A JSTL primer: Presentation is everything

Formatting and internationalization through custom tags

Level: Intermediate

Mark A. Kolb (mak@taglib.com)
Software Engineer
April 15, 2003

Localizing content for visitors is a critical element for developers who want their Web applications to have global impact. Internationalization features have been part of the Java programming language since JDK 1.1, and the JSP Standard Tag Library (JSTL) `fmt` library provides convenient access to all of these features through a focused set of custom tags. Mark Kolb returns to the topic of JSTL in this third installment of his four-part series with a look at the `fmt` tags for formatting and internationalizing data.

Related content:
Java internationalization basics
Take control of your JSP pages with custom tags
JSP taglibs: Better usability by design
Subscribe to the developerWorks newsletter
developerWorks Toolbox subscription

Also in the Java zone:
Tutorials
Tools and products
Code and components
Articles

In the previous articles in this series, we discussed the JSTL and its expression language (EL). We also examined the custom tags defined by the `core` library. Specifically, in "The expression language" we said the EL provides a simplified language for accessing and manipulating data within a JSP application and making that data available to JSTL custom tags as dynamic attribute values. The `core` library, which includes custom tags for managing scoped variables, displaying EL values, implementing iterative and conditional content, and interacting with URLs, was the topic of "Getting down to the core."

The next JSTL library we'll discuss is the `fmt` library. The custom tags in the `fmt` library support localizing textual content through resource bundles and the display and parsing of numbers and dates. These tags leverage the Java language's internationalization API as realized in the `java.util` and `java.text` packages, so if you're already familiar with classes such as `ResourceBundle`, `Locale`, `MessageFormat`, and `DateFormat`, you'll find much to appreciate in the `fmt` library. If not, the `fmt` library's tags encapsulate the internationalization API in an intuitive manner that makes it easy to incorporate localization features into your JSP applications.

Localization

Within the Java language internationalization API, there are two major factors that influence how data is localized. One is the user's *locale*, the other is the user's *time zone*. A locale represents the linguistic conventions of a particular region or culture, including the formatting of dates, numbers, and currency amounts. A locale will always have an associated language, which in many cases is a dialect of a language shared by multiple locales. For example, there are different locales for the American, British, Australian, and Canadian dialects of the English language, and for the French, Belgian, Swiss, and Canadian dialects of the French language.

Time zone is the second factor in the localization of data, simply because some locales span very large geographic regions. When you display time-of-day information for a continent-spanning locale such as Australian English, customizing the data for a user's time zone is just as important as formatting it properly.

This begs the question, though: How does an application determine a user's locale and time zone? In the case of

Java applications, the JVM is able to set a default locale and time zone by interacting with the local operating system. While this approach works fine for desktop applications, it's not really appropriate for a server-side Java application that is handling requests from locations that may be halfway around the world from the server the application resides on.

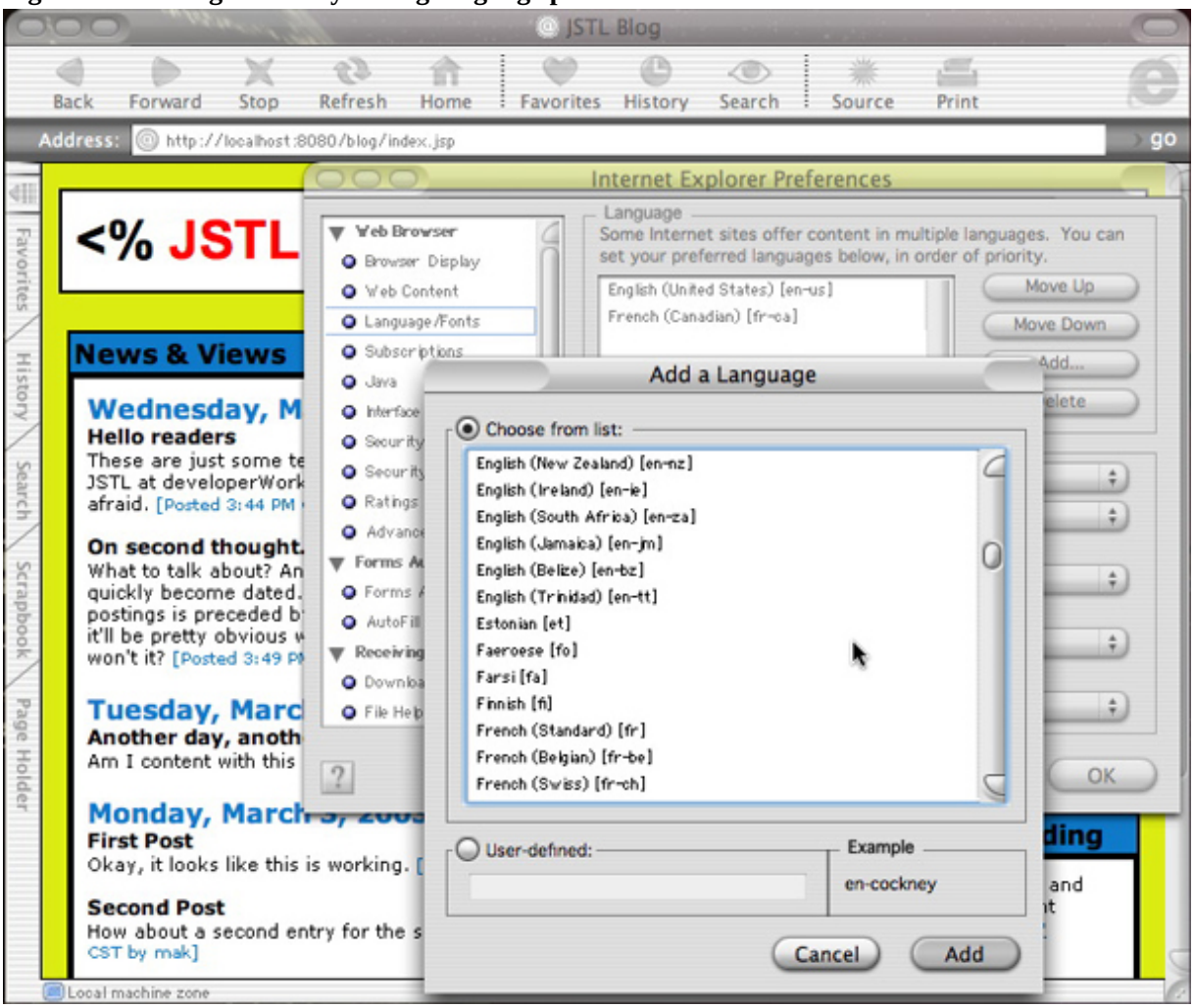
Don't miss the rest of this series

Part 1, "The expression language" (February 2003)

Part 2, "Getting down to the core" (March 2003)

Fortunately, the HTTP protocol includes provisions for relaying localization information from the browser to the server by means of the `Accept-Language` request header. Many Web browsers allow users to customize their language preferences, as illustrated in Figure 1. Typically, those browsers that don't provide explicit settings for one or more preferred locales will instead interrogate the operating system to determine what value (or values) to send in the `Accept-Language` header. The servlet specification automatically takes advantage of this feature of the HTTP protocol through the `getLocale()` and `getLocales()` methods of the `javax.servlet.ServletRequest` class. The custom tags in the JSTL `fmt` library in turn leverage these methods to automatically determine a user's locale and adjust their output accordingly.

Figure 1. Selecting locales by setting language preferences for a browser



Unfortunately, however, no standard HTTP request headers exist for transmitting a user's time zone from the browser to the server. As a result, users who want their Web applications to localize time data will need to implement their own mechanisms for determining and keeping track of user-specific time zones. For example, the Weblog application introduced in Part 2 of this series, "Getting down to the core," includes a form that stores a user's time zone preference in a cookie.

The fmt library

The custom tags in the JSTL `fmt` library fall into four major groupings. The first set allows you to set the

localization context in which the other tags operate. In other words, this group of tags allows the page author to explicitly set the locale and time zone that the other `fmt` tags will use when formatting data. The second and third sets of tags support the formatting and parsing of dates and numbers, respectively. The final group of tags is focused on localizing text messages.

Now that we've set the stage, let's focus our attention on each of these four sets of tags in turn, and demonstrate their use.

Localization context tags

As we already discussed, the locale used by the JSTL tags when formatting data is normally determined by examining the `Accept-Language` header sent by a user's browser as part of each HTTP request. If no such header is present, then JSTL provides a set of JSP configuration variables you can set to specify a default locale. If these configuration variables have not been set, then the JVM's default locale is used, which is obtained from the operating system the JSP container is running on.

The `fmt` library provides its own custom tag for overriding this process of determining a user's locale: `<fmt:setLocale>`. As the following snippet shows, the `<fmt:setLocale>` action supports three attributes:

```
<fmt:setLocale value="expression"
    scope="scope" variant="expression"/>
```

Only one of the attributes, the `value` attribute, is required. The value of this attribute should be either a string naming a locale, or an instance of the `java.util.Locale` class. A locale name is constructed from a two-letter lowercase ISO country code, optionally followed by an underscore or hyphen and a two-letter uppercase ISO language code.

For example, `en` is the language code for English and `US` is the country code for the United States, so `en_US` (or `en-US`) would be the locale name for American English. Similarly, `fr` is the language code for French and `CA` is the country code for Canada, so `fr_CA` (or `fr_CA`) is the locale name for Canadian French. (See Resources for links to all the valid ISO language and country codes.) Of course, because the country codes are optional, `en` and `fr` are themselves valid locale names and would be appropriate for applications that do not distinguish between specific dialects of the corresponding languages.

The optional `scope` attribute of `<fmt:setLocale>` is used to specify the scope of the locale. The `page` scope indicates that the setting is only applicable over the current page, while the `request` scope applies it to all JSP pages accessed during a request. If the `scope` attribute is set to `session`, then the specified locale is used for all JSP pages accessed over the course of the user's session. A value of `application` indicates that the locale is applied to all requests for all of the Web application's JSP pages and for all users of that application.

The `variant` attribute (also optional) allows you to further customize the locale to a specific Web browser platform or vendor. For example, `MAC` and `WIN` are variant names for the Apple Macintosh and Microsoft Windows platforms, respectively.

The following snippet shows how the `<fmt:setLocale>` tag is used to explicitly specify the locale setting for a user's session:

```
<fmt:setLocale value="fr_CA" scope="session"/>
```

After the JSP container processes this JSP fragment, the language preferences as specified in the user's browser settings will be ignored.

The `<fmt:setTimeZone>` action, like `<fmt:setLocale>`, may be used to set the default time zone value for use by the other `fmt` custom tags. Its syntax is shown here:

```
<fmt:setTimeZone value="expression"
    var="name" scope="scope"/>
```

Like the <fmt:setLocale>, only the value attribute is required, though in this case it should be either the name of a time zone or an instance of the java.util.TimeZone class.

Unfortunately, there aren't any widely accepted standards for naming time zones. Time zone names you may use for the value attribute of the <fmt:setTimezone> tag are therefore Java platform-specific. You can retrieve a list of valid time zone names by calling the getAvailableIDs() static method of the java.util.TimeZone class. Examples include US/Eastern, GMT+8, and Pacific/Guam.

As was the case for <fmt:setLocale>, you can use the optional scope attribute to indicate the scope of the time zone setting. The code below shows the use of <fmt:setTimeZone> to specify the time zone to be applied to an individual user's session:

```
<fmt:setTimeZone value="Australia/Brisbane" scope="session"/>
```

You can also use the <fmt:setTimeZone> action to store the value of a TimeZone instance in a scoped variable. In this case, you use the var attribute to name the scoped variable and the scope attribute specifies the variable's scope (just as these two attributes are used in the <c:set> and <c:if> actions, for example). Note that when you use the <fmt:setTimeZone> action in this manner, its only side effect is setting the specified variable. When the var attribute is specified, no change is made to the JSP environment with respect to what time zone is used by any other JSTL tags.

The final tag in this group is the <fmt:timeZone> action:

```
<fmt:timeZone value="expression">
  body content
</fmt:timeZone>
```

Like <fmt:setTimeZone>, you use this tag to specify the time zone to be used by other JSTL tags. The scope of the <fmt:timeZone> action, however, is limited to its body content. Within the body of an <fmt:timeZone> tag, the time zone specified by the tag's value attribute overrides any other time zone setting present in the JSP environment.

As was the case for <fmt:setTimeZone>, the value attribute of the <fmt:timeZone> tag should be either the name of a time zone or an instance of java.util.TimeZone. An example of how to use <fmt:timeZone> appears later in Listing 1.

Date tags

The fmt library includes two tags for interacting with dates and time: <fmt:formatDate> and <fmt:parseDate>. As their names suggest, <fmt:formatDate> is used to format and display dates and times (data *output*), while <fmt:parseDate> is used to parse date and time values (data *input*).

The syntax for <fmt:formatDate> is shown here:

```
<fmt:formatDate value="expression"
    timeZone="expression"
    type="field" dateStyle="style"
    timeStyle="style"
    pattern="expression"
    var="name" scope="scope"/>
```

Only the value attribute is required. Its value should be an instance of the java.util.Date class, specifying the date and/or time data to be formatted and displayed.

The optional timeZone attribute indicates the time zone in which the date and/or time are to be displayed. If no timeZone attribute is specified explicitly, then the time zone specified by any surrounding <fmt:timeZone>

tag is used. If the `<fmt:formatDate>` action is not enclosed in the body of an `<fmt:timeZone>` tag, then the time zone set by any applicable `<fmt:setTimeZone>` action is used. If there is no relevant `<fmt:setTimeZone>` action, then the JVM's default time zone is used (that is, the time zone setting specified for the local operating system).

The `type` attribute indicates which fields of the specified `Date` instance are to be displayed, and should be either `time`, `date`, or `both`. The default value for this attribute is `date`, so if no `type` attribute is present, the `<fmt:formatDate>` tag -- true to its name -- will only display the date information associated with the `Date` instance, specified using the tag's `value` attribute.

The `dateStyle` and `timeStyle` attributes indicate how the date and time information should be formatted, respectively. Valid styles are `default`, `short`, `medium`, `long`, and `full`. The default value is, naturally, `default`, indicating that a locale-specific style should be used. The semantics for the other four style values are as defined by the `java.text.DateFormat` class.

Rather than relying on the built-in styles, you can use the `pattern` attribute to specify a custom style. When present, the value of the pattern attribute should be a pattern string following the conventions of the `java.text.SimpleDateFormat` class. These patterns are based on replacing designated characters within the pattern with corresponding date and time fields. For example, the pattern `MM/dd/yyyy` indicates that two-digit month and date values and a four-digit year value should be displayed, separated by forward slashes.

If the `var` attribute is specified, then a `String` value containing the formatted date is assigned to the named variable. Otherwise, the `<fmt:formatDate>` tag will write out the formatting results. When the `var` attribute is present, the `scope` attribute specifies the scope of the resulting variable.

Listing 1 (which is an extension of Listing 8 from Part 2 of this series) includes two uses of the `<fmt:formatDate>` tag. In the first usage, `<fmt:formatDate>` is used to display only the date portion of the creation timestamp for the first weblog entry. In addition, a value of `full` is specified for the `dateStyle` attribute, so that all date fields will be displayed in a locale-specific format.

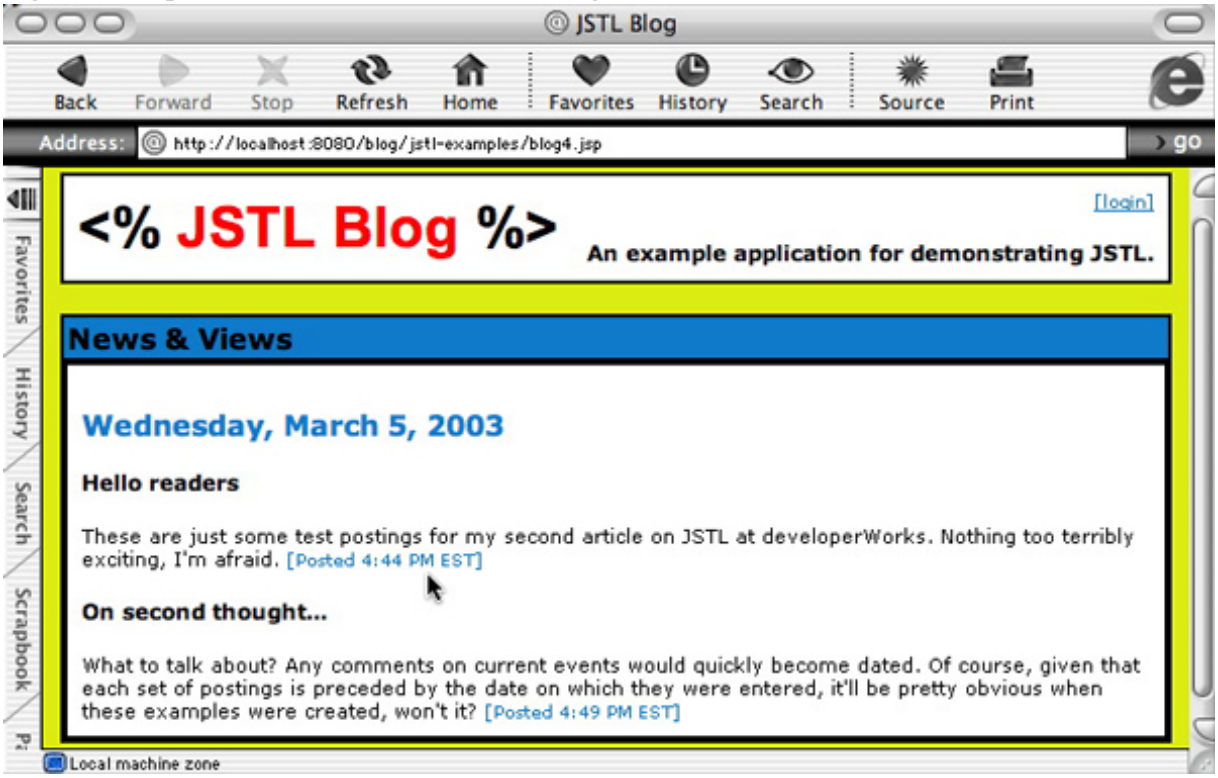
Listing 1. Using the <fmt:formatDate> tag to display date and time values

```
<table>
<fmt:timeZone value="US/Eastern">
<c:forEach items="${entryList}" var="blogEntry" varStatus="status">
<c:if test="${status.first}">
        <tr><td align="left" class="blogDate">
          <fmt:formatDate value=
              "${blogEntry.created}" dateStyle="full"/>
        </td></tr>
      </c:if>
      <tr><td align="left" class="blogTitle">
        <c:out value="${blogEntry.title}" escapeXml="false"/>
      </td></tr>
    <tr><td align="left" class="blogText">
      <c:out value="${blogEntry.text}" escapeXml="false"/>
      <font class="blogPosted">
        [Posted <fmt:formatDate value="${blogEntry.created}"
                                pattern="h:mm a zz"/>]
      </font>
    </td></tr>
  </c:forEach>
  </fmt:timeZone>
</table>
```

Within the body of the `<c:forEach>` loop, a second `<fmt:formatDate>` action is used to display only the time portion of each entry's creation date. In this case, the `pattern` attribute is used to control the formatting of the time value, specifying a single-digit hour display (when possible), a twelve-hour clock, and output of an

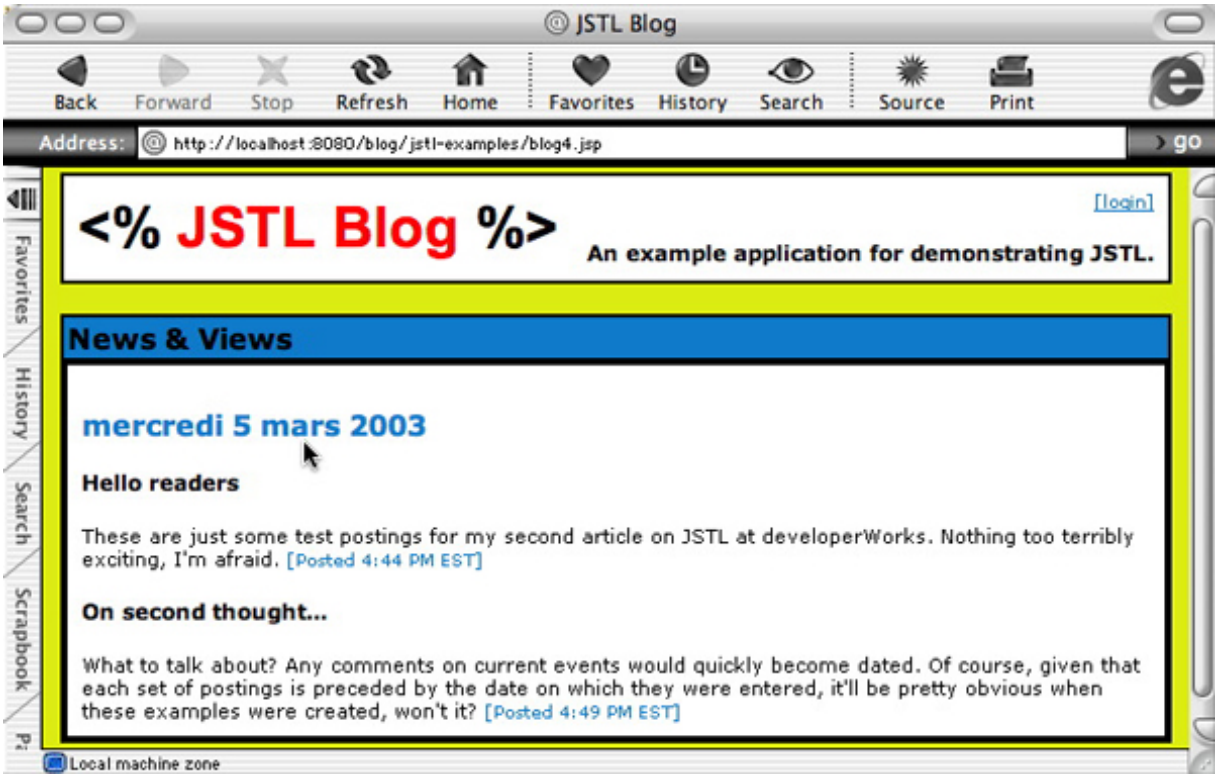
abbreviated time zone. The output is shown in Figure 2:

Figure 2. Output of the en_US locale from Listing 1



To be more precise, the output shown in Figure 2 results when the user's browser settings indicate a preference for the English language. Because `<fmt:formatDate>` is sensitive to a user's locale, however, a change in browser preferences will cause different content to be generated. For example, when a French-language locale is given precedence, the results will instead be like those shown in Figure 3:

Figure 3. Output of the fr_CA locale from Listing 1



Whereas `<fmt:formatDate>` generates a localized character-string representation of a `java.util.Date` instance, the `<fmt:parseDate>` action performs the inverse operation: Given a character string representing a date and/or time, it generates the corresponding `Date` object. There are two forms of the `<fmt:parseDate>` action, as shown here:

```
<fmt:parseDate value="expression"
    type="field" dateStyle="style" timeStyle="style"
    pattern="expression"
    timeZone="expression" parseLocale="expression"
    var="name" scope="scope"/>

<fmt:parseDate
    type="field" dateStyle="style" timeStyle="style"
    pattern="expression"
    timeZone="expression" parseLocale="expression"
    var="name" scope="scope">
  body content
</fmt:parseDate>
```

For the first form, only the `value` attribute is required, and its value should be a character string specifying a date, time, or combination of the two. For the second form, there are no required attributes, and the character string representing the value to be parsed is specified as the required body content of the `<fmt:parseDate>` tag.

The `type`, `dateStyle`, `timeStyle`, `pattern`, and `timeZone` attributes play the same role for `<fmt:parseDate>` as they do for `<fmt:formatDate>`, except that they control the parsing of a date value, rather than its display. The `parseLocale` attribute is used to specify a locale that the tag's value is to be parsed against, and should be either the name of a locale or an instance of the `Locale` class.

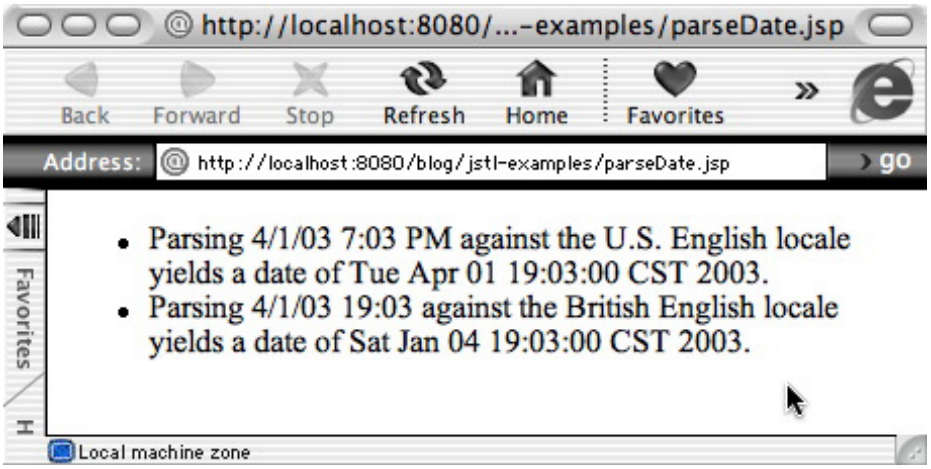
The `var` and `scope` attributes are used to specify a scoped variable that -- as a result of `<fmt:parseDate>` -- the `Date` object is assigned to. If the `var` attribute is not present, the result is written to the JSP page using the `Date` class's `toString()` method. Listing 2 shows an example of the `<fmt:parseDate>` action:

Listing 2. Using the <fmt:parseDate> tag to parse dates and times

```
<c:set var="usDateString">4/1/03 7:03 PM</c:set>
<fmt:parseDate value="${usDateString}" parseLocale="en_US"
               type="both" dateStyle="short" timeStyle="short"
               var="usDate"/>

<c:set var="gbDateString">4/1/03 19:03</c:set>
<fmt:parseDate value="${gbDateString}" parseLocale="en_GB"
               type="both" dateStyle="short" timeStyle="short"
               var="gbDate"/>

<ul>
<li> Parsing <c:out value="${usDateString}"/> against the
U.S. English
     locale yields a date of <c:out value="${usDate}"/>.</li>

<li> Parsing <c:out value="${gbDateString}"/> against the
British English
     locale yields a date of <c:out value="${gbDate}"/>.</li>
</ul>
```

The output of Listing 2 is shown in Figure 4.

Figure 4. Output of Listing 2



It is important to note that the parsing performed by `<fmt:parseDate>` is not at all lenient. As Listing 2 suggests, the value to be parsed must strictly adhere to the specified (locale-specific) styles or pattern. This is, of course, rather limiting. On the other hand, the parsing of data is not a task well-suited to the presentation layer. For production code, validating and transforming textual input is better handled by back-end code (a servlet, for example), rather than by means of JSP custom tags.

Number tags

Just as the `<fmt:formatDate>` and `<fmt:parseDate>` tags are used for formatting and parsing dates, the `<fmt:formatNumber>` and `<fmt:parseNumber>` tags perform similar functions for numeric data.

The `<fmt:formatNumber>` tag is used to display numeric data, including currencies and percentages, in a locale-specific manner. The `<fmt:formatNumber>` action determines from the locale, for example, whether to use a period or a comma for delimiting the integer and decimal portions of a number. Here is its syntax:

```
<fmt:formatNumber value="expression"
    type="type" pattern="expression"
    currencyCode="expression" currencySymbol="expression"
    maxIntegerDigits="expression" minIntegerDigits="expression"
    maxFractionDigits="expression" minFractionDigits="expression"
    groupingUsed="expression"
    var="name" scope="scope"/>
```

As was the case for `<fmt:formatDate>`, only the `value` attribute is required. It is used to specify the numeric value that is to be formatted. The `var` and `scope` attributes also play the same role for the `<fmt:formatNumber>` action as they do for `<fmt:formatDate>`.

The value of the `type` attribute should be either `number`, `currency`, or `percentage`, and indicates what type of numeric value is being formatted. The default value for this attribute is `number`. The `pattern` attribute takes precedence over the `type` attribute and allows more precise formatting of numeric values following the pattern conventions of the `java.text.DecimalFormat` class.

When the `type` attribute has a value of `currency`, the `currencyCode` attribute can be used to explicitly specify the currency for the numerical value being displayed. As with language and country codes, currency codes are governed by an ISO standard. (See Resources for links to all the valid ISO currency codes.) This code is used to determine the currency symbol to display as part of the formatted value.

Alternatively, you can use the `currencySymbol` attribute to explicitly specify the currency symbol. Note that as of JDK 1.4 and the associated introduction of the `java.util.Currency` class, the `currencyCode` attribute of the `<fmt:formatNumber>` action takes precedence over the `currencySymbol` attribute. For earlier versions of the JDK, however, the `currencySymbol` attribute takes precedence.

The `maxIntegerDigits`, `minIntegerDigits`, `maxFractionDigits`, and `minFractionDigits` attributes are used to control the number of significant digits displayed before and after the decimal point. These attributes require integer values.

The `groupingUsed` attribute takes a Boolean value and controls whether digits before the decimal point are grouped. For example, in English-language locales, large numbers have their digits grouped by threes, with each set of three delimited by a comma. Other locales delimit such groupings with a period or a space. The default value for this attribute is `true`.

Listing 3 shows a simple currency example, which is itself an extension of Listing 1. In this case, neither the `currencyCode` nor the `currencySymbol` attributes are specified. The currency is instead determined from the locale setting.

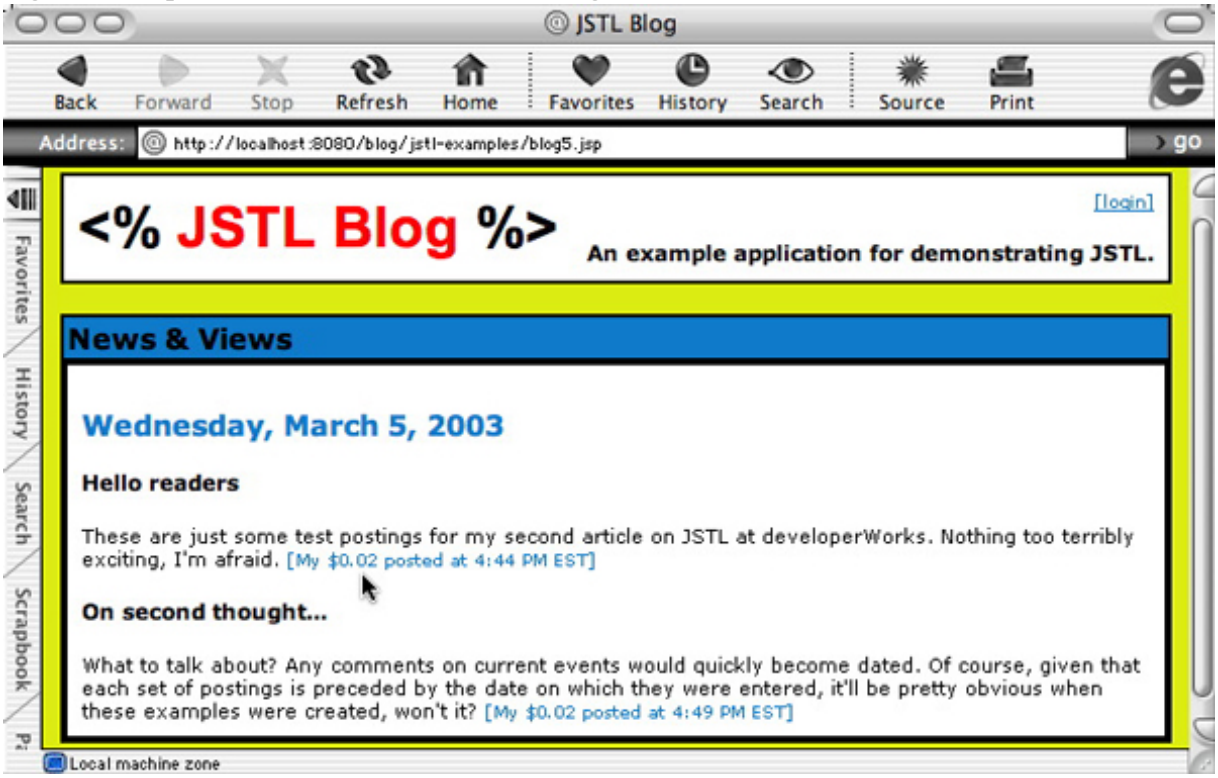
Listing 3. Using the <fmt:formatNumber> tag to display currency values

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

```
      <c:out value="${blogEntry.text}" escapeXml="false"/>
      <font class="blogPosted">
        [My <fmt:formatNumber value="0.02" type="currency"/>
         posted at <fmt:formatDate value="${blogEntry.created}"
                                   pattern="h:mm a zz"/>]
      </font>
    </td></tr>
  </c:forEach>
  </fmt:timeZone>
</table>
```

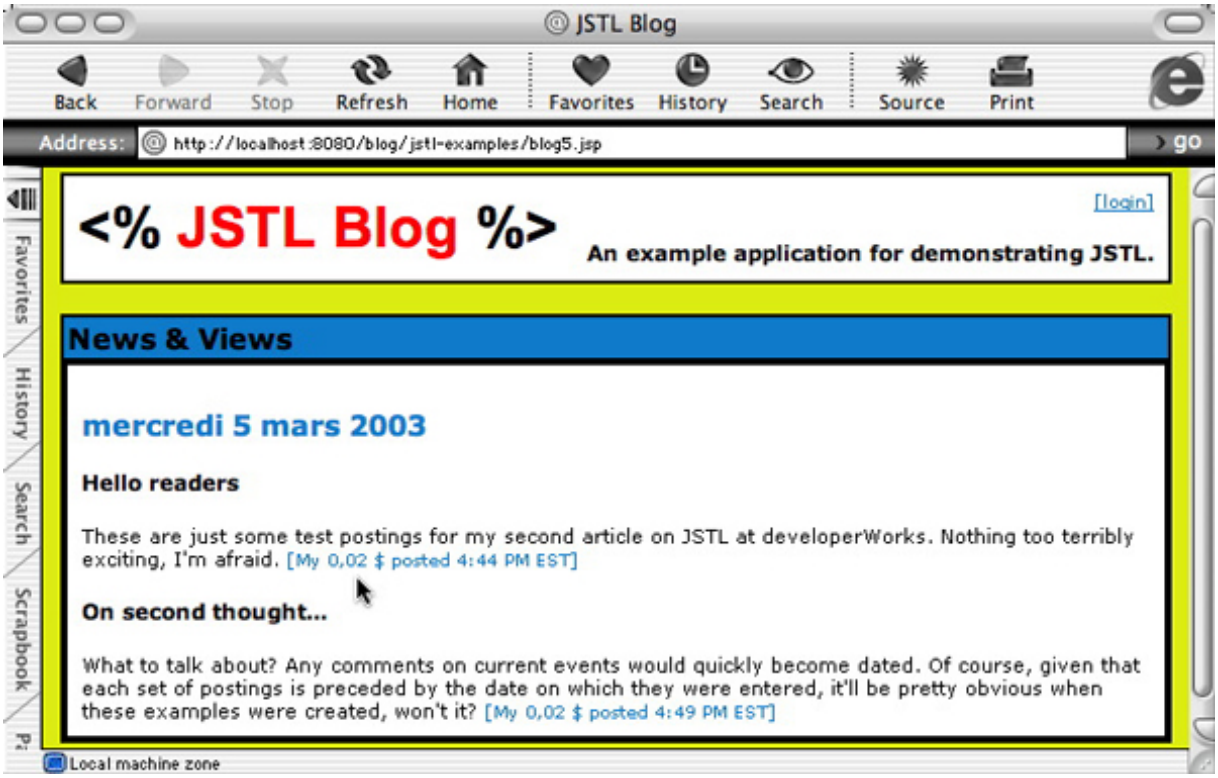
The output for the `en_US` locale is shown in Figure 5:

Figure 5. Output of the en_US locale from Listing 3



The output for the `fr_CA` locale is shown in Figure 6:

Figure 6. Output of the fr_CA locale from Listing 3



The `<fmt:parseNumber>` action, shown below, parses a numerical value provided through either its `value` attribute or its body content in a locale-specific manner, and returns the result as an instance of the `java.lang.Number` class. The `type` and `pattern` attributes play the same role for `<fmt:parseNumber>` as they do for `<fmt:formatNumber>`. Likewise, the `parseLocale`, `var`, and `scope` attributes play the same role for `<fmt:parseNumber>` as they do for `<fmt:parseDate>`.

```
<fmt:parseNumber value="expression"
    type="type" pattern="expression"
    parseLocale="expression"
    integerOnly="expression"
    var="name" scope="scope"/>

<fmt:parseNumber
    type="type" pattern="expression"
    parseLocale="expression"
    integerOnly="expression"
    var="name" scope="scope">
  body content
</fmt:parseNumber>
```

The comment made earlier regarding `<fmt:parseDate>` is equally applicable to `<fmt:parseNumber>`: Parsing data is not a task well-suited to the presentation layer. Software maintenance will be simplified if parsing and validating data are implemented as part of the application's business logic. For this reason, it is generally advisable to avoid the use of both `<fmt:parseDate>` and `<fmt:parseNumber>` in production JSP pages.

Only the `integerOnly` attribute is unique to `<fmt:parseNumber>`. This attribute takes a Boolean value indicating whether only the integer portion of the provided value should be parsed. If this attribute's value is `true`, any digits following the decimal point within the character string being parsed are ignored. The default value for this attribute is `false`.

Message tags

Localizing text in JSTL is accomplished with the `<fmt:message>` tag. This tag allows you to retrieve text messages from a locale-specific resource bundle and display it on a JSP page. Furthermore, because this action leverages the capabilities provided by the `java.text.MessageFormat` class, parameterized values can be substituted into such text messages to customize localized content dynamically.

Resource bundles for storing locale-specific messages take the form of classes or property files that adhere to a standard naming convention, in which a basename is combined with a locale name. Consider, for example, a property file named `Greeting.properties` that resides in our weblog application's classpath in the subdirectory corresponding to the `com.taglib.weblog` package. You could localize the resource bundle represented by this property file for the English and French languages by specifying two new property files in the same directory, named by appending the appropriate language codes. Specifically, these two files would be named `Greeting_en.properties` and `Greeting_fr.properties`, respectively. If additional localization for the Canadian dialect of the French language were desired, you could introduce a third property file that includes the appropriate country code in its name (such as `Greeting_fr_CA.properties`).

Each of these files would define the same properties, but the values for those properties would be customized to the corresponding language or dialect. This approach is shown in Listings 4 and 5, which provide sample contents for the `Greeting_en.properties` and `Greeting_fr.properties` files. In these examples, two localized messages are defined. They are identified by the `com.taglib.weblog.Greeting.greeting` and `com.taglib.weblog.Greeting.return` keys. The values associated with these keys, however, have been localized for the language identified in the file's name. Note that the `{0}` pattern appearing in both values for the `com.taglib.weblog.Greeting.greeting` message enables a parameterized value to be dynamically inserted into the message during content generation.

Listing 4. Contents of the Greeting_en.properties localized resource bundle

```
com.taglib.weblog.Greeting.greeting=Hello {0}, and welcome to the JSTL Blog.
com.taglib.weblog.Greeting.return=Return
```

Listing 5. Contents of the Greeting_fr.properties localized resource bundle

```
com.taglib.weblog.Greeting.greeting=Bonjour {0}, et bienvenue au JSTL Blog.
com.taglib.weblog.Greeting.return=Retournez
```

The first step in displaying such localized content with JSTL is to specify the resource bundle. The `fmt` library provides two custom tags for accomplishing this -- `<fmt:setBundle>` and `<fmt:bundle>` -- which are analogous in their behavior to the `<fmt:setTimeZone>` and `<fmt:timeZone>` tags introduced earlier. The `<fmt:setBundle>` action sets a default resource bundle for use by `<fmt:message>` tags within a particular scope, whereas `<fmt:bundle>` specifies the resource bundle for use by any and all `<fmt:message>` actions nested within its body content.

The code snippet below shows the syntax for the `<fmt:setBundle>` tag. The `basename` attribute is required, and identifies the resource bundle to be set as the default. Note that the value for the `basename` attribute should not include any localization suffixes or filename extensions. The basename for the example resource bundle presented in Listings 4 and 5 is `com.taglib.weblog.Greeting`.

```
<fmt:setBundle basename="expression"
    var="name" scope="scope"/>
```

The optional `scope` attribute indicates the JSP scope that the setting of the default resource bundle applies to. If this attribute is not explicitly specified, `page` scope is assumed.

If the optional `var` attribute is specified, then the resource bundle identified by the `basename` attribute will be assigned to the variable named by this attribute's value. In this case, the `scope` attribute specifies the variable's

scope; no default resource bundle is assigned to the corresponding JSP scope.

You use the <fmt:bundle> tag, whose syntax is shown below, to set the default resource bundle within the scope of its body content. Like <fmt:setBundle>, only the basename attribute is required. You use the optional prefix attribute to specify a default prefix for the key values of any nested <fmt:message> actions.

```
<fmt:bundle basename="expression"
prefix="expression">
  body content
</fmt:bundle>
```

Once the resource bundle has been set, it is the role of the <fmt:message> tag to actually display a localized message. Two different syntaxes are supported by this action, depending on whether any nested <fmt:param> tags are required:

```
<fmt:message key="expression" bundle="expression"
    var="name" scope="scope"/>

<fmt:message key="expression" bundle="expression"
    var="name" scope="scope">
  <fmt:param value="expression"/>
  ...
</fmt:message>
```

For <fmt:message>, only the key attribute is required. The value of the key attribute is used to determine which of the messages defined in the resource bundle is to be displayed.

You can use the bundle attribute to specify an explicit resource bundle for looking up the message identified by the key attribute. Note that the value of this attribute must be an actual resource bundle, such as is assigned by the <fmt:setBundle> action when its var attribute is specified. String values, such as the basename attribute of <fmt:bundle> and <fmt:setBundle>, are not supported by the bundle attribute of <fmt:message>.

If the var attribute of <fmt:message> is specified, then the text message generated by the tag is assigned to the named variable, rather than written to the JSP page. As usual, the optional scope attribute is used to specify the scope for the variable named by the var attribute.

You use the <fmt:param> tag to provide parameterized values for the text message, where needed, through the tag's value attribute. Alternatively, the value can be specified as body content of the <fmt:param> tag, in which case the attribute is omitted. Values specified with the <fmt:param> tag are spliced into the message retrieved from the resource bundle wherever parameterized value patterns appear in the message text, in accordance with the behavior of the java.text.MessageFormat class. Because parameterized values are identified by their indices, the order of the nested <fmt:param> tags is significant.

The interaction of the <fmt:bundle>, <fmt:message>, and <fmt:param> tags is shown in Listing 6. Here, the <fmt:bundle> tag specifies the bundle localized messages are to be retrieved from by the two nested <fmt:message> tags. The first of these two <fmt:message> tags corresponds to a message with a single parameterized value, for which a corresponding <fmt:param> tag appears.

Listing 6. Using the <fmt:message> tag to display localized messages

```
<fmt:bundle basename="com.taglib.weblog.Greeting">
<fmt:message key="com.taglib.weblog.Greeting.greeting">
<fmt:param value="${user.fullName}"/>
</fmt:message>
  <br>
  <br>
  <center>
    <a href=
      "<c:url value='/index.jsp'/>"><fmt:message
          key="com.taglib.weblog.Greeting.return"/></a>
  </center>
</fmt:bundle>
```

Listing 7 shows the use of `<fmt:bundle>`'s `prefix` attribute; the value provided for the `prefix` attribute is automatically prepended to all `key` values in the nested `<fmt:message>` actions. Listing 7 is therefore equivalent to Listing 6, but takes advantage of this convenience feature to enable the use of abbreviated `key` values in the two `<fmt:message>` tags.

Listing 7. Effect of the prefix attribute of <fmt:bundle> on the <fmt:message> tag

```
<fmt:bundle basename="com.taglib.weblog.Greeting"
            prefix="com.taglib.weblog.Greeting.">
<fmt:message key="greeting">
<fmt:param value="${user.fullName}"/>
</fmt:message>
  <br>
  <br>
  <center>
    <a href="<c:url value='/index.jsp'/>"><fmt:message key="return"/></a>
  </center>
</fmt:bundle>
```

Figure 7 and Figure 8 show the `fmt` library's message-related tags in action, showing the output resulting from the code in Listing 7 and the localized resource bundles in Listing 4 and Listing 5. Figure 7 shows the results when the browser preferences reflect an English-language locale.

Figure 7. Output of the en_US locale from Listing 7

Figure 8 shows the output for a locale specifying the French language.

Figure 8. Output of the fr_CA locale from Listing 7

Summary
The custom tags in the JSTL `fmt` library provide straightforward access to the Java platform's internationalization API for JSP developers. Text messages, numerical values, and dates can all be displayed in a locale-sensitive manner, while times can also be adjusted for specific time zones. The locale for a particular user can be determined automatically from the user's browser settings, or specified explicitly by the page author. Finally, in addition to providing actions for generating and displaying formatted data, the `fmt` library also includes custom tags for parsing numerical and time-oriented data.

Resources

- 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.
- Mark Davis and Helena Shih provide an extensive overview of the Java platform's internationalization features in "The Java International API: Beyond JDK 1.1" (*developerWorks*, October 1998).
- If you're interested in internationalization, you won't want to miss Joe Sam Shirah's comprehensive tutorial, Java internationalization basics (*developerWorks*, April 2002).
- Valid language codes are defined by the ISO 639 specification, while the ISO 3166 specification defines the valid country codes.
- Similarly, currency codes are defined by the ISO 4217 specification.

- "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 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 platform topics and the co-author of *Web Development with JavaServer Pages, 2nd Edition*. You can contact Mark at mak@taglib.com.