

12

JSP and XML

XML and JSP are two important tools available in producing a web application. This chapter examines the potential of mixing these two technologies in order to enhance the capabilities of JSP. While this chapter will cover many things about XML, this chapter will not attempt to teach XML. Instead it focuses on how JSP and XML can be used together as a highly flexible and powerful tool. In general the usage of XML in these examples will be kept simple and should cause no problems for users who are starting XML.

In short the chapter will be broken down into five main sections:

- ❑ **A quick look at XML**
Why is XML valuable? Before even dealing with XML combined with JSP we need to understand why it would be beneficial to do so. As mentioned this is not going to be a tutorial on how to write your own XML and XSLT. Instead the first section will be dealing with concepts of XML and its implementation in your project.
- ❑ **An overview of Java-XML tools**
Using XML with JSP is much easier if you have the right tools. Before diving right in to some examples this section will give a brief overview of some of the most popular Java-XML tools. Along with overviews we will also cover which tools this chapter requires and where to get them.
- ❑ **Focus on the DOM, JDOM and SAX**
Several pre-built Java based code libraries are available to access XML. This section will go more in depth about dealing with the Document Object Model, Java Document Object Model and Simple API for XML. While DOM, JDOM, and SAX can be of great aid to a developer, the reader should understand the benefits and drawbacks for each API. This section will cover the DOM in the greatest detail as it can be considered to be the baseline standard for working with XML.

- ❑ **A Step By Step Tutorial**
The best way to learn is to walk through and build some useful code. This section will show you a practical example on how JSP and XML can be combined to work together on a project. The best part is the code will be reusable for any project. The tutorial will help you create a JSP tag library to use with XML.
- ❑ **JSP Documents**
A review of the merging of XML and JSP in the JSP 1.2 Specification. All of the examples up to this point are implemented using the JSP 1.1 specifications simply because most developers are already familiar with them. JSP 1.2 shows great promise in allowing JSP to be authored in a fully XML compliant syntax. This section is devoted to understanding the new XML based JSP syntax.

What Is XML?

Besides being a common buzzword, what exactly is XML? Before diving right in to the code let's take some time to examine what exactly XML is and what it is good for. For those of you that are already quite familiar with XML this section should only need a skim. However, if XML is completely new to you then this section will explain why XML is so important and give a brief introduction to XML.

XML stands for Extensible Markup Language. The official XML recommendation is made by the W3C and is publicly available at the W3C's website, <http://www.w3.org/XML/>. Reading through the entire XML recommendation can be quite tedious so we will summarize some of the most important points:

- ❑ XML is a markup language that is designed for easy use over the Internet. XML is compatible with the SGML (Standard Generalized Markup Language) specifications and can be easily created, edited or viewed by a simple text editor.
- ❑ XML markup gives data a logical structure that is both easily human-legible and easily processed by applications. While XML markup may resemble other markup languages, such as HTML, here is where a big difference can be seen. An application using XML can verify a document's structure before using the document's content, via either a Document Type Definition (DTD), or a schema. If an XML document is malformed then an application can identify the error before producing an undesired result. However, this doesn't concern us in this chapter.
- ❑ Optional features in XML are kept to an absolute minimum, currently zero. This means that an XML document will be universally accepted by any XML compliant parser or application. Porting an XML document between operating systems or projects will not require a syntax change for compatibility.
- ❑ XML is a syntax for defining data and meta-data. It allows you to self describe and serialize information in a universal method. This is one of the most important features of the XML specification. Consider the fact that literally everything can be described in terms of data.

As an example, even a programming language could have its rules and definitions defined with XML. This means you could use XML to form and describe any programming language. In fact the JSP 1.2 spec allows for just that and your JSP can now be coded as XML. Why is this important? This means we will be able to apply the tools we use in XML to many new tasks which would have been harder to perform in the past. We will examine this idea a little more towards the end of the chapter.

So the critical word is 'data'. XML doesn't change the data we use, it merely gives us a way to store and describe it more easily. XML gives us a way to store items that in the past we might not have thought of as data, but now can express in XML as a collection of data. It is this standard way of defining data and storing data that empowers XML. This means over time as programmers, we will use XML to replace other methods of storing and using data. Many of the techniques we have honed over the years are still applicable, it is just we have a new format to apply these skills against.

While XML has many benefits it can still be difficult to understand these benefits especially if you have never used XML. To clarify let's examine a mock case where initially using an XML compatible language saved a lot of work later on.

The Value of XML: An Example

Imagine you are the webmaster of an online publication. The publication has been around for years and consists of thousands of HTML pages. Since you are quite the HTML guru, each HTML page has been crafted to look perfect for the average computer screen. Then one day you walk in and are told every page needs to be changed so they could appear in a paper based book.

The new format poses quite a problem. When constructing the site it was satisfactory to make each page look good on the average web browser. Now each page needs to have its content extracted and reformatted for the book. If all the pages share an identical layout a custom built utility to change the formatting might be a solution, however no fore thought was given to strictly following a standard structure.

While all the pages are coded in a similar fashion they still have enough difference to toss out a custom code-changing tool. The only working solution is to manually go through each page and copy the content. Not only is this inefficient but also the amount of work would easily overburden a single webmaster.

The importance of a common format should now be fairly easy to recognize, but one could still argue that the project above did use a common HTML structure for all of the documents. There is no fault in this argument. Only a misunderstanding of what we are defining as a strictly followed and standard format. For our definition a standard format should allow for clear and easy understanding for both a person and a program.

HTML falls short of our standard format because it does not enforce a common coding syntax throughout a document. HTML tag attributes can be surrounded by quotes or not. Some HTML tags have optional ending tags. HTML even allows for markup syntax to be intermixed with content to be displayed. All of these little allowances work for what HTML was intended for; however, they make it much more difficult for a program to work with the markup correctly.

With a few changes HTML could easily be made in to a format that is easier on a program. The changes might require all attribute values to be surrounded by quotes, all tags to have a clear start and end and markup to be clearly separated from content.

By requiring all of these little changes HTML would provide the same functionality but have a more clearly defined format. Because of the more clearly defined format a program to read HTML would need to do less guessing at optional rules and could display content correctly following the strict rules. In fact this is exactly where XML comes in to play. Don't think of XML as some totally new and different technology. Instead think of XML as enforcing a strict format on markup that does not require a loss of functionality.

One of the most powerful aspects of XML is its ability to define a language that follows these strict formatting rules. In fact XML has already been used to do this for the above issues with HTML. XHTML almost identically resembles normal HTML, but is made using XML for a strict format and structure. Since XHTML also complies with XML standards it may also easily be used by any utility built to support XML. The official XHTML recommendation is hosted publicly at the W3C's website, <http://www.w3.org/TR/xhtml1/>.

If the troubled webmaster from above had used XHTML he would have a much easier job changing the pages in to new formats. Keep in mind XML is a markup language for easy reading and understanding. XML does not restrict what a program does with the information after it is read. The webmaster could design a custom utility that followed XML rules and performed the format conversion automatically, or the webmaster could go out in search utilities already built to read XML and change its format.

Here is where XML shines some more and the next section proves its worth. The above webmaster would not have to search far for utilities that work with XML. Many developers, companies and other individuals have already decided to support XML and have created software to use its functionality. Some of the most current and popular free software will be reviewed in the next section. We will also take a look at what software will be required to use the XML examples from this chapter.

Useful Tools for XML and JSP

Objects are used to represent data in Java. XML is a mark-up language, but by itself it does nothing, so it must be parsed into a Java object before it is useful to a Java programmer. Fortunately many fine free implementations of Java XML parsers already exist.

Here is an overview of some of the tools used in the examples of this chapter. Each overview includes a location on the Internet where you can find the tool. Most of the tools listed are open-source and all the tools are freely available for your use.

XSLT

XSLT is an XML defined language for performing transformations of XML documents from one form in to another. XSLT by itself does not do much, but relies on other software to perform its transformations.

XSLT is very flexible and becoming quite popular; however, it does not have the same level of support as XML. A few good utilities are available for XSLT and will be listed below. For the XSLT examples in this chapter we are using the default XSLT support that is packaged with the JAXP 1.1 release.

The official XSLT recommendations are made by the W3C and are publicly available at the W3C's website, <http://www.w3.org/Style/XSL/>.

JAXP

JAXP is meant to be an API to simplify using XML within Java. The JAXP isn't built to be an XML parser. Instead, it is set up with a solid interface with which you can use any XML parser. To further aid developers it does also include a default XML parser.

The JAXP supports XSL transformations and by default uses the Apache Group's Xalan and part of Sun's Project X, renamed to Crimson, for XSLT. Sun and the Apache Group are cooperating for Java XML functionality and because of this Crimson was donated to the Apache Group for future integration with XML projects.

Just about every example in this chapter requires that you have the JAXP resource files available to your JSP container. If you do not have the JAXP 1.1 release installed we recommend you do so now before trying out any code examples.

To download or learn more about JAXP you can visit the Sun web site,
<http://java.sun.com/xml/download.html>.

JDOM

JDOM is an XML utility designed to create a simple and logical Java Document Object Model representation of XML information. The W3C DOM, which we will cover more in depth later, creates a fully accurate representation of a document and is sometimes thought of as too complex.

JDOM simplifies the DOM by only covering the most important and commonly used aspects of the DOM. By taking this approach JDOM is both faster and easier to use but at the cost of limited functionality compared to the standard W3C DOM. While JDOM doesn't have all the features of DOM it does have more than enough features to be a solid tool for a Java-XML developer.

JDOM is only required for the JDOM specific section and the final example of this chapter. You will need to download at least JDOM beta 5 to try those examples, but you do not need it for the rest of the chapter.

To download or learn more about JDOM you can visit the JDOM organization site, <http://www.jdom.org/>.

Xerces

Xerces is the Apache Group's open-source XML parser. Xerces is 100% W3C standards compliant and represents the closest thing to a reference implementation of a Java parser for the XML DOM and SAX.

JAXP comes with packaged support for XML parsing by Crimson. Crimson does not have the widespread support and documentation of Xerces, but if you would like to try another XML parser with JAXP then Xerces is recommended. The JDOM portion of the chapter use Xerces and it is included within the JDOM package.

To download or learn more about Xerces you can visit the Apache XML site,
<http://xml.apache.org/xerces-j/index.html>.

Xalan

Xalan is the Apache Group's open-source XSLT processor for transforming XML documents into HTML, text or other XML document types. Xalan implements the W3C Recommendations for XSL Transformations. It can be used from the command line, in an applet or a Servlet or as a module for other programs.

Xalan is packaged with the normal JAXP 1.1 so you will not need to download it separately for use with examples in this chapter.

To download Xalan or learn more about it visit the Apache Group's Xalan webpage,
<http://xml.apache.org/xalan-j/index.html>.

Other Software

All of the examples in this chapter are built using the Tomcat 4 beta release. Earlier versions of Tomcat will work for all the examples except the ones found in the JSP in XML syntax section.

If you do not have a JSP container or would like to download Tomcat visit the Apache Group's Jakarta project website; here is the address, <http://jakarta.apache.org/tomcat/index.html>.

Before continuing on we feel there is need for a word of caution. Tomcat already uses some of the same tools we listed above. Chapter 19 and Appendix A discuss in some detail how classloading works in Tomcat; the easiest way around any potential problems is to just dump all of the JAR files from the JAXP 1.1 release into each web applications's WEB-INF\lib directory.

If you do continue and get a 'sealing violation' error, it probably means you have conflicting JAR files. Double check your environment resources and fix any duplications of JAR or class files.

Extracting and Manipulating XML Data With Java

There is not one be all and end all way of accessing XML data with Java. The JAXP supports two of the most commonly used methods know as the Document Object Model (DOM) and the Simple API for XML (SAX). In addition to the support found in the JAXP the Java Document Object Model (JDOM) is also becoming a commonly used and popular method. At the writing of this material only the DOM is a formal recommendation by the W3C.

This section will briefly give an introduction to these three methods and then compare the advantages and disadvantages of using each.

Extracting XML Data with the DOM

The first example is fast and easy to code. In this example we will examine how to a parse and expose XML information using the JAXP with a JSP page. This example is only geared towards showing how to construct a Java object from an XML document. In a production system you would use a set of JavaBeans to perform most of the work being done within this JSP page. We are keeping the first example simple on purpose to illustrate the much-repeated process of parsing XML to Java. In future examples we will incorporate this code into a JavaBean for repeated use in our JSP pages.

We first need a sample XML document and some code to parse it. The sample XML document will be a simple message. All XML files required for this chapter are referenced as being in the `C:/xml/` directory. If you are copying examples verbatim place all XML files in this directory.

Here is `message.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<messages>
  <message>Good-bye serialization, hello Java!</message>
</messages>
```

Next we need to parse the XML file into a Java object. The JAXP makes this easy requiring only three lines of code. Define a factory API that allows our application to obtain a Java XML parser:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
```

Create a `DocumentBuilder` object to parse an `org.w3c.dom.Document` from XML:

```
DocumentBuilder db = dbf.newDocumentBuilder();
```

Call the `parse` method to actually parse the XML file to create our `Document` object:

```
Document doc = db.parse("c:/xml/message.xml");
```

As noted in the JAXP API documentation, the `Document` object supports the Document Object Model Level 2 recommendations of the W3C. If you are a W3C standard savvy individual the above three lines of code are all that is needed to place this Java object into your field of experience. If you are not familiar with the W3C's recommendation don't worry. Later in the chapter we will spend some time getting acquainted with the standard DOM as well as some of the other options available for using XML with JSP.

For now it is important to understand that the DOM is a model to describe your data. The whole and only purpose of the DOM is to be a tool for manipulating data. The DOM comes with methods and properties with which you can read, modify and describe the data that the DOM models.

The model the DOM uses is a tree structure of nodes. These nodes are the placeholders for the data and everything else contained in the DOM. For example, if you wanted to reference the overall data tree you would reference the document node, but if you wanted to reference some comments about the data file you could check the comment nodes.

Keeping that brief introduction of the DOM in mind let's finish retrieving our example message. From the `Document` object we can get a `NodeList` object that represents all of the elements in our XML document named `message`. Each slot in the `NodeList` is a single node that represents a message element:

```
NodeList nl = doc.getElementsByTagName("message");
```

A `NodeList` can be thought of as an array starting from 0 and going up to the length of the array. We know this example only has one message element so the list should have only one node for the message element. To return a node from a `NodeList` the `item()` method is used with the index of the node wanted. In this case `item(0)` of the `NodeList` representing the message element would return the first message in the example XML file:

```
Node my_node = nl.item(0);
```

Once the first node is retrieved it is possible to query it to get more information about that node. In this case we would like to get the message stored within the node. Fortunately convenient self-named methods such as `getNodeValue()` are available for extracting this data:

```
String message = my_node.getFirstChild().getNodeValue();
```

Some readers may ask why we have to use the `getFirstChild()` method when our example node has no attributes or another node besides the text. The reason for this comes from the fact that with the W3C DOM data representation of the node really has more sub-nodes in its tree-like structure. The one sub-node we are interested in contains the message text. After calling `getFirstChild()` the desired text node is returned and we can use `getNodeValue()` for our message.

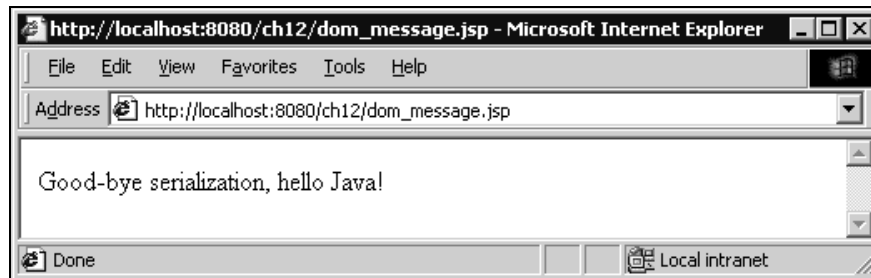
Here is where a difference can be seen between the various parsers. In this simple example the DOM is tracking many pieces of information we really don't need. JDOM creates a simpler representation of the XML file. This means JDOM uses less memory to represent the XML file and the function calls would be easier.

Since we know no other sub-nodes are present in the message node why even bother with the text node? JDOM would let us concentrate on a simpler model. The standard W3C DOM provides these sub-nodes for extra flexibility regardless of the programming language or situation. JDOM on the other hand is built specifically for Java and ease of use. However, the ease JDOM provides comes at the loss of some of the standards such as these sub-nodes. In the long run either API would accomplish the same goal.

Putting all of the above together gives us a JSP that will read in our XML document and display the message. Here is the code for `dom_message.jsp`:

```
<%@ page contentType="text/html"%>
<%@ page import="javax.xml.parsers.DocumentBuilderFactory,
    javax.xml.parsers.DocumentBuilder,
    org.w3c.dom.*"
%>
<%
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse("c:/xml/message.xml");
NodeList nl = doc.getElementsByTagName("message");
%>
<html>
<body>
<%= nl.item(0).getFirstChild().getNodeValue() %>
</body>
</html>
```

The output for this example should be a plain HTML page that says the example message. Here is a screenshot of our results:



The above example should help illustrate what an XML parser does and what exactly a DOM is, but don't think the DOM is restricted to the above example. Let's look a little closer at the Document Object Model and the flexibility it provides.

Focusing on the DOM

The Document Object Model is an important and commonly used object when dealing with XML. Remember how earlier we mentioned the DOM is a tool for creating a structure to represent data. Having a complete and well-defined structure is what allows us to both manipulate the data and the structure itself. Now let's learn a little more about the DOM and how it is used.

A document never starts off as a DOM object for use with Java. Instead a data source must be processed and converted into a DOM object. For practical purposes in Java, the DOM object is the intersection of a data object, such as an XML file, and your Java. The intersection formed provides a JSP programmer's interface to the XML document.

Over the years several different DOM objects have been created to handle different document types. This can make it confusing to understand the exact nature of a DOM object. When we use the term DOM we are referencing the standard W3C DOM built to support an XML structured document.

The W3C Document Object Model Level 2 Core Specification can be found at, <http://www.w3.org/DOM/>.

For the next example we will continue to use the W3C standard DOM object in the JAXP. Keep in mind while the DOM is based upon a recommendation and is a specification, we are also using Java-based libraries to create a DOM representation. This can be confusing since we are referring to the DOM as both the specification and the Java representation.

In the next section we are reviewing the objects that comprise the Java representation of the DOM. We will present a brief overview for some of the most commonly used objects and methods. Keep in mind this is not a complete listing by any means. This list will serve to make these objects familiar for a future example. For a complete list, reference the documentation with your JAXP download or visit Sun's web site for the online version, <http://java.sun.com/xml/docs/api/>.

Common DOM Objects

Below are some of the commonly used DOM objects found in the `org.w3c.dom` package. Each object has a short description along with a list of relative method information for our examples.

Node

A node is the primary data type of the DOM tree. An object with the `Node` interface implements methods needed to deal with children objects but is not required to have children. Some common objects with the `Node` interface are `Document` and `Element`:

Method	Description
<code>appendChild(org.w3c.dom.Node)</code>	Adds a child node to this node and returns the node added
<code>getFirstChild()</code>	Returns the first child of the node if it exists
<code>getNextSibling()</code>	Returns the node immediately after this node
<code>getNodeName()</code>	Returns the name of the node depending on its type (see API)
<code>getNodeTypes()</code>	Returns the node's type (see API)
<code>getNodeValue()</code>	Returns the value of the node

Element

Elements are an extension of the `Node` interface and provide additional methods similar to the `Document` object. When retrieving nodes by using the `getElementsByTagName()` method often times a cast to the element type is needed for further manipulation of sub-trees:

Method	Description
<code>getElementsByTagName(String)</code>	Returns a <code>NodeList</code> of all of the elements with the specified tag name.
<code>getTagName()</code>	Returns a <code>String</code> representing the tag name of the element.
<code>getAttribute(String)</code>	Returns a <code>String</code> value of the attribute. Caution should be used because XML allows for entity references in attributes. In such cases the attribute should be retrieved as an object and further examined.
<code>getAttributeNode(String)</code>	Returns the attribute as an <code>Attr</code> object. This <code>Attr</code> may contain nodes of type <code>Text</code> or <code>EntityReference</code> . See API.

Document

The document object represents the complete DOM tree of the XML source:

Method	Description
<code>appendChild(org.w3c.dom.Node)</code>	Adds a node to the DOM tree
<code>createAttribute(String)</code>	Create an <code>Attr</code> named by the given <code>String</code>
<code>createElement(String)</code>	Creates an element with a name specified by the given <code>String</code>
<code>createTextNode(String)</code>	Creates a node of type <code>Text</code> that contains the given <code>String</code> as data
<code>getElementsByTagName(String)</code>	Returns a <code>NodeList</code> of all of the elements with the specified tag name
<code>getDocumentElement()</code>	Returns the node that is the root element of the document

NodeList

The `NodeList` interface acts as an abstraction for a collection of nodes. A `NodeList` can be thought of much like an array. Any item in the `NodeList` may be manipulated by making reference to its index in the list:

Method	Description
<code>getLength()</code>	Returns the number of nodes in the list
<code>Item(int)</code>	Returns the specified node from the collection

Putting the DOM to Work

With the next example we will use some of the above objects and methods. Instead of explaining the syntax for each bit of code we will focus on what exactly the code is doing. If you get lost on syntax just reference the above section.

For the next example we will create a JSP to verify the status of a DOM full of URLs. The JSP page will have a small form for adding or clearing the URLs from our DOM. Ideally for this example we would like to stash a Document object throughout the session. In the Java API the Document object is only an interface. We will need an object implementing the Document interface for this example to work. In the JAXP the XmlDocument is an ideal object to use and it can be found within the `org.apache.crimson.tree` package.

Before going farther we should warn you the Crimson documentation isn't easily found. The JAXP 1.1 does not bind a specific XML parser or XSLT processor to itself. As a result the documentation for these two parts of the JAXP is found from the suppliers of the XML and XSLT tool sets used within JAXP.

The Apache Group happens to be the owner of the XML parser and XSLT processor that comes packaged by default with the JAXP 1.1. At the time of writing the Apache web site lacks pre-built documentation for the Crimson package. If you would like to make your own documentation you can download the Crimson source files from the Apache Group and run the javadoc utility yourself. For your aid we will also javadoc the Crimson source files and include the documentation files with this chapter's download. Xalan, the default XSLT processor with the JAXP 1.1, has excellent documentation but we will get to that later.

As mentioned at the start of this section, the first part of this example stashes a DOM tree to the session context. See if you can pick out where the XmlDocument object is used.

The `dom_links` JSP

Here is the code for `dom_links.jsp`. The three lines of code below use the XmlDocument object and are the same three we have been using throughout the chapter. Because the Crimson package is being used `db.newDocument()` creates an XmlDocument object even though we treat it as a W3C compliant Document object:

```
<%@ page
  import=" org.w3c.dom.*,
  javax.xml.parsers.*" %>
<%
  DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
  DocumentBuilder db = dbf.newDocumentBuilder();
  Document doc = db.newDocument();
```

The new code below places our DOM tree within the session and then creates a root node so we can add URLs:

```
session.setAttribute("doc", doc);

Element newLink = doc.createElement("root");
doc.appendChild(newLink);
%>
<jsp:forward page="dom_links_checker.jsp" />
```

With our object stashed in the session the request is forwarded to a JSP that will check and modify our DOM.

The `dom_links_checker` JSP

Here is the code for `dom_links_checker.jsp`:

```
<%@ page
import="org.w3c.dom.*,
javax.xml.parsers.*,
java.net.*"%>
<html>
```

The easy part comes first; two simple HTML forms. One form will add the URL submitted while the other will clear all the set of URLs:

```
<table>
<tr>
<td colspan="2">
<form action="dom_links_checker.jsp" method="post">
Add a url: <INPUT name="add" size="25">
</td>
</tr>
<tr>
<td align="center"><INPUT type="submit" value=" Send "></form></td>
<td align="center">
<form action="dom_links_checker.jsp" method="post">
<INPUT name="clear" type="hidden" value="true">
<INPUT type="submit" value=" Clear List">
</form>
</td>
</tr>
</table>
<%
```

After the forms we need to add the functionality in our JSP. In order to manipulate our tree of URLs it must first be snagged from the session:

```
org.w3c.dom.Document doc = (org.w3c.dom.Document)session.getAttribute("doc");
```

Next we need some code for adding URLs from the form. For adding a URL we must first make a new element in our DOM. After a `url` element is created we then toss a text node in with the URL. You can see we name each of these elements "url" for convenience. Later on we will retrieve every `url` element to check the actual URL:

```
if (request.getParameter("add") != null)
{
Element newLink = doc.createElement("url");
org.w3c.dom.Text linkText =
(org.w3c.dom.Text)doc.createTextNode(request.getParameter("add"));
newLink.appendChild(linkText);
doc.getDocumentElement().appendChild(newLink);
}
```

When the clear button is clicked our tree of URLs will be reset. Removing all the URLs from our DOM is as easy as looping through and taking out each `url` element:

```

if (request.getParameter("clear") != null)
{
    int count = doc.getElementsByTagName("url").getLength();
    for(int i = 0; i < count; i++)
        doc.getDocumentElement().removeChild(doc.getElementsByTagName("url").item(0));
}

```

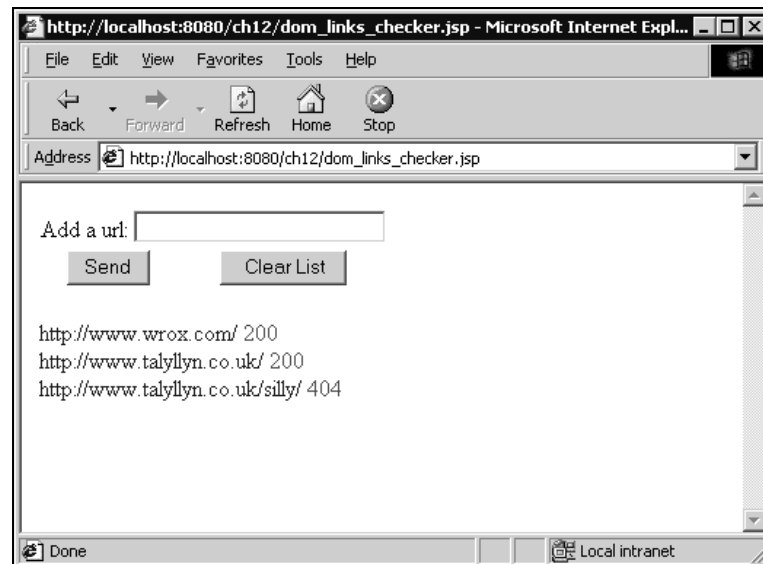
After making our changes to the DOM object, we still need to verify the URLs stored within the DOM object are valid. The following code loops through all our `url` elements and performs a quick connection to see if they are available over the Internet. The only addition from above is that a URL is created and checked for each `url` element. As the URLs are validated the code returns the name of the URL, and the response code for the URL connection attempt is sent back to the user:

```

for(int i = 0; i < doc.getElementsByTagName("url").getLength(); i++)
{
    URL url = new
    URL(doc.getElementsByTagName("url").item(i).getFirstChild().getNodeValue());
    HttpURLConnection link = (HttpURLConnection)url.openConnection();
    %>
    <font color="blue">
    <%= doc.getElementsByTagName("url").item(i).getFirstChild().getNodeValue() %>
    </font>
    <font color="red"><%= link.getResponseCode() %></font><br />
    <% } %>
</html>

```

Just about everyone knows that a 404 response-code means trouble, however you should expect to see the "OK" 200 code if you typed in a real URL. Here is a screen shot after we typed in a few URLs:



Now the above example seems easy, but we haven't gained much over a simple array. The power of a tool based on a DOM would be that it could read any XML source. If we were maintaining a web site with all the links in XML compatible format we could use a JSP page to check the entire site.

Following that thought let's create an XML file of URLs to plug in with `dom_links.jsp`. The XML source will not only be helpful to this example but later we will reuse it to generate things like an HTML page for a web browser and a WML page for WAP devices.

The URL File

Here is the code for `links.xml`:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<links>
```

The document is a set of links. Each link element represents a URL and some important information pertaining to it. The first link is for Wrox publishing:

```
<link>
  <text>Wrox publishing</text>
  <url newWindow="no">http://www.wrox.com</url>
  <author>Wrox</author>
  <date>
    <day>1</day>
    <month>1</month>
    <year>2001</year>
  </date>
  <description>Check out Wrox for more books.</description>
</link>
```

The next link is structured identically but with information for JSP Insider:

```
<link>
  <text>JSP Insider</text>
  <url newWindow="no">http://www.jspinsider.com</url>
  <author>JSP Insider</author>
  <date>
    <day>2</day>
    <month>1</month>
    <year>2001</year>
  </date>
  <description>A JSP information site.</description>
</link>
```

Another link, but this time for Sun Microsystems main Java page:

```
<link>
  <text>The makers of Java</text>
  <url newWindow="no">http://java.sun.com</url>
  <author>Sun Microsystems</author>
  <date>
    <day>3</day>
    <month>1</month>
    <year>2001</year>
  </date>
  <description>Sun Microsystem's website.</description>
</link>
```

A final link to the JSP container reference implementation:

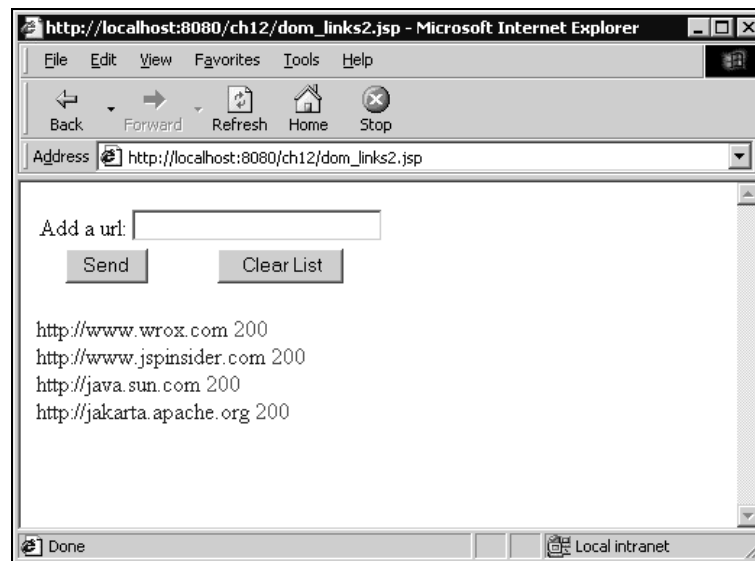
```
<link>
  <text>The standard JSP container</text>
  <url newWindow="no">http://jakarta.apache.org</url>
  <author>Apache Group</author>
  <date>
    <day>4</day>
    <month>1</month>
    <year>2001</year>
  </date>
  <description>Some great software.</description>
</link>
</links>
```

To plug the XML source in to our `dom_links.jsp` we need to change three lines of code. Here is the new code for `dom_links2.jsp`:

```
<%@ page
  import=" org.w3c.dom.* ,
  javax.xml.parsers.*" %>
<%
  DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
  DocumentBuilder db = dbf.newDocumentBuilder();
  Document doc = db.parse("c:/xml/links.xml");

  session.setAttribute("doc", doc);
%>
<jsp:forward page="dom_links_checker.jsp" />
```

With that fix here is the screen shot after running `dom_links2.jsp`:



Now the value of a DOM in `dom_links2.jsp` can be seen over a simple array. Instead of reading a file we could tweak `dom_links2.jsp` one more time to accept request parameter specifying an XML compatible source. The source could then be from a client, database or just about anything else.

DOM: Pros and Cons

After the above example you should know enough to start working on your own with the DOM; however remember what we said in the beginning: there is not one be all and end all way of accessing XML data with Java.

With that in mind, let's look at a few reasons to use the DOM as well as some of the limitations of the DOM:

- ❑ The DOM is very flexible and generic. The W3C DOM can describe many different documents, including anything in XML syntax. Since the DOM provides such broad support it can be thought of as a generic tool, especially when dealing with XML.
- ❑ By gaining skills with the standard W3C DOM you can apply them wherever a W3C DOM might appear. For example, many browsers are now supporting the W3C DOM. Currently Mozilla and Opera both have excellent support for the W3C DOM and IE has fairly good support as well. Using client-side scripting such as JavaScript you can use the same DOM manipulating methods described in the previous section.
- ❑ A DOM is not customized for any one type of project. The memory requirements of a standard DOM and processing time are greater than a customized object. For large XML resources a DOM will have a very noticeable speed difference.

Moving away from the W3C DOM, let's take a look at a tool aimed at solving the third of these issues.

Focusing on the JDOM

DOM issues such as memory requirements and a desire to create a simpler model for working with XML data has prompted several Java developers to create an API called JDOM. JDOM is a Java specific Document Object Model.

The most important fact we must make clear is that JDOM is not a layer that sits over the DOM. JDOM takes a different approach by taking an XML document and creating a Java object representation of the XML file. In addition JDOM takes a simplified approach in comparison to what the DOM object implements. JDOM has 80% to 90% of the DOM functionality.

However, JDOM steers clear on some of the less used but highly complex areas of the DOM. This means JDOM will accomplish most things you would need but a few exceptions exist where you still might need to use DOM. The other good thing about the JDOM design is that it is easy to integrate JDOM and SAX together.

As JDOM is still a new and evolving product you should check in at the JDOM site to get the latest specifications. Popular open-source projects like the Apache Group's Xerces are also working JDOM support in to future releases. Another big bonus to JDOM is that it is starting the Java Community Process. Overall JDOM appears to have a bright future.

For more information on JDOM, visit the official website, <http://www.jdom.org/>.

Installing JDOM

You will want to install JDOM to work with your container. With Tomcat this means copying the `jdom.jar` and `xerces.jar` files into the web application's `WEB-INF\lib` directory.

Now this introduces a slight problem, many versions of `xerces.jar` are in existence and it is possible you will have several copies from different programs using Xerces. So this means you need to be careful on managing your JAR files. If you are getting strange results make sure you have the version of `xerces.jar` that comes bundled with JDOM. With all of the different versions of Java parsing tools floating around it is easy to get confused by using the wrong JAR file.

Revisiting the `dom_message JSP`

Remember the slight difficulty we had getting the message from `message.xml` with `dom_message.jsp`? Let's now take a look at how to accomplish the same simple task with JDOM.

The `jdom_message JSP`

This example uses the `message.xml` within the `C:/xml` directory from the earlier example:

```
<%@ page contentType="text/html"%>
<%@ page import="java.io.File, org.jdom.*, org.jdom.input.SAXBuilder" %>
<%
SAXBuilder builder = new SAXBuilder("org.apache.xerces.parsers.SAXParser");

Document l_doc = builder.build(new File("c:/xml/message.xml"));
%>
<html>
<body>
  <%= l_doc.getRootElement().getChild("message").getText() %>
</body>
</html>
```

This produces the exact same output as the `dom_message.jsp` example. As you can see the code actually appears to be a bit simpler. Some programmers feel the syntax within JDOM is easier to use than the DOM syntax.

For example, `getText()` vs. `getFirstChild().getNodeValue()`.

However, this is a matter of personal preference, and usually depends on which style one is exposed to first as a programmer. In fact many programmers will have experienced DOM-like syntax from other tools.

In this example, you will notice the use of `SAXBuilder`. A nice feature of JDOM is the great integration with SAX it offers. The code illustrates the ease of creating a `SAXBuilder` object and directly importing an XML file into our JSP code. In fact since JDOM uses builders to import an XML file it is easy to choose which builder fits your needs the best.

Currently JDOM has two builders, one for SAX and one for DOM. Usually it is best to use the SAX builder over the DOM builder. It usually doesn't make sense to use the DOM builder unless you are using a DOM that is already created. This is due to the fact you are already using the tree structure of JDOM. The act of creating a DOM would be redundant in most cases ending up being an inefficient use of resources. The SAX builder is the quickest method to use in importing an XML file.

A Different Example of Using JDOM

This next example will read in the `links.xml` file from the DOM example, modify data within it, and then display the modified results. The actual change performed will be to simply change the year, but this will show how to access and modify multiple records several layers down within the XML file.

The `ldom_example` JSP

The `links.xml` file saved within the `C:/xml` directory is also required for this example:

```
<%@ page contentType="text/html"%>
<%@ page import="java.io.File,
                java.util.*,
                org.jdom.*,
                org.jdom.input.SAXBuilder,
                org.jdom.output.*" %>
```

We will need to import the XML file. As stated earlier, JDOM uses builders to actually create the document object and for speed purposes we will use SAX to import the XML file into memory:

```
<%
String ls_xml_file = "c:/xml/links.xml";

SAXBuilder builder = new SAXBuilder("org.apache.xerces.parsers.SAXParser");

Document l_doc = builder.build(new File(ls_xml_file));
```

Now that a JDOM document has been created we can perform queries upon it and modify the data. We will need to get a handle on the root element of the document. Once we have the root, it is possible to ask JDOM to give us an iterator. The iterator permits us to generically loop through all elements under the root. Using this technique we can access any element under the root:

```
Element root = l_doc.getRootElement();

/* get a list of all the links in our XML document */
List l_pages = root.getChildren("link");

Iterator l_loop = l_pages.iterator();
```

Now the code will loop through each link record. Since the year element is actually an element under the date tag, some additional drilling down must be performed by the code. Once we get the child record for the year we can reset the data with a quick `setText()` function call:

```
while ( l_loop.hasNext() )
{
    Element l_link = (Element) l_loop.next();
    Element l_year = l_link.getChild("date").getChild("year");
    l_year.setText("2002");
}
```

Finally, we can take the JDOM document and create a string representation of the XML data. In this case we are left with data that is formatted as an XML file:

```
XMLOutputter l_format = new XMLOutputter();
String ls_result = l_format.outputString(l_doc);
```

Since we want to display our data in an HTML file we must format our data to display correctly. This means we have to encode all of the `<` and `>` characters as `<` and `>`. However, we will use a special feature of JDOM to illustrate the difference between plain text and XML.

When you use the `setText()` function in JDOM, two things happen. The first is that it replaces everything within the tag with the text you supply. If you wanted to insert text and XML into a tag then you would use the `setMixedContent()` function. The second thing `setText()` does is to encode all of the `<` and `>` characters for us:

```

root.setText(ls_result);

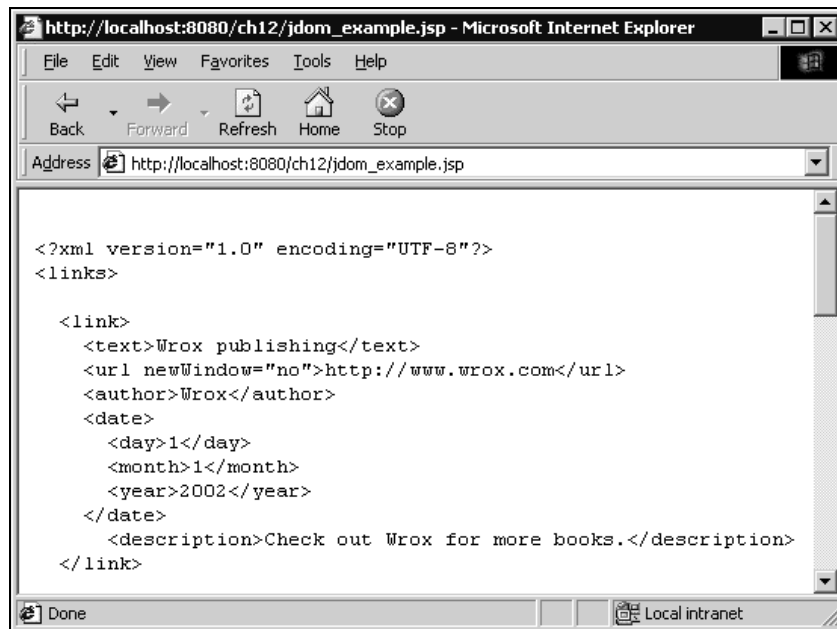
ls_result = l_format.outputString(l_doc);
%>

<html><head><title></title></head>
  <body>
    <pre>
      <%=ls_result%>
    </pre>
  </body>
</html>

```

So in the last step, the call `root.setText(ls_result)` replaces everything within the JDOM object under the root element with a string representation of the XML object. The important point to realize is a string of XML data is not always treated as XML data, it might be treated as a simple string, depending on the functions you use.

This example will produce a result that looks like this:



This example shows several things:

- One thing to keep in mind is that when accessing an element you are only dealing with that level of data. To access sub-elements you need to drill down to that sub-elements level. This means you have to drill down to get to your final destination. This actual drill down is relatively simple as shown in the code above.

- ❑ JDOM is merely a tool to represent and access an XML data source as a collection of Java objects. In many respects using JDOM doesn't change the way we approach programming and using data. From a practical viewpoint the only change is reducing the dependence of using string logic and switching to using elements and nodes to store and change your data. This will become clearer in the last example of the chapter.

Now that we have used JDOM a little let's examine the benefits of JDOM.

JDOM: Pros and Cons

Just like we highlighted in the DOM section, there is no be all and end all way for accessing XML information with Java. Here are some good points to help decide if JDOM is meant for your project:

- ❑ JDOM is specific for Java and has smaller memory requirements than a generic DOM.
- ❑ JDOM has a simpler and more logically based set of methods for accessing its information. This difference can be both a blessing and a curse. What JDOM trades off for ease is some flexibility.
- ❑ JDOM currently does not have support for XSLT. To drive an XSLT processor you would have to use the `XMLOutputter` class to get XML from your JDOM. Hopefully in the future Java XSLT processors and APIs like the JAXP will have native support for JDOM and XSLT transformations.
- ❑ JDOM can suffer memory problems when dealing with large files. The issue boils down to the fact that you can only use JDOM if the final document it generates fits within RAM memory. Future releases of JDOM should address this issue.
- ❑ JDOM is Java specific and can offer support to access other data from sources other than XML. For example classes are being built to access data from SQL queries.

Focusing on the SAX

And now for something completely different. The Simple API for XML is a valuable tool for accessing XML; however, it is not similar at all to its Document Object Model counterparts. Instead the SAX is made for quickly reading through a stream of XML and appropriately firing off events to a listener object. We will cover some of these SAX parsing events later. By using parsing events and having an event handler object SAX is very efficient for handling even large XML sources. You may ask why does this make SAX efficient? Unlike the DOM, which handles everything, events within SAX let us get selective in what our code processes.

JAXP 1.1 supports the SAX 2 API and SAX 2 Extensions developed cooperatively by the XML-DEV mailing list hosted by `XML.org`. Here are the links for the official information. We will give a brief example of using the SAX next:

- ❑ SAX 2 API
<http://www.megginson.com/SAX/index.html>
- ❑ SAX 2 Extensions
<http://www.megginson.com/Software/sax2-ext-1.0.zip>
- ❑ XML-DEV mailing list
<http://www.xml.org/xml-dev/index.shtml>

Before creating an object to handle SAX events we must use a few lines of code to create a `SAXParser`. Similar to `DocumentBuilderFactory` for a DOM there is a `SAXParserFactory` for making `SAXParsers`:

```
SAXParserFactory spf = SAXParserFactory.newInstance();
```

By calling the `newSAXParser()` method we can now get a `SAXParser` object:

```
SAXParser sp = spf.newSAXParser();
```

The only thing left to do is call the `parse()` method on our `SAXParser`. When calling the `parse()` method we must pass in the source to be parsed and an object that listens to SAX events as parameters. From the DOM section we still have `links.xml` to use as our XML source. The only thing left for us to do is create our SAX event listener object.

A SAX event listener object must implement the correct interface for the appropriate SAX events. Interfaces such as `ContentHandler`, `DTDHandler` and `ErrorHandler` all exist in the SAX API for listening to events.

As you might have guessed all of these interfaces are named after the type of event they handle. `ContentHandler` deals with events such as the start of a document or the beginning of an element. `DTDHandler` handles events associated with the Document Type Definition such as notation declarations. `ErrorHandler` deals with any sort of error encountered when parsing through the XML document.

The `DocumentHandler` interface also exists; however, it is only around for legacy support of SAX 1.0 utilities. `ContentHandler` should be used for SAX 2.0 applications because it also supports namespaces.

For our example object we will use the `ContentHandler` interface. In the `org.xml.sax.helpers` package a `DefaultHandler` object already implements the `ContentHandler` interface. Our example will extend this object to ease the amount of code required for the example. The goal of our SAX utility will be to parse through `links.xml` and notify us of a few events as well as counting the number of URLs in the file.

The *SAXExample* Class

Save this file to `WEB-INF/classes/com/jspinsider/jspkit/examples`:

```
package com.jspinsider.jspkit.examples;

import org.xml.sax.helpers.*;
import org.xml.sax.*;
import javax.xml.parsers.*;
import javax.servlet.jsp.*;
import java.io.*;
```

First we must extend the `DefaultHandler` object so that we can implement the `ContentHandler` interface. Next some objects are declared that will be used throughout the code. One of these is a `Writer` object. We will use this to stash a reference to our JSP out implicit object:

```
public class SAXExample extends DefaultHandler{
    private Writer w;
    String currentElement;
```


For the most part SAX events are intuitive with the exception of the `characters()` method. The `characters()` method is called whenever character data is encountered in your XML source. Unfortunately, the parameters passed in to this function don't describe from what element the character data came from. If needed you will have to keep track of this information yourself. For this example, you can see we track this information by having the `currentElement` object updated each time an element is encountered. If the `currentElement` is a URL we will display the URL:

```
public void characters(char[] ch, int start, int length) throws SAXException{
    try{
        if (0 == currentElement.compareTo("url")){
            int count = 0;
            while(count < length)
            {
                w.write(ch[start + count]);
                count++;
            }
            w.write("\n");
        }
    }
    catch(Exception e){throw new SAXException(e.toString());}
}
```

This example doesn't require any further events; however, there are many other different types of event you can track within SAX. Depending on your need different events are available to use in your own custom event handling objects. SAX 2 provides support for every logical event that occurs when parsing an XML document. Consult the JAXP 1.1 documentation to see all of the available SAX events supported.

Now that we have an object ready to listen to SAX events let's tie it in to a JSP.

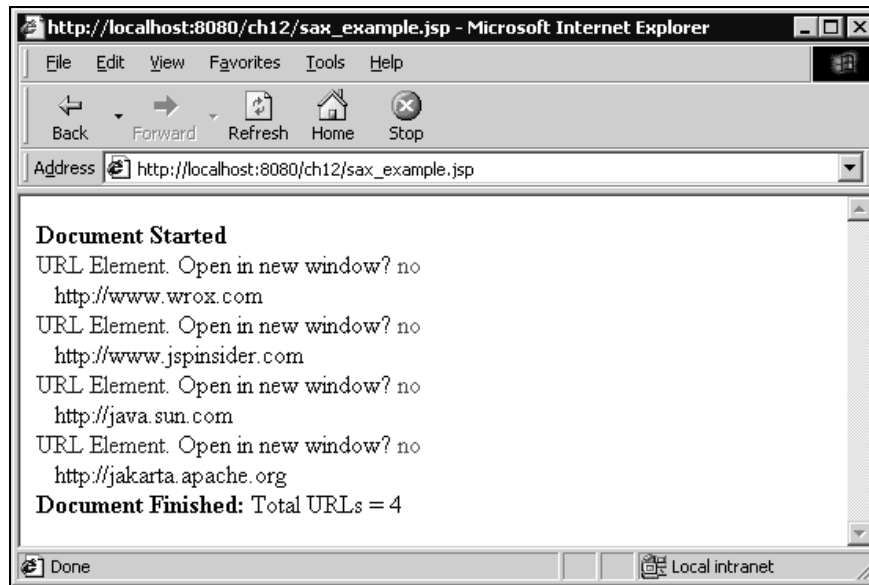
The sax_example JSP

Here is the code for `sax_example.jsp`:

```
<%@ page
    import="org.xml.sax.helpers.*,
           javax.xml.parsers.*,
           com.jspinsider.jspkit.examples.*,
           org.xml.sax.*" %>
<html>
<%
    SAXParserFactory spf = SAXParserFactory.newInstance();
    SAXParser sp = spf.newSAXParser();
    SAXExample se = new SAXExample(out);

    sp.parse(new java.io.File("c:/xml/links.xml"), se);
%>
</html>
```

The only new code that was required is the parse method telling our SAXParser to parse links.xml and notify our SAXExample object of events. The output for this JSP looks like this:



As you can see SAX-style and DOM-style handling of XML is very different. Both can be used effectively for different purposes and should be used as needed. Compared to the example we used in the DOM section you can see we could have used the SAX to verify each of the links in our XML; however, we could not have allowed for the links to be manipulated similarly by stashing a SAX object in the client's session. On the other hand if we wanted to use DOM-style manipulation on a 20mb XML file it would most certainly cause trouble for our system whereas a SAX-style would work.

SAX: Pros and Cons

To conclude the final of our three main Java XML accessing methods we will give a similar list as in the DOM and JDOM example. After this we will give one final reminder on the key differences of the DOM, JDOM, and SAX all at the same time. We will also mention when it might be appropriate to use each:

- ❑ SAX is sequential event based XML parsing. SAX represents an XML document by providing a method to transform the XML as a stream of data, which then can be processed by the programmer.
- ❑ SAX cannot directly modify the streaming document it creates. You can consider SAX to be a read only process. Once the programmer has received a parsed bit of data from SAX, it is then up to the programmer to decide what to do with this received data.
- ❑ SAX is the hardest method to use when performing parsing in non-sequential order. Jumping around in a SAX stream removes any efficiency gain you achieved over a DOM and will usually cause a headache.

DOM / JDOM / SAX: A Final Comparison

Do we really need all of these tools to handle XML? The short answer is yes. While XML is simple it is being used in countless different ways on different projects. The simple fact is that XML represents data and in dealing with data it is important to have several different ways to handle and process this data. This guarantees that no single XML API will ever meet everyone's needs.

These API's all have one thing in common as they all present methods to represent XML data. The strange aspect of these API's is that you might think they share more in common, but in reality what each tool offers is something distinct and unique relative to their specifications.

All the talk about DOM, JDOM and SAX can be a bit confusing to someone encountering these beasts for the first time. In conclusion of this section we would like to give a summary of key points regarding each API along with when each API might be appropriate to use:

- ❑ The streaming nature of SAX makes it generally the fastest way to work through an XML source. When speed is a key issue with your XML SAX is a good place to start.
- ❑ SAX requires the least memory requirements and you can start working with the results as the parser processes the XML stream. For very large XML sources SAX is usually the only viable option.
- ❑ JDOM relies on other processors to actually perform the first step transformation of the XML data into the JDOM model. Of course if you are not using an XML source in the first place this is not an issue.
- ❑ JDOM is usually faster than a DOM and offers a simple Java interface to use in working with an XML document. JDOM also slightly simplifies the syntax required within your Java code.
- ❑ Both the DOM and JDOM have a tree-like structure. The tree-like structure is usually preferred when representing an entire XML document or when needing to access any part of the tree at will.
- ❑ DOM is based on recommendations from W3C and as such is the closest to being a 'standard' of the three systems listed here. SAX and JDOM are not standards, but rather are open source projects that were created to resolve problems that exist within the DOM recommendations. However, while not official standards, both SAX and JDOM have become unofficial standards to address XML parsing issues. At the writing of this book JDOM has started the official JSR process at Sun to become a standard under the Java code umbrella.

In all the above sections we have been describing each of these XML tools separately. Keep in mind there are no restrictions keeping you from mixing and matching the DOM, JDOM and SAX. Use what works best for you.

JSP and XML: A Step By Step Tutorial

The first part of this chapter was a gentle introduction to using XML with Java and the various methods with JSP. Now let's work on a more practical example that will illustrate using the JAXP and JSP together to produce many different formats from the same XML content. For styling XML to different formats we will use something called XSLT (eXtensible Styling Language Transformations).

Styling XML with XSLT

We introduced XSLT when describing some of the XML tools available for use with JSP. XSLT is used to transform an XML document into another form such as HTML, plain text, or even a different XML layout. XSLT is a very rich and comprehensive language. Like the XML in this chapter, for brevity's sake we will not attempt to give a tutorial on XSLT.

*The XSLT examples we do use should be fairly easy to follow even without XSLT experience; however, if you would like to read more on XSLT here is the link once more, <http://www.w3.org/Style/XSL/>. Also see *XSLT Programmer's Reference 2nd Edition* from Wrox Press, ISBN 1861005067.*

Before explaining XSLT further let's explain why we are using it at all. A series of JSP templates could be constructed in place of an XSLT transformation sheet; however, a JSP template approach has a few disadvantages:

- ❑ A JSP template would be project specific and would tend to be cumbersome and impractical for reuse amongst projects. XSLT documents are natively authored in XML and in comparison are extremely portable between projects.
- ❑ A JSP template system would have a higher training cost since each developer has to learn the rules of the unique JSP templates. XSLT already exists for styling XML and is becoming a standard format that many developers are learning.

To be fair XSLT has its own drawbacks. The major disadvantage of XSLT is ironically its biggest advantage. XSLT and the supporting standards form a large and rich environment. As a result fully mastering XSLT is something that can take a while for a programmer.

From a JSP programmer's perspective XSLT also brings a valuable new resource to the programming mix because web browsers are beginning to support it. Client-side XSLT support gives the option to transfer work to the client instead of keeping it on the server. Reducing a server's workload is key in scalability.

As more client-side tools besides web-browsers begin to support XML and XSLT this option could be much more heavily used by web applications. As a result XSLT should spark interest because of its flexibility and be regarded as a powerful tool to watch for a JSP developer. Here are some points to be aware of when choosing between client-side and server-side XSLT.

Some benefits of server-side XSL transformations:

- ❑ Simply put, the client doesn't have to support XML or XSLT. This is the factor that often determines the decision on where you apply the XSLT.
- ❑ Higher degree of data security. You can control the data before it is sent to a client. This means each user will only receive their specific data. This is a great security tool as you can finely control both the data being sent to the client and what client can see the data. Of course with security there are many other factors involved, however, having central control of the data is an important first step.
- ❑ Since you apply the style sheet on the server-side, there is 100% control over the format the user is given. (Of course different client tools could still change the way the final data appears). Keeping tight control over format is helpful when you use one dataset to drive presentations for different users. So on the server you could use one set of data and upon request of the data apply different style sheets depending on the user.
- ❑ Extending on the previous point, for large datasets you can reduce the amount of data sent to the client. Formatting can be conscious of bandwidth issues as well.

However, client-side XSL transformations also have some benefits:

- ❑ Most importantly, client-side transformations distribute the workload so the server doesn't have as much processing to perform. For high load systems this could be a major reason to use client-side XSLT.
- ❑ An XML data island is downloaded to the user. Since the user has local access to the raw data, it is possible to apply many different XSL transformations to the same data without having to go back to the web server. The major benefit is the data only needs to be downloaded once. Once the user has the data, then they are free to apply an infinite amount of different transformations on the same dataset. This reason is often a factor when your users have consistently low bandwidth or limited connection times.

Currently, performing the XSL transformation within the server environment is the most popular method of using XSLT. The major reason is that XSLT is still very new and client-side support is limited. Over the next few years as client-side support increases there will be more applications porting XSLT support to the client.

For our example we will keep all the XSLT on the server-side because of limited support in current web-browsers.

Step 1: Build the XML Source

Before styling we do need some XML content. Again we will reuse `links.xml` from the DOM example. If you have `links.xml` saved from the previous examples this step is already done. If not head back up and snag it from the DOM section.

Next we will work on the XSLT documents that will be needed to perform each of the transformations.

Step 2: The XSLT File

To transform the XML file we will need a few XSLT documents, one XSLT document for each desired output format. Each of these XSLT documents will need to be saved locally on your hard drive for use in the later steps. In our examples we will use the `C:\xml\` directory but you may use anything as long as the URI matches appropriately.

The first XSLT document will be for making the commonly known format of HTML. This first example will have many comments with the code to aid understanding of what exactly is going on. The rest of the XSLT documents will follow the same format so they will have much less explanation. In brief, this XSLT is going to make an HTML document with links to each of our 'link' elements in the original XML source.

The `links_html` XSL

Here is our XSLT document for `links_html.xsl`:

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

The header for XSLT documents is the default XML header. Remember XSLT is 100% XML compatible. Like the rest of the XML in this chapter you can manipulate XSLT documents in Java with tools such as the DOM, JDOM or SAX.

After the header is the XSLT namespace declaration along with a special `xsl` prefixed tag named `output`. The `output` tag will remove the default XML header from the output and also convert some XML syntax tags to HTML 4.0 tags. Since HTML is commonly used XSLT inherently provides support.

Next is the first template match on our XML document. This match will process every element titled `links` in our XML document. All tags inside a template without the `xsl` namespace will be sent directly to the output. Notice that all the tags without the `xsl` namespace are simply the HTML:

```
<xsl:template match="links">
<HTML>
  <HEAD>
    <TITLE>Fun with XSL</TITLE>
  </HEAD>
  <BODY>
    <table cellspacing="0" width="100%">
      <tr>
        <td class="clear" align="center">
          <table width="95%">
            <xsl:apply-templates select="link"/>
          </table>
        </td>
      </tr>
    </table>
  </BODY>
</HTML>
</xsl:template>
```

The first template has a call to another template for each element named `link` in our main `links` element. In our source XML document each `link` element describes an actual link to a resource. Here is where we add the appropriate HTML for each specific link:

```
<xsl:template match="link">
  <tr>
    <td class="clear">
      <table cellpadding="0" cellspacing="0" width="100%">
        <tr>
          <td style=" font-family: verdana; font-size: 10pt; background: #aabbbb;
border-style: groove; border-width:2px; border-color:#ffffff; padding:2px;">
```

Two `xsl` namespace tags called `use-attribute-sets` and `value-of` are new here. The `xsl` tag `use-attribute-sets` will call another set of tags similar to the `xsl` template tag. Instead of sending more tags to the output the `use-attribute-sets` will add attributes to the tag it was called in. The `xsl` `value-of` tag does exactly what it says. The value of our element named `text` will be inserted in the XSLT template:

```
  <A xsl:use-attribute-sets="a"><xsl:value-of select="text" /></A><br />
  </td>
</tr>
<tr>
  <td style=" font-family: verdana; font-size: 10pt; border-style: groove;
border-width:2px; border-color:#ffffff; padding:2px; background: #f5f5f5; border-
top-width:0px; padding-left:10px; padding-top:0px; margin-top:0px;">
```

Here the `value-of` tag is seen again. This time the author's name will be used from our XML document:

```

        <xsl:value-of select="author"/><br />
    </td>
</tr>
<tr>
    <td style=" font-family: verdana; font-size: 10pt; border-style: groove;
border-width:2px; border-color:#ffffff; padding:2px; background: #eeeeee; border-
top-width:0px; padding-left:10px; padding-top:0px; margin-top:0px;">

```

One last time the `value-of` tag is used for the description of the link:

```

        <xsl:value-of select="description"/><br />
    </td>
</tr>
</table>
</td>
</tr>
</xsl:template>

```

After the end of our template a description is needed for the `attribute-set` we called earlier. Inside this attribute set a name and value is placed for each attribute tag. We only have one attribute named `href` for our link and its value is set as the URL in our XML document:

```

<xsl:attribute-set name="a">
    <xsl:attribute name="href"><xsl:value-of select="url"/></xsl:attribute>
</xsl:attribute-set>

```

With all the templates and attribute sets done this XSLT is finished:

```

</xsl:stylesheet>

```

That is it for the first template. The above might be a little confusing if you are unfamiliar with XML and XSLT. The key point to remember with each of these templates is that they will transform our XML source into a new format. For this template the result will be basic HTML. The benefit of using XSLT for this is that we can add support for a new format by adding a new template.

The links_wml XSL

Here is the next template for supporting WML. WML is the Wireless Markup Language and is used on smaller devices such as web-enabled cell phones. WML is not the only markup for wireless devices but is currently popular. WML is fully XML compliant and has many similar tags to HTML. The WAP Forum standardizes WML.

The current WML specification can be found in pdf format at:
<http://www.wapforum.org/what/technical.htm>.

With no more delay here is the template (`links_wml.xsl`) for creating WML. Again this template is the same as the previous with the HTML tags swapped to WML:

```
<?xml version="1.0"?>

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

Again the default XML header is seen and the `xsl` output tag is called. This time the output will be XML compatible.

Next the same set of templates will be used substituting WML for the HTML. Don't worry about understanding the WML syntax; we will give a screen shot of what the result looks like on a WAP device:

```
<xsl:template match="links">
<xsl:text disable-output-escaping="yes">&lt;!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD
WML 1.1//EN" "http://www.wapforum.org/DTD/wml_1.1.xml"&gt;</xsl:text>
<wml>
  <card id="card0" title="JSPInsider.com">
    <do type="prev" label="Back"><prev/></do>
    <p>
      <table columns="1">
        <xsl:apply-templates select="link"/>
      </table>
    </p>
  </card>
</wml>
</xsl:template>
```

The same link template is here as well. Each link from our XML is used when creating the WML output:

```
<xsl:template match="link">
<tr>
  <td class="clear"></td>
</tr>
<tr>
  <td>
    <a xsl:use-attribute-sets="a"><xsl:value-of select="text"/></a><br />
  </td>
</tr>
<tr>
  <td>
    <xsl:value-of select="author"/><br />
  </td>
</tr>
<tr>
  <td>
    <xsl:value-of select="description"/><br />
  </td>
</tr>
</xsl:template>
```

The identical attribute set is being reused from the first template. Since many of the markup languages have similar syntax, it is possible to reuse much of the XSLT templates for these various examples:

```
<xsl:attribute-set name="a">
  <xsl:attribute name="href"><xsl:value-of select="url" /></xsl:attribute>
</xsl:attribute-set>
```

The document is now done so we end it:

```
</xsl:stylesheet>
```

WML support is now available for our XML source. Once we create the JSP to combine these XSLT documents with our XML we will be in business. Let's make one more template before moving on to the JSP.

The final XSLT document will be for sending out a simplified XML tree. XSLT is often used to reformat XML data itself so different systems can use the same data. To show how this is done, we will take the original XML file and create a new one with only a subset of the data we need for our code.

The new XML file created is what we are calling the simplified tree. The simplified tree will contain nothing more than what is needed for the link check example in the DOM section of this chapter. We could use this method to simplify what our DOM in `dom_links.jsp` will represent. After all why have the DOM use memory to represent information we are not even using?

The `links_xml` XSL

Here is the XSLT document for `links_xml.xsl`:

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

<xsl:template match="links">
<links>
  <xsl:apply-templates select="link"/>
</links>
</xsl:template>

<xsl:template match="link">
  <url>
  <xsl:value-of select="url"/>
  </url>
</xsl:template>

</xsl:stylesheet>
```

That is it for the XSLT templates. As we mentioned previously all of the example transformations will be done on the server-side using JSP. If you would like to try out a client-side transformations use the latest Internet Explorer or Mozilla browser.

For extra XSLT help for IE 5.5 see, <http://www.netcrucible.com/xslt/msxml-faq.htm>, or for Mozilla XSLT information see, <http://www.mozilla.org/projects/xslt/>.

Step 3: Build a Source Object

Source objects represent the XML data that will be used to create a final formatted output from our XSLT. Combining the source with our XSLT is done in the JAXP with a transformation. Do not confuse what we are calling a JAXP transformation with the 'T' at the end of XSLT. XSLT defines what a transformation should do. A JAXP transformation does what the XSLT document defines.

In JAXP a source is an actual interface. To define a source for a transformation we need to use an object that implements the source interface. Three objects in the JAXP 1.1 release implement the source interface and they are called `StreamSource`, `DOMSource` and `SAXSource`.

StreamSource

A `StreamSource` is a placeholder for a transformation source in the form of an XML stream. A `StreamSource` is a convenient object when you have an XML file or a string that you will be using as a source.

Here is code you would use for creating a `StreamSource` object from an XML file:

```
StreamSource ss = new StreamSource(new File("Your_xml_file.xml"));
```

Turning a `String` object in to a `StreamSource` object can be done similarly:

```
StreamSource ss = new StreamSource(new StringReader(new String()));
```

DOMSource

A `DOMSource` represents a node from a DOM object. The node could be as small as one data element or it could be the entire document the DOM is representing. You create the `DOMSource` to represent the piece of the DOM you would like to modify or work with. Once you have the `DOMSource` you can then use it to perform a transformation.

Here is some code showing how a `Document` object can be used as a `DOMSource`:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document d = db.parse("Your_xml_file.xml");
DOMSource ds = new DOMSource(d);
```

SAXSource

A `SAXSource` object represents a SAX-style source. `SAXSource` objects can be constructed out of an `XMLReader` object or any `InputSource` object. Check the JAXP 1.1 javadocs for more information about `XMLReader` and `InputSource`.

Here is some code showing how a `SAXSource` can be created from a `FileReader` object:

```
SAXSource ss = new SAXSource(
    new InputSource(new FileReader("Your_xml_file.xml")));
```

Creating a StreamSource

For step five in this tutorial we will need an XML source and an XSLT source for a JAXP transformation. The sources we will use are two `StreamSource` objects made from `links.xml` and `links_html.xsl`.

Here is the code for our XML source:

```
StreamSource xml = new StreamSource(new File("c:/xml/links.xml"));
```

Here is the code for our XSLT source:

```
StreamSource xsl =  
    new StreamSource(new File("c:/xml/links_html.xsl"));
```

Step 4: Build a Result Object

Result objects are the output of a JAXP transformation. The result does not have to be in XML format, it can be in many different forms such as HTML, XML, WML or even plain text. The result object is actually an interface and should not be declared as an object itself. In the JAXP 1.1 three objects use the `Result` interface: `StreamResult`, `DOMResult` and `SAXResult`.

StreamResult

A `StreamResult` acts as a holder for a transformation result that is a stream. `StreamResult` is commonly used when writing the transformation result to a file or the JSP output stream.

Here is some code for writing a JAXP transformation to a file:

```
StreamResult sr = new StreamResult(new File("example.out"));
```

Here is some code for writing a JAXP transformation to output of a JSP:

```
StreamResult sr = new StreamResult(out);
```

DOMResult

A `DOMResult` holds the result of a transformation in a DOM tree object. The object can then be further manipulated as a `Document` object. When it is convenient to pass a `Document` object between applications this type of result should be used:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();  
DocumentBuilder db = dbf.newDocumentBuilder();  
Document d = db.newDocument();  
DOMResult dr = new DOMResult(d);
```

SAXResult

A `SAXResult` is a holder for a JAXP transformation in an object that implements the `ContentHandler` interface for handling SAX events. These objects can be customized to a great extent for working with SAX.

For an example snippet we will use the basic `DefaultHandler` object. In a practical situation you would use an object customized to accept SAX events and do whatever task you were trying to accomplish:

```
SAXResult sr = new SAXResult(new DefaultHandler());
```

Displaying the Results

For this step-by-step tutorial we will use the `out` implicit object to display the result of our JAXP transformations. The code from the second `StreamSource` example is exactly what we need.

Here is the code for our `StreamResult`:

```
StreamResult result = new StreamResult(out);
```

Step 5: Transform the Data

Taking steps one through four we now have enough information to do a sample JAXP transformation. Using scriptlets we will make a JSP that performs a transformation using our sources from step three and our result from step four.

Here is the code for `example_transformation.jsp`:

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

<%
StreamSource xml = new StreamSource(new File("c:/xml/links.xml"));
StreamSource xsl = new StreamSource(new File("c:/xml/links_html.xsl"));

StreamResult result = new StreamResult(out);
```

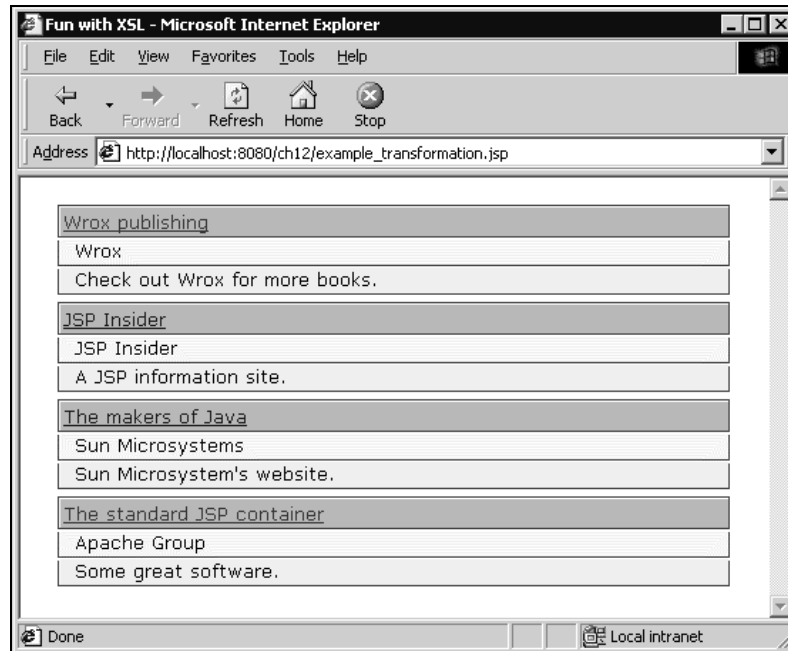
For a transformation a `TransformerFactory` is required so we can then produce a transformer object. When constructing a `Transformer` object we will take advantage of the option to associate an XSLT source. For this example we will use the `StreamSource` representing `links_html.xsl`:

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

Once we have a `Transformer` object it is possible to perform a JAXP transformation. For the transformation we need to provide a parameter specifying the XML source and where we want the destination of the transformation result:

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

The result will be an HTML page with links and descriptions to the resources in our XML file:



We will not end the tutorial here. As mentioned in the beginning we are trying to create a generic tag library providing easy access to the functionality of the JAXP. For this tag library we will need some custom JavaBeans and some tags.

Step 6: Build a JavaBean to Access XML Data

Now that the basic mechanics have been explained it is now possible to begin writing reusable code. The first step is to build a JavaBean for specifying sources and the result for a transformation. The Bean will be very useful so that we may specify information for the transformation anywhere and then pass it along as needed. To keep all of the flexibility of the JAXP this Bean will need to allow each of the various `Source` and `Result` objects. To implement this flexibility three main objects will be created and named `XMLSource`, `XSLSource` and `transformationResult` and be used as follows:

- ❑ `XMLSource`
Holds the XML resource to be used in the transformation. Since this could be either a `StreamSource`, `DOMSource` or `SAXSource` initially it will be declared as an object and later cast into the appropriate type.
- ❑ `XSLSource`
Holds the XSL resource to be used in the transformation. Exactly like the `XMLSource`, `XSLSource` will initially be declared as an object and later cast into the appropriate type at transformation.
- ❑ `transformationResult`
Represents the output of the transformation. Since the JAXP is capable of `StreamResult`, `DOMResult` and `SAXResult` this will also initially be declared as an object and later cast as needed.

All of the above objects will also have get and set methods for each type of possible cast. This means that each object will have three get and set methods for Stream, DOM and SAX.

With the above said here is our code that does exactly that. The next section will cover creating the JAXP transformer.

The JAXPTransformerResources Class

Save this file to WEB-INF/classes/com/jspinsider/jspkit/jaxp:

```
package com.jspinsider.jspkit.jaxp;

import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.sax.SAXResult;
import javax.xml.transform.sax.SAXSource;
import java.lang.Object;

public class JAXPTransformerResources extends java.lang.Object implements
java.io.Serializable
{
```

Here are the three main objects of the Bean. Each of these resources is kept as an object to allow for the DOM, SAX and Stream sources:

```
private Object XMLSource;
private Object XSLSource;
private Object transformResult;

public JAXPTransformerResources(){}
```

Each resource needs the appropriate get method:

```
public Object getXMLSource(){ return XMLSource; }
public Object getXSLSource(){ return XSLSource; }
public Object getTransformResult(){ return transformResult; }
```

Next each resource has the appropriate set method to complement the gets:

```
public void setXMLStreamSource(StreamSource newXMLStreamSource)
{ XMLSource = newXMLStreamSource; }
public void setXMLDOMSource(DOMSource newXMLDOMSource)
{ XMLSource = newXMLDOMSource; }
public void setXMLSAXSource(SAXSource newXMLSAXSource)
{ XMLSource = newXMLSAXSource; }
public void setXSLStreamSource(StreamSource newXSLStreamSource)
{ XSLSource = newXSLStreamSource; }
public void setXSLDOMSource(DOMSource newXSLDOMSource)
{ XSLSource = newXSLDOMSource; }
public void setXSLSAXSource(SAXSource newXSLSAXSource)
{ XSLSource = newXSLSAXSource; }
public void setStreamResult(StreamResult newStreamResult)
```

```

    { transformResult = newStreamResult; }
    public void setDOMResult(DOMResult newDOMResult)
    { transformResult = newDOMResult; }
    public void setSAXResult(SAXResult newSAXResult)
    { transformResult = newSAXResult; }
}

```

Step 7: Build an XML JavaBean to Transform the XML Data

The next object we require is one that does the actual transformation. All that is needed for this object is a few constructors and a method calling for a JAXP transformation using the above resource object. The next step will be to make this Bean into a custom tag for easy use with a JSP.

The JAXPTransformer Class

Save the file to WEB-INF/classes/com/jspinsider/jspkit/jaxp:

```

package com.jspinsider.jspkit.jaxp;

import java.lang.Object;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.sax.SAXResult;
import javax.xml.transform.sax.SAXSource;
import javax.xml.transform.TransformerException;

public class JAXPTransformer extends java.lang.Object
    implements java.io.Serializable {

```

First an object is declared for holding our JAXPTransformerResources object. Next the default public constructor made along with a constructor that will take a JAXPTransformerResources object:

```

    JAXPTransformerResources source;

    public JAXPTransformer(){}

    public JAXPTransformer(JAXPTransformerResources source)
        throws TransformerException {
        transform(source);
    }

```

The transform method of this object will do the actual JAXP transformation. If the resources are not set properly then an error will be thrown to reflect that:

```

    public void transform (JAXPTransformerResources source)
        throws TransformerException {

```

Inside the transform method our three requirements for a JAXP transformation are extracted from the JAXPTransformationResources object:

```

Object XMLSource = source.getXMLSource();
Object XSLSource = source.getXSLSource();
Object transformResult = source.getTransformResult();

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

```

The code for the JAXP transformation is the same as seen in step five except that we now need to determine what type of source is being used from our main objects. The `XSLSource` object is compared against its three possibilities and set when correct:

```

if(XSLSource.getClass().getName().compareTo
    ("javax.xml.transform.stream.StreamSource") == 0)
    transformer = tFactory.newTransformer((StreamSource)XSLSource);

if(XSLSource.getClass().getName().compareTo
    ("javax.xml.transform.dom.DOMSource") == 0)
    transformer = tFactory.newTransformer((DOMSource)XSLSource);

if(XSLSource.getClass().getName().compareTo
    ("javax.xml.transform.sax.SAXSource") == 0)
    transformer = tFactory.newTransformer((SAXSource)XSLSource);

```

The last step in the transformation is accomplished in the same fashion as the previous example. To determine what class `XMLSource` and `transformResult` are they are compared against the possibilities. When the correct two matches are found the transformation is finally called and our method is complete:

```

if(XMLSource.getClass().getName().compareTo
    ("javax.xml.transform.stream.StreamSource") == 0) {
    if(transformResult.getClass().getName().compareTo
        ("javax.xml.transform.stream.StreamResult") == 0)
        transformer.transform((StreamSource)XMLSource,
            (StreamResult)transformResult);
    if(transformResult.getClass().getName().compareTo
        ("javax.xml.transform.dom.DOMResult") == 0)
        transformer.transform((StreamSource)XMLSource, (DOMResult)transformResult);
    if(transformResult.getClass().getName().compareTo
        ("javax.xml.transform.sax.SAXResult") == 0)
        transformer.transform((StreamSource)XMLSource, (SAXResult)transformResult);
}

if(XMLSource.getClass().getName().compareTo
    ("javax.xml.transform.dom.DOMSource") == 0) {
    if(transformResult.getClass().getName().compareTo
        ("javax.xml.transform.stream.StreamResult") == 0)
        transformer.transform((DOMSource)XMLSource, (StreamResult)transformResult);
    if(transformResult.getClass().getName().compareTo
        ("javax.xml.transform.dom.DOMResult") == 0)
        transformer.transform((DOMSource)XMLSource, (DOMResult)transformResult);
    if(transformResult.getClass().getName().compareTo
        ("javax.xml.transform.sax.SAXResult") == 0)
        transformer.transform((DOMSource)XMLSource, (SAXResult)transformResult);
}
if(XMLSource.getClass().getName().compareTo

```

```

        ("javax.xml.transform.sax.SAXSource") == 0) {
    if(transformResult.getClass().getName().compareTo
        ("javax.xml.transform.stream.StreamResult") == 0)
        transformer.transform((SAXSource)XMLSource, (StreamResult)transformResult);
    if(transformResult.getClass().getName().compareTo
        ("javax.xml.transform.dom.DOMResult") == 0)
        transformer.transform((SAXSource)XMLSource, (DOMResult)transformResult);
    if(transformResult.getClass().getName().compareTo
        ("javax.xml.transform.sax.SAXResult") == 0)
        transformer.transform((SAXSource)XMLSource, (SAXResult)transformResult);
    }
}

```

The last two methods are the simple get and set for the `JAXPTransformerResources` object:

```

public void setTransformerResources(JAXPTransformerResources new_resources) {
    source = new_resources;
}

public JAXPTransformerResources getTransformerResources() {
    return source;
}
}

```

Step 8: Create a JAXP Tag Library Interface

Finally let's put all of the code into an easy and reusable tag library. By doing this we can use a few tags in our JSP to accomplish a complete JAXP transformation. To picture what we are talking about, imagine using the simple tags below instead of steps one through four. That is what we are building here:

```

<transform>
  <xmlFile>c:/xml/links.xml</xmlFile>
  <xslFile>c:/xml/links_html.xsl</xslFile>
</transform>

```

In Chapters 8 to 11 tag libraries are extensively covered. For now, the important thing to know about tag libraries is that they are a method of making your code easy to use. They do this by moving scriptlets from your JSP page and enclosing them within special tag handler objects. The intersection of these tag handler objects and your JSP container are some special XML configuration files that describe your tag library. As we build the JAXP tags we will walk you through where to place everything.

The first thing needed for our tag library are some custom tag handler objects. The first of these tag handler objects will be the transformer tag seen surrounding the `xslFile` and `xmlFile` tags above.

The `JAXPTransformerTag` Class

Save the file to `WEB-INF/classes/com/jspinsider/jspkit/jaxp`:

```

package com.jspinsider.jspkit.jaxp;

import java.io.IOException;
import java.io.File;
import javax.xml.transform.dom.DOMSource;

```

```

import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.sax.SAXResult;
import javax.xml.transform.sax.SAXSource;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import javax.servlet.*;

public class JAXPTransformerTag extends BodyTagSupport {
    JAXPTransformer transformer = new JAXPTransformer();
    JAXPTransformerResources resources = new JAXPTransformerResources();
    boolean isResources = false;

    public void setTransformerResources(String new_resources) {
        resources = (JAXPTransformerResources)pageContext.getAttribute
            (new_resources, pageContext.getAttributesScope(new_resources));
        isResources = true;
    }
}

```

Set methods are available in this tag for each of the three types of the two Source objects. Without these we could not preserve the flexibility the JAXP offers for source types:

```

public void setXSLStreamSource(StreamSource new_XSLStreamSource)
    { resources.setXSLStreamSource(new_XSLStreamSource); }
public void setXSLSAXSource(SAXSource new_XSLSAXSource)
    { resources.setXSLSAXSource(new_XSLSAXSource); }
public void setXSLDOMSource(DOMSource new_XSLDOMSource)
    { resources.setXSLDOMSource(new_XSLDOMSource); }
public void setXMLStreamSource(StreamSource new_XMLStreamSource)
    { resources.setXMLStreamSource(new_XMLStreamSource); }
public void setXMLSAXSource(SAXSource new_XMLSAXSource)
    { resources.setXMLSAXSource(new_XMLSAXSource); }
public void setXMLDOMSource(DOMSource new_XMLDOMSource)
    { resources.setXMLDOMSource(new_XMLDOMSource); }

```

Set methods are also made available for all of the result types in the JAXP:

```

public void setStreamResult(StreamResult new_StreamResult)
    { resources.setStreamResult(new_StreamResult); }
public void setSAXResult(SAXResult new_SAXResult)
    { resources.setSAXResult(new_SAXResult); }
public void setDOMResult(DOMResult new_DOMResult)
    { resources.setDOMResult(new_DOMResult); }

```

When the first transformer tag is used in a JSP page the output of our transformation will default to the JSP implicit out object. Later on this can be changed at will:

```

public int doStartTag() throws JspTagException {
    if (!isResources) resources.setStreamResult
        (new StreamResult(pageContext.getOut()));
    // Have the JSP Container continue processing the JSP page as normal.
    return EVAL_PAGE;
}

```

When the end transform tag is encountered a JAXP transformation will then take place:

```
public int doEndTag() throws JspTagException {
    try {
        transformer.transform(resources);
    }
    catch (Exception e) {
        throw new JspTagException("JSP Kit JAXPTransformerTag Error:" +
            e.toString());
    }
    // Have the JSP Container continue processing the JSP page as normal.
    return EVAL_PAGE;
}
```

The above tag definitely has a lot of code but does nothing without having the resources needed in a transformation. For this tag two different mechanisms exist to allow resources to get set for our transformation. A `JAXPTransformationResources` Bean could be initialized for the process, but this would only work for experienced Java programmers because scriptlets would be needed to complement the Bean. Instead let's create some more custom tags for setting resources by simply typing in the resource or result URI.

If you wish to make some of your own custom tags these next two sections of code will serve as a good template. The only thing you will need to change is what gets done with the `BodyContent`, the text that is placed inside your custom tag. The next two pieces of code will be used for the tags named `xmlFile` and `xslFile` seen at the beginning of this step.

The `JAXPTransformerXMLFileTag` Class

Save this file to `WEB-INF/classes/com/jspinsider/jspkit/jaxp`:

```
package com.jspinsider.jspkit.jaxp;

import java.io.IOException;
import java.io.File;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import javax.servlet.*;
import java.io.StringReader;
import java.io.StringWriter;

public class JAXPTransformerXMLFileTag extends BodyTagSupport
{
    /* The JSP container calls this function when it encounters the
       end of the Tag. */
    public int doEndTag() throws JspTagException {
        try {
```

The only part of this code that we will comment on is this section including the `BodyContent` object. The `BodyContent` contains what is typed in between the start and end of our custom tag. You can see here that we use this text as the URI to the XML source file:

```

        BodyContent lbc_bodycurrent = getBodyContent();
        StringWriter sw = new StringWriter();
        lbc_bodycurrent.writeOut(sw);
        File f = new File(sw.toString());
        JAXPTransformerTag tag =
            (JAXPTransformerTag)findAncestorWithClass(this,
                com.jspinsider.jspkit.jaxp.JAXPTransformerTag.class);
        tag.setXMLStreamSource(new StreamSource(f));
    }
    catch (Exception e) {
        throw new JspTagException("JSP Kit JAXPTransformerXMLFileTag Error:" +
            e.toString());
    }

    // Have the JSP Container continue processing the JSP page as normal.
    return EVAL_PAGE;
}
}

```

Next comes the code for the `xslFile` tag seen at the beginning of this step. As you might have guessed the code for this tag is almost identical to the above except for its name and one function call.

The *JAXPTransformerXSLFileTag* Class

Save this file to `WEB-INF/classes/com/jspinsider/jspkit/jaxp`:

```

package com.jspinsider.jspkit.jaxp;

import java.io.IOException;
import java.io.File;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import javax.servlet.*;
import java.io.StringReader;
import java.io.StringWriter;

public class JAXPTransformerXSLFileTag extends BodyTagSupport {
    public int doEndTag() throws JspTagException {
        try {
            /* Gather up the contents between the start and end of our tag */
            BodyContent lbc_bodycurrent = getBodyContent();
            StringWriter sw = new StringWriter();
            lbc_bodycurrent.writeOut(sw);
            File f = new File(sw.toString());
            JAXPTransformerTag tag = (JAXPTransformerTag)findAncestorWithClass(this,
                com.jspinsider.jspkit.jaxp.JAXPTransformerTag.class);

```

Please note the one difference from the previous object is that the `setXSLStreamSource()` method is called instead of the `setXMLStreamSource()`:

```

        tag.setXSLStreamSource(new StreamSource(f));
    }
    catch (Exception e) {
        throw new JspTagException("JSP Kit JAXPTransformerXSLFileTag Error:" +
            e.toString());
    }

    // Have the JSP Container continue processing the JSP page as normal.

```

```

    return EVAL_PAGE;
  }
}

```

Now that we have built our custom tag objects to perform the work, we need to tell our JSP container about them, so we can use them.

Step 9: Using the JAXP Tag Library in a JSP Page

Finally we can actually use the tags seen in step eight. The JSP for this example will actually be quite small since all of our code is in the above Beans and tags. This is good because now just about anyone can use the transformer tag. Even if the page editors have no experience with Java they can now specify a few simple tags and everything will work. Of course your page editors will still need to be able to match up a URI to what they type between tags. So if your editors are smart enough to know about what a URI is then you are set, other wise you might have to help in the set up a little.

To get this tag to work we need to do two more simple steps. First we need to insert into the `web.xml` some XML to tell the JSP container about the tags. The second step is to include a special file called a tag library descriptor (TLD) that describes the custom tag.

Both the XML file fragment to be inserted within `web.xml` and the TLD file are included with the code for this book.

Additions to `web.xml`

To let your JSP container know about these new custom tags you must edit the `web.xml` file in your JSP project's `WEB-INF` directory to include the code below:

```

<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
  <taglib>
    <taglib-uri>
      http://www.jspinsider.com/jspkit/JAXP
    </taglib-uri>
    <taglib-location>
      /WEB-INF/JAXP.tld
    </taglib-location>
  </taglib>
</web-app>

```

Next you will need to add the `JAXP.tld` file in your `WEB-INF` folder for this JSP project. The TLD just tells the system where to find our tags and what attributes are within the tag on the JSP page. Once you have installed this file and updated the `web.xml` file, restart Tomcat so it can learn about the JAXP tag and then everything is set to go.

The JAXP TLD

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>JAXPTransformerTag</shortname>
  <uri></uri>
  <info>
    A tag for XML transformations using XSL via Sun's JAXP.
  </info>

  <tag>
    <name>transformer</name>
    <tagclass>com.jspinsider.jspkit.jaxp.JAXPTransformerTag</tagclass>
    <info>Uses Sun's JAXP to do a transformation given an XML source, XSL source
and output result.</info>
    <attribute>
      <name>transformerResources</name>
      <required>>false</required>
    </attribute>
  </tag>
  <tag>
    <name>xmlFile</name>
    <tagclass>com.jspinsider.jspkit.jaxp.JAXPTransformerXMLFileTag</tagclass>
    <info>XML file Source tag for the JAXPTransformerTag</info>
  </tag>
  <tag>
    <name>xslFile</name>
    <tagclass>com.jspinsider.jspkit.jaxp.JAXPTransformerXSLFileTag</tagclass>
    <info>XSL file Source tag for the JAXPTransformerTag</info>
  </tag>
</taglib>

```

Here is an example JSP page that uses the transformer tag. Notice it looks very much like the example we mentioned in step eight:

```

<%@ taglib uri="http://www.jspinsider.com/jspkit/JAXP" prefix="JAXP"%>
<JAXP:transformer>
  <JAXP:xmlFile>Your_XML_File.xml</JAXP:xmlFile>
  <JAXP:xslFile>Your_XSL_File.xsl</JAXP:xslFile>
</JAXP:transformer>

```

Using the XSLT documents built in step three we can now have some real fun. Everything is now set and all we need to do is use our templates. Let's start by again demonstrating the HTML example but this time using the custom tags.

The links_html JSP

```

<%@ taglib uri="http://www.jspinsider.com/jspkit/JAXP" prefix="JAXP"%>
<JAXP:transformer>
  <JAXP:xmlFile>c:/xml/links.xml</JAXP:xmlFile>
  <JAXP:xslFile>c:/xml/links_html.xsl</JAXP:xslFile>
</JAXP:transformer>

```

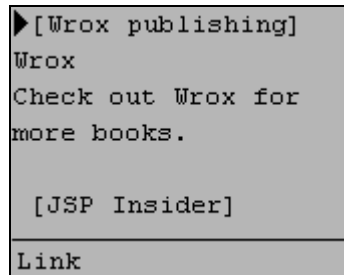
We encourage you to actually run these examples mostly because they are fun. Since this is a book we will also show you a screenshot of the output. The output looks the same as before.

For our WML JSP we use the same tags but change the XSLT sheet to be `links_wml.xsl`.

The `links_wml` JSP

```
<%@ page contentType="text/vnd.wap.wml" %>
<%@ taglib uri="http://www.jspinsider.com/jspkit/JAXP" prefix="JAXP"%>
<JAXP:transformer>
  <JAXP:xmlFile>c:/xml/links.xml</JAXP:xmlFile>
  <JAXP:xslFile>c:/xml/links_wml.xsl</JAXP:xslFile>
</JAXP:transformer>
```

And here is what the phone displays (if you have the Opera web browser you can also view this page with it):



Setting up an actual WAP server to serve the WML to real WAP devices is a completely different story. If you noticed we actually just used a WAP emulator to show our WML. You can download this WAP browser yourself from http://developer.openwave.com/download/license_41.html and test the output.

The `links_xml` JSP

```
<%@ taglib uri="http://www.jspinsider.com/jspkit/JAXP" prefix="JAXP"%>
<JAXP:transformer>
  <JAXP:xmlFile>c:/xml/links.xml</JAXP:xmlFile>
  <JAXP:xslFile>c:/xml/links_xml.xsl</JAXP:xslFile>
</JAXP:transformer>
```

And the simplified XML tree ends up like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<links>
<url>http://www.wrox.com</url>
<url>http://www.jspinsider.com</url>
<url>http://java.sun.com</url>
<url>http://jakarta.apache.org</url>
</links>
```

Fine Points of JAXP

Be sure to try the above examples when you get a chance. The combination of XML, XSLT and JSP permits a developer to accomplish quite a bit of work with relative ease. Now is the time to become creative. The output and input can be from any sort of resource.

In building JSP applications, developers tend to think of a traditional web application delivering data to human clients on the web. However, this is rapidly changing and many web applications are dealing with other applications. One of the fastest growing markets is getting application servers to share data with each other. The marketplace of SOAP and inter-application communication should be one of the hot areas of development over the next few years. Code like JAXP will be one of the tools needed to build these new systems.

If anything the above example should illustrate some of the benefits of using XML in general. For JSP we have just constructed a very flexible and generic tool that links our XML to Java. Just about anything could be tied into this process. Using a database? No problem, many databases can now produce pure XML result sets. If not just add another step to the chain that converts your database results to XML and pump it through your new custom tags.

As mentioned before the above tag library will be available from Wrox with this book.

XML Syntax for JSP 1.2

All of the above sections have focused on using content described by XML with your JSP. Now we will briefly look at an XML related enhancement promised with the JSP 1.2 specification. The enhancement we are talking about is JSP code that is 100% compatible XML syntax.

With JSP compliant in XML syntax we will be able to use JSP code with any XML utility. Keep in mind we mentioned XML based JSP is promised with the JSP 1.2 specification. As of publication of this book the JSP 1.2 specs are not official and have plenty of disclaimers in place for change. However, it is highly doubtful that these specs will significantly change before being finished so we will give a little preview. Tomcat 4.0 is geared towards supporting all the JSP1.2 specs.

Goodbye Normal JSP Syntax?

No, in the proposed final draft the original syntax for JSP is not being deprecated. Just because JSP will be XML compliant in its new syntax does not mean it is the best solution. The new XML JSP syntax has both advantages:

- ❑ JSP code can now be authored in an XML compatible syntax. Tools designed to work with XML can be used with JSP code.
- ❑ XML content can be natively added in to a JSP document using the new XML syntax.
- ❑ A container that follows the JSP 1.2 specifications will automatically validate XML syntax JSP. Same as every other XML compatible language you will be forced to use correct XML syntax when using XML syntax JSP code.
- ❑ JSP can now be 100% dynamic in XML syntax. Complete applications can be produced on the fly using already existing XML tools. The usefulness of this will have to be seen as more developers have time to work with the JSP 1.2 specification.

and disadvantages:

- ❑ Full support for JSP 1.2 specifications will take time to implement. First generation utilities will not truly show all the impacts of XML syntax JSP.
- ❑ Projects using JSP documents will require everyone coding to know basic XML syntax. XML does have a learning curve regardless if it is an easy one.

- ❑ A web designer using JSP as HTML with custom tags will not be able to use JSP in the same method if the XML syntax is being used.

Making JSP Compatible with XML

XML syntax JSP is not a very difficult transition from traditional JSP. Since parts of JSP were already XML based some tags had no need for a change. Here is a summary of some of the important features of JSP in XML syntax and a chart for some of the easy changes:

- ❑ XML syntax JSP works the same as JSP always has and uses a JSP 1.2 compliant container. If you are already using a JSP 1.2 compliant container you will not have to install any new software.
- ❑ When using an include directive it is valid to have the included fragment be in either syntax; however, the same fragment cannot intermix syntax styles.
- ❑ If a tag is already XML compliant, the syntax is unchanged.
- ❑ Request-time attributes are in the form of '%= text %' with optional white space around the argument.
- ❑ Plain text may not be outside a tag because it breaks XML rules. New 'cdata' tags are used for this.

A lot of JSP syntax is already XML compliant including any custom tag library. For the tags that did need a change of syntax it was usually intuitive. Here is a chart of XML versus traditional JSP syntax from the JSP 1.2 proposed final draft specification. After the chart we will give a more in depth explanation of some of the more significant changes:

Traditional JSP Syntax	XML JSP Syntax
<% page ... %>	<jsp:directive.page ... />
<%@ taglib ... %>	See <jsp:root> element
<%@ include ... %>	<jsp:directive.include ... />
<%! ...%>	<jsp:declaration> ... </jsp:declaration>
<% ...%>	<jsp:scriptlet> ... </jsp:scriptlet>
<%= ...%>	<jsp:expression> ... </jsp:expression>

As you can see we were not lying when saying most of the changes were intuitive. Scriptlets, expressions, declarations and directives are now just placed in a self-named XML tag. Only the `taglib` directive is left as a slight mystery in the above chart. Below we will cover the `taglib` directive mystery as well as some of the new additions needed for JSP in XML syntax.

<jsp:root> ... </jsp:root>

The root element for all XML syntax JSP documents is `jsp:root`. Inside this element is an attribute `xmlns` that allows for all of the standard JSP 1.2 XML tags to be used as well as any custom taglibs. The `xmlns` attribute follows the syntax of `xmlns:prefix="uri"`.

Here is an example given in the JSP 1.2 proposed final draft:

```
<jsp:root
  xmlns:jsp="http://java.sun.com/jsp_1_2"
  xmlns:prefix1="URE-fortaglib1"
  xmlns:prefix2="URI-for-taglib2" ... >
  JSP Page
</jsp:root>
```

This example shows how the required first `xmlns` attribute is used to enable the standard JSP document tags. The next prefix declarations would be for other taglibs being used in the JSP document. No other attributes should be declared inside the `jsp:root` element.

<jsp:cdata> ... </jsp:cdata>

Something new with JSP but familiar in XML is CDATA. CDATA is information that does not pertain to the XML structure but needs to be included in your document. You can think of this as the plain text you normally would write outside of tags in HTML. This plain text is referred to as the template data stored within your `jsp:cdata` tags.

Here is the example given in the JSP 1.2 proposed final draft:

```
<jsp:cdata> template data </jsp:cdata>
```

template data would be passed to the current value of `out`.

`jsp:cdata` tags have no attributes and are allowed anywhere that template data may appear. In your final JSP document template data should only appear inside a `jsp:cdata` element. If you want to use XML compliant tags outside of `jsp` namespace tags you may. Content used in a JSP document outside of `jsp` namespace tags is called an XML fragment.

XML Fragments

XML fragments are allowed anywhere that a `jsp:cdata` is allowed in a JSP document. The interpretation of the XML fragment gets passed to the current output of the JSP page with the fragment's white space preserved. A key difference between `jsp:cdata` and XML fragments is that an XSL transformation will see an XML fragment; however, it will not see the content of a `jsp:cdata` tag. Remember anything inside a `cdata` section is considered content of the `cdata` tag.

An Example JSP Document

Now let's look at a real example of a JSP page in the older style syntax and then the same page converted to XML syntax JSP. Earlier in the chapter we created a tag library for a web site using XML content. For this example we will take that same page and rewrite it in to a JSP document.

Here is the previous example:

```
<%@ taglib uri="http://www.jspinsider.com/jspkit/JAXP" prefix="JAXP"%>
<JAXP:transformer>
  <JAXP:xmlFile>c:/xml/links.xml</JAXP:xmlFile>
  <JAXP:xslFile>c:/xml/links_html.xsl</JAXP:xslFile>
</JAXP:transformer>
```

And now here would be the equivalent in XML syntax. Keep in mind not all JSP containers yet support the JSP 1.2 specifications, though our examples work on Tomcat 4.0:

```
<jsp:root
  xmlns:jsp="http://java.sun.com/jsp_1_2"
  xmlns:JAXP="http://www.jspinsider.com/jspkit/JAXP">
<JAXP:transformer>
  <JAXP:xmlFile><jsp:cdata>c:/xml/links.xml</jsp:cdata></JAXP:xmlFile>
  <JAXP:xslFile><jsp:cdata>c:/xml/links_html.xsl</jsp:cdata></JAXP:xslFile>
</JAXP:transformer>
</jsp:root>
```

The above was rather easy and it should not be much of a shock. While XML syntax JSP is new it is not a totally different beast. Both of the above examples will work identically in a JSP 1.2 container.

The decision to use XML syntax JSP should be based on your needs. Not all projects will benefit from using XML syntax. Along with the benefits of XML specifications also comes the drawback of well-formed coding rules. Sometimes it will just be easier to write your JSP in the normal syntax and allow for non-XML programmers to edit away. Also XML syntax can dramatically increase the size of a simple JSP page. Overall the new syntax for JSP should be seen as an additional tool to be used as needed. The original syntax of JSP can still be a blessing.

Dynamic JSP Made Easy with XML

To end the chapter we wanted to show you one of the most powerful aspects of JSP in XML syntax.

With JSP in XML syntax we can now use all the already existing XML tools to make dynamic JSP. In the past this was rarely done since parsing a JSP page would be a difficult and slow task. Now with the combination of the XML, JSP syntax and XML tools, new doors of opportunity have been opened to the programmer.

It should be noted right away that this is not a technique you will want to use everywhere since it comes at the high price of interpreting your JSP page every time you modify it. However, in certain circumstances there can be practical purposes for this technique. Some of the ways this technique can be useful are:

- ❑ JSP sites can be self-updating. At first glance this doesn't seem like a big issue, after all, database sites already change when you update the database. However, changing the database doesn't change the internal logic of the JSP pages that drives the site. With JSP you now have easy access to the code stored on your page. This opens up the possibility for a web site to grow and expand itself driven by its own logic.
- ❑ It is common to see an IDE tool generically build a site quickly for a user. What these tools usually don't do would be to go back and change a site once the tool has generated the initial version. Since a JSP site can be 100% XML compatible, it becomes easy for the tool to go back and re-modify the site. Over the next few years some extremely advanced JSP site management tools might appear.

We will build a simple example which will show you how to dynamically modify an XML based JSP. This example uses JDOM and we will also expand your knowledge of how to use JDOM.

An Example Dynamic JSP

This example consists of three parts. The first page is `dynamic_test.jsp`, which is the JSP page we will dynamically modify with another JSP page. The second page is `dynamic_link.jsp` which when called modifies `dynamic_test.jsp`. Finally you will need the `links.xml` from the previous examples. This XML file will provide the data which we will use to add links to our `dynamic_test.jsp` page.

The `dynamic_test` JSP

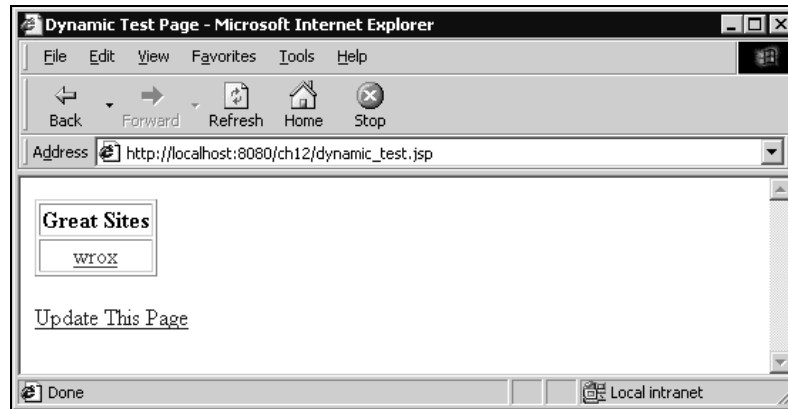
Keep in mind your JSP container must support JSP 1.2 to run this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root xmlns:jsp="http://java.sun.com/jsp_1_2">
  <html>
    <head><title> Dynamic Test Page </title>
      <ExampleCount views="0">
        <jsp:scriptlet> String ls_Display = ""; </jsp:scriptlet>
      </ExampleCount>
    </head>
    <body>
      <ExampleLinks>
        <table border="1">
          <tr><th> Great Sites </th></tr>
          <tr>
            <td align="center">
              <a href="http://www.wrox.com">wrox</a>
            </td>
          </tr>
        </table>
      </ExampleLinks>
      <p><jsp:expression>ls_Display</jsp:expression></p>
      <p><a href="dynamic_links.jsp">Update This Page</a></p>
    </body>
  </html>
</jsp:root>
```

The first thing you will notice in this example is that it uses the new XML based JSP syntax. The actual XML syntax isn't that strange; however, you will notice two tags `<ExampleCount>` and `<ExampleLinks>`. These tags are just XML tags we added to the page to make it easier to perform our modifications. We could have called the tags anything we wanted as long as we followed the rules of XML.

The `<ExampleCount>` tag is used to keep track of how many times this page gets modified and also to store our simple JSP logic that we want to modify. The `<ExampleLinks>` tag is where we are storing the data for the links we dynamically add to the page. We didn't need to create these tags; however, by using these tags we now have extremely convenient markers within our page to help us both navigate the JDOM model and to insert our modified code within.

The first time you run this page it will look like this:



To make life simple we added a link on the first page to execute the second page. The second file is where we will perform all of our work that will update the `dynamic_test.jsp` page.

The `dynamic_links JSP`

This file is saved to the same directory in which you saved the `dynamic_test.jsp` file:

```
<%@page contentType="text/html"%>
<% page import="java.io.*,
               java.util.*,
               org.jdom.*,
               org.jdom.input.SAXBuilder,
               org.jdom.output.*" %>

<%

/* determine where all our files are located */
String ls_path = request.getServletPath();
```

The first logical step is to load in our XML file, in this case our first JSP page:

```
ls_path = ls_path.substring(0,ls_path.indexOf("dynamic_links.jsp"));

String ls_Jsp_Template = application.getRealPath(ls_path + "dynamic_test.jsp");
String ls_XML_Links = "c:/xml/links.xml";

SAXBuilder builder = new SAXBuilder("org.apache.xerces.parsers.SAXParser");
Document l_jspdoc = builder.build(new File(ls_Jsp_Template));
```

Once we have our JSP page represented as a JDOM structure we will define the base HTML element of our page for a common reference point. It should be noted that within JDOM if you 'drill down' through the XML with the wrong path you would receive a `NullPointerException` error.

So if you are getting a mysterious `NullPointerException` check the syntax in the code matches to the actual XML layout. So for example if in this code if we had used `getChild("HTML")` it would have caused a `NullPointerException` since XML is case sensitive and "HTML" is considered to be a different node from "html":

```
Element l_page = l_jspdoc.getRootElement().getChild("html");
```

We then determine how many times we have modified the page and increment the count. The actual count is saved within the `<ExampleCount>` element stored in the attribute `views`. Retrieving and saving an attribute is similar to working with an element (or node). The main difference is in using the `Attribute` object associated with the `Element` object rather than just the `Element` object:

```
Element l_count = l_page.getChild("head").getChild("ExampleCount");
String ls_number = l_count.getAttributeValue("views");
int li_modcount = Integer.parseInt(ls_number) + 1;

ls_number = Integer.toString(li_modcount);
l_count.getAttribute("views").setValue(ls_number);
```

After updating the modify count the next step is to update the JSP scriptlet. This means we must get the data from within the `<jsp:scriptlet>` tag. This tag is using a namespace (the `jsp` portion of `jsp:scriptlet`). Working with namespaces is a little more complicated than working with a plain tag. It requires defining a `Namespace` object and then using this `Namespace` object as part of the key for determining the element:

```
Namespace jsp = Namespace.getNamespace("jsp", "http://java.sun.com/jsp_1_2");
Element l_change = l_page.getChild("head").getChild("ExampleCount")
    .getChild("scriptlet", jsp);

String ls_message = "This page has been modified : " + ls_number + " times.";
l_change.setText("String ls_Display =\" + ls_message + "\";");
```

Now we are ready to import the `links.xml` file. The code here is similar to previous code. The file is imported and we create a `Document` object. Once we have a document, it is possible to get a reference to the basic XML structure. In addition, a handle to the `<ExampleLinks>` tag is created so we can update our JSP page:

```
Document l_doc = builder.build(new File(ls_XML_Links));

Element root = l_doc.getRootElement();
Element example_links = l_page.getChild("body").getChild("ExampleLinks");

List l_links = root.getChildren("link");
```

The `setText()` function is used to set the `<ExampleLinks>` element to `null`. This effectively clears the element and all of its children, letting us start with a clean slate:

```
example_links.setText(null);
```

Now for the fun part. In the past, the task of modifying HTML consisted of appending many strings together. This style of coding has the problem of being not very readable and difficult to debug at times. We are modifying an XML file and the table we are adding is an XML data structure.

So instead of appending many strings, we create the elements we need, loop through the data, populate the elements with data and finally add the elements to our node (in our case the `<ExampleLinks>` node). The code is much cleaner and easier to automate than the older append string methodology:

```

Element l_table = new Element("Table");
Element l_tr = new Element("tr");
Element l_th = new Element("th");
Element l_td = new Element("td");
Element l_anchor= new Element("a");

l_th.setText(" Great Sites ");
l_tr.addContent(l_th);
l_table.addContent(l_tr);
l_table.addAttribute("border", "1");

Iterator l_loop = l_links.iterator();
while ( l_loop.hasNext() ) {
    Element l_link = (Element) l_loop.next();
    l_tr = new Element("tr");
    l_td = new Element("td");
    l_anchor = new Element("a");
    l_anchor.addContent(l_link.getChild("author").getText());
    l_anchor.addAttribute("href", l_link.getChild("url").getText());
    l_td.addContent(l_anchor);
    l_tr.addContent(l_td);
    l_table.addContent(l_tr);
}

example_links.addContent(l_table);

```

After we are done modifying our JSP page we need to rewrite the results back to the original file:

```

try {
    FileOutputStream l_write_file = new FileOutputStream(ls_Jsp_Template);
    XMLOutputter l_format = new XMLOutputter();
    l_format.output(l_jspdoc, l_write_file);
}
catch (IOException e) {
    out.print(e.toString());
}
%>

```

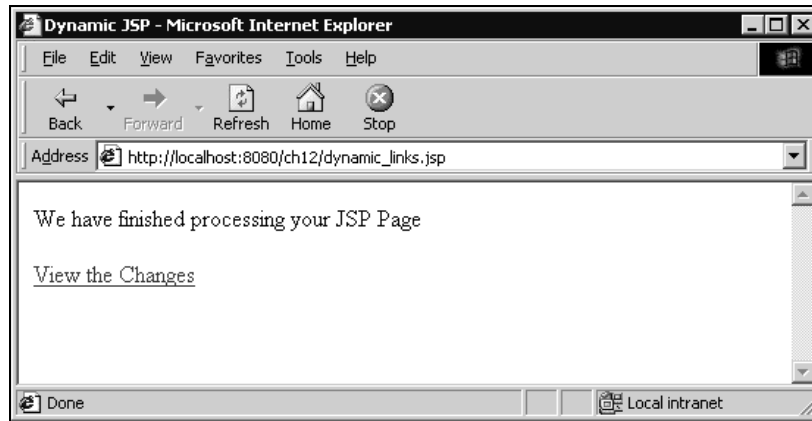
Finally when done, a simple HTML page will be displayed to indicate we have processed the first JSP page:

```

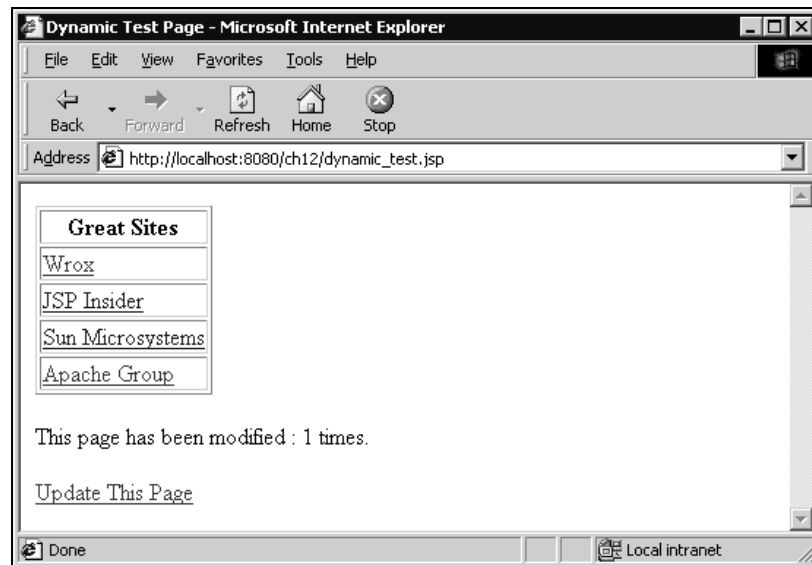
<html><head><title>Dynamic JSP</title></head>
<body>
    We have finished processing your JSP Page <br>
    <p><a href="dynamic_test.jsp">View the Changes</a></p>
</body>
</html>

```

When you run this page it will produce a screen indicating when it gets done processing which will look like:



So at this point you go back to our first page to find it has indeed been modified to look like this:



And more importantly our JSP page has now been changed to be a totally new page.

The following code is from `dynamic_test.jsp` after being modified by the `dynamic_link.jsp` page. This code has been reformatted to fit in this page. The actual output is identical but without the line breaks. The presence or non-presence of line breaks has no effect on the functionality of the code:

```
<?xml version="1.0" encoding="UTF-8"?><jsp:root
xmlns:jsp="http://java.sun.com/jsp_1_2">
<html>
  <head><title> Dynamic Test Page </title>
  <ExampleCount views="1">
  <jsp:scriptlet>
    String ls_Display ="This page has been modified : 1 times.";
```

```

    </jsp:scriptlet>
  </ExampleCount>
</head>
<body>
  <ExampleLinks>
    <Table border="1"><tr><th> Great Sites </th></tr>
      <tr><td><a href="http://www.wrox.com">Wrox</a></td></tr>
      <tr><td><a href="http://www.jspinsider.com">JSP Insider</a></td></tr>
      <tr><td><a href="http://java.sun.com">Sun Microsystems</a></td></tr>
      <tr><td><a href="http://jakarta.apache.org">Apache Group</a></td></tr>
    </Table>
  </ExampleLinks>

  <p><jsp:expression>ls_Display</jsp:expression></p>

  <p><a href="dynamic_links.jsp">Update This Page</a></p>

</body>
</html>
</jsp:root>

```

Keep in mind this is a simple example. Use your creativity to expand it. For example, we could have modified this JSP page to keep track of the last time it was modified by the server. Then it would become possible to set a time when the page would automatically run the update process on its own. So you could fine-tune your JSP page to run its own scheduler to update on the first of every month, or any time period you specify.

Summary

Using JSP with XML can add a lot of flexibility and functionality to your JSP. Some of the important points we covered are:

- ❑ XML allows for information to be dealt with in a manner that is easily used by both people and machines. Content stored in XML syntax can easily be reused and changed into many different formats. The only downside to XML is that it will never be optimised for speed on a specific project.
- ❑ To use XML with JSP you need to have a parsing utility to create a Java object from serialized XML. The W3C makes the official recommendations for XML and many open-source parsers are available that follow these specifications.
- ❑ There is no one best way for using XML with JSP. The DOM, JDOM and SAX are all excellent tools and should be used when best suited for your project.
- ❑ XSLT is an XML syntax language used for XML transformations. Many browsers are starting to integrate support for these transformations and many Java utilities exist for these transformations. Using XSL to style XML has many benefits such as reusability and spreading your server workload to your clients.
- ❑ JSP 1.2 specification allows for fully compatible XML syntax JSP. Since JSP is already semi-compliant with XML syntax the new format is not hard to use. Since XML based JSP syntax is new it will take a little time before the full power of this development can be taken for best advantage.

Even though we covered quite a bit of ground, this chapter only gives a brief glimpse of using XML with JSP. If this entire book focused on using JSP with XML even then it would not have been enough to fully document all the possibilities. Remember the examples from this chapter but don't be limited by them. With JSP and XML together you have a very powerful tool to work with.

The next chapter examines how we can use the JDBC API to access relational databases.

