operators is shown in Listing 5. The result of applying an arithmetic operator to a pair of EL expressions is the result of applying that operator to the numeric values returned by those expressions.

Listing 5. An EL expression utilizing arithmetic operators

```
${item.price * (1 + taxRate[user.address.zipcode])}
```

The relational operators allow you to compare either numeric or textual data. The result of the comparison is returned as a boolean value. The logical operators allow boolean values to be combined, returning a new boolean value. The EL logical operators can therefore be applied to the results of nested relational or logical operators, as demonstrated in Listing 6.

Listing 6. An EL expression utilizing relational and logical operators

```
${(x >= min) && (x <= max)}
```

The final EL operator is `empty`, which is particularly useful for validating data. The `empty` operator takes a single expression as its argument (that is, `${empty input}`), and returns a boolean value indicating whether or not the expression evaluates to an "empty" value. Expressions that evaluate to `null` are considered empty, as are collections or arrays with no elements. The `empty` operator will also return `true` if its argument evaluates to a `String` of zero length.

Operator precedence for the EL operators is shown in Table 3. As suggested in Listings 5 and 6, parentheses may be used to group expressions and override the normal precedence rules.

Table 3. EL operator precedence (top to bottom, left to right)

`[]`, `.`
`()`
unary `-`, `not`, `!`, `empty`
`*`, `/`, `div`, `%`, `mod`
`+`, binary `-`
() <, >, <=, >=, `lt`, `gt`, `le`, `ge`
`==`, `!=`, `eq`, `ne`

&&, `and`
`||`, `or`

Literals

Numbers, character strings, booleans, and `null`s can all be specified as literal values in EL expressions. Character strings are delimited by either single or double quotes. Boolean values are designated by `true` and `false`.

Taglib directives

As we discussed earlier, JSTL 1.0 includes four custom tag libraries. To illustrate the interaction of JSTL tags with the expression language, we will look at several of the tags from the JSTL `core` library. As is true with any JSP custom tag library, a `taglib` directive must be included in any page that you want to be able to use this library's tags. The directive for this specific library appears in Listing 7.

Listing 7. The taglib directive for the EL version of the JSTL core library

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

Actually, there are two `taglib` directives that correspond to the JSTL `core` library because in JSTL 1.0 the EL is optional. All four of the JSTL 1.0 custom tag libraries have alternate versions that use JSP expressions rather than the EL for specifying dynamic attribute values. Because these alternate libraries rely on JSP's more traditional request-time attribute values, they are referred to as the *RT* libraries, whereas those using the expression language are referred to as the *EL* libraries. Developers distinguish between the two versions of each library using alternate `taglib` directives. The directive for using the RT version of the core library is shown in Listing 8. Given our current focus on the EL, however, it is the first of these directives that is needed.

Listing 8. The taglib directive for the RT version of the JSTL core library

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

Variable tags

The first JSTL custom tag we will consider is the `<c:set>` action. As already indicated, scoped variables play a key role in JSTL, and the `<c:set>` action provides a tag-based mechanism for creating and setting scoped variables. The syntax for this action is shown in Listing 9, where the `var` attribute specifies the name of the scoped variable, the `scope` attribute indicates which scope the variable resides in, and the `value` attribute specifies the value to be bound to the variable. If the specified variable already exists, it will simply be assigned the indicated value. If not, a new scoped variable is created and initialized to that value.

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

```
<c:set var="name" scope="scope" value="expression"/>
```

The `scope` attribute is optional and defaults to `page`.

Two examples of the `<c:set>` are presented in Listing 10. In the first example, a session-scoped variable is set to a `String` value. In the second, an expression is used to set a numeric value: a page-scoped variable named `square` is assigned the result of multiplying the value of a request parameter named `x` by itself.

Listing 10. Examples of the <c:set> action

```
<c:set var="timezone" scope="session" value="CST"/>
<c:set var="square" value="${param['x'] * param['x']}"/>
```

Rather than using an attribute, you can also specify the value for the scoped variable as the body content of the `<c:set>` action. Using this approach, you could rewrite the first example in Listing 10 as shown in Listing 11. Furthermore, as we will see momentarily, it's acceptable for the body content of the `<c:set>` tag to employ custom tags itself. All content generated within the body of `<c:set>` will be assigned to the specified variable as a `String` value.

Listing 11. Specifying the value for the <c:set> action through body content

```
<c:set var="timezone" scope="session">CST</c:set>
```

The JSTL core library includes a second tag for managing scoped variables, `<c:remove>`. As its name suggests, the `<c:remove>` action is used to delete a scoped variable, and takes two attributes. The `var` attribute names the variable to be removed, and the optional `scope` attribute indicates the scope from which it should be removed and defaults to `page`, as shown in Listing 12.

Listing 12. An example of the <c:remove> action

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

Output
While the `<c:set>` action allows the result of an expression to be assigned to a scoped variable, a developer will often want to simply display the value of an expression, rather than store it. This is the role of JSTL's `<c:out>` custom tag, the syntax of which appears in Listing 13. This tag evaluates the expression specified by its `value` attribute, then prints the result. If the optional `default` attribute is specified, the `<c:out>` action will instead print its value if the `value` attribute's expression evaluates either to `null` or an empty `String`.

Listing 13. Syntax for the <c:out> action

```
<c:out value="expression" default="expression" escapeXml="boolean"/>
```

The `escapeXml` attribute is also optional. It controls whether or not characters such as "<", ">", and "&", which have special meanings in both HTML and XML, should be escaped when output by the `<c:out>` tag. If `escapeXml` is set to true, then these characters will automatically be translated into the corresponding XML entities (`&lt;`, `&gt;`, and `&amp;`, respectively, for the characters mentioned here).

For instance, suppose there is a session-scoped variable named `user` that is an instance of a class that defines two properties for users, `username` and `company`. This object is automatically assigned to the session whenever a user accesses the site, but the two properties are not set until the user actually logs in. Given this scenario, consider the JSP fragment shown in Listing 14. Once the user has logged in, this fragment will display the word "Hello," followed by his or her username and an exclamation point. Before the user has logged in, however, the content generated by this fragment will instead be the phrase, "Hello Guest!" In this case, because the `username` property has yet to be initialized, the `<c:out>` tag will instead print out the value of its `default` attribute (that is, the character string, "Guest").

Listing 14. An example of the <c:out> action with default content

```
Hello <c:out value="${user.username}" default=="Guest"/>!
```

Next, consider Listing 15, which uses the `<c:out>` tag's `escapeXml` attribute. If the `company` property has in this case been set to the Java `String` value `"Flynn & Sons"`, then the content generated by this action will, in fact, be `Flynn & Sons`. If this action is part of a JSP page generating HTML or XML content, then the ampersand in the middle of this string of characters may end up being interpreted as an HTML or XML control character and interrupt the rendering or parsing of this content. If the value of the `escapeXml` attribute is instead set to `true`, however, the generated content will instead be `Flynn &amp; Sons`. A browser or parser encountering this content should have no problems with its interpretation. Given that HTML and XML are the most common content types in JSP applications, it should come as little surprise that the default value for the `escapeXml` attribute is `true`.

Listing 15. An example of the <c:out> action with escaping disabled

```
<c:out value="${user.company}" escapeXml=="false"/>
```

Setting variables with default values

In addition to simplifying the display of dynamic data, the ability of `<c:out>` to specify a default value is also useful when setting variable values through `<c:set>`. As highlighted in [Listing 11](), the value to be assigned to a scoped variable can be specified as the body content of the `<c:set>` tag, as well as through its value attribute. By nesting a `<c:out>` action in the body content of a `<c:set>` tag, the variable assignment can leverage its default value capability.

This approach is illustrated in Listing 16. The behavior of the outer `<c:set>` tag is straightforward enough: it sets the value of the session-scope `timezone` variable based on its body content. In this case, however, that body content is generated through a `<c:out>` action. The value attribute of this nested action is the expression `${cookie['tzPref'].value}`, which attempts to return the value of a cookie named `tzPref` by means of the `cookie` implicit object. (The `cookie` implicit object maps cookie names to corresponding `Cookie` instances, which means you must use the dot operator to retrieve the actual data stored in the cookie through the object's `value` property.)

Listing 16. Combining <c:set> and <c:out> to provide default variable values

```
<c:set var="timezone" scope=="session">
   <c:out value="${cookie['tzPref'].value}" default=="CST"/>
</c:set>
```

Consider the case, however, in which this is the user's first experience with the Web application using this code. As a result, there is no cookie named `tzPref` provided in the request. This means the lookup using the implicit object will return `null`, in which case the expression as a whole will return `null`. Since the result of evaluating its `value` attribute is `null`, the `<c:out>` tag will instead output the result of evaluating its `default` attribute. Here, this is the character string `CST`. The net effect, then, is that the `timezone` scoped variable will be set to the time zone stored in the user's `tzPref` cookie or, if none is present, use a default time zone of `CST`.

The EL and JSP 2.0
For now, the expression language is only available for specifying dynamic attribute values in JSTL custom tags. An extension of the JSTL 1.0 expression language has been proposed, however, for inclusion in the JSP 2.0 specification, now undergoing final review. This extension will allow developers to leverage the EL with their own custom tags. Page authors will be able to use EL expressions anywhere they are currently allowed to use JSP expressions, such as to insert dynamic values into template text:

```
<p>Your preferred time zone is ${timezone}.</p>
```

This JSP 2.0 feature will, like JSTL itself, enable page authors to further reduce their dependence on JSP scripting elements, leading to improved maintainability for JSP applications.

Summary

The EL, in concert with the actions provided by the four JSTL custom tag libraries, allows page authors to implement presentation-layer logic without resorting to scripting elements. Contrast, for example, the JSP code in Listing 1 at the beginning of this article with the same functionality as implemented through the JSTL highlighted in Listing 17. (The remaining tags in the JSTL `core` library, including `<c:choose>` and its children will be covered in the next article in this series.) Although it is still clear that conditional logic is being performed, the JSTL version has no Java language source code in it, and the relationships between the tags -- particularly with respect to nesting requirements -- should be familiar to anyone comfortable with HTML syntax.

Listing 17. Implementing conditional content via JSTL

```
<c:choose><c:when test="${user.role == 'member'}">
    <p>Welcome, member!</p>
  </c:when><c:otherwise>
    <p>Welcome, guest!</p>
  </c:otherwise></c:choose>
```

By providing standard implementations of functionality common to most Web applications, JSTL helps accelerate the development cycle. In concert with the EL, JSTL can remove the need for program code in the presentation layer, greatly simplifying the maintenance of JSP applications.

Resources

- Sun's JSP Standard Tag Library page is a good starting point for learning 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 Publications Co., 2002) provides excellent coverage of all JSTL features, having been written by the reference implementation lead.
- David Geary, a popular author on Java technology, has also written a book on JSTL, entitled *Core JSTL*.

- JSPTags.com is a directory of JSP technology 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, JSPs 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.
- For more details on EcmaScript, see Sing Li's "Quick-and-dirty Java programming" (*developerWorks*, July 2001).
- 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*. Mark can be contacted at mak@taglib.com.

PDF e-mail it!

What do you think of this document?

○ Killer! (5) ○ Good stuff (4) ○ So-so; not bad (3) ○ Needs work (2) ○ Lame! (1)

Comments?

Submit feedback