

# 4

## A Quick Start to JSP and XML Together

### IN THIS CHAPTER

- The Relationship Between XML and JSP
- Java XML/XSL APIs

The goal of this chapter is to throw you right into the JSP/XML mix. This chapter will show you how easy it is to mix XML and XSLT together within a JSP. A second goal is to introduce the various APIs that enable you to use XML within Java.

The approach this chapter takes is to start with a brief discussion of how XML and JSP relate. Then the chapter will introduce the reader to three basic APIs: JAXP, Xerces, and Xalan. Using these Java coded libraries, we will expand what we learned from Chapter 2, “Introduction to XML/XSL,” and put it into practice with a JSP environment. After discussing the code from these simple examples, we can turn up the heat and show a fuller example of building a newsletter processor that creates a weekly news page. Finally, the chapter shifts focus to describing the other APIs that a JSP/XML programmer will find in common use.

### The Relationship Between XML and JSP

Using JSP and XML together makes quite a bit of sense. JSP is all about character-based output and XML is all about character-based data; mixing the two together ends up producing a natural match. Simply understanding XML is not enough, because XML is about marking up the data, not about moving the data around. JSP allows us to move data into and out of an XML data source. To use XML successfully in a project, we must understand how to pull and push character data around within a JSP. Several common ways of using JSP and XML together are listed here:

- The data is stored within an XML file. Using JSP, the data is imported and then converted to an HTML-based page to be viewed by a user with a browser.
- The data is stored within a database. A Java process is built to read the data from the database and convert it to an XML document. Once the data is stored as XML, either the data is shipped to the user as an XML file, or the XML file is transformed to another format such as an HTML page to be viewed by a user.
- Data is stored within an XML file. Java is used to parse the file and format the data for database insertion. Once the data is in the database, a JSP is used to access and view the data.
- Web application initialization information is stored within an XML file. The JSP reads the file and changes application behavior based on what it finds in the file.
- Data is transferred between two systems in an XML-formatted file. JSP is used as a front-end interface to initiate the process of sending or receiving the data.

These examples point out that data is being pushed and pulled around and converted to various formats. It's important to learn to use JSP as a layer to facilitate the transport of data using either XML or some other means.

### A Warning

This is the appropriate time to give a warning regarding XML. XML is a wonderful creation, and it can do amazing things. However, be warned that it is not the appropriate solution to a number of problems. Used appropriately, it can improve performance and save time. Misused, XML *will* slow down Web applications, waste development time, and cost money.

Reporting systems are generally good places to use XML and XSL because of the filtering and formatting that can be implemented on the same datasets to achieve different reports. Examples of this can be found in Chapter 11, "Using XML in Reporting Systems," and Chapter 12, "Advanced XML in Reporting Systems." Also, data stored as XML within a Web page can be used to allow the client to reorganize the data in the reports without needing the server again. Examples of this can be found in Chapter 13, "Browser Considerations with XML."

In addition, XML gives an abstraction layer between business logic and presentation logic. This means that changing a report only requires updating an XSL stylesheet while the data generation level remains untouched by changes. Likewise, if the business objects generating the report change, it's possible to still generate the same XML, so the presentation layer still works. The XML layer ends up working like a rubber gasket that permits your Web application to flex with less effort.

As stated earlier in the book, XML is not designed to be a replacement for a database. While XML files can be used to store data, they are best used for initialization files or small data stores. Using XML to store and retrieve large amounts of data is a slower method of data storage than using a database. Databases provide additional features, such as transaction management, that are not part of XML. When designing larger systems using XML, you should balance the design relative to XML strengths and weaknesses. XML doesn't replace a database; rather, it complements a database. Our system designs should reflect and use each technology to the benefit of the other. In later chapters, we will demonstrate blending the two technologies together.

There are times when using XML as a data store may be a valid choice. Say, for instance, you need to enable many clients to view unchanging data without requiring either a database or network connection. A very simple system could be developed relatively quickly that would use a browser, XML, and some stylesheets to achieve this goal.

Sometimes projects convert data to XML just to use XML. This is a bad idea, because the process of creating and using XML adds another layer of processing to a project. This will both increase costs due to development time and slow down processing. Data should only be converted to XML when the use of XML can be shown to provide the best solution.

The basic rule to use is this: You should never transform data into XML format if the data isn't being packaged for a purpose. Don't use XML for XML's sake.

### **JAXP, Xerces, and Xalan**

Many tools exist for the processing and transformation of XML. Each has strengths and weaknesses for differing circumstances. However, you have to start somewhere, so we will evaluate the APIs that are most commonly used with JSP. This translates into writing an example using the JAXP API. This API will use the Xerces Java parser and the Xalan stylesheet processor. If you haven't yet placed the `xerces.jar` and `xalan.jar` files in the `lib` directory found in the root of your Tomcat installation, do so now. Download locations follow.

#### **Xerces (XML Parser)**

Simply described, Xerces is an XML parser and generator. Named after the Xerces Blue butterfly, it implements the W3C XML and DOM (Levels 1 and 2), as well as the SAX standard.

Xerces is included with Tomcat, and therefore you will not need to install it. However, there may be times when the newest version is necessary for newly added functionality. Use the following URL to download the newest version of Xerces: <http://xml.apache.org/xerces2-j/index.html>. To date, Tomcat has been keeping pace with Xerces.

DOM and SAX are different ways in which to process XML and will be discussed in Chapter 5, “Using DOM,” and Chapter 6, “Programming SAX.” Simply put, DOM is the creation of a tree-like structure in memory that can be programmatically manipulated. SAX is a sequential event-driven XML reader.

This book uses Xerces version 1.4.3.

#### **Xalan (XSLT Processor)**

Xalan is an XSLT stylesheet processor. Named after a rare musical instrument, it implements the W3C XSLT and XPath recommendations. Sadly, Xalan is not included with Tomcat, so we must install it. Download Xalan from <http://xml.apache.org/xalan-j/index.html>. Place the `xalan.jar` file in the `lib` directory found under the Tomcat installation. Stop and restart the server so that Tomcat registers the new classes.

The examples in this book use Xalan version 2.2.d10.

#### **JAXP (Generic XML Processing)**

JAXP is technically an API for XML processing. More accurately, it's an abstraction layer because it contains no parsing functionality. JAXP is a single API through which to access the various supported XML and XSL processors. It's a lot like JDBC for XML parsers. It does not provide any new functionality to XML processing, except that it provides a common way to use different XML processor implementations. Through JAXP, several of the industry standard XML processor implementations may be used, such as SAX and DOM.

In order to use JAXP effectively, it's important to use its API only, and avoid directly using any of its underlying implementation-dependent APIs. The reason for this comes from the fact the JAXP layer is an abstraction layer. Any direct calls to the underlying XML APIs defeats the purpose of having the generic layer. This rule is especially important if your application will be deployed in different situations and if each deployment needs to use a different underlying XML processor. JAXP enables your application to be XML processor independent.

JAXP was created to facilitate the use of XML on the Java platform. It does this through the common interface to whichever XML processor is chosen.

JAXP comes bundled with the Tomcat server, and therefore requires no installation. If you want to review the source of JAXP or check the details of the latest version, check out <http://java.sun.com/xml/jaxp.html>.

This book uses JAXP version 1.1.

**An Example with JAXP, XSL, and XML** This example simulates the randomly generated banner ads that are seen all over the Web. The difference is that this version is simplified to use only text. In Chapter 1, “Integrating JSP and Data,” we created a

similar example using various methods. This version's data will be stored in a static XML file. Using JAXP we will apply an XSL stylesheet to the XML document and output the results. As a simple starting point, the example in this section will display all the data contained in the XML file. Then in the example in the next section, the code will randomize which link is displayed.

Let's begin with the XML data file, shown in Listing 4.1. This and all files throughout this chapter should be saved in the Tomcat webapps directory with the path `webapps/xmlbook/chapter4/BannerAds.xml`.

**LISTING 4.1** BannerAds.xml; XML Data File for JAXP Example

---

```
<?xml version="1.0"?>
<BANNERS>
  <BANNERAD>
    <NAME>JSPInsider</NAME>
    <LINK>http://www.jspinsider.com</LINK>
    <LINKTEXT>JSP News</LINKTEXT>
  </BANNERAD>
  <BANNERAD>
    <NAME>Sun</NAME>
    <LINK>http://www.sun.com</LINK>
    <LINKTEXT>The Home of Java</LINKTEXT>
  </BANNERAD>
  <BANNERAD>
    <NAME>SAMS</NAME>
    <LINK>http://www.sampublishing.com</LINK>
    <LINKTEXT>Java Books</LINKTEXT>
  </BANNERAD>
  <BANNERAD>
    <NAME>Jakarta</NAME>
    <LINK>http://jakarta.apache.org</LINK>
    <LINKTEXT>Kewl Tools</LINKTEXT>
  </BANNERAD>
</BANNERS>
```

---

Notice that this file has four groups of data, each of which contains information for a banner link. Each group of data is properly nested under the root element `BANNERS`.

Next, the XSL stylesheet, shown in Listing 4.2, needs to be built to apply the formatting to the XML file. Briefly, this stylesheet creates a table row for each child `BANNERAD` found within the root element `BANNERS`. Then the text data found at various nodes is selected and placed within table cells.

**LISTING 4.2** BannerAds.xml; XSL Stylesheet for JAXP Example

---

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<HTML>
<HEAD><TITLE>Banner Ads With JAXP</TITLE></HEAD>
<BODY>
  <TABLE border="1">
    <TR>
      <TH>Name</TH>
      <TH>Link</TH>
      <TH>LinkText</TH>
    </TR>
    <xsl:for-each select="BANNERS/BANNERAD">
      <TR>
        <TD><xsl:value-of select="NAME" /></TD>
        <TD><xsl:value-of select="LINK" /></TD>
        <TD><xsl:value-of select="LINKTEXT" /></TD>
      </TR>
    </xsl:for-each>
  </TABLE>
</BODY>
</HTML>
</xsl:template>
</xsl:stylesheet>

```

---

This XPath `select` statement begins by selecting all `BANNERAD` children of the root element `BANNERS`.

```
<xsl:for-each select="BANNERS/BANNERAD">
```

This causes the body of this `for-each` element to be processed for each element that is matched through the `select` expression.

Now it's time to build the JSP that uses JAXP to apply the stylesheet transformation to the XML file. The code is shown in Listing 4.3.

**LISTING 4.3** BannerAds\_JAXP.jsp; JSP for JAXP Example

---

```

<%@ page
  import="javax.xml.transform.*,
        javax.xml.transform.stream.*,
        java.io.*"
%>

```

---

**LISTING 4.3** Continued

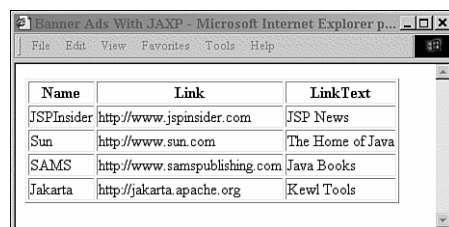
```
<%  
String ls_path = request.getServletPath();  
    ls_path = ls_path.substring(0,ls_path.indexOf("BannerAds_JAXP.jsp")) ;  
  
String ls_xml = application.getRealPath(ls_path + "BannerAds.xml");  
String ls_xsl = application.getRealPath(ls_path + "BannerAds.xsl");  
  
StreamSource xml = new StreamSource(new File(ls_xml));  
StreamSource xsl = new StreamSource(new File(ls_xsl));  
StreamResult result = new StreamResult(out);  
  
TransformerFactory tFactory = TransformerFactory.newInstance();  
Transformer transformer = tFactory.newTransformer(xsl);  
transformer.transform(xml, result);  
%>
```

This page begins by declaring those packages used by the code. The first package, `javax.xml.transform.*`, is the package that contains the `Transformer` and `TransformerFactory` classes. These classes transform a source tree, the XML and XSL, into a result tree. The next package declared, namely `javax.xml.transform.stream.*`, is used for the `StreamResult` and `StreamSource` classes. The `StreamSource` is the XML source being handled. It makes no difference to us whether it is a file being read in a stream of data or an existing representation of the XML data. The `StreamSource` takes care of the details for us in this matter. The `StreamResult` represents where we want to send our final XML document. These classes serve as holders for the XML and XSL files before transformation, and for the final output after processing. Last, we have the `java.io.*` package, which handles external file management.

The JSP then requests the path to itself. This path check is performed so that we can be sure that we are using the proper path when we grab the XML and XSL files. Once we have the path, we use it to put the XML and XSL files into the `StreamSource` objects that serve as holders. The `StreamResult` object is built to send the results of the transformation. In this page, the example just needs to send the output directly to the JSP's output, or the `out` implicit object of type `javax.servlet.jsp.JspWriter`.

All of the work we've done so far just sets up everything for the XML/XSLT transformation to occur. At this point, the code creates an instance of the `TransformerFactory` object. Factories are used to create specific transformations. A factory sets up all the reusable information, which the XML process will need later when we perform the actual XSL transformation of the XML data. We use the `TransformerFactory` to create a specific `Transformer`. Each `Transformer` created

handles a single XSL file. If we had five different XSL files to process, we would create five transform instances, or reinitialize a single Transformer five times. So when creating the Transformer object, we tell it which XSL file to use. Then we call the transform method to process an XML document. This requires that we know which XML file to process (which `StreamSource` object) and also where to send the output (the `StreamResult`). The overall transformation only used a few lines of JAXP. Figure 4.1 shows the results.



Name	Link	LinkText
JSPInsider	<a href="http://www.jspinsider.com">http://www.jspinsider.com</a>	JSP News
Sun	<a href="http://www.sun.com">http://www.sun.com</a>	The Home of Java
SAMS	<a href="http://www.samspublishing.com">http://www.samspublishing.com</a>	Java Books
Jakarta	<a href="http://jakarta.apache.org">http://jakarta.apache.org</a>	Kewl Tools

**FIGURE 4.1** Results of JAXP transformation.

The ability to display an XSLT transformation is very powerful, but let's expand our example to have some more functionality. A real banner ad system would only display one randomly selected banner ad at a time. We'll modify our example to accommodate this behavior.

**An Example with XSL Parameters** We will begin by adding a parameter tag to the XSL stylesheet. This tag will create a parameter that will tell the stylesheet which banner ad to display. The value of this parameter will be set through our JSP (more specifically, through JAXP). Listing 4.4 shows the new stylesheet with the changes noted in boldface print.

**LISTING 4.4** BannerAds\_Param.xsl; Externally Set Parameter

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="Choose" />
<xsl:template match="/">
<HTML>
<HEAD><TITLE>Banner Ads With Parameter</TITLE></HEAD>
<BODY>
  <TABLE border="1">
    <TR>
      <TH>Name</TH>
      <TH>Link</TH>
      <TH>LinkText</TH>
    </TR>
<xsl:for-each select="BANNERS/BANNERAD[ $Choose ]">
```

**LISTING 4.4** Continued

```
<TR>
  <TD><xsl:value-of select="NAME" /></TD>
  <TD><xsl:value-of select="LINK" /></TD>
  <TD><xsl:value-of select="LINKTEXT" /></TD>
</TR>
</xsl:for-each>
</TABLE>
</BODY>
</HTML>
</xsl:template>
</xsl:stylesheet>
```

Within the stylesheet element we've added a parameter element named Choose.

```
<xsl:param name="Choose" />
```

This parameter element has global scope for the entire stylesheet because it was placed outside an `xsl:template` tag. Normally when a parameter is declared it is also set to a default value with the `select` attribute. We chose not to use a default value; if we had used one, it would look like this:

```
<xsl:param name="Choose" select="2" />
```

The other place that parameters may be declared is in the beginning of a template. In this case, the parameter scope is limited to within the template in which it resides.

The next changed part of the original stylesheet from Listing 4.2 is the `select` statement of the `for-each` loop.

```
<xsl:for-each select="BANNERS/BANNERAD[$Choose]">
```

This XPath `select` statement still begins by selecting all `BANNERAD` children of the root element `BANNERS`. Now, however, only the `BANNERAD` element whose position number matches our parameter in the square brackets will continue into the `for-each` loop. This is, in effect, only selecting one of all the possible options to be output through code found within this loop.

**NOTE**

To reference a parameter anywhere within its scope, use a dollar sign before the case-sensitive parameter name.

Finally, we have a new JSP, shown in Listing 4.5. This page still contains the original code from Listing 4.3 with some added processing. We now need to find out how many banner ads are contained within the XML file, and then randomly choose a number within that range. Once that number is chosen, we have to set the parameter in the XSL stylesheet so only that data is displayed.

**LISTING 4.5** BannerAds\_Param.jsp; Elements and XSL Parameters

```
<%@ page
  import="javax.xml.transform.*,
         javax.xml.transform.stream.*,
         java.io.*,
         javax.xml.parsers.*,
         org.w3c.dom.*"
%>

<%
String ls_path = request.getServletPath();
      ls_path = ls_path.substring(0,ls_path.indexOf("BannerAds_Param.jsp"));

String ls_xml = application.getRealPath(ls_path + "BannerAds_Param.xml");
String ls_xsl = application.getRealPath(ls_path + "BannerAds_Param.xsl");

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setValidating(false);

DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(new File(ls_xml));

NodeList nodes = doc.getElementsByTagName("BANNERAD");
int intNodeLength = nodes.getLength();

int intRandom = 1 + (int) (Math.random() * intNodeLength);

StreamSource xml = new StreamSource(new File(ls_xml));
StreamSource xsl = new StreamSource(new File(ls_xsl));

StreamResult result = new StreamResult(out);

TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer transformer = tFactory.newTransformer(xsl);
```

**LISTING 4.5** Continued

---

```
transformer.setParameter("Choose", new Integer(intRandom));

transformer.transform(xml, result);
%>
```

---

In this new JSP we added two packages. The first package, `javax.xml.parsers.*`, gives us access to the JAXP DOM builder implementations and thus allows us to create a tree-like structure in memory using the `DocumentBuilder` and `Document` classes. The other package is `org.w3c.dom.*`. This package allows us to access the DOM definition of the W3C. Through this we can create a set of chosen nodes using the `NodeList` object.

At this point we create an instance of the `DocumentBuilderFactory` object. Once created, this object will be used to create a `Document` object into which we will parse the XML. Notice the line

```
factory.setValidating(false);
```

This line is important, as it determines whether an XML file is validated as it loads or not. Because this file doesn't have a DTD or schema, we will leave validation off.

In short, validation means that we are comparing the XML document structure to a definition of that structure. If they are the same, the document is valid.

Now that the XML file has been loaded into the `Document` object, we are ready to figure out how many banner ads' worth of data exists in the XML file:

```
NodeList nodes = doc.getElementsByTagName("BANNERAD");
int intNodeLength = nodes.getLength();
```

In looking at the XML file shown in Listing 4.1, we see that each `BANNERAD` element contains information for one banner ad. We create a `NodeList` and select only the elements called `BANNERAD`. This `NodeList` now contains a single node for each banner ad. Using `getLength()` in this list returns the total number of banner ads.

**NOTE**

---

At this point we would like to mention that counting child elements is a bad idea when large amounts of data are involved. It is fine for the XML documents we are using because they are small. However, when working with large XML documents, loading the entire document into memory just to count elements would result in a huge performance hit.

---

Using the length of the list, it is possible to use the random method with that number as a scaling factor to obtain a random number:

```
int intRandom = 1 + (int) (Math.random() * intNodeLength);
```

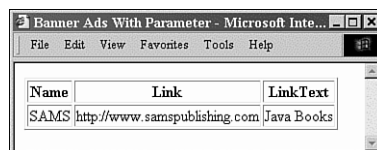
Then the code sets the parameter in the stylesheet:

```
transformer.setParameter("Choose", new Integer(intRandom));
```

We are setting the parameter named Choose within the XSL stylesheet to the random number from the last step. It's this number that will select which banner ad to display.

This is just the beginning. Using similar techniques, we could determine which data a user gets to see based upon the user's login security rating. It is also possible to pass in parameters that change the way an XSL stylesheet formats reports or that change sorting order dynamically. You will learn how to implement these options later when we begin to delve into the realm of creating XML documents based on data obtained from a database.

Figure 4.2 shows the output produced by this JSP. The output, of course, is random, so you will not see the same thing each time the JSP is executed.

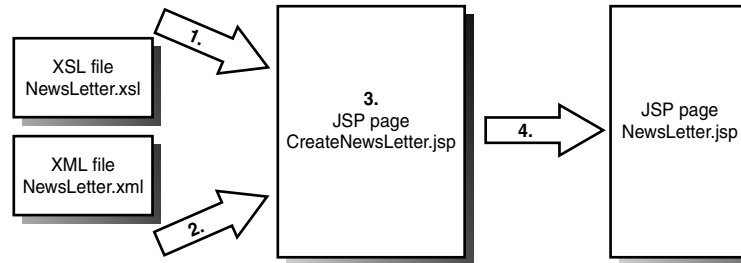


**FIGURE 4.2** Results of random banner ad.

**Outputting Transformations to a JSP File** There may be times when processing XML and XSL files is redundant because the same output is created repeatedly. An example of this would be a newsletter. Let's say we have a Web site that posts a newsletter. The contents of that newsletter, which are in XML, change once a week, but the rest of the time they remain the same. In this case, it would be a waste of processing power to transform the XML with the XSL stylesheet each time a user requested the page. Instead, we should output the transformation results into another JSP file and have the user request the preprocessed page. This way, we can make the transformation happen only when the XML source file has changed by calling the processing page.

Now, let's look at a JSP that processes an XML and XSL file and places the results into another JSP. The processing page is called `CreateNewsLetter.jsp` and the results are placed in `NewsLetter.jsp`. This example is loosely based on the JSPBuzz newsletter, which is published at the JSP Insider Web site.

The flow of logic is shown in Figure 4.3.



**FIGURE 4.3** Creating a new file from output.

The steps illustrated in Figure 4.3 are as follows:

1. CreateNewsLetter.jsp loads the XSL document.
2. CreateNewsLetter.jsp loads the XML document.
3. The XML and XSL are transformed into the results.
4. The results are put into Newsletter.jsp and the file is closed.

The code will create a newsletter from an XML data source and an XSL stylesheet. The newsletter will have articles and links related to Java and XML. We will output the results from the processing JSP to another JSP file so that the transformation won't be called repeatedly and reprocessed redundantly. Using this design, the links in the Web site can point to the output file instead of the processing file.

We will begin with a new XML file found in Listing 4.6. This XML file has header information and content information. Save this file as `webapps/xmlbook/chapter4/NewsLetter.xml`.

**LISTING 4.6** NewsLetter.xml; Data for Newsletter Example

```

<?xml version="1.0" encoding="UTF-8"?>
<newsletter volume="II" issue="15">
  <header>
    <title>Newsletter Example</title>
    <description>Demonstrate XML/XSL Transformation Output
    Into JSP File</description>
    <date>12.14.2001</date>
  </header>
  <article position="1">
    <author>Dennis M. Sosnoski </author>
    <title>XML in Java : Document models, Part 1: Performance</title>
  </article>
</newsletter>
  
```

**LISTING 4.6** Continued

---

```

    <description>A must read article for anyone playing around with Java
      and XML. It's both a review of current XML parsers and comparison on
      performance results. Great article.</description>
    <link>http://www-106.ibm.com/developerworks
      /xml/library/x-injava/index.html</link>
    <date>September 2001</date>
  </article>
  <article position="2">
    <author>Marshall Lamb</author>
    <title>Generate dynamic XML using JavaServer Pages technology</title>
    <description>A good article describing how to use JSP to create dynamic
      content beyond HTML, such as XML.</description>
    <link>http://www-106.ibm.com/developerworks/library/j-dynxml.html</link>
    <date>December 2000</date>
  </article>
  <links position="1">
    <title>Tomcat Home Page</title>
    <link>http://jakarta.apache.org/tomcat/index.html</link>
  </links>
  <links position="2">
    <title>W3C XML Home Page</title>
    <link>http://www.w3.org/XML/</link>
  </links>
  <links position="3">
    <title>Java Tutorial</title>
    <link>http://java.sun.com/docs/books/tutorial/</link>
  </links>
</newsletter>

```

---

Next we have the XSL stylesheet. This file, shown in Listing 4.7, should be saved in `webapps/xmlbook/chapter4/NewsLetter.xsl`.

**LISTING 4.7** NewsLetter.xsl; Newsletter Stylesheet

---

```

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

<xsl:template match="/">
  <html><head><title>Example Newsletter</title></head>
  <body bgcolor="#ECFCEA">

```

**LISTING 4.7** Continued

```

<center><font size="4pt"><b>
  <xsl:value-of select="newsletter/header/title" /><br/>
  Volume&#160;<xsl:value-of select="newsletter/@volume" />&#160;#
  <xsl:value-of select="newsletter/@issue" /><br/>
  <xsl:value-of select="newsletter/header/date" /><br/>
  <xsl:value-of select="newsletter/header/description" />
</b></font></center>
<table border="0" width="100%">
  <tr>
    <td bgcolor="#9EC49A">
      <b><font color="#000000">Table of Contents</font></b>
    </td>
  </tr>
  <tr>
    <td>
      <br/><a href="#article">Articles of Interest</a>
      <ul>
        <xsl:for-each select="newsletter/article" >
          <li><a href="#article{@position}">
            <xsl:value-of select="title" /></a>
          </li>
        </xsl:for-each>
      </ul>
    </td>
  </tr>
  <tr>
    <td><a href="#article">Useful Links</a>
      <ul>
        <xsl:for-each select="newsletter/links" >
          <li><a href="#links{@position}">
            <xsl:value-of select="title" /></a>
          </li>
        </xsl:for-each>
      </ul>
    </td>
  </tr>
</table>
<table width="100%"><tr>
  <td bgcolor="#9EC49A"><a name="article">
    <font color="#000000"><b>Articles of Interest</b></font></a>
  </td></tr>
</table>
<xsl:apply-templates select="newsletter/article" />

```

**LISTING 4.7** Continued

---

```

<table width="100%"><tr>
  <td bgcolor="#9EC49A"><a name="links">
    <font color="#000000"><b>Useful Links</b></font></a>
  </td></tr>
</table>
<xsl:apply-templates select="newsletter/links" />
</body>
</html>
</xsl:template>
<xsl:template match="newsletter/article">
  <table style="border-style:groove;border-width:2px;" width="100%">
    <tr><td>
      <table border="0" width="100%">
        <tr><td colspan="2">
          <a href="{link}" name="article{@position}">
            <xsl:value-of select="title"/></a>
          </td>
        </tr>
        <tr>
          <td width="50%"><xsl:value-of select="author"/>
          </td>
          <td><xsl:value-of select="date"/></td>
        </tr>
        <tr>
          <td colspan="2"><xsl:value-of select="description"/></td>
        </tr>
      </table>
    </td></tr>
  </table><br/>
</xsl:template>

<xsl:template match="newsletter/links">
  <table border="0" width="100%">
    <tr>
      <td><a href="{link}" name="link{@position}">
        <xsl:value-of select="title"/></a>
      </td>
    </tr>
  </table>
</xsl:template>
</xsl:stylesheet>

```

---

**More on XSL Templates** This stylesheet uses most of the same elements introduced in Chapter 2, “Introduction to XML/XSL.” However, there are a couple of new things that we need to go over. The first of these is the use of templates. In Chapter 2, we introduced templates. Again, templates provide the means by which we match and extract data from an XML document. The `template` tag’s `match` attribute determines what of the XML document will make it into the template element to be matched. Here’s an example:

```
<xsl:template match="/">
```

This template tag, found near the beginning of Listing 4.7, selects the root of the XML document using `/`. The entire XML document will make it into this template to be processed.

However, templates can be defined that only format or transform a select part of an XML document. Notice that an XPath statement is used in a template’s `match` attribute to determine which part of an XML document will be selected into the template. This means that we can be as specific or as general as XPath expressions allow in selecting which XML elements will be matched into a template.

In the stylesheet in Listing 4.7, we have created three templates. The first one is discussed earlier in this section, and selects the entire XML document. The other two each format only the `article` and `links` child elements of the root. These templates are defined towards the bottom of the listing. The article template looks like this:

```
<xsl:template match="newsletter/links">
  <table border="0" width="100%">
    <tr>
      <td><a href="{link}" name="link{@position}">
        <xsl:value-of select="title"/></a>
      </td>
    </tr>
  </table>
</xsl:template>
```

Notice that the `select` attribute of this template tag only chooses those `links` elements that are children of the root element `newsletter`. Templates are very helpful in keeping large stylesheets organized and modular. Using a different template to format specific elements makes it easier to find and quickly update or change formatting rather than searching through a single large template.

---

**NOTE**

If there are no templates that match parts of an XML document, the unmatched parts of the document will be output as text data without the markup. This is because the default template for elements discards the tags and outputs the text. Imagine, for example, the

stylesheet from Listing 4.7 without the first template that matches the root. If you'd like to see what this looks like, once we have our JSPs running (later in the chapter), comment out the first template and see what happens.

---

The first part of the XML document output will be the text data from the elements unmatched in any templates. In this example, that is the header element and children. (This text is output first because processing occurs from start to finish in the order of the source XML file when there are no templates that match the root.) Next, the `link` and `article` elements, which match the remaining templates, are formatted accordingly.

Finally, let's look at another new template-related tag:

```
<xsl:apply-templates select="newsletter/article" />
```

This element provides a way to state at a particular point within another template that you'd like to apply the template that formats those elements indicated by the value of the `select` attribute of this tag. In this case, it's calling the template that formats the `article` children of the root `newsletter` element. In Listing 4.6 these are used to output the result of the called template in the appropriate place.

**More on Accessing XML Element Data** Now we need to change gears away from templates to discuss another new syntax that is used in Listing 4.7. Let's say an HTML page is being created from the XML/XSL transformation. How can we select XML text values from a tag to place within the `href` attribute of HTML? Well, we can use the `value-of` tag, right? That would create something looking like this:

```
<a href="<value-of select="elementname" />">
```

Do you see the problem with this? It is not valid XML because the tags are not properly nested, or more specifically, a tag is opened inside another tag. We need a way around this. It turns out that we have a shorthand notation that uses curly braces to delimit the node that needs to be selected. This notation is used in the stylesheet in Listing 4.7 and looks like the following:

```
<td><a href="{link}" name="link{@position}">
```

The shorthand is used twice in the above code snippet. The value of the `href` attribute of the anchor tag becomes the text data from the currently selected `link` element of the XML document. The name of the anchor tag is the combination of the string `link` and the value of the `position` attribute found in the XML document. Isn't that clever?

Now that we understand the XML and XSL files, let's move on to the JSP. This page is very similar to the JSPs found in Listings 4.3 and 4.5. The only difference is that instead of putting the result of the transformation into the output buffer through the `StreamResult` object, we are going to put it into another JSP file.

Listing 4.8 shows the source code for the JSP file that will transform the XML with the XSL stylesheet. The differences between it and Listing 4.3 are noted in boldface type. Save this file as `webapps/xmlbook/chapter4/CreateNewsLetter.jsp`.

**LISTING 4.8** CreateNewsLetter.jsp; Outputting to File

---

```
<%@ page
  import="javax.xml.transform.*,
        javax.xml.transform.stream.*,
        java.io.*"
%>

<%
  String ls_path = request.getServletPath();
  ls_path = ls_path.substring
    (0,ls_path.indexOf("CreateNewsLetter.jsp")) ;

  String ls_JSPResult = application.getRealPath
  (ls_path + "NewsLetter.jsp");
  String ls_xml = application.getRealPath(ls_path + "NewsLetter.xml");
  String ls_xsl = application.getRealPath(ls_path + "NewsLetter.xsl");

  FileWriter l_write_file = new FileWriter(ls_JSPResult);

  StreamSource xml = new StreamSource(new File(ls_xml));
  StreamSource xsl = new StreamSource(new File(ls_xsl));

  StreamResult result = new StreamResult(l_write_file);

  TransformerFactory tfactory = TransformerFactory.newInstance();
  Transformer transformer = tfactory.newTransformer(xsl);
  transformer.transform(xml, result);

  l_write_file.close();
%>
<jsp:forward page="NewsLetter.jsp"/>
```

---

This file starts out the same way as the other JSP built in this chapter. We get the path to the JSP file itself and then use that path to create a string that states where each of the XML, XSL, and result JSP files resides. This assumes that all files will reside in the same directory. The difference this time is that we are creating an extra string for the result JSP file:

```
String ls_JSPResult = application.getRealPath(ls_path + "NewsLetter.jsp");
```

Next, we create a `FileWriter` object that will be used to write a character file, namely our result file called `NewsLetter.jsp`:

```
FileWriter l_write_file = new FileWriter(ls_JSPResult);
```

Then `StreamSource` objects are again used to hold the XML and XSL. The `TransformerFactory` object is used to create the transformer that will apply the stylesheet to our XML file and put the result into our `StreamResult` object, which is outputting to the JSP file.

Now that the transformation is complete, we need to close the JSP file that contains the results:

```
l_write_file.close();
```

And last but not least, we will perform a page forward so that we can see the results. Using the following JSP action element will redirect the server to a different file to send to the requesting Web browser:

```
<jsp:forward page="NewsLetter.jsp" />
```

Normally, we will be using the JSP file named `NewsLetter.jsp` to view our newsletter. This file doesn't perform any transformations; rather, it is the result of that transformation. After running the `CreateNewsLetter.jsp` file, go and look at the source of `NewsLetter.jsp`. This is a flat HTML file that has the result of the transformation and looks like Figure 4.4 in a browser.

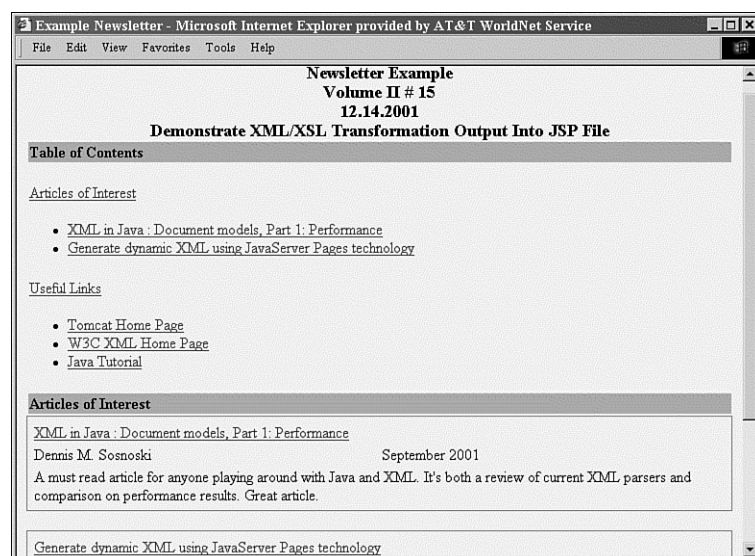


FIGURE 4.4 Result of running `CreateNewsLetter.jsp`.

Now you may ask why we output the result as a JSP, when this example only has HTML? Typically, there will be other processing that has to happen on the destination JSP. As an example, on the JSPBuzz newsletter where a similar process occurs, the final output page has special JSP include and footer files to deal with our standard site header and footers. The advantage of this system is that the newsletter only performs the processing that has to happen on every call, and the actual XML/XSL transformation only happens once every two weeks with the publication of the newsletter.

**Lots of XML and XSL and Not Much JSP** Now we need to spend some time discussing the bigger picture of what we just covered. You may wonder why we are spending so much time with what goes on within the XML and XSL. In fact, the JSP part of this chapter seems quite minimal. The fact is that JSP can be viewed as glue that holds everything together.

A typical JSP/XML page requires only a little bit of JSP code to hold everything together, and requires lots of XML and XSL to produce the results. So coverage of XML and XSL is as important as JSP, since in these examples the XML and XSL really constitute the majority of the work that needs to be performed. The true secret of JSP design is the knowledge of how to weave together the many elements before they are sent to a client. The JSP logic in most JSP applications is actually the smallest portion of your logic.

Most of the logic on any JSP is distributed to other processes such as the database, XSLT transformations, JavaBeans, Web services, and other external processes. JSPs are used just to hold it all together. The fact is that the JSP element is the simplest part of the learning equation. The first big step toward becoming a great JSP programmer is realizing that your skills are more about all the different technologies being brought together and being able to weave them to produce an output for the client. You might say that the best JSPs are the ones that contain the least JSP code and have the greatest modularity.

### JSP and XML: An Overview

Where do we stand? Earlier in this chapter we examined JAXP and JSP. JAXP lets us easily access XML and apply XSL to the XML files. JAXP will also allow event-driven types of XML processing. However, we need to begin peeling away the layers of the processing that we performed in the examples.

First, we have XML, which is a language used to create documents. These documents are a mixture of data and the markup used to describe the stored data. Then JSPs are used as a scripting template for running code on the server side.

The problem is that a JSP and an XML document are not in the same place at the same time. In the code, it looks as though it's all in one place. However, the JSP is a process being executed in memory, while the XML file is stored in another location.

This means that JSP never directly works on an XML file. Rather, we must create a *representation* of the XML data and use it for the code to access. A document representation is a logical construct of the XML document, which resides in memory. We have several options in the types of representations we can use, but in the end, all are based on some model representation of the data.

JAXP is wonderful in that it permits us to easily transform an XML file with XSLT. In all of our earlier examples, the main effort was geared towards supporting these XSLT transformations to an XML file. However, what do we do when we need to modify the XML file? Or what happens when we want to build some output by using Java logic rather than an XSLT transformation? In these cases we need to scratch the surface of JAXP and look at what is happening underneath the JAXP polish.

In one example, the code actually obtained some specific data from the XML file and then appended it to the XSL document. However, the code didn't explain what was happening behind the scenes to get to the data. In this case, we were accessing other APIs more directly to examine the XML files. This leads us to our next discovery: when we dig deeper, we find a whole alphabet soup of APIs that we can use to accomplish different tasks with XML. You may wonder whether it's really necessary to know all the APIs. We only need to know about the major APIs, but each one brings a solution to a different problem. This means that we should examine the primary APIs that are available for our use.

## Java XML/XSL APIs

This section covers the Java APIs that we haven't yet discussed. In later chapters, we will use several of these APIs to make life easier for XML users.

### DOM (XML Document Object Model)

The DOM will be covered in detail in Chapter 5, "Using DOM." The DOM represents the standard W3C representation of an XML document. The advantage that the DOM offers is that it's a standard, and the same DOM methods that work within Java code will work within a Web browser using JavaScript, or within a C application. The DOM is very widely understood and has extensive support in the user community.

The biggest problem that the DOM has is that it creates large in-memory representations of XML documents. This makes it slow and often impractical when dealing with large XML documents because the entire XML document has to be parsed before processing can begin. This large footprint was a driving force in the creation of many of the other XML APIs because a definite need exists to handle larger documents.

The most common Java DOM parser is Xerces. Another frequently encountered DOM parser is Crimson. However, Xerces 2.0 is considered the replacement DOM parser for Crimson. As mentioned earlier, Xerces 1.4.3 forms the best baseline to use as a DOM parser for this book. This is due to the fact that Tomcat 4.0 uses version 1.4.3 of Xerces. Xerces 2.0 is still very much in development at the time of this book's writing.

The home page for the DOM specification is <http://www.w3.org/DOM/>.

### **SAX (XML Parser)**

The Simple API for XML (SAX) is a tool for event-driven XML parsing. SAX is widely used, and it will be covered in detail in Chapter 6, "Programming SAX."

Briefly, SAX enables programmers to parse through an XML document by defining *events*. An event is a callback from the SAX parser when certain conditions are met while the SAX parser is reading sequentially through an XML document. Users can take any action they desire on the callback, from reading the data in the XML file to acting on the data. In the end, it's the programmer's decision whether to create the callbacks. Also, it is up to the programmer to determine what action to take upon receiving a callback from SAX. SAX is quickly becoming a lower level API. This means that while many programmers will use SAX, it will be indirectly through another tool or API such as JAXP. Using SAX directly is a lot of work, but it is fast and efficient. Many of the other APIs will use SAX as a quick way to read in an XML file. In fact, many XML tools will use SAX as the default XML parser because of speed considerations. However, you should always check your tool's documentation to be sure, as the default parser can vary from tool to tool.

The home page for SAX can be found at <http://www.saxproject.org/>.

### **JDOM (XML Document Representation)**

JDOM is a method of creating a Java representation of an XML document.

JDOM has several classes that permit it to use SAX or the DOM to read XML data into its own Java representation. JDOM supports many of the commonly found features that the DOM supports. Unlike many of the other APIs discussed in this section, which use Java interfaces to define the Java representation, JDOM uses concrete classes to define the Java representation. This approach has the advantage of making JDOM simpler to use out of the box than either SAX or the DOM. It also gives JDOM the disadvantage of limited flexibility in implementation. This gives JDOM a different flavor than most of the other APIs. One result is that you won't see factory objects within JDOM.

JDOM makes heavy use of the Java Collection API, especially the List and Map Java interfaces, to define its representation. This means that a programmer can use standard Java structures to navigate around the JDOM representation.

JDOM doesn't offer XML parsing; instead, you must use SAX or read in a DOM representation. JDOM offers several classes that make it easy to use SAX or the DOM as a data source.

JDOM is listed as JSR-102 at the Java Community Process Web site.

The JDOM project can be found at <http://www.jdom.org/>.

### **dom4j (XML Document Representation)**

dom4j is an open source project that has a set of Java APIs used for handling XML, XSLT, and XPath.

This API is similar to JDOM in the sense that it creates a Java-based XML document representation. However, it also differs as it uses Java interfaces. Using interfaces offers advantages (a flexible and expandable design) and disadvantages (you have several extra layers to learn) to the dom4j API. This API permits the use of Java Collection objects, which makes navigating the XML structure easy for an experienced Java programmer.

With dom4j it's easy to use the DOM standard while also providing an easy-to-use Java document representation. Also, dom4j is able to easily integrate with JAXP, as is SAX and DOM. An extremely nice feature of dom4j is the fact that it incorporates an event-based model for the processing of XML documents. This permits a dom4j parser to handle large XML files in sections rather than as a single document. This means that an entire XML document doesn't need to be in memory (a plus when dealing with a huge XML file). We will use this feature in Chapter 14, "Building a Web Service."

The dom4j project can be found at <http://dom4j.org/>.

### **JAXB (Parser and XML Document Representation)**

Java Architecture for XML Binding (JAXB) is used to automate mapping between XML documents and Java. JAXB's unique angle is to use the XML schema (currently the DTD, and in future releases the XML schema) to build an optimized Java class to perform the parsing of the XML document representation. This permits JAXB to automatically use these customized Java classes to process an XML document faster than SAX, while giving a document representation of the XML data structure, which is similar to DOM. The drawback is that the resultant XML document representation is only functional against the XML schema it was generated to match. JAXB uses a binding language to permit the programmer to optimize the resulting XML document representation relative to the schema and design needs.

JAXB is a solution that works best when you need a document representation that requires fast content validation, and the XML document is stable in format.

At the time of writing, this is still a young project. However, it will most likely end up being one of the standard APIs to use in XML document handling.

The JAXB project can be found at <http://java.sun.com/xml/jaxb/>.

JAXB is listed as JSR-031 at the Java Community Process Web site.

## Summary

This chapter gave you a first taste of using XML and JSP together. JAXP makes it easy to combine an XML file and an XSL stylesheet to produce new pages. However, XSLT is only part of the picture in using XML and JAXP. In exploring what else we can do, it quickly becomes apparent that there are many APIs available. Each API offers unique features, advantages, and disadvantages. While on paper we could discuss the merits of each, it's within the code that we will find out the benefits of the various tools. In the next chapters, you will learn how to programmatically examine, modify, and act upon an XML document in detail.

In working with JAXP, we discovered a few surprising facts about JSPs. The first is that the secret to using JSPs is more about weaving multiple technologies together than about using JSPs. The second is that solid JSP design is about modularity and a JSP is really just a glue or template to connect the modules together.