

Help for Server-Side Java Developers

3rd Edition
Covers JSP 2.0 & JSTL 1.1

JavaServer Pages™



O'REILLY®

Hans Bergsten

Working with XML Data

There's no escape from Extensible Markup Language (XML) these days. It's everywhere: in configuration files, messages between servers, web pages, even databases. Wherever there's structured data, XML is often found close by.

As I mentioned earlier, JSP pages can generate any type of text, including XML. In the simplest case, the JSP page includes static XML elements as template text and a few actions to add the dynamic data, similar to the HTML examples in previous chapters. A more sophisticated page gets raw XML data from somewhere and transforms it to different XML formats depending on the type of browser making the request.

More and more, web applications also consume XML data generated by an external source, perhaps a database or another server. Such an application may extract price information from different vendors' product catalogs, published as XML documents, and create a side-by-side comparison.

In this chapter we first look at the things you need to be aware of when generating XML responses with JSP, including device-dependent transformations, and then how to process XML data in different ways.

Generating an XML Response

XML is a set of syntax rules for how to represent structured data using markup elements represented by an opening tag (optionally with attributes), a body, and a closing tag:

```
<employee id="123">
  <first-name>Hans</first-name>
  <last-name>Bergsten</last-name>
  <telephone>310-555-1212</telephone>
</employee>
```

This XML example contains four elements: `<employee>`, `<first-name>`, `<last-name>`, and `<telephone>`.

By selecting sensible element names, an XML file may be understandable to a human, but to make sense to a program, it must use only a restricted set of elements in which each element has a well-defined meaning. This is known as an *XML application* (the XML syntax applied to a certain application domain). A couple of examples are the Wireless Markup Language (WML) used for browsers in cellular phones and other small devices, and XHTML, which is HTML 4.0 reformulated as an XML application. Another example is the web application deployment descriptor, used to configure various aspects of a standard Java web application, as you have seen in the previous chapters.

As I mentioned in Chapters 3 and 5, everything in a JSP page that is not a JSP element is template text. In all examples so far, I have used HTML as the template text, but it can be any markup, for instance XHTML or WML XML elements. Example 15-1 shows a JSP page that sends a simple phone book to a wireless device, using the XML elements defined by the WML specification as the template text.

Example 15-1. WML phone book JSP page (phone_wml.jsp)

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<%@ page contentType="text/vnd.wap.wml" %>
<wml>
  <card id="list" newcontext="true">
    <p>Phone List</p>
    <p>
      <anchor>Bergsten, Hans
        <go href="#Bergsten_Hans"/>
      </anchor>
      <br/>
      <anchor>Eckstein, Bob
        <go href="#Eckstein_Bob"/>
      </anchor>
      <br/>
      <anchor>Ferguson, Paula
        <go href="#Ferguson_Paula"/>
      </anchor>
    </p>
  </card>

  <card id="Bergsten_Hans">
    <p>Bergsten, Hans</p>
    <p>
      Phone: 310-555-1212
      <do type="prev" label="Back">
        <prev/>
      </do>
    </p>
  </card>
  <card id="Eckstein_Bob">
    <p>Eckstein, Bob</p>
```

Example 15-1. WML phone book JSP page (*phone_wml.jsp*) (continued)

```
<p>
  Phone: 800-555-5678
  <do type="prev" label="Back">
    <prev/>
  </do>
</p>
</card>
<card id="Ferguson_Paula">
  <p>Ferguson, Paula</p>
  <p>
    Phone: 213-555-1234
    <do type="prev" label="Back">
      <prev/>
    </do>
  </p>
</card>
</wml>
```

A discussion of the WML elements is outside the scope of this book, but let's look at some important details of the JSP page. The first line in Example 15-1 is an *XML declaration*, telling which version of XML the document conforms to. Some WML browsers are very picky about this being the first thing in an XML document, and even whitespaces—regular spaces, linefeed characters, and tab characters—before the declaration can throw them off. In all examples you have seen so far, the JSP page directive has been on the first line. Here, I have moved it down so that the linefeed character that ends the directive line doesn't cause any problems.

The second and third lines in Example 15-1 contain an *XML document type declaration*. This identifies the so-called Document Type Definition (DTD) for the document, basically the definition of all XML elements a conforming document of this type can contain. Here, it's the DTD for the WML 1.1 elements.

The JSP page directive on the fourth line is important. The content type for a JSP page is `text/html` by default. For a WML document, you must instead specify the content type `text/vnd.wap.wml`. Otherwise the WML browser doesn't accept the document.

The rest of the page in Example 15-1 is just static WML code. To run this example, you need a WML browser. I've used the WML browser included in the Openwave Systems Inc. SDK 4.1, available at <http://developer.openwave.com/resources/sdk.html>, to test the examples in this chapter. Figure 15-1 shows what the phone-list menu card and one details card look like in this WML browser.

Transforming XML into HTML

You may also have heard about the Extensible Stylesheet Language (XSL). XSL defines one set of XML elements to transform an XML document into some other type of document, and another set of elements to produce a formatted version of an



Figure 15-1. Phone list in WML browser (UP.SDK image courtesy of Openwave Systems Inc.)

XML document suitable for display. Browsers and other programs that need to render an XML document with different styles for different elements, such as a bold large font for a header and a regular font for paragraph text, use the formatting part of XSL. The transformation part of XSL is referred to as XSLT. XSLT can turn a source XML document, such as a document representing an order, into different forms using different stylesheets. This is useful in business-to-business (B2B) applications, where different partners often require the same information in slightly different formats. You can read more about XSL and XSLT at <http://www.w3.org/TR/xsl/>.

In a web application, XSLT can transform structured XML data into HTML. Example 15-2 shows an example of a JSP page in which the same phone book information used in Example 15-1 is transformed into an HTML table.

Example 15-2. Transforming XML to HTML (phone_html.jsp)

```
<%@ page contentType="text/html" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
<html>
  <head>
    <title>Phone List</title>
  </head>
  <body bgcolor="white">

    <x:import url="htmltable.xml" var="stylesheet" />
    <x:transform xslt="${stylesheet}">
```

Example 15-2. Transforming XML to HTML (*phone_html.jsp*) (continued)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<employees>
  <employee id="123">
    <first-name>Hans</first-name>
    <last-name>Bergsten</last-name>
    <telephone>310-555-1212</telephone>
  </employee>
  <employee id="456">
    <first-name>Bob</first-name>
    <last-name>Eckstein</last-name>
    <telephone>800-555-5678</telephone>
  </employee>
  <employee id="789">
    <first-name>Paula</first-name>
    <last-name>Ferguson</last-name>
    <telephone>213-555-1234</telephone>
  </employee>
</employees>
</x:transform>

</body>
</html>
```

At the top of the page, the `taglib` directive for the JSTL XML library is included, along with the directive for the JSTL core library used in previous chapters.

To transform the XML data, you first need to get hold of the XSLT stylesheet. The JSTL `<c:import>` action, described in Table 15-1, loads the stylesheet from the file specified by the `url` attribute and saves it in the variable named by the `var` attribute.

Table 15-1. Attributes for JSTL `<c:import>`

Attribute name	Java type	Dynamic value accepted	Description
<code>url</code>	String	Yes	Mandatory. A page- or context-relative path, or an absolute URL.
<code>context</code>	String	Yes	Optional. The context path for another application.
<code>charEncoding</code>	String	Yes	Optional. The character encoding for the imported content. Default is the encoding specified by the protocol used for the import or ISO-8859-1 if no encoding is found.
<code>var</code>	String	No	Optional. The name of the variable to hold the result as a String.
<code>scope</code>	String	No	Optional. The scope for the variable, one of <code>page</code> , <code>request</code> , <code>session</code> , or <code>application</code> . <code>page</code> is the default.
<code>varReader</code>	String	No	Optional. The name of the variable to expose the result as a Reader to the body.

The `<c:import>` action is very versatile. You can use it to import data from resources in the same application, another application on the same server (identified by the context attribute), and even from an external server by specifying an absolute URL for any protocol supported by the web container, such as HTTP, HTTPS, or FTP. Parameters can be defined either in the URL as a query string or using nested `<c:param>` actions. The imported data can be saved as a `String` in any scope, or exposed as a `java.io.Reader` to actions within the element's body. Using a `Reader` is slightly more efficient, because the `<c:import>` action doesn't have to read the input in this case; it just wraps a `Reader` around the input stream that a nested action then reads directly. I'll show you an example of this later. When you import a resource (such as a JSP page) that belongs to the same application, the target resource has access to all request parameters and variables in the request scope, the same way as when you use the `<jsp:forward>` action (Chapter 10).

The transformation is performed by a JSTL action named `<x:transform>`, described in Table 15-2.

Table 15-2. Attributes for JSTL `<x:transform>`

Attribute name	Java type	Dynamic value accepted	Description
<code>doc</code>	<code>String</code> , <code>java.io.Reader</code> , <code>javax.xml.transform.Source</code> , <code>org.w3c.dom.Document</code> , or the types exposed by <code><x:parse></code> and <code><x:set></code>	Yes	Mandatory, unless specified as the body. The XML document to transform.
<code>xslt</code>	<code>String</code> , <code>java.io.Reader</code> , <code>javax.xml.transform.Source</code>	Yes	Mandatory. The XSLT stylesheet.
<code>docSystemId</code>	<code>String</code>	Yes	Optional. The system identifier for the XML document.
<code>xsltSystemId</code>	<code>String</code>	Yes	Optional. The system identifier for the XSLT stylesheet.
<code>result</code>	<code>javax.xml.transform.Result</code>	Yes	Optional. A <code>Result</code> object used to capture or process the transformation result.
<code>var</code>	<code>String</code>	No	Optional. The name of the variable to hold the result as a <code>org.w3c.dom.Document</code> .
<code>scope</code>	<code>String</code>	No	Optional. The scope for the variable; one of <code>page</code> , <code>request</code> , <code>session</code> , or <code>application</code> . <code>page</code> is the default.

The XML document to transform can be specified as the body, as in Example 15-2, or as a variable through the `doc` attribute. The example XML document contains elements representing information about employees. The `xsl` attribute is set to the XSL

stylesheet imported by the `<c:import>` action. It contains XSLT elements for transforming the XML document into an HTML table. In Example 15-2, both the `var` and the `result` attributes are omitted, so the `<x:transform>` action adds its result to the response. This is the most common use, but the `var` and `result` attributes can be used if the transformation result needs to be captured and processed further.

Descriptions of all the XSLT elements would fill a book all by itself, but Example 15-3 shows the stylesheet used here to give you an idea of how XSLT looks.

Example 15-3. XSL stylesheet that generates an HTML table (htmltable.xml)

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="employees">
    <table border="1" width="100%">
      <tr>
        <th>ID</th>
        <th>Employee Name</th>
        <th>Phone Number</th>
      </tr>
      <xsl:for-each select="employee">
        <tr>
          <td>
            <xsl:value-of select="@id"/>
          </td>
          <td>
            <xsl:value-of select="last-name"/>,
            <xsl:value-of select="first-name"/>
          </td>
          <td>
            <xsl:value-of select="telephone"/>
          </td>
        </tr>
      </xsl:for-each>
    </table>
  </xsl:template>

</xsl:stylesheet>
```

The XSLT elements are similar to JSP action elements in that they perform some action rather than identify information types. The XSLT elements select and process pieces of the source XML document. Here, the `<xsl:template>` element selects the top `<employees>` element in the source XML document, the `<xsl:for-each>` element loops over all nested `<employee>` elements, and the `<xsl:value-of>` elements extract the attribute values and nested elements for each `<employee>` element. The non-XSLT elements are used as template data, the same way as in JSP. You get the idea.

An XSLT stylesheet can use parameters to represent dynamic data, provided to the XSLT processor when a document is transformed:

```

<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:param name="empName" />

  <xsl:template match="employees/employee[name = $empName]">
  ...

```

The parameter in this example limits the `<employee>` elements to be processed to those that have a `<name>` element with the value specified by the parameter.

To pass the parameter value to the XSLT stylesheet, you must use a nested `<x:param>` action in the `<x:transform>` body:

```

<x:transform xslt="{stylesheet}">
  <x:param name="empName" value="{param:empName}" />
  <?xml version="1.0" encoding="ISO-8859-1"?>
  <employees>
    <employee id="123">
      <first-name>Hans</first-name>
      <last-name>Bergsten</last-name>
      <telephone>310-555-1212</telephone>
    </employee>
    ...
  </x:transform>

```

Here I pass on a request parameter value to the stylesheet, but you can, of course, use any EL expression as the value.

XML documents, including XSLT stylesheets, can contain references to external entities, for instance in the XSL `<xsl:include>` and `<xsl:import>` elements. If these references are written as relative paths in the document, a base URI must be used to establish what they are relative to. You can pass base URIs for the XSLT stylesheet and the XML source to the `<x:transform>` action through the `xsltSystemId` and the `docSystemId` attributes. The value can be any valid URI, such as an absolute file or HTTP URL or a context- or page-relative path.

Transforming XML into a Device-Dependent Format

A web application can use XSLT to respond with different content depending on the type of device making the request. Example 15-4 shows a page that serves both HTML and WML browsers by applying different stylesheets to the same XML document, transforming it to the appropriate markup for the browser that requests it.

Example 15-4. XSL stylesheet that generates HTML or WML (phone.jsp)

```

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

```

Example 15-4. XSL stylesheet that generates HTML or WML (*phone.jsp*) (continued)

```
%><%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions"
%><%@ taglib prefix="ora" uri="orataglib"
%><c:choose><c:when
test="{fn:contains(header.Accept, 'text/vnd.wap.wml')}"
><ora:setHeader name="Content-Type" value="text/vnd.wap.wml"
/><c:import url="wml.xsl" var="stylesheet"
/></c:when><c:otherwise><ora:setHeader name="Content-Type" value="text/html"
/><c:import url="html.xsl" var="stylesheet"
/></c:otherwise></c:choose><x:transform xslt="{stylesheet}">
<?xml version="1.0" encoding="ISO-8859-1"?>
<employees>
  <employee id="123">
    <first-name>Hans</first-name>
    <last-name>Bergsten</last-name>
    <telephone>310-555-1212</telephone>
  </employee>
  <employee id="456">
    <first-name>Bob</first-name>
    <last-name>Eckstein</last-name>
    <telephone>800-555-5678</telephone>
  </employee>
  <employee id="789">
    <first-name>Paula</first-name>
    <last-name>Ferguson</last-name>
    <telephone>213-555-1234</telephone>
  </employee>
</employees>
</x:transform>
```

There are a number of things to note here. First, see how messy the page looks. That's because the start tag for all JSP directives and actions in this page are written on the same line as the end tag for the preceding element, to make sure that no extra linefeeds are added to the response. As described earlier, leading whitespace (such as linefeed characters) in a WML page can cause a WML browser to reject the page.

Because the page can serve both HTML and WML content, the page directive's `contentType` attribute cannot be used to set the content type. Instead, the content type needs to be set dynamically. This page uses a JSTL function and a custom action to handle this. The JSTL `fn:contains()` function checks if the HTTP Accept header contains the content type for WML. This piece of information is used to decide which type of content to return. If the browser accepts WML, the `<ora:setHeader>` custom action sets the Content-Type header dynamically to `text/vnd.wap.wml`, otherwise to `text/html`. The `<c:import>` actions import the appropriate stylesheet, `wml.xsl` or `html.xsl`, based on the device type making the request, and the `<x:transform>` action finally transforms the XML document accordingly.

For a simple example like this, letting an XSLT stylesheet transform the XML source into a complete web page works fine. However, on most real web sites, the HTML version of the site differs significantly from the WML version. You want to provide a

rich interface for HTML browsers with a nice layout, navigation bars, images, colors, and fonts, and typically as much content as you can fit on each page. A WML browser, on the other hand, has a very small screen with limited layout, font, and graphics capabilities. Developing an efficient interface for this type of device is very different. A more practical approach for combining XML, XSLT, and JSP to serve different types of browsers is to keep the actual content (articles, product information, phone lists, etc.) in a device-independent XML format, but use separate JSP pages for each device type. The JSP pages can then use the `<x:transform>` action to transform the key content and merge it with the device-dependent template text to form a complete page suitable for each specific device type, like in Example 15-1.

Processing XML Data

XSLT is great for transforming an XML source into another format, but sometimes you need to process the XML data in other ways. For instance, you may want to use part of the XML data in a database query to get additional information and compose a response that merges the two data sources, or reformat date and numeric information in the XML source according to the user's preferred locale. To process XML data in this way, the JSTL XML library includes a number of actions for picking out pieces of an XML document, as well as iteration and conditional actions similar to the ones in the core library, but adapted to work specifically with XML data.

In this section, we look at an example that uses most of the JSTL XML actions. The XML data comes from the O'Reilly Meerkat news feed. Meerkat scans a large set of Rich Site Summary (RSS)—an XML application suitable for news, product announcements, and similar content—sources frequently and makes the aggregated data available in a number of formats, including a superset of the RSS format that includes category, source, and date information for each story. You can learn more about Meerkat and how to use it at http://www.oreillynet.com/pub/a/rss/2000/05/09/meerkat_api.html. Example 15-5 shows a sample of the XML data that Meerkat can deliver.

Example 15-5. Meerkat XML news feed format

```
<?xml version="1.0"?>
<!DOCTYPE meerkat_xml_flavour
  SYSTEM "http://meerkat.oreillynet.com/dtd/meerkat_xml_flavour.dtd">

<meerkat>

  <title>Meerkat: An Open Wire Service</title>
  <link>http://meerkat.oreillynet.com</link>
  <description>
    Meerkat is a Web-based syndicated content reader providing
    a simple interface to RSS stories. While maintaining the original
    association of a story with a channel, Meerkat's focus is on
    chronological order -- the latest stories float to the top,
```

Example 15-5. Meerkat XML news feed format (continued)

```
    regardless of their source.
</description>
<language>en-us</language>

<image>
  <title>Meerkat Powered!</title>
  <url>http://meerkat.oreillynet.com/icons/meerkat-powered.jpg</url>
  <link>http://meerkat.oreillynet.com</link>
  <width>88</width>
  <height>31</height>
  <description>
    Visit Meerkat in full splendor at meerkat.oreillynet.com
  </description>
</image>

<story id="881051">
  <title>
    Clay Shirky: What Web Services Got Right ... and Wrong
  </title>
  <link>
    http://www.oreillynet.com/pub/a/network/2002/04/22/clay.html
  </link>
  <description>
    Web Services represent not just a new way to build Internet
    applications, says Clay Shirky in this interview, but the second
    stage of peer-to-peer, in which distinctions between clients and
    servers are all but eliminated.
  </description>
  <category>General</category>
  <channel>O'Reilly Network</channel>
  <timestamp>2002-04-23 17:02:50</timestamp>
</story>
...
</meerkat>
```

The example application processes this XML data in a number of ways. First, it extracts some information about the Meerkat service itself and adds it to the page, so the user can see where the data comes from. It then gets all `<category>` elements and builds a list of unique category names. This list is used to build an HTML select list, from which the user can pick one category to filter the data. The XML data is then filtered accordingly, and an HTML table with matching stories is generated. Just for fun and to illustrate the use of the conditional XML actions, all stories in the General category are displayed against a light green background. The result is shown in Figure 15-2.

Example 15-6 shows the JSP page that does all the processing.

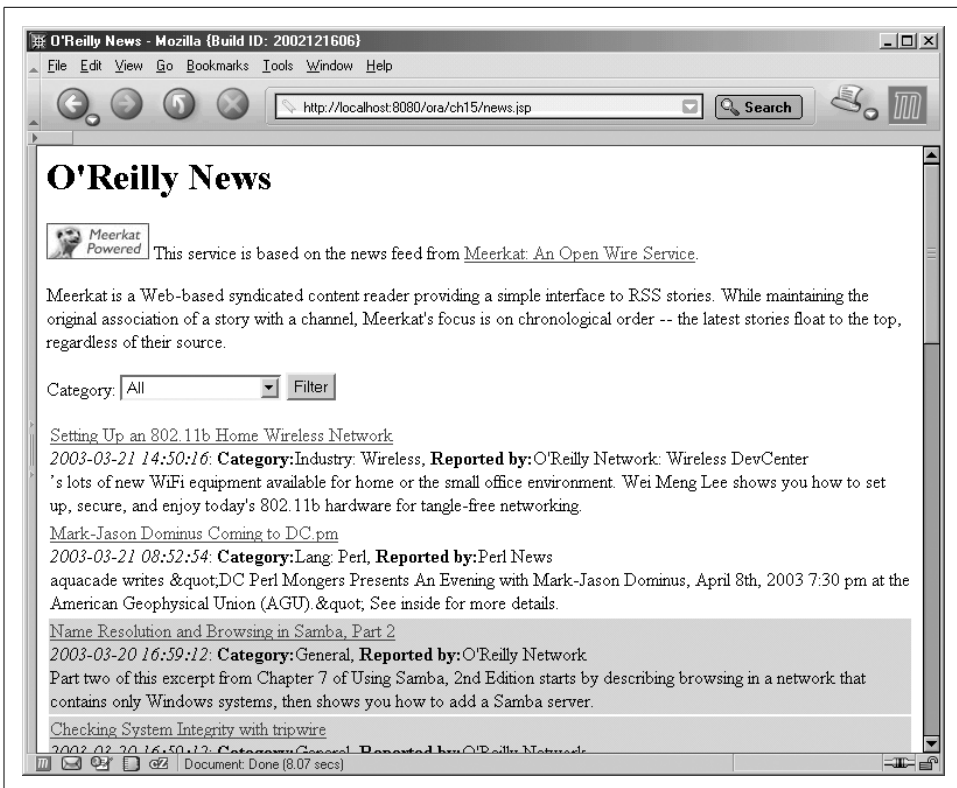


Figure 15-2. The XML-base news service application

Example 15-6. Processing XML data (news.jsp)

```

<%@ page contentType="text/html" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>

<%--
    Get new XML data if the cached version is older than
    1 hour.
--%>
<c:set var="cachePeriod" value="{60 * 60 * 1000}" />
<jsp:useBean id="now" class="java.util.Date" />
<c:if test="{(now.time - cacheTime) > cachePeriod}">
    <c:import url="http://meerkat.oreillynet.com/?p=4999&_fl=xml&t=ALL"
        varReader="xmlSource">
        <x:parse var="doc" doc="{xmlSource}" scope="application" />
    </c:import>
    <c:set var="cacheTime" value="{now.time}" scope="application" />
</c:if>

<html>
<head>

```

Example 15-6. Processing XML data (news.jsp) (continued)

```
<title>O'Reilly News</title>
</head>
<body bgcolor="white">
  <h1>O'Reilly News</h1>
  " />
  This service is based on the news feed from
  <a href="<x:out select="$doc/meerkat/link" />" />
    <x:out select="$doc/meerkat/title" /></a>.
  <p>
  <x:out select="$doc/meerkat/description" />

  <!--
  Create a list of unique categories present in the XML feed
  -->
  <jsp:useBean id="uniqueCats" class="java.util.TreeMap" />
  <x:forEach select="$doc/meerkat/story/category" var="category">
    <!-- Need to convert the XPath node to a Java String -->
    <x:set var="catName" select="string($category)" />
    <c:set target="{uniqueCats}" property="{catName}" value="" />
  </x:forEach>

  <form action="news.jsp">
    Category:
    <select name="selCat">
      <option value="ALL">All
      <c:forEach items="{uniqueCats}" var="current">
        <option value="<c:out value="{current.key}" />" />
          <c:if test="{param.selCat == current.key}">
            selected
          </c:if>
          <c:out value="{current.key}" />
        </option>
      </c:forEach>
    </select>
    <input type="submit" value="Filter">
  </form>

  <!-- Filter the parsed document based on the selection -->
  <c:choose>
    <c:when test="{empty param.selCat || param.selCat == 'ALL'}">
      <x:set var="stories" select="$doc//story" />
    </c:when>
    <c:otherwise>
      <x:set var="stories"
        select="$doc//story[category = $param:selCat]" />
    </c:otherwise>
  </c:choose>

  <!-- Generate a table with data for the selection -->
  <table>
    <x:forEach select="$stories">
      <tr>
```

Example 15-6. Processing XML data (news.jsp) (continued)

```
<x:choose>
  <x:when select="category[. = 'General']">
    <td bgcolor="lightgreen">
      </x:when>
    <x:otherwise>
      <td>
        </x:otherwise>
      </x:choose>
      <a href="<x:out select="link" />">
        <x:out select="title" /></a>
      <br>
      <i><x:out select="timestamp" /></i>:
      <b>Category:</b><x:out select="category" />,
      <b>Reported by:</b><x:out select="channel" />
      <br><x:out select="description" />
    </td>
  </tr>
</x:forEach>
</table>
</body>
</html>
```

At the top of the page, the XML source is retrieved from the Meerkat server using the same `<c:import>` action used in the previous examples. There are two noteworthy differences, though: the `url` attribute specifies an absolute URL and the imported data is exposed as a `Reader` instead of as a `String`. I mentioned both these features earlier. In this example, using a `Reader` is appropriate because the data may be large, and it's only of interest to the nested `<x:parse>` action.

Caching Data

Before we look at the `<x:parse>` action in detail, I'd like to say a few words about the caching technique used in this example. The Meerkat data is updated only on an hourly basis, so it's pointless to ask for it more frequently. It's also expensive in terms of time and computing resources to import and parse the XML data. By caching the parsed data for an hour, the web application gets more responsive and avoids putting load on the Meerkat server unnecessarily.

The caching technique used here simply creates a timestamp for the data in the form of a `java.util.Date` object and saves it together with the data itself in the application scope, using standard and JSTL core actions. When a new request is received, it's tested to see if the cache is older than the predefined cache period (one hour in this example). If it is, a fresh copy is imported, parsed, and saved in the application scope again, along with the timestamp. Otherwise the cached data is used. You can use this technique for any type of processing that's expensive, for instance retrieving data from a database or performing complex calculations.

Parsing XML Data

Before you can access the XML data with the JSTL XML actions, the imported document must be parsed and converted to a data structure the actions can read. That's what the `<x:parse>` action does (see Table 15-3).

Table 15-3. Attributes for JSTL `<x:parse>`

Attribute name	Java type	Dynamic value accepted	Description
<code>doc</code>	<code>String</code> or <code>java.io.Reader</code>	Yes	Mandatory, unless specified as the body. The XML document to parse.
<code>systemId</code>	<code>String</code>	Yes	Optional. The system identifier for the XML document.
<code>filter</code>	<code>org.xml.sax.XMLFilter</code>	Yes	Optional. An <code>XMLFilter</code> to be applied to the XML document.
<code>var</code>	<code>String</code>	No	Optional. The name of the variable to hold the result as an implementation-dependent type.
<code>scope</code>	<code>String</code>	No	Optional. The scope for the variable, one of <code>page</code> , <code>request</code> , <code>session</code> , or <code>application</code> . <code>page</code> is the default.
<code>varDom</code>	<code>String</code>	No	Optional. The name of the variable to hold the result as a <code>org.w3c.dom.Document</code> .
<code>scopeDom</code>	<code>String</code>	No	Optional. The scope for the DOM variable, one of <code>page</code> , <code>request</code> , <code>session</code> , or <code>application</code> . <code>page</code> is the default.

The XML document to parse can be specified as the body or as a `String` or `Reader` variable. In Example 15-3, I use the `Reader` exposed by the `<c:import>` action to get the best performance. A base URI for interpretation of relative URIs in the document can be specified by the `systemId` attribute, the same way as for the `<x:transform>` action.

The parse result can be saved either as an implementation-dependent data structure (named by the `var` attribute) or as a standard `org.w3c.dom.Document` object (named by the `varDom` attribute), in any scope. You should use the latter only if you need to process the parse result with a custom action or other custom code because the implementation-dependent type is typically optimized in terms of memory use and ease of access, and it's supported by all the other JSTL XML actions that use a parse result. The implementation-dependent data structure is saved as an application scope variable in Example 15-3, where it's picked up by the other XML actions in the page.

If the XML document is large and you're only interested in a very small part of it, you can provide an implementation of the `org.xml.sax.XMLFilter` interface to the action, typically created and configured by a servlet, a filter, or a listener (the filter and listener component types are described in Chapter 19). As the name implies, an

XMLFilter can remove the parts you don't need, making the parsing process more efficient. For more about XML filters, I suggest you look at the documentation of the interface or read a book about Java and XML, such as Brett McLaughlin's *Java and XML* (O'Reilly).

Accessing XML Data Using XPath Expressions

With the parsing out of the way, we can turn to how to access parts of the XML data. The JSTL XML library contains a number of actions for this purpose, similar to the ones you're familiar with from the JSTL core library: `<x:out>`, `<x:set>`, `<x:if>`, `<x:choose>`, `<x:when>`, `<x:otherwise>`, and `<x:forEach>`. The main difference between the XML and the core flavor is that the XML actions use a special language for working with XML data, named XPath, instead of the standard JSP EL. XPath 1.0 is a W3C recommendation that has been around since 1999, and it's used in XSLT stylesheets and other XML applications.* The language details are beyond the scope for this book, but here's a brief summary.

An XPath expression identifies one or more nodes (root, elements, attributes, namespace attributes, comments, text, and processing instructions) in an XML document. The simplest expression type is a plain *location path*, similar to a Unix filesystem path, to a set of nodes in the document. For instance, the path `/meerkat/image/url` identifies the `<url>` element in the Meerkat XML document:

```
...
<meerkat>

  <title>Meerkat: An Open Wire Service</title>
  <link>http://meerkat.oreillynet.com</link>
  <description>
    Meerkat is a Web-based syndicated content reader providing
    a simple interface to RSS stories. While maintaining the original
    association of a story with a channel, Meerkat's focus is on
    chronological order -- the latest stories float to the top,
    regardless of their source.
  </description>
  <language>en-us</language>

  <image>
    <title>Meerkat Powered!</title>
    <url>http://meerkat.oreillynet.com/icons/meerkat-powered.jpg</url>
  ...
```

A location path that starts with double forward slashes identifies all nodes of a certain type, regardless of their position in the document hierarchy. For instance, `//description` identifies all `<description>` elements, so it finds two elements in the sample XML document:

* Available at <http://www.w3.org/TR/xpath>.

```

...
<meerkat>
  ...
  <description>
    Meerkat is a Web-based syndicated content reader providing
    a simple interface to RSS stories. While maintaining the original
    association of a story with a channel, Meerkat's focus is on
    chronological order -- the latest stories float to the top,
    regardless of their source.
  </description>
  ...
  <image>
    ...
    <description>
      Visit Meerkat in full splendor at meerkat.oreillynet.com
    </description>
    ...

```

A path is always interpreted relative to a specific context, such as the complete document or a subset of its nodes. When you use XPath expressions as JSTL XML action attributes, the context can be represented by a variable and can also be adjusted by actions such as the `<x:forEach>` action. Besides the type of paths described here, an XPath expression can also include function calls, literals, operators, and special syntax for identifying attributes. Some of these features are used in Example 15-3, but I recommend that you learn more about them if you're going to use the JSTL XML actions. Check out the XPath chapter from Elliotte Rusty Harold and W. Scott Means's *XML in a Nutshell* (O'Reilly), available online at <http://www.oreilly.com/catalog/xmlnut/chapter/ch09.html>, and Robert Eckstein's *XML Pocket Reference* (O'Reilly). The XPath tutorial by Miloslav Nic and Jiri Jirat, available at <http://www.zvon.org/xxl/XPathTutorial/General/examples.html>, is another good resource.

Let's look at how XPath expressions are used with the JSTL `<x:out>` action (see Table 15-4) to add the general Meerkat information that appears at the beginning of the page:

```

" />
This service is based on the news feed from
<a href="<x:out select="$doc/meerkat/link" />" />
  <x:out select="$doc/meerkat/title" /></a>.
<p>
<x:out select="$doc/meerkat/description" />

```

Table 15-4. Attributes for JSTL `<x:out>`

Attribute name	Java type	Dynamic value accepted	Description
select	String	No	Mandatory. An XPath expression to be evaluated.
escapeXml	boolean	Yes	Optional. <code>true</code> if special characters in the value should be converted to character entity codes. Default is <code>true</code> .

All JSTL actions that accept XPath expressions do so only for their select attribute, to avoid confusion with other attributes that accept JSP EL expressions. For the first `<x:out>` action, the select attribute contains an XPath expression that starts with the doc variable (containing the parse result) followed by a location path for the `<url>` element. The `<x:out>` action converts the XPath evaluation result to a Java String and adds it to the response.

The way the doc variable is used here establishes the context for the XPath expression. Variables can appear anywhere in an XPath expression and always start with a dollar sign, followed by the name of the variable. XPath expressions used with the JSTL actions have access to almost the same type of dynamic data as an EL expression. Any application variable in any JSP scope can be accessed by its name, just as in an EL expression. The doc variable is an example of this. Important differences are that in an XPath expression, all variable names start with a dollar sign, and the EL property and element access operators (. and []) aren't recognized, so you can't use syntax like `bean.propertyName` in an XPath expression. A workaround is to save the property or element value in a new variable, and use it in the XPath expression:

```
<c:set var="myProperty" value="{myBean.myProperty}" />
<x:out select="$doc/root/myElement[@myAttribute = $myProperty]" />
```

Here the property value finds elements with an attribute that matches a bean property value. Also note that the XPath expression itself is not identified by any special syntax, as opposed to an EL expression that must always be enclosed by `{` and `}`.

In addition to application data, most of the information represented by EL implicit variables is available to an XPath expression with a slightly different syntax, most noticeable that a colon is used as a separator instead of a dot (see Table 15-5).

Table 15-5. XPath implicit variables

XPath expression	Description
<code>\$param:myParam</code>	The myParam request parameter
<code>\$header:Accept</code>	The Accept request header
<code>\$cookie:password</code>	The password cookie
<code>\$initParam:myConfig</code>	The myConfig context parameter
<code>\$pageScope:myVariable</code>	The myVariable variable from the page scope
<code>\$requestScope:myVariable</code>	The myVariable variable from the request scope
<code>\$sessionScope:myVariable</code>	The myVariable variable from the session scope
<code>\$applicationScope:myVariable</code>	The myVariable variable from the application scope

The JSTL `<x:forEach>` action (Table 15-6) lets you loop through the nodes that matches an XPath expression.

Table 15-6. Attributes for JSTL `<x:forEach>`

Attribute name	Java Type	Dynamic value accepted	Description
select	String	No	Mandatory. An XPath expression to be evaluated.
var	String	No	Optional. The name of the variable to hold the value of the current element.
varStatus	String	No	Optional. The name of the variable to hold a <code>LoopTagStatus</code> object.
begin	int	Yes	Optional. The first index, 0-based.
end	int	Yes	Optional. The last index, 0-based.
step	int	Yes	Optional. Index increment per iteration.

This action is used in Example 15-3 to extract the text from all `<category>` elements and build a sorted list of unique category names, that is then used to generate an HTML selection list:

```
<jsp:useBean id="uniqueCats" class="java.util.TreeMap" />
<x:forEach select="$doc/meerkat/story/category" var="category">
  <!-- Need to convert the XPath node to a Java String --%>
  <x:set var="catName" select="string($category)" />
  <c:set target="{uniqueCats}" property="{catName}" value="" />
</x:forEach>
```

A `<jsp:useBean>` action creates a `java.util.TreeMap` to hold the list. By using a map, the list of category names is automatically trimmed to unique names, since the keys in a map must be unique.* The `TreeMap` is a map type that sorts its keys, taking care of the sorting requirement. The XPath expression used for the `<x:forEach>` action matches all `<category>` elements. The action then evaluates its body once per element node, where the `<c:set>` action adds a map entry with the text value as the key and an empty string as the value.

An important detail here is that the value of the loop variable (`category`) contains an instance of an XPath node object, not the string needed for the Map. One way to convert an XPath node to a string is to use the XPath `string()` function. That's what I do here. The `<x:set>` action (Table 15-7) converts the current node to a Java String and saves it as a variable that is then used by `<c:set>` to set the Map entry. Tricks like this are unfortunately needed to bridge the XPath and Java domains in some cases.

* A `java.util.TreeSet` would actually be more appropriate, but there is no JSTL action that can add elements to a set.

Table 15-7. Attributes for JSTL `<x:set>`

Attribute name	Java type	Dynamic value accepted	Description
select	String	No	Mandatory. An XPath expression to be evaluated.
var	String	No	Mandatory. The name of the variable to hold the value of the current element.
scope	String	No	Optional. The scope for the variable; one of page, request, session, or application. page is the default.

You can look at Example 15-3 to see how a `<c:forEach>` action is then used to loop over all map entries and use the key values to build the HTML select list.

Next we need to decide which stories to display. This is also accomplished with the help from the `<x:set>` action:

```
<c:choose>
  <c:when test="{empty param.selCat || param.selCat == 'ALL'}">
    <x:set var="stories" select="$doc//story" />
  </c:when>
  <c:otherwise>
    <x:set var="stories"
      select="$doc//story[category = $param:selCat]" />
  </c:otherwise>
</c:choose>
```

The JSTL core `<c:choose>` action with nested `<c:when>` and `<c:otherwise>` actions tests the value of the `selCat` request parameter. The first time the page is requested, this parameter is not present. In this case, as well as when it has the value `ALL`, an `<x:set>` action with an XPath expression that matches all `<story>` elements (and their subnodes) extracts the data to be displayed and saves it in a variable named `stories`.

If the user selects a specific category and clicks the **Filter** button, however, the `selCat` parameter is received with the request. In this case, another `<x:set>` action extracts only the `<story>` elements that match the selected category. It does this by using an XPath expression that contains a predicate with a Boolean expression:

```
$doc//story[category = $param:selCat]
```

XPath processes this expression by first collecting all nodes matching `//story` in the context represented by the `doc` variable, and then removing all nodes where the Boolean expression evaluates to false. In the Boolean expression, the text for the `<category>` element of each selected node is compared to the value represented by the `$param:selCat` variable: the `selCat` request parameter value.

The final part of the sample application loops over the selected nodes and generates an HTML table, with a light green background for the cells that contains stories in the General category:

```
<table>
  <x:forEach select="$stories">
```

```

<tr>
  <x:choose>
    <x:when select="category[. = 'General']">
      <td bgcolor="lightgreen">
        </x:when>
      <x:otherwise>
        <td>
          </x:otherwise>
        </x:choose>
        <a href="{x:out select='link' }">
          {x:out select='title' }</a>
        <br>
        <i>{x:out select='timestamp' }</i>:
        <b>Category:</b>{x:out select='category' },
        <b>Reported by:</b>{x:out select='channel' }
        <br>{x:out select='description' }
      </td>
    </tr>
  </x:forEach>
</table>

```

The `<x:forEach>` action is again used to loop through the set of nodes, but as opposed to when it was used to create the category name list, the current element is not exposed to the body as a variable. This illustrates another `<x:forEach>` feature, namely that the action adjusts the current XPath context seen by nested JSTL XML actions. When the body is evaluated, the current context for XPath expressions is the current node. An expression such as `category[. = 'General']` used by the nested `<x:when>` action, is therefore evaluated in the context of the current story node, checking the value of its `<category>` element. The expression evaluates to true if the text in the `<category>` element equals the string “General”. The `<x:out>` actions use similar XPath expressions to extract data from the current `<story>` element.

The last part of Example 15-3 also illustrates the use of most of the conditional JSTL XML actions: `<x:choose>`, `<x:when>`, and `<x:otherwise>`. They have the same function as the corresponding JSTL core elements; `<x:choose>` groups a number of `<x:when>` actions and optionally one `<x:otherwise>` action, where the body of the first `<x:when>` action with a `select` attribute that evaluates to true, or the `<x:otherwise>` body if none of them do, is processed. Only the `<x:when>` action has attributes, described in Table 15-8.

Table 15-8. Attributes for JSTL `<x:when>`

Attribute name	Java type	Dynamic value accepted	Description
<code>select</code>	String	No	Mandatory. An XPath expression to be evaluated as a Boolean.

The result of the XPath expression in the `select` attribute is converted to a Boolean using the XPath `boolean()` function; any valid number except 0, a nonempty string, and an expression that matches at least one node is converted to true. All other

values are converted to false. Note that this means that the string “false” evaluates to true.

The only JSTL XML action I don’t use in this example is `<x:if>`, described in Table 15-9.

Table 15-9. Attributes for JSTL `<x:if>`

Attribute name	Java type	Dynamic value accepted	Description
select	String	No	Mandatory. An XPath expression to be evaluated as a Boolean value.
var	String	No	Optional. The name of the variable to hold the Boolean result.
scope	String	No	Optional. The scope for the variable; one of page, request, session, or application. page is the default.

It works exactly like the corresponding action in the JSTL core library, except that the `select` attribute is evaluated as `XPath boolean()` the same way as for `<x:when>`.

The examples in this chapter show how the JSTL XML actions let you process XML documents pretty much any way you can think of. You can transform a document using a stylesheet, parse and access parts of the document in many ways, save a part as a variable, or add it to the response. As illustrated by the examples in this chapter, you can mix the JSTL XML actions with the other JSTL actions (or custom actions) and use application variables and request data in XPath expressions to select parts based on runtime conditions.