Beginning JSP 2: From Novice to Professional

PETER DEN HAAN, LANCE LAVANDOWSKA,
SATHYA NARAYANA PANDURANGA,
AND KRISHNARAJ PERRUMAL

EDITED BY MATTHEW MOODIE

Apress™

ISBN (pbk): 1-59059-339-1

Printed and bound in the United States of America 10987654321

Lead Editor: Matthew Moodie

Technical Reviewers: Scott Davis and Matthew Moodie

Editorial Board: Steve Anglin, Dan Appleman, Gary Cornell, James Cox, Tony Davis, John Franklin, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Julian Skinner, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Project Manager: Sofia Marchant

Copy Manager: Nicole LeClerc

Copy Editor: Kim Wimpsett

Production Manager: Kari Brooks

Production Editor: Laura Cheu

Compositor: Kinetic Publishing Services, LLC

Proofreader: Liz Welch

Indexer: Michael Brinkman

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 8

Using JSP and XML Together

Extensible Markup Language (XML) has become the de facto standard for data interchange on the Internet these days. It has revolutionized the way the Web works by defining a standard for electronic exchange of information. It has done the same thing for data that Java did for code—it has made it portable.

What is XML, and why is it important? We'll try and answer these and other questions you may have in this chapter, and we'll show how you can use XML in your JavaServer Pages (JSP) applications.

Introducing XML

Before delving deeper into the guts of XML, you'll examine the acronym XML itself. X, for Extensible, means you can extend the language to meet various requirements. ML, for Markup Language, means it's a language for identifying structures within a document.

XML is extensible because it isn't a fixed language. You can extend it to create your own languages in order to match your particular needs. In a way, XML isn't really a language in itself; rather, it's a standard for defining other languages to fit various computing scenarios, typically in the business or academic spheres.

A markup language is used to add meaning to different parts of a document. Hypertext Markup Language (HTML) is another type of markup language. Its tags give particular meaning to parts of a document. For instance, the `<table>` tag *marks up* a section to represent a table.

To clarify things further, you'll look at some XML documents. The following is a file called `web.xml` used to configure Tomcat:

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

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
      version="2.4">
</web-app>
```

Another XML-like file type you're acquainted with is an HTML file:

```
<html>

  <head><title>A date formatting form</title></head>

  <body>
    <form action="./format.jsp" method="POST">
      <table>
        <tr>
          <td>Date to format (mm/dd/yyyy): </td>
          <td><input type="text" name="date"/></td>
        </tr>
        <tr>
          <td>Format: </td>
          <td><input type="text" name="format"/></td>
        </tr>
      </table>
      <input type="submit"/>
    </form>
  </body>

</html>
```

In many ways, XML is similar to HTML. Both these markup languages use tags to enclose items with particular meaning; these tags are enclosed within angle brackets, as in `<table>`, `<web-app>`, `<html>`, and so on. Also, there will be a corresponding closing tag that has the same name but starts with `</`. These closing tags mark the end of the item referred to by the tag, and in XML, the whole section including the start tag and the end tag is known as an *element*. Note that although many HTML elements don't require a closing tag (such as `<p>` and `<br>`), XML elements must always have a start and an end tag. Both HTML and XML allow elements to be contained, or *nested*, within each other, as you can see from the previous examples (the `<tr>` element is nested within the `<table>` element because it appears after the `<table>` start tag and before the `</table>` end tag). However, XML is much stricter than HTML in this regard because it doesn't allow elements to overlap (that is, if an element's start tag appears after another's start tag, the nested element's end tag must appear before the other's end tag).

There are many other ways in which XML is quite different from HTML. XML isn't limited to a preexisting set of tags, as HTML is, and although HTML is primarily concerned with describing how a document should be laid out in a browser, XML documents are more concerned with describing the data contained in a document, and they're generally quite independent of how that data may be rendered for display.

Both these documents use an XML language, often called an *XML dialect*, to give meaning to the data they contain. The actual markup tags (or XML *elements*) each uses are different, though. For instance, `web.xml` starts with the `<web-app>` element, and the first element in an HTML file is `<html>`.

Methods exist to enforce the dialect so that a document is always understandable. In `web.xml` the `<web-app>` element contains a reference to an XML Schema. An XML Schema is a document that defines what can and can't appear in a dialect and is itself an XML document. Another method for defining a dialect is a Document Type Definition (DTD). These aren't XML documents but rather are written in their own language. You'll see more of these later the "Well-Formed vs. Valid Documents" section.

Understanding the Structure of XML Data

XML is a vendor-neutral standard, regulated by the World Wide Web Consortium (W3C; `http://www.w3c.org/`). You can find the specification at `http://www.w3c.org/TR/WD-xml`, which contains the formal details of XML syntax. We'll cover the essentials here, starting with a look at a very simple XML document:

```
<?xml version="1.0"?>
<!DOCTYPE Book SYSTEM "book.dtd">
<Book>
  <Author-Name>
    <Last>
      Einstein
    </Last>

    <First>
      Albert
    </First>
  </Author-Name>

  <Book-Name>
    General Relativity
  </Book-Name>

  <Edition Year="1930"/>
  <Bestseller/>
</Book>
```

XML documents are composed of data, enclosed within tags that describe that data. There may also be *processing instructions*, which provide specific details required for an application that will read the XML document, and other elements, such as the `xml` declaration or a DTD declaration as shown here.

Note that the `xml` declaration starts with `<?` and ends with `?>`, and it doesn't require a closing tag. This notation is also used for processing instructions, but despite the common misconception, the `xml` declaration isn't strictly speaking a processing instruction. The `xml` declaration may also specify `encoding` and `standalone` attributes, which specify the character encoding used by the document and whether the document depends on any others:

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

The DTD declaration, which starts with `<!` and ends with `>` and also doesn't require a closing tag, specifies what the name of the first element in the file must be (which here is `Book`) and also specifies where to find the DTD file that details rules that XML elements in this particular dialect must obey. The `SYSTEM` part indicates that the DTD can be found at either a relative or an absolute uniform resource locator (URL). If, however, the DTD is officially sanctioned by a standards body, you'd use `PUBLIC` instead.

Note that DTDs are falling out of favor now, and the more recent XML Schema standard is often preferred because it lets you specify more rigorous rules for an XML dialect. However, DTDs are still used in many companies, partly because of legacy issues and also because DTDs are somewhat easier to create. We'll cover this issue further in the section "Well-Formed vs. Valid Documents."

The actual content of an XML document (that is, the data and XML elements it contains) must obey certain syntax rules as we've hinted at already. We've already mentioned the first element in a document, and this element is known as the *root element*—all other elements must be contained within it. That is to say, all valid XML documents must have one and only one root element. In the XML file shown previously, the root element is `<Book>`.

All XML elements can contain other XML elements and/or text data as required; such contained elements are known as *child elements* of the containing element. For instance, the previous `<Author-Name>` element has the child elements `<Last>` and `<First>`.

If an element has neither child elements nor text data—that is, if it could be written as so:

```
<Bestseller>
</Bestseller>
```

then you can refer to it using a shorthand form like this:

```
<Bestseller/>
```

Element names can be almost anything you like as long as they start with either a letter or an underscore and don't contain whitespace (such as return characters or spaces). Be aware that XML element names are case-sensitive. As stated previously, all XML elements must have both a start tag and an end tag,

unless they use the short form shown previously (`<Bestseller/>`). As we've also said already, XML elements may not overlap. The following is invalid nesting:

```
<Book>
  <Author>
  </Book>
</Author>
```

The following is valid nesting:

```
<Book>
  <Author>
  </Author>
</Book>
```

Because the `<Book>` element is the first to be opened, it should be the last to be closed. In the invalid example, its closing element appears before `</Author>` and is therefore wrong.

XML documents that satisfy these rules are known as *well-formed* documents.

Attributes

Attributes of an element are listed in the element's start tag and give further meaning to that element. Consider the `<Edition>` element from the example:

```
<Edition Year="1930"/>
```

Its `Year` attribute has the value `1930`. Attributes follow the same rule for names as elements and must have their value between quotes. Only one attribute of the same name may be given for an element.

Comments

To improve the readability of your XML documents, you can place comments within them using the same syntax as used in HTML files: You place them between `<!--` and `-->` sequences. Well-placed comments can be invaluable when others try to read your XML files (or even when you read them a while after writing them!). As in all programming contexts, prudent use of comments is a very good habit to have. Note that comments don't form part of the actual content of an XML document and may be ignored when the document is read, so you should avoid using code that depends on them.

Entity References

There are cases when the data in an XML document needs to contain characters that normally have a special purpose, such as the less-than symbol (<) or the apostrophe ('). You can represent these using *entity references*, just as you would in HTML.

An entity reference uses the syntax &entityname; XML has a total of five entity references (see Table 8-1).

Table 8-1. XML's Entity References

Entity Reference	Character	Notes
<	<	lt stands for *less than.*
>	>	gt stands for *greater than.* Only required in attribute values.
&	&	The ampersand sign.
"	"	Double quotes.
'	'	A single quote—the apostrophe.

So, if you had a book with the title *Learn HTML from < to >*, you could describe it by using the following XML element:

```
<Book>
  <Book-Name>
    Learn HTML from &lt; to >
  </Book-Name>
</Book>
```

You don't need to use > for the > character because when this character appears in text data, it's clear that it doesn't mark the end of a tag.

Character DATA (CDATA) Sections

Entity references let you use characters that normally have a reserved meaning but wouldn't be a great solution if your text data contains many instances of such characters.

The better solution is to place such data inside a character DATA (CDATA) section. These sections begin with the sequence <![CDATA[and end with]]>. Text

within such sections can contain any characters at all, and their content is preserved as it appears, including any whitespace. Say you had an XML element that contained programming code like so:

```
<Code>
for(int i = 0; i < 10; i++)
{
  if(i < 5) System.err.println("I would rather be fishing.");
  if(i == 5)System.err.println("I would rather be at my PlayStation.");
  if(i > 5) System.err.println("I would rather be at DreamWorld.");
}
</Code>
```

As you can see, this code contains a lot of special characters, and it's formatted in a way you may want to keep. Thus, it'd probably be a good idea to put the whole section in a CDATA section:

```
<Code>
<![CDATA[
for(int i = 0; i < 10; i++)
{
  if(i < 5) System.err.println("I would rather be fishing.");
  if(i == 5)System.err.println("I would rather be at my PlayStation.");
  if(i > 5) System.err.println("I would rather be at DreamWorld.");
}
]]>
</Code>
```

Well-Formed vs. Valid Documents

As mentioned before, XML documents follow a set of rules that dictates how an XML document must be structured. XML documents that satisfy these rules are said to be *well-formed*. This isn't to be confused with the similar concept of validity. A *valid* XML document conforms to the rules specified in the corresponding DTD or Schema for that dialect. The DTD (or XML Schema) details rules that documents in a particular XML dialect must obey. They specify which elements are defined by that dialect and the attributes that particular elements can have.

Thus, although you can say that the previous example file, `book.xml`, is well-formed because it satisfies the rules of XML, you can't say just by looking at it whether it's valid. To do that, you'd need to see the DTD that it specifies, namely `book.dtd`. Look at that DTD file now:

```
<!ELEMENT Book   (Author-Name+, Book-Name, Edition?, Bestseller?) >
<!ELEMENT Author-Name  (Last, First, Middle?) >
<!ELEMENT Last       (#PCDATA) >
<!ELEMENT First      (#PCDATA) >
<!ELEMENT Book-Name (#PCDATA) >
<!ELEMENT Edition EMPTY>
<!ATTLIST Edition
           Year   CDATA  #REQUIRED>
<!ELEMENT Bestseller EMPTY>
```

The first thing to notice is that all items of a DTD start with `<!` and end with `>`. Items define either elements or attributes.

Elements

The `<!ELEMENT>` item describes an element that may appear in XML documents that conform to this DTD:

```
<!ELEMENT Book   (Author-Name+, Book-Name, Edition?, Bestseller?) >
```

This line describes an XML element called `Book` and states that the `<Book>` element can have the child elements named in the comma-separated list in parentheses. The order in which these elements are listed in the brackets corresponds to the order that these elements must appear in XML documents. If the order doesn't matter, you can list the elements using just a space rather than a comma to separate them:

```
<!ELEMENT Book   (Author-Name+ Book-Name Edition? Bestseller?) >
```

The `+` and `?` characters indicate how many times the preceding element may occur, according to Table 8-2.

Table 8-2. XML Symbols

Symbol	Meaning
,	Strict ordering: Elements must be in the specified order.
+	One or more.
*	Zero or more.
?	Optional (zero or one).

So the DTD states that the <Author-Name> element may appear one or more times, the <Edition> element is optional, and so on. If no symbol is present, that element may appear only once, as is the case with the <Book-Name> element.

The <Last> element follows this rule:

```
<!ELEMENT Last      (#PCDATA) >
```

This simply means that this element must contain text data only, as indeed it does in the book.xml file:

```
<Last>
  Einstein
</Last>
```

The final element to look at is this:

```
<!ELEMENT BestSeller EMPTY>
```

EMPTY indicates that the <Bestseller> element must always be empty and not contain any textual data or other nested tags. In other words, it must be either this:

```
<Bestseller/>
```

or this:

```
<Bestseller></Bestseller>
```

Attributes

The only remaining item in the DTD we haven't discussed is that for the <Edition> element. There are in fact two DTD items that relate to this element:

```
<!ELEMENT Edition EMPTY>
<!ATTLIST Edition
          Year   CDATA  #REQUIRED>
```

These two rules state that the <Edition> element may not contain any text data or child elements and that it has an attribute called Year. This attribute is required, and it's of the type CDATA (character data). Simply omit the #REQUIRED keyword if the attribute isn't required.

An XML element may have multiple attributes, and a DTD can restrict each attribute to one of a given set of values. It can also specify one as a default like this:

```
<!ATTLIST Edition
          Year CDATA #REQUIRED
          Month (Jan | Feb | Mar) 'Jan'>
```

This would add a second attribute to the `<Edition>` element that may have a value of `Jan`, `Feb`, or `Mar`, and the default value is `Jan` if the attribute isn't specified.

The Problem with DTDs

DTDs have a number of problems:

- The DTD content model is hard to manage. In the DTD application, you've seen how to define a list of subelements with cardinality 1, but allowing them to appear in any particular order is difficult.

- DTDs aren't written in XML. This is a very big disadvantage. You can't use XML applications to read, edit, or build DTDs. If you want to write DTDs from a database or a document, it'd be easier if they could be written in XML.

- DTDs can't validate data types. If you want to restrict an element or an attribute to be a number, a date, or some other specific data type, you can't do that because DTDs can use only the `#PCDATA` and `CDATA` data types.

- DTDs can't validate element contents. You can use an enumeration to validate an attribute value against a list of admitted values, but you can't do the same with attributes. In many situations you'll want to define an element that can only have some specific values; this is very useful when you have to exchange information between different applications using XML.

- You can't use DTDs with namespaces. Namespaces are heavily used to combine two different vocabularies in the same XML document. Because the document can refer to only one external DTD, you can't use a DTD to validate a document that uses multiple namespaces.

That just about wraps it up for DTDs. You'll now look at XML Schemas, which are rapidly gaining ground over DTDs.

Defining Validity with Schemas

Before delving into the world of XML Schemas, you need to understand the concept of *XML namespaces*.

XML Namespaces

Namespaces crop up in other areas of programming as well as XML. In general, namespaces serve two basic purposes:

- To group related information under one umbrella
- To avoid name collision between different groups

Namespaces in XML also serve these purposes by associating elements that belong together with a unique identifier. The unique identifier is a uniform resource indicator (URI), and because these can be quite long and unwieldy, a shorthand form is almost always associated with a namespace. Elements that belong to that namespace are then prefixed by the short form, differentiating them from other elements with the same name but belonging to a different namespace. For instance, the XML file has an element called `<Last>` that contains an author's last name. This same element could quite easily be used in another XML dialect, perhaps one that describes the results of a book awards ceremony. This other dialect's `<Last>` element would probably have a quite different meaning, and by having a unique namespace for each dialect, you can quite easily have an XML document that contains both types of element without ambiguity.

You specify the namespace used through the `xmlns` attribute of the root element of an XML document like this:

```
<Book xmlns="http://www.apress.com/bookCatalog"
      xmlns:prize="http://www.apress.com/bookAwards">
```

The unique identifier for these namespaces is given by the value of the `xmlns` attribute in question, and the short prefix for that namespace is the part preceded by the colon (`:`).

You can see that the first namespace doesn't have any prefix—it's called the *default* namespace, and any elements that don't have a prefix are assumed to belong to it. The second namespace has a prefix of `prize`. The XML document can then contain an element such as this without any ambiguity between the two types of `<Last>` element:

```
<Book-Name>
  <First>
    Joey
  </First>
  <Last>
    Gillespie
  </Last>
```

```
  <prize:Awards>
    <prize:Last/>
  </prize:Awards>
</Book-Name>
```

XML Schemas

Schemas do the same thing as DTDs, but they overcome many of the shortcomings that DTDs exhibit when applied to XML. Many of their advantages actually stem from the fact that they are themselves written in an XML dialect.

When elements in an XML document must conform to a certain Schema, you specify where a copy of the appropriate Schema can be found using the `schemaLocation` attribute on the document's root element. This attribute is defined by the *schema instance namespace*, which you must therefore declare using an `xmlns` attribute, typically with the prefix `xsi`:

```
<Book
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="Book.xsd">
```

Note the file extension of `.xsd` for the Schema file (which stands for *XML Schema Definition*).

The following is this `Book.xsd` file, equivalent to the DTD you've already seen:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Book">
    <xs:complexType>
       <xs:sequence>
        <xs:element name="Author-Name">
          <xs:complexType>
            <xs:element name="Last" type="xs:string"/>
            <xs:element name="First" type="xs:string"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="Book-Name" type="xs:string"/>
        <xs:element name="Edition">
          <xs:complexType>
            <xs:attribute name="Year" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
```

```
        <xs:element name="Bestseller" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The first thing to spot is the first line of this Schema:

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

This shouldn't be surprising. It is, as we said before, an XML document. After this comes the `<schema>` root element, which defines the `xs` namespace.

The Root Element

The `<schema>` element is the root element of every XML Schema. Notice that its `xmlns` attribute defines the `xs` namespace that qualifies the `<schema>` element itself.

Usually, there are a couple more attributes—namely, `targetNamespace` and `elementFormDefault`. `targetNamespace` specifies the URI to uniquely identify this Schema, and `elementFormDefault` lets you require your elements to always be qualified with the namespace prefix:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.apress.com/begjsp2"
           elementFormDefault="qualified">
```

Elements

Elements within a Schema can be either simple or complex.

A *simple element* doesn't contain any nested elements and can't have attributes: It may contain text only between the start and the end tags. This is an example of a simple element from the previous example:

```
<xs:element name="Last" type="xs:string"/>
```

which corresponds to the `Last` tag in the XML:

```
<Last>
  Einstein
</Last>
```

Notice that the `type` attribute qualifies this element as a string type. This means the value of this element is to be interpreted as plain text. Other possible types include `decimal`, `integer`, `Boolean`, `date`, and `time`.

Finally, you can place restrictions on how many times this element may appear by the attributes `minOccurs` and `maxOccurs`. If neither is specified, the element can occur only once. `maxOccurs="unbounded"` allows unlimited occurrences of the element.

Complex Elements

Complex elements can contain child elements and attributes. This is how you define the `<Author-Name>` and `<Edition>` elements:

```
<xs:element name="Author-Name">
  <xs:complexType>
    <xs:element name="Last" type="xs:string"/>
    <xs:element name="First" type="xs:string"/>
  </xs:complexType>
</xs:element>
<xs:element name="Edition">
  <xs:complexType>
    <xs:attribute name="Year" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

You define child elements for `<Author-Name>` and an attribute for `<Edition>` using the Schema `<complexType>` element. Just as simple elements, attributes can have a default value or a fixed value specified. Any specified default value is automatically given for an attribute when no other value is supplied. If a fixed value is specified, no other value may be specified for that element.

All attributes are optional by default. To explicitly specify whether the attribute is optional, use the `use` attribute. It can have one of two values, `optional` or `required`.

Restricting the Content

You saw some examples of how to restrict the values contained in your XML documents when you learned how to use the `type` keyword. With XML Schemas, there are several other ways in which you can restrict content.

Restriction on Element Values

The following example restricts the value of the element `<length>` to the range from `5` to `10`:

```
<xs:element name="length">
 <xs:simpleType>
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="5"/>
    <xs:maxInclusive value="10"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
```

Restriction on a Set of Values

In the following example, the `<enumeration>` element restricts the value of the `<language>` element to `C++`, `SmallTalk`, or `Java`:

```
<xs:element name="language">
  <xs:simpleType>
   <xs:restriction base="xs:string">
    <xs:enumeration value="C++"/>
    <xs:enumeration value="SmallTalk"/>
    <xs:enumeration value="Java"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
```

To limit the content of an XML element to a series of numbers or letters, you use the `<pattern>` element:

```
<xs:element name="choice">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[abcd]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

In the previous example, the `<choice>` element can take a value between `a` and `d`. You could also do this like so:

```
<xs:pattern value="[a-d]"/>
```

The next example defines an element called `<license-key>` where the only acceptable value is a sequence of five digits, and each digit must be in the range `0` to `9`:

```
<xs:element name="license-key">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:pattern value="[0-9][0-9][0-9][0-9][0-9]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

The following defines an element called `<gender>`, which can be either `Male` or `Female`:

```
<xs:element name="gender">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="Male|Female"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

There are several other ways of restricting content. However, to cover them all would probably take another book. This concludes the discussion about XML Schemas. You'll next concentrate on how to read and write XML documents. It's time to get your hands dirty by writing some code!

Reading and Writing XML Documents

To be able to programmatically read and write XML documents, you need an *XML parser*. An XML parser is a program, typically an application programming interface (API) that can read (and write) XML documents. It *parses* the document (breaks it up into its constituent elements) and makes these available to your own programs. To do this, the XML document must generally be well-formed although some parsers can handle XML fragments (these are XML documents that lack a single root element). Validating parsers can tell you if it's valid according to a specified DTD or a Schema.

The following are some popular XML parsers:

- Apache Xerces: `http://xml.apache.org/xerces2-j/index.html`
- IBM's XML4J: `http://alphaworks.ibm.com/tech/xml4j`
- Microsoft's MXSML Parser: `http://msdn.microsoft.com/xml/default.asp`

This book concentrates on the Apache Xerces parser because it's part of the open-source project at Apache and freely available.

Parsers internally use a set of low-level APIs that allow them to interact with the XML documents in question. There are two approaches to parsing XML documents, known as the *push* and *pull* models. An example of a push parser is Simple API for XML (SAX) while the Document Object Model (DOM) uses a pull model.

The push model reads an XML document in sequence, firing events whenever a new part of the document is encountered (such as a start or end tag). You can link these events to your own methods that are then called when that event occurs. This model is quite lightweight and fast, but its main limitation is the sequential access, meaning you can't go back and forth through elements at random.

DOM, on the other hand, reads in the entire XML document at once, creating a model of the document in memory (the DOM tree). This allows you to jump about from element to element as you want, but its main drawback is that loading the whole document can consume a lot of memory and be relatively time consuming, particularly when a document is large.

Trying It Out: Downloading, Installing, and Running the Samples for Xerces

In this section, you'll download, install, and run some of the samples provided with the Xerces parser to illustrate how a parser works and to determine the well-formedness of an XML document. Follow these steps:

1. You can download Xerces from `http://xml.apache.org/xerces2-j/download.cgi`. Download the latest ZIP or TAR file depending on whether you're running Windows or Linux. Look for a file named `Xerces-J-bin.2.6.1.zip` or `Xerces-J-bin.2.6.1.tar.gz`.

2. Unzip the contents of this file into a folder of your choice, such as `C:\java`. The Xerces files will be placed in a subdirectory called `xerces-version_no` (depending on the actual version you downloaded).

3. Add the following paths to your `CLASSPATH` environment variable (as described in Chapter 1) to point to the Xerces Java Archive (JAR) files:

   ```
   ;C:\java\ xerces-2_6_1\xercesImpl.jar;
   C:\ java\ xerces-2_6_1\xercesSamples.jar;
   C:\ java\ xerces-2_6_1\xmlParserAPIs.jar
   ```

   Make sure that you include the first semicolon when adding this to your existing `CLASSPATH` and don't include any spaces. Note that you've included the `xercesSamples.jar` also because you'll run these samples. They aren't required to run the parser itself. Change the semicolons to colons if you're running Linux or Mac OS X.

4. Staying at the command prompt, navigate to the `samples` directory under the `xerces` folder and type the following:

```
> cd \java\xerces-2_6_1\samples
> javac ui\TreeView.java
```

5. Now run the compiled class like this:

```
> java ui.TreeView ..\data\personal.xml
```

This should bring up the utility that shows the structure of the XML document in the `data` folder called `personal.xml` (see Figure 8-1).

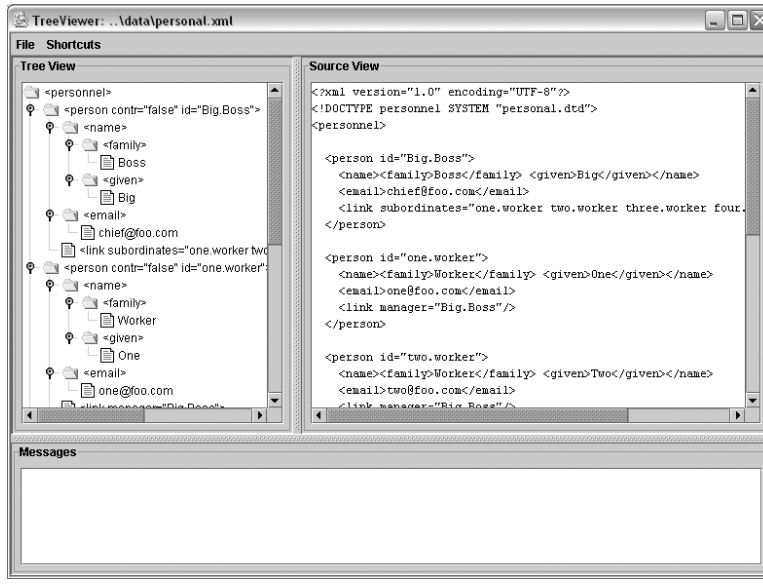

Figure 8-1. The `personal.xml` *structure*

6. Now open this `personal.xml` in a text editor, and remove the end tag for the first `<person>` element (the one with `id=Big.Boss`). Rerun the program by repeating step 5. This time, the screen will show you an error in big red letters, indicating the XML document isn't well-formed (see Figure 8-2).

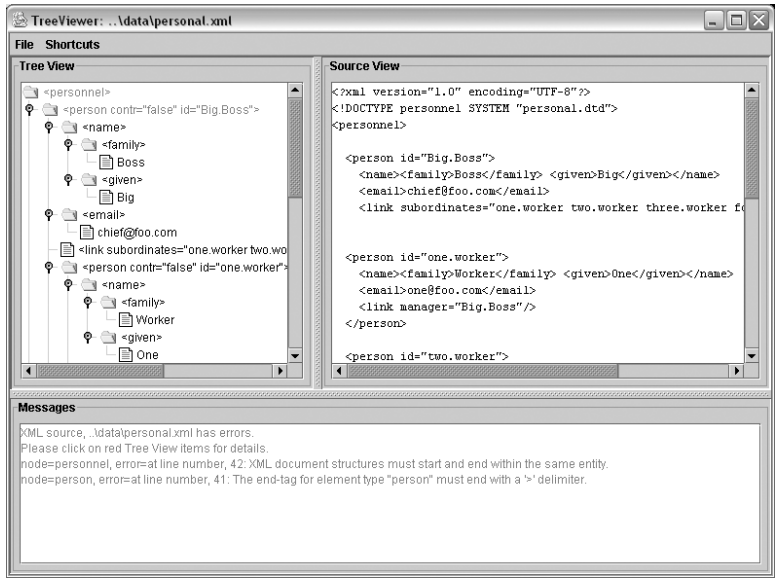

Figure 8-2. Getting an error

How It Works

This example reads and displays an XML document using a tree structure. The left pane shows the XML elements, and the right shows the actual document, which is basically the source view. The bottom shows messages, such as the error message you got when you removed the closing tag.

You can find the source code for this particular file, `TreeView.java`, in the `samples\ui` folder. A quick look at the source code will show that it uses the DOM as the low-level API for parsing the document (line 127 in the source code).

XML and JSP

Knowledge of XML is becoming increasingly necessary in all areas of programming, and especially so in the Web application sphere. You've seen XML in use in configuration files such as `web.xml`. Also, XML is finding a place in data sharing, data storing, messaging, and many other areas of application development. Some of the reasons for XML's widespread acceptance include the following:

Content in plain text: Because XML isn't a binary format, you can create and edit files with anything from a standard text editor to a visual development environment. That makes it easy to debug your programs and makes it useful for storing small amounts of data. At the other end of the spectrum, an XML front end to a database makes it possible to efficiently store large amounts of XML data as well. So XML provides scalability for anything from small configuration files to a company-wide data repository.

Data identification: The markup tags identify the information and break up the data into parts, an e-mail program can process it, a search program can look for messages sent to particular people, and an address book can extract the address information from the rest of the message. In short, because the different parts of the information have been identified, they can be used in different ways by different applications.

Ease of processing: As mentioned earlier, regular and consistent notation makes it easier to build a program to process XML data. And because XML is a vendor-neutral standard, you can choose among several XML parsers, any one of which takes the work out of processing XML data.

With these points in mind, you'll now look at some areas where you can apply XML in JSP.

Delivering XML Documents with JSP

So far you've learned a lot about JSP as a technology. You know that JSP pages produce HTML content so they can be displayed in a browser as a regular Web page. However, some devices can't interpret HTML and instead use another markup language called Wireless Markup Language (WML), which makes optimal use of the bandwidth and processing capability available in many mobile phones. Another exciting technology is VoiceXML-based voice services. These voice services let users interact with your applications through speech. One thing common with all these technologies is that the content is authored using XML. WML and VoiceXML are XML documents that conform to a particular Schema. To be able to display this XML in user-friendly output, you need to transform it into either HTML or WML as the case might be. A process known as, unsurprisingly, *transformation* can help you then use the same XML to create multiple forms of output. Transformation is done with the help of another standard XML dialect called Extensible Stylesheet Language (XSL). XSL is similar to Cascading Style Sheets (CSS) and defines styling information for HTML documents; it's also a specification regulated by the W3C. You'll see this in the "Understanding XSL Transformations" section later in the chapter.

If your Web application creates output as XML, you can apply XSL stylesheets to transform these documents into HTML, WML, or any other XML form that the browser in use may require (see Figure 8-3).

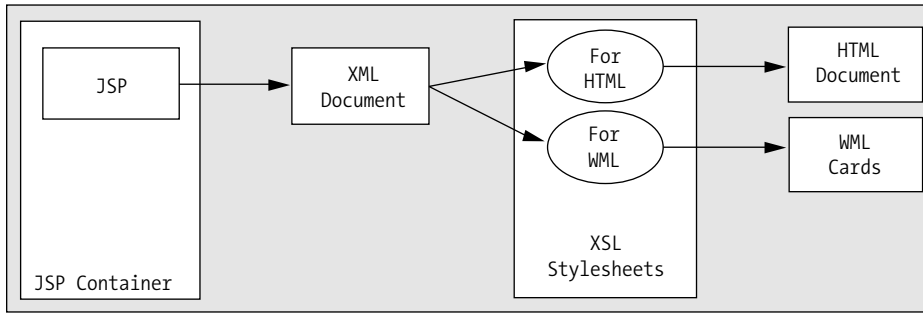

Figure 8-3. Transformation

Trying It Out: Creating XML with JSP

To create XML with JSP, follow these steps:

1. Create a subdirectory under `%TOMCAT%/webapps` called **XML**.

2. Create a subdirectory under `XML` called **WEB-INF**.

3. Copy the `lib` directory containing the JSTL JARs from a previous example to the `WEB-INF` directory.

4. Create the following JSP page and save it as **date.jsp** in `%TOMCAT%/webapps/XML`:

```
<?xml version="1.0"?>
<%@ taglib uri="http://java.sun.com/jstl/fmt_rt" prefix="fmt" %>
<jsp:useBean id="now" class="java.util.Date" />
<date-example>
  <title>Date Example</title>
  <content>
    <heading>Sample JSP XML File</heading>
    <text>The date and time is :
      <fmt:formatDate value="${now}" pattern="dd-MMM-yyyy hh:mm"/>
    </text>
  </content>
</date-example>
```

5. Now restart Tomcat. Visit `http://localhost:8080/XML/date.jsp` in your Web browser. It will produce XML as shown in Figure 8-4. This isn't how it will appear on the screen; you must view the source to see the XML because the browser won't render it.

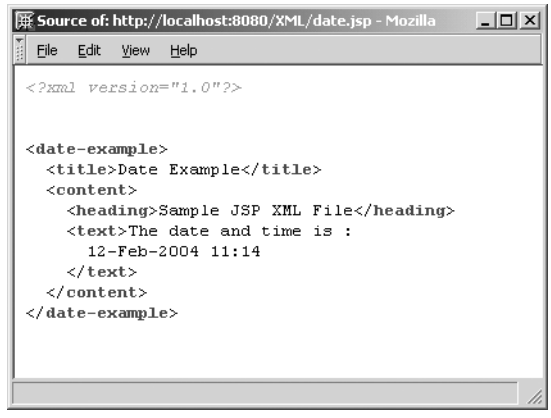

Figure 8-4. The XML produced by the JSP page

How It Works

As this example shows, creating XML output using JSP is just the same as creating HTML output. The JSP source contains the XML elements that you want to appear in the output, mixed in with the same JSP tags you'd use when creating HTML output. These JSP tags create dynamic content in your XML output just as they do when creating HTML output.

By applying a suitable XSL stylesheet, you can transform XML such as this into HTML or other kinds of documents, including Portable Document Format (PDF), WML, and VoiceXML. In other words, you can create the same output for all your users as XML and, simply by supplying suitable stylesheets, provide a readable display for their browser type.

JSTL XML Tags

JSTL XML tags provide easy access to XML content. The JSTL XML tags use XPath, another W3C standard, which provides a way to specify and select parts of XML documents.

There are three classes of JSTL XML tags:

- **Core**: Parse, read, and write XML documents.
- **Flow control**: Provide looping and decision capability based on XML content.
- **Transformation**: Provide utilities to transform XML content into other classes of documents.

The XML tags use XPath as a local expression language. XPath expressions are specified in `select` attributes. This means that only values specified for `select` attributes are evaluated using the XPath expression language. All other attributes are evaluated using the rules associated with the global expression language or the EL.

In addition to the standard XPath syntax, the JSTL XPath engine supports the following scopes to access Web application data within an XPath expression:

- `$param`: Request parameter
- `$header`: Header content
- `$cookie`: Cookie identification
- `$initParam`: Context parameters
- `$pageScope`: Any page scope variable
- `$requestScope`: To access request scope variables
- `$sessionScope`: Session scope variable access
- `$applicationScope`: Application scope variable access

These scopes are defined in the same way as their counterparts in the JSTL expression language. For example, `$sessionScope:profile` retrieves the session attribute called `profile`.

You'll now try out a few simple JSTL XML tags.

Trying It Out: Parsing XML with JSP

For this example, you'll need the JSTL XML tag library. Follow these steps:

1. Save the following as **parseXML.jsp** in the `XML` folder:

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

<html>
  <head>
    <title>JSTL XML Support -- Parsing</title>
  </head>
  <body bgcolor="#FFFFFF">
    <h3>Parsing an XML Document using XML tags</h3>
```

```
<x:parse var="xml">
  <BOOK>
   <AUTHOR>
    <LAST-NAME>
     EINSTEIN
    </LAST-NAME>
   </AUTHOR>
   <BOOK-NAME>
     GENERAL RELATIVITY
   </BOOK-NAME>
  </BOOK>
</x:parse>

<!--printing the LAST-NAME element-->
LAST NAME :
<x:out select="$xml/BOOK/AUTHOR/LAST-NAME" />
<br />


<!--printing the BOOK-NAME element -->
BOOK NAME :
<x:out select="$xml/BOOK/BOOK-NAME"/>

<hr />

  </body>
</html>
```

2. Visit `http://localhost:8080/XML/parseXML.jsp` in your Web browser. You should see a screen like Figure 8-5.

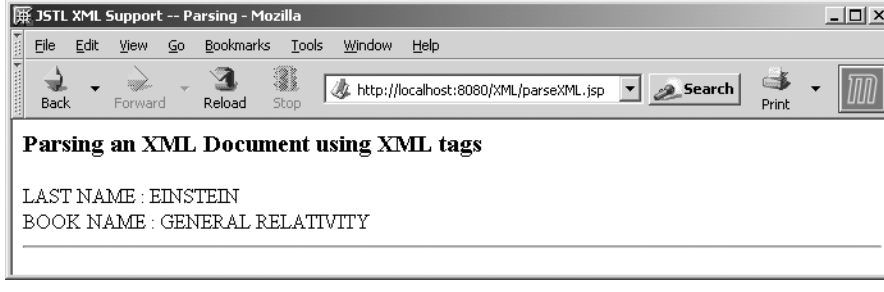

Figure 8-5. XPath has extracted the required information.

How It Works

Before the result can be displayed, you need to parse the XML document for which you use the `<x:parse>` tag as shown previously. Generally, you'd specify a URI rather than embedding XML in the JSP page by including an appropriate `uri` attribute on the `<x:parse>` tag:

```
<x:parse uri="/book.xml" var="xml"/>
```

The `var` attribute specifies the variable in which to store the parsed XML. This is then used in the XPath expression.

The root element in the previous XML document is named `<BOOK>`, which contains the two child elements, `<AUTHOR>` and `<BOOK-NAME>`. The element `<AUTHOR>` has another child element called `<LAST-NAME>`. Among these tags, only `<LAST-NAME>` and `<BOOK-NAME>` have content. You retrieve this content using the `<x:out>` tag:

```
<x:out select="$xml/BOOK/AUTHOR/LAST-NAME" />
```

Trying It Out: Using XML Flow Control Tags

In the previous example, you retrieved the required XML elements using XML tags. Now you'll look at another example, which iterates through all the elements in an XML document. This example demonstrates the use of XML flow control tags.

1. Save the following as **iterate.jsp** in the `XML` folder:

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

<html>
  <head>
    <title>JSTL XML Support -- Flow Control</title>
  </head>
  <body bgcolor="#FFFFFF">
    <h3>Iterating through an XML document</h3>

    <x:parse var="xml">
     <items>
      <item>
       <Fruit Name="Apple">Red Apple</Fruit>
      </item>
```

```
        <item>
         <Vegetable Name="Okra">Fresh Okra</Vegetable>
        </item>
        <item>
         <Beer Name="Dark Island">Fine Beer</Beer>
        </item>
        <item>
         <Beer Name="Kingfisher">Lager Beer</Beer>
        </item>
       </items>
      </x:parse>

      <!--iterate through all the elements -->
      <x:forEach select="$xml/items/item">
        <!--print the current element -->
        ->  <x:out select="." />

        <!--check if the selected element is Beer -->
        <x:if select="./Beer" >
          <!--yes it is beer -->
          * is a Beer
        </x:if>
        <br />
      </x:forEach>

    </body>
  </html>
```

2. Visit `http://localhost:8080/XML/iterate.jsp` in your browser (see Figure 8-6).

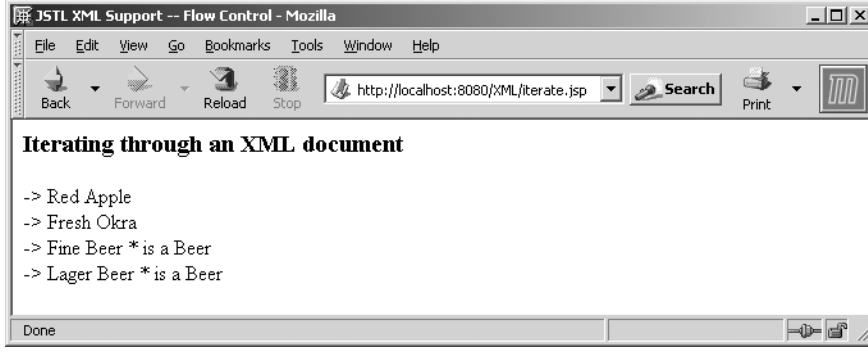

Figure 8-6. Conditional processing with XML

How It Works

Here you parse an embedded XML element called `<items>`. You then iterate over all the child elements of `<items>` and print `* is a Beer` next to any `<Beer>` elements. You use a `<x:forEach>` tag to iterate through the document and `<x:if>` to find out if the element name is `Beer`:

```
<x:if select="./Beer" >
```

Understanding XSL Transformations

XSL is an XML dialect that describes rules dictating how a source document can be transformed into a target document. This means that one XML document can be converted into another XML document, an HTML document, or simply plain text. Although these inputs and outputs can be documents, they may also be strings that contain XML content.

The simplest XSL document contains an empty `<stylesheet>` element:

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

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

This isn't much use to you because no transformation takes place. Try an example.

Trying It Out: Transforming XML with XSL

You'll look at the XSL stylesheet (not to be confused with CSS stylesheets), and then we'll explain it once you've finished the example:

1. Create **book.xsl**, and store it in the `XML` folder of `webapps`. This is like your parsing page and contains all of the HTML you'll need. It's simply a template and is similar in concept to the template technique you used in Chapter 6:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
                  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" indent="yes" />
  <xsl:template match="/">
    <html>
      <head>
        <title>JSTL XML Support -- Transforming</title>
      </head>
```

```
        <body bgcolor="#FFFFFF">
          <h3>Transforming an XML Document using XML tags</h3>
          <xsl:apply-templates />
        </body>
      </html>
    </xsl:template>

    <xsl:template match="Author-Name">
      <!--printing the  element-->
      LAST NAME :
      <xsl:apply-templates select="Last"/>
    </xsl:template>

    <xsl:template match="Last">
      <xsl:value-of select="text()" />
      <br />
    </xsl:template>

    <xsl:template match="Book-Name" >
      <!--printing the BOOK-NAME element -->
      BOOK NAME :
      <xsl:value-of select="text()" />

      <hr />
    </xsl:template>
  </xsl:stylesheet>
```

2. You'll be using `book.xml` in this example. When you transform the imported file, the DTD will be checked. You may have to change the following line of code to pass the validation checks:

```
<!DOCTYPE Book SYSTEM "http://localhost:8080/XML/book.dtd">
```

3. Now you need the JSP page that will transform your XML (`transformXML.jsp`):

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

<c:import url="/book.xml" var="XMLDocument"/>
<c:import url="/book.xsl" var="XSLDocument" />

<x:transform xml="${XMLDocument}" xslt="${XSLDocument}"/>
```

4. Now navigate to `http://localhost:8080/XML/transformXML.jsp`, and you should see the same page as Figure 8-7.

Figure 8-7. Transforming XML with XSLT

How It Works

You'll look at the XSL stylesheet first.

First you declare the root element, which must always be `<stylesheet>` and is in the `xsl` namespace:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

`<xsl:output>` specifies the format of the end result, in this case HTML. You want the HTML to be indented so you set `indent` to `"yes"`:

```
<xsl:output method="html" indent="yes" />
```

The first `<xsl:template>` tag is used to match the root element using `"/"`; in this case, it will match `<Book>`. Therefore, this template is always called when processing begins. It's a good idea to place all your main structural elements, such as `<html>` and `<body>`, in this default template:

```
<xsl:template match="/">
  <html>
    <head>
      <title>JSTL XML Support -- Transforming</title>
    </head>
    <body bgcolor="#FFFFFF">
      <h3>Transforming an XML Document using XML tags</h3>
```

You want to place your transformed XML at this point in the HTML document, so you call `<xsl:apply-templates>` to use all the other templates defined in this document:

```
        <xsl:apply-templates />
      </body>
    </html>
  </xsl:template>
```

The first specific template matches any `<Author-Name>` elements in the document regardless of their location. If you wanted to match only `<First>` elements that were subelements of `<Author-Name>`, which in turn was a subelement of the root element, you'd use `"/Author-Name/First"`. Note the `"/"` to specify the root element:

```
  <xsl:template match="Author-Name">
    <!--printing the Last element-->
    LAST NAME :
```

If there's a match, you want to select the `<Last>` element, so you call the template that will match it:

```
    <xsl:apply-templates select="Last"/>
  </xsl:template>
```

The next specific template matches the `<Last>` element:

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

The `<xsl:value-of>` tag can obtain the value of many things, such as the text of an element or the value of its attributes. In this case you use XSL's `text()` method to obtain the text of the element:

```
    <xsl:value-of select="text()" />
    <br />
  </xsl:template>
```

The final template matches the `<Book-Name>` element and displays its contents:

```
  <xsl:template match="Book-Name" >
    <!--printing the BOOK-NAME element -->
    BOOK NAME :
    <xsl:value-of select="text()" />
    <hr />
  </xsl:template>
</xsl:stylesheet>
```

Once the last template has been called, the processing continues to the end of the default template.

In the JSP page, you import the XML and core tag libraries:

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

You're using external XML and XSL files, so you need to use the `<c:import>` tag to get their contents and place the imported strings into variables for use later:

```
<c:import url="/book.xml" var="XMLDocument"/>
<c:import url="/book.xsl" var="XSLDocument" />
```

The transformation is one line of JSP where you take the two strings and apply a transformation. The results are printed to the screen; however, they can also be stored in a variable as is the case with most JSTL tag results:

```
<x:transform xml="${XMLDocument}" xslt="${XSLDocument}"/>
```

This was a very simple example, and it showed the principles of XSL. However, there's so much more to XSL that we just can't cover it properly in this book. Try these other resources:

- `http://www.w3.org/Style/XSL/`
- `http://www.w3schools.com/xsl/`

Why Not Try?

- Write a simple JSP Web application that stores data in XML format using XML tags.

Summary

In this chapter you learned the following:

- XML: its relevance, advantages, and uses
- The difference between validity and well-formedness in the context of XML
- XML parsers
- XSL and XML transformation
- JSP tags to deal with and output XML

You've taken a small tour into the exciting world of XML. This chapter serves only as a very quick introduction, and there's a huge ocean of knowledge out there to be explored.