An Overview of Extensible Markup Language

By Carol E. Wolf, Professor Emeritus, Pace University, NY

An Overview of Extensible Markup Language

Some History

Standard Generalized Markup Language (SGML) was developed in the 1960s and 1970s. Its purpose was to provide a standard way to annotate (markup) documents that was system independent. It became an ISO (International Standards Organization) standard in 1986. It was widely used for text processing up until 2000, but now most applications use XML.

Extensible Markup Language (XML) grew out of SGML. It is somewhat easier to use and so has to a large extent replaced SGML. It was published as a Recommendation by the W3C[1] in 1998. This recommendation (essentially equivalent to a standard) has been undated with a number of additions and modifications. XML itself has a number of subsets, including RSS[2], used to exchange new bulletins, and MathML, a markup language for mathematics.

Hypertext Markup Language (HTML) was developed by Tim Berners-Lee in 1992[3] along with his invention of Hypertext Transfer Protocol (HTTP). Together HTML and HTTP created the World Wide Web. Berners-Lee adapted SGML *tags* for HTML, carrying over some basic ones. His most important contribution was the addition of the *anchor* tag <a> together with its hypertext reference, href..

It is noteworthy that HTML predated XML. The first version of HTML was quite simple enabling the development of the first browser, Mosaic.[4] Mosaic used the tags to guide the way it displayed a document. It and subsequent browsers were quite forgiving of markup errors. If the tags did not determine how to display something, they either omitted the text or displayed the HTML itself.

Recently a need was seen for a version of HTML that also obeyed all the requirements of XML. The result was XHTML (Extensible HTML). The W3C came out with a Recommendation for XHTML[5] in 2000, which was revised in 2002. XHTML is a subset of XML. It follows all the rules of XML and adds a few additional restrictions.

XML and HTML actually have differing purposes. XML is used to annotate and so describe the contents of documents. HTML on the other hand is used to specify the way documents are to be displayed on a web page. Tags used in HTML are pre-defined so that browsers all know how to interpret them. A number of XML subsets have pre-defined tags, but this is not necessary. XML users may create their own tags as they go along.

[1] The W3C Recommendation for XML Version 1.0, Third Edition is at http://www.w3.org/TR/REC-xml/.
[2] RSS originally was an acronym for Really Simple Syndication. Other acronyms are Rich Site Summary and RDF Site Summary.
[3] Dave Raggett , A History of HTML, Chapter 2, Addison Wesley Longman, 1998, http://www.w3.org/People/Raggett/book4/ch02.html.
[4] Mosaic was developed by Mark Andreesen and others at the National Center for Supercomputer Applications at the University of Illinois.
[5] The W3C Recommendation for XHTML may be found at http://www.w3.org/TR/xhtml1/.

Unicode

XML uses Unicode to code for character data[6]. There are a number of different versions of Unicode, but all have ASCII as the first 128 characters. After that the versions may differ. The most common version used in the West, and the XML default, is UTF-8. It is a variable length code that encodes some characters in a byte, some in two bytes and even some in four bytes.

Since many applications just use ASCII, this is the most efficient way to handle data, and it wastes the least space. The remaining 128 characters from code 128 to code 255 are used for some of the more common non-ascii characters used in western nations. The two-byte codes are used for some other language systems including some Asian ideographs. And finally the four-byte codes are used for more complicated ideographs.

There are a number of other flavors of Unicode. If you expect to be coding languages other than the common western ones, you should investigate all the possibilities. We will use UTF-8 for our documents.

XML Tags and Rules

All the markup languages use *tags*, names enclosed by angle brackets. In HTML there are tags such as <p> … </p> that begin and end paragraphs, <b> … </b> that delimit boldface text, and <i> … </i> that indicate text in italics. Tags in XML tend to be spelled out, such as <name>Alice</name> or <telephone>123-45-6789</telephone>. The naming requirements are similar to those in many computer languages. Tags are case sensitive, and along with letters, digits, and underscores, names may include hyphens, periods, and a single colon.

Some of the rules for XML tags are:

- Opening tags all have matching closing tags. Empty tags such as <br> and <input> that have no closing tags are to be written as <br /> and <input />.
- Attribute values (such as text or size) must be in quotes.
- Tags must be nested. That means that <b><i>…</i></b> is correct but <b><i>…</b></i> is not.
- Comments contain double hyphens (<!-- … -->), and no double hyphens are allowed inside comments otherwise.
- Values must be added to boolean attributes, e.g. multiple = “multiple”.
- The entities &lt; and &amp; must be used in place of <, less than, and &, ampersand.

XML documents, including XHTML ones, must be *well-formed*. That means that they adhere to all the rules listed above. If they do not, they cannot be properly interpreted. Most browsers are very forgiving and will display web pages that do not comply with all the requirements. However this is not the case with XML parsers, programs used to extract information from XML documents. They reject XML documents that are not well-formed.

An XML document may also be *valid*. A valid document is checked against either a Document Type Definition (DTD) or a Schema. These will both be described later on.

[6] Elliotte Rusty Harold and Scott Means, *XML In a Nutshell*, Third Edition, Chapter 5, O’Reilly & Associates, Inc., 2004

XML Example for an Address

The following is a very simple XML document.

```
<?xml version = "1.0" ?>
<address>
    <name>Alice Lee</name>
    <email>alee@aol.com</email>
    <phone>123-45-6789</phone>
    <birthday>1983-07-15</birthday>
</address>
```

The first line is a *processing instruction*. It begins with '<?' and indicates the version of xml that is used in the document. The rest of the document is essentially self-explanatory. It is clear that it refers to a person whose name is Alice Lee, email address is alee@aol.com, etc. While the first two elements would be clear without the markup, the tags clarify the meaning of the last two elements.

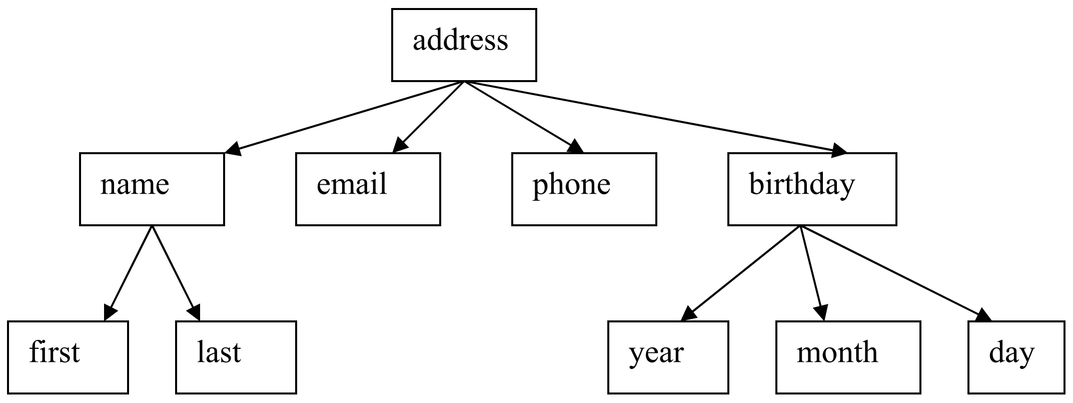
Tree Structure

An XML document exhibits a tree structure. It has a single root node, <address> in the example above. The tree is a general ordered tree. There is a first child, a next sibling, etc. Nodes have parents and children. There are leaf nodes at the bottom of the tree. The declaration at the top is not part of the tree, but the rest of the document is.

We could expand the document above so that name and birthday have child nodes.

```
<?xml version = "1.0" ?>
<address>
    <name>
        <first>Alice</first>
        <last>Lee</last>
     </name>
    <email>alee@aol.com</email>
    <phone>123-45-6789</phone>
    <birthday>
        <year>1983</year>
        <month>07</month>
        <day>15</day>
    </birthday>
</address>
```

Now <name> has two children and <birthday> has three. Most processing on the tree is done with a preorder traversal. One way to view the tree is shown on the next page.



Attributes

As in html, tags can have attributes. These are name-value pairs such as width = "300". We have seen these in applet and image tags. They can be used in XML and are required in some places.

An example from the preceding might be

```
<name first = "Alice" last = "Lee" />
```

While this is legal, it is not very useful for data. It makes it more difficult to see the structure of the document.

However, there are places where attributes are necessary. One that we will be using shortly is for the xml processing instruction.

```
<?xml version="1.0" encoding="UTF-8" standalone ="no"?>
```

The attribute, encoding, refers to the version of Unicode used in the document. The standalone attribute indicates whether a DTD (document type definition) is part of the document. The default is "no", meaning that the document does not have an inline DTD. XML documents do not require either processing instructions or DTDs. But it is a good idea to include a processing instruction at the top of any XML file.

There are a number of attribute types. See one of the standard references on XML for a list.[7]

Entities and CDATA

As in html, certain characters are not allowed in XML documents. The most obvious ones are less than signs and quotation marks. Also, ampersands are used to start the escape string, so they too have a substitution. These are escaped with the following substitutions with the greater than sign thrown in for symmetry.

[7] Elliotte Rusty Harold and Scott Means, *XML In a Nutshell*, Third Edition, Chapter 3, O'Reilly & Associates, Inc., 2004

<	&lt;
>	&gt;
&	&amp;
"	&quot;
'	&apos;

Users can create their own entities in a number of different types. These will not be described here. But they can be found at most references for XML.[8]

CDATA stands for character data. XML can have sections that contain characters of any kind that are not *parsed*. This means that they will be ignored by the XML parser, which is used to put the document into a tree. These sections are similar to the *pre* sections in html that a browser displays unchanged.

CDATA sections begin with <![CDATA[and end with]]>. An example might be an equation like the following:

```
<![CDATA[
    x + 2*y = 3
]]>
```

XHTML

Extensible Hypertext Markup Language is a subset of XML, unlike HTML. An XHTML document must be well-formed, i.e. follow all XML rules. XHTML has a few additional requirements.

- Tags must be in lower case.
- The image tag, <img … > must include an *alt* attribute.
- Documents must begin with a DOCTYPE declaration.

DOCTYPE declarations refer to the W3C Recommendations for HTML. There are three levels, *Transitional*, *Strict*, and *Frameset*. Strict declarations are used for documents that separate out all layout information into a Cascading Style Sheet (CSS).[9] Transitional declarations are used for documents that include some layout information. An example would be

```
<body bgcolor="blue">
```

Some older browsers do not support CSS. Frameset declarations are used for documents that use HTML frames.

To understand an XHTML document, it is necessary to learn what the various tags mean. There are many guides available for this, including the requirements defined by the W3C committee.[10] The following is an example of an XHTML document containing a form. Once the form is filled out, it can be submitted to a server for processing.

[8] Elliotte Rusty Harold and Scott Means, *XML In a Nutshell*, Third Edition, Chapter 3, O'Reilly & Associates, Inc., 2004

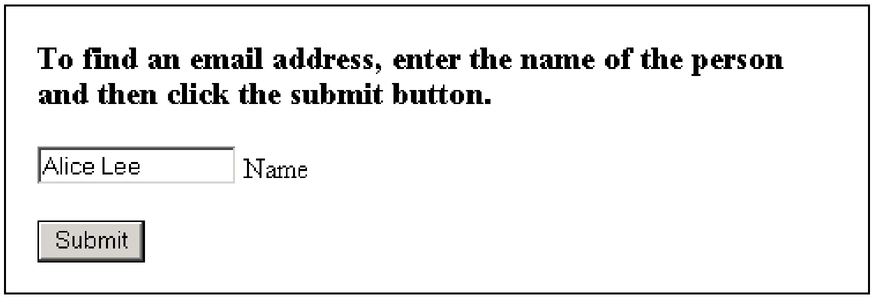
[9] Information on Cascading Style Sheets is available in many places including the W3Schools web site, http://www.w3schools.com.

[10] The W3C Recommendation for XHTML is at http://www.w3.org/TR/xhtml1/.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
	<head><title>E-Mail Form</title></head>

	<body>
		<h3>To find an email address, enter the name of the person
		<br /> and then click the submit button.</h3>
		<form method = "get" action="http://url-address/processor-name ">
			<input type = "text" name = "keyName" size = "15" /> Name
			<p><input type="submit" value="Submit"></p>
		</form>
	</body>
</html>
```

When displayed, the web page appears as shown below. Here the form has been filled out but not yet submitted to the server for processing.


To find an email address, enter the name of the person and then click the submit button.

Alice Lee Name

Submit

Document Type Definitions

A Document Type Definition (DTD) for an xml file is a list of elements (tags) used in the file, together with some information about how they are defined. The document must have a single root node. This is followed by the children of the root and either their children or data type. In a DTD there are only two data types, PCDATA (parsed character data) or CDATA, unparsed character data. Most of the examples use parsed character data.

A DTD also indicates how many times an element can occur in the file. The default is once. But most files use the same tag names a number of times. The notation used is similar to that used in regular expressions.

- * means zero or more occurrences.
- + means one or more occurrences.
- ? means zero or one occurrence.

A DTD also allows for choice. A vertical bar (|) is used to indicate one element or another.

A DTD for the Address Example

A DTD for the address example on page 4 might be:

```
<!ELEMENT address (name, email, phone, birthday)>
<!ELEMENT name (first, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT birthday (year, month, day)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT month (#PCDATA)>
<!ELEMENT day (#PCDATA)>
```

This says that address is the root of the document. It has four children: name, email, phone, and birthday. The element, name, also has two children, first and last. And the element, birthday, has three children, year, month, and day. All the rest of the elements consist of PCDATA, parsed character data.

If the above definitions are contained in a file called *address.dtd*, the following declaration should be added to the top of the xml file.

```
	<!DOCTYPE  address SYSTEM "address.dtd">
```

This assumes that the file, address.dtd, is in the same folder as the xml file. This is the best way to handle finished DTDs.

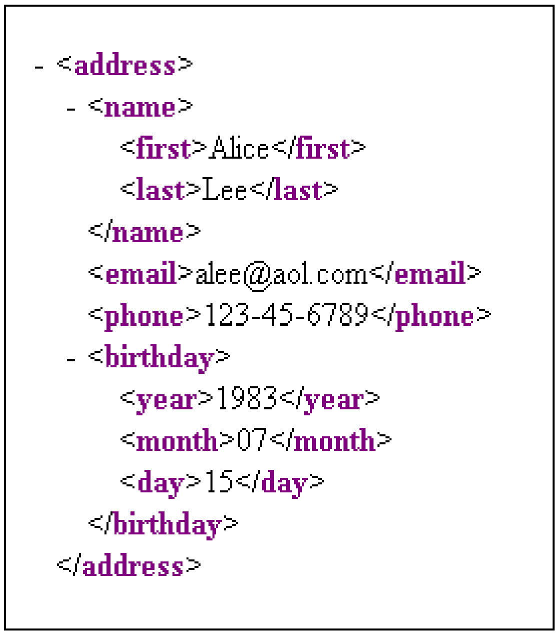
However when developing a DTD, it is more convenient to have it in-line. In that case, the entire DTD is placed at the top of the xml file enclosed by <!DOCTYPE address […]>. The entire in-line example for the preceding xml file follows.

```
<?xml version="1.0" encoding="UTF-8" standalone ="no"?>
<!DOCTYPE address [
	<!ELEMENT address (name, email, phone, birthday)>
	<!ELEMENT name (first, last)>
	<!ELEMENT first (#PCDATA)>
	<!ELEMENT last (#PCDATA)>
	<!ELEMENT email (#PCDATA)>
	<!ELEMENT phone (#PCDATA)>
	<!ELEMENT birthday (year, month, day)>
	<!ELEMENT year (#PCDATA)>
	<!ELEMENT month (#PCDATA)>
	<!ELEMENT day (#PCDATA)>
]>
<address>
	<name>
		<first>Alice</first>
		<last>Lee</last>
	</name>
	<email>alee@aol.com</email>
	<phone>123-45-6789</phone>
	<birthday>
		<year>1983</year>
		<month>07</month>
		<day>15</day>
	</birthday>
</address>
```

This is a *valid* document. That means that the XML file is an *instance* of the DTD and adheres to all its requirements. Documents can be validated using an XML parser. Parsers are programs that read the document and verify its tree structure. In addition, the parser can determine whether or not the document is valid.

The above document was validated by a parser made available by the Refsnes Data Company of Norway, a web consulting firm. They have a web site http://www.w3schools.com/ that features a number of excellent tutorials on web development. A parser called Xerces is also available from the Apache Software Foundation at http://www.apache.org/. It will be discussed later.

Most web browsers will also parse XML files. If no layout information is provided, they display the tree structure of the document. The hyphens can be used to collapse the tree. When collapsed, the hyphens are replaced by plus signs. Clicking on these opens up the tree again. The following shows the address example as displayed by the Firefox browser from Mozilla.[11]



```
- <address>
  - <name>
      <first>Alice</first>
      <last>Lee</last>
    </name>
    <email>alee@aol.com</email>
    <phone>123-45-6789</phone>
  - <birthday>
      <year>1983</year>
      <month>07</month>
      <day>15</day>
    </birthday>
  </address>
```

A Grocery Store Example

Another example could be used to describe some products at a grocery store. It contains fields for a product's name, id, quantity, and price. These must be included, but the number of entries for each type of product may vary. A DTD for this example follows:

```
<!--  A document type definition for grocery.xml.  -->
<!ELEMENT grocery (heading+, fruit*, vegetables*, bakery*)>


<!--  The elements that have children.  -->
<!ELEMENT heading (name, id, quantity, price)>
```

[11] http://www.mozilla.org/

```
<!ELEMENT fruit (name, id, quantity, price)>
<!ELEMENT vegetables (name, id, quantity, price)>
<!ELEMENT bakery (name, id, quantity, price)>

<!--  Definition of the data types.  -->
<!ELEMENT name (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

From this DTD you can see that the root element is <grocery>. This element has four different kinds of children. There may be zero or one heading. The DTD also indicates that there may be zero or more fruit, vegetables, and bakery elements. But it also mandates that all fruit elements come first, vegetable elements next, and bakery elements last.

A file that satisfies all these requirements follows:

```
<?xml version="1.0" encoding="UTF-8" standalone ="no"?>
<!DOCTYPE  grocery SYSTEM "grocery.dtd">
<!--
    An xml file that shows names, ids, quantities, and prices of fruit, vegetables, and bakery items.
-->
<grocery>
    <heading>
        <name>Name</name>
        <id>ID</id>
        <quantity>Quantity</quantity>
        <price>Price</price>
    </heading>
    <fruit>
        <name>apples</name>
        <id>A123</id>
        <quantity>25</quantity>
        <price>1.25</price>
    </fruit>
    <fruit>
        <name>pears</name>
        <id>P234</id>
        <quantity>50</quantity>
        <price>2.55</price>
    </fruit>
    <vegetables>
        <name>beans</name>
        <id>B345</id>
        <quantity>10</quantity>
        <price>.85</price>
    </vegetables>
    <vegetables>
        <name>corn</name>
        <id>C456</id>
        <quantity>60</quantity>
```

```
		<price>.50</price>
	</vegetables>
	<bakery>
		<name>bread</name>
		<id>B567</id>
		<quantity>15</quantity>
		<price>2.30</price>
	</bakery>
	<bakery>
		<name>cake</name>
		<id>C678</id>
		<quantity>4</quantity>
		<price>4.25</price>
	</bakery>
</grocery>
```

A Cascading Style Sheet for the Grocery Example

A Cascading Style Sheet (CSS) can also be used to display the xml file in another way. The following link must be added to the beginning of the xml file.

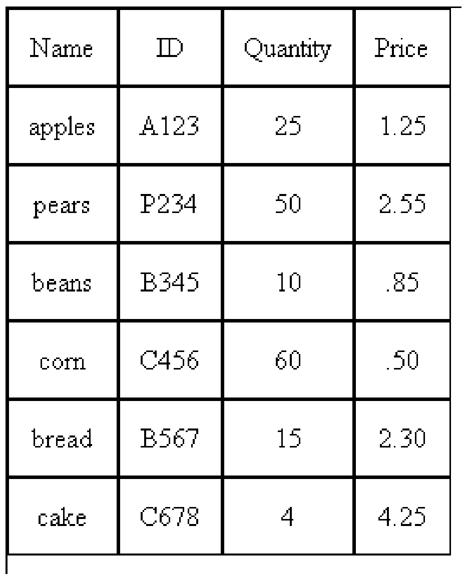
```
	<?xml-stylesheet type="text/css" href="grocery.css"?>
```

Some browsers will use this information to display the file while others will ignore it. Both Firefox and Netscape version 7.2 use the style sheet for display, while Internet Explorer version 6.0 does not.

The following style sheet will display the document in a table.

```
/* Style sheet for address application. */
grocery
{
	font-family: "Times New Roman", serif
	display: table;
	border-style: solid;
	border-width: thin;
	margin-left: 1.0cm;
	margin-top: 1.0cm;
}
heading, fruit, vegetables, bakery
{
	display: table-row;
}
name, id, quantity, price
{
	display: table-cell;
	border-style: solid;
	border-width: thin;
	padding: 0.3cm;
	text-align: center;
}
```

If this style sheet is applied, the display looks like the following in Firefox.



Name	ID	Quantity	Price
apples	A123	25	1.25
pears	P234	50	2.55
beans	B345	10	.85
corn	C456	60	.50
bread	B567	15	2.30
cake	C678	4	4.25

This style sheet says that the root element, grocery, should be displayed as a table.
    display: table;
The other styles determine the font and table properties such as a solid, thin border and 1 cm margins.

The columns of the table are given by the next elements: heading, fruit, vegetables, and bakery. The style for these is display: table-row. This will display these elements as rows.

Finally the data elements: name, id, quantity, and price, will be displayed in the table cells.
    display: table-cell;
The cell styles must also have instructions as to how the borders should appear.

There are many other applicable styles. W3Schools has an extensive list in their tutorial on CSS.

Attributes and DTDs

XML tags may include attributes. These are name-value pairs such as standalone="no". They can be used in XML and are required in some places. An example might be the following XML file that contains information about students in a course. Each exam grade has a *weight* attribute to indicate how it should factor into the course grade.

```
<?xml version="1.0"?>
<!DOCTYPE roster SYSTEM "roster.dtd">

<roster>
    <student>
        <name>
            <first>Alice</first>
            <last>Lee</last>
        </name>
        <midterm weightMidterm = "40">85</midterm>
        <final weightFinal = "60">92</final>
    </student>
    <student>
        <name>
            <first>Barbara</first>
            <last>Smith</last>
        </name>
        <midterm weightMidterm = "40">78</midterm>
        <final weightFinal = "60">84</final>
    </student>
    <student>
        <name>
            <first>Cathy</first>
            <last>Jones</last>
        </name>
        <midterm weightMidterm = "40">82</midterm>
        <final weightFinal = "60">87</final>
    </student>
</roster>
```

Attributes must also be listed in the DTD for the document. They are defined in an attribute list given by an ATTLIST definition. A DTD for this example follows.

```
<!--  A document type definition for roster.xml.  -->
<!ELEMENT roster (student+)>

<!--  Each student must have midterm and final grades..  -->
<!ELEMENT student (name, midterm, final)>
<!ELEMENT name (first, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT midterm (#PCDATA)>
<!ELEMENT final (#PCDATA)>

<!--  The midterm and final attributes consist of CDATA with a default value of "0".  -->
<!ATTLIST midterm weightMidterm CDATA "0">
<!ATTLIST final weightFinal CDATA "0">
```

While there are only two datatypes for elements, PCDATA and CDATA, there are quite a few for attribute lists. To learn about the others types, see the references.[12]

XML Schema

Both Schema and Document Type Definitions (DTDs) are used to make sure that those that use the documents agree on their contents and form. DTDs were developed first. They define the tree structure of a document, but they only provide two data types, CDATA and PCDATA. This was fine when XML

[12] Elliotte Rusty Harold and Scott Means, *XML In a Nutshell*, Third Edition, Chapter 3, O'Reilly & Associates, Inc., 2004.

was primarily used for marking up documents, such as books and articles. Most of that content consists of character data.

However, now XML is widely used to interchange data from files and databases. These documents can have a number of data types other than strings, including integers, decimals, dates and booleans. Also, a schema is itself an XML document. The recommendations for schemas[13] only date from May 2001, but they are now probably more widely used than DTDs. Some people are suggesting that DTDs be retired in favor of schemas.

Schemas, like DTDs, are used to validate a document. A parser that can be used for validation with either a schema or DTD is available from the Apache Software Foundation.[14] It is open source and is called Xerces. A sample program called Writer.java comes with it.[15] It was written by Andy Clark at IBM and supplied by IBM to Apache.

If an XML document is valid, Writer will simply echo it on the console screen. However, if there is an error, it will first point it out and then echo the document. As with all such software, some of the error messages are easier to understand than others. When you are developing a schema for an XML document, it is wise to check it regularly for validity. Schemas are complicated, so errors are common.

Namespaces

Before discussing schemas, it is necessary to explain what a namespace is in XML. A namespace is used to make a distinction between items with the same name but different meanings. The most common example is that of a 'table', which could refer to either an html table or a piece of furniture.

In order to keep the meanings straight, a *prefix* is added to the beginning of the tag. In another example, we could have h:form for an html form and m:form for a medical form. The entire name is said to be the *qualified* name. It consists of the prefix and the *local* part. Since XML tag names may contain only a single colon, the local part must be colon free.

Namespaces are described by a Uniform Resource Identifier (URI). The identifier doesn't actually have to point to a real web page, but it is preferable that it do so. The page only needs to have some explanation about the uses for the prefix. The main one that we will use is

```
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

This is the namespace for XML schema. The prefix is "xmlns", which stands for xml namespace.

For the form example above, we could have namespaces

```
    xmlns:m="http://csis.pace.edu/~wolf/medical/"
```

and

```
    xmlns:h="http://www.w3.org/TR/html4/"
```

[13] http://www.w3.org/XML/Schema
[14] http://www.Apache.org
[15] A version of Writer.java is on my web site. In order to compile it in JCreator, you will need some .jar files. These are also on the website. Get all five of them, resolver.jar, xercesImpl.jar, xercesSamples.jar, xml-apis.jar, and xmlParserAPTs.jar. You can store them with the other .jar files in your jdk folder. Configure JCreator to find them by adding them to the profile for the Java editor. When you execute the file it will first ask you for an option and the name of the xml file. To validate using a DTD, the option is –v, and to validate using a schema, the option is –s.

The latter is a real website, the W3C HTML 4.01 Specification. However there is no medical folder on my website. If you try to link to it, you will get a **Not Found** page.

If you put a namespace attribute in a tag, all its children will inherit it. This way you do not have to add the prefix to every tag. This provides a *default* namespace for the tag and its children.

```
<form xmlns="http://csis.pace.edu/~wolf/medical/">
        <doctor>Dr. Stein</doctor>
        <patient>Alice Lee</patient>
</form>
```

Simple Address Example

The first address example we used had one root with four children. It is repeated below.

```
<?xml version = "1.0" ?>
<address>
        <name>Alice Lee</name>
        <email>alee@aol.com</email>
        <phone>123-45-6789</phone>
        <birthday>1983-07-15</birthday>
</address>
```

It might represent a row in a database. We previously saw a DTD that described it. The following schema does also. Note the first two lines of the schema. They are standard and must be copied exactly as is into the document.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="address">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="name" type="xs:string"/>
                        <xs:element name="email" type="xs:string"/>
                        <xs:element name="phone" type="xs:string"/>
                        <xs:element name="birthday" type="xs:date"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
</xs:schema>
```

This looks more complicated than the DTD. But it also contains more information. It says that address is an element, that its type is complex, and that the elements called name, email, phone, and birthday must occur in the order shown. If <xs:sequence> had been left out, the four elements could appear in any order, but they would all have to be there. Also while three of the elements are strings, the fourth is a date. Date fields in XML are of the form yyyy-mm-dd. If they are not in this form, they are not valid.

Also since a schema is an XML document itself, it can mirror the form of the document it describes. The one above shows that address is the root node and that name, email, phone, and birthday are its children. This schema also says that each element must occur once and only once. The default is exactly once. This can be changed by adding a constraint to an element.

```
	<xs:element name="phone" type="xs:string" maxOccurs="unbounded"/>
```

This says that there may be one or more phone numbers listed. There must be at least one, however. To change that, we would have to add another constraint, minOccurs="0".

An XML document is known as an *instance* of the schema. To use the schema, the document must contain a link to it. This is put into the root tag.

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

Notice that it not only identifies the W3C site for schema, but it also indicates that this is an instance of that schema. Since namespaces are not used inside this document, it says that the location of the schema is in xsi:noNamespaceSchemaLocation. If a namespace had been used, this would change to xsi:schemaLocation.

Schemas are not unique. Another one for address.xml uses a reference in one element to another element. Thus the *address* element has references to the *name, email, phone*, and *birthday* elements. This can be used to divide the schema into manageable parts. Note that comments follow the usual rules for html and xml.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- Definition of simple elements. -->
<xs:element name="name" type="xs:string"/>
<xs:element name="email" type="xs:string"/>
<xs:element name="phone" type="xs:string"/>
<xs:element name="birthday" type="xs:date"/>

<!-- Definition of complex elements. -->
<xs:element name="address">
	<xs:complexType>
		<xs:sequence>
			<xs:element ref="name"/>
			<xs:element ref="email"/>
			<xs:element ref="phone" maxOccurs="unbounded"/>
			<xs:element ref="birthday"/>
		</xs:sequence>
	</xs:complexType>
</xs:element>
</xs:schema>
```

Dividing the elements up this way will make it easier to handle more complicated xml documents.

Attributes in Schema

The XML document that described a class roster contained attributes for exam weights. As in DTDs, attributes are treated separately by schemas. An element with an attribute has a complex type and is not listed the same way as a simple element.

The midterm and final both have attributes and so are considered complex types. First, they are *extensions* of a simple base type, xs:positiveInteger. The attributes themselves are simple types, so they contribute simple content to the element. This part of the schema looks as follows:

```
    <xs:element name="midterm">
        <xs:complexType>
            <xs:simpleContent>
                <xs:extension base="xs:positiveInteger">
                    <xs:attribute ref="weightMidterm"/>
                </xs:extension>
            </xs:simpleContent>
        </xs:complexType>
    </xs:element>
```

The following is a schema for roster.xml. It has

1. one simple element: name,
2. two attributes: weightMidterm and weightFinal,
3. a root: roster,
4. a child of the root: student,
5. three children of student: name, midterm, and final.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- definition of simple elements -->
<xs:element name="name" type="xs:string"/>

<!-- definition of attributes -->
<xs:attribute name="weightMidterm" type="xs:positiveInteger"/>
<xs:attribute name="weightFinal" type="xs:positiveInteger"/>

<!-- definition of complex elements -->

<xs:element name="midterm">
    <xs:complexType>
        <xs:simpleContent>
            <xs:extension base="xs:positiveInteger">
                <xs:attribute ref="weightMidterm"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
</xs:element>

<xs:element name="final">
    <xs:complexType>
```

```
			<xs:simpleContent>
				<xs:extension base="xs:positiveInteger">
					<xs:attribute ref="weightFinal"/>
				</xs:extension>
			</xs:simpleContent>
		</xs:complexType>
</xs:element>

<xs:element name="student">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="name"/>
				<xs:element ref="midterm"/>
				<xs:element ref="final"/>
			</xs:sequence>
		</xs:complexType>
</xs:element>

<xs:element name="roster">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="student" maxOccurs="unbounded"/>
			</xs:sequence>
		</xs:complexType>
</xs:element>
</xs:schema>
```

There is a lot more to know about schema besides the above. For more information, see the References.[16]

Extensible Stylesheet Language Transformations

Extensible Stylesheet Language Transformations (XSLT) is a language used to *transform* documents. The source document is often an XML file and the destination document is a web page. However, other transformations are possible. In fact, the language is a full (though awkward) programming language. It allows for a number of computations.

An XML document forms a tree. The nodes can be the root node, elements, attributes, text, comments, processing instructions (tags beginning with <?), or namespaces. A language called XPath is used to find particular nodes in the tree. In conjunction with XSLT, it can be used to include selected data in the output.

Transformation for the Address Example

We can attach an XSL stylesheet link to the address example used previously. Note that the type in the link is now ="application/xml", and the reference is to an XSL document.

[16] Elliotte Rusty Harold and Scott Means, *XML In a Nutshell*, Third Edition, Chapter 17, O'Reilly & Associates, Inc., 2004.

```
<?xml version = "1.0" ?>
<?xml-stylesheet type="application/xml" href="address.xsl"?>

<address>
	<name>Alice Lee</name>
	<email>alee@aol.com</email>
	<phone>123-45-6789</phone>
	<birthday>1983-7-15</birthday>
</address>
```

The simplest stylesheet for it has no content.

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

	</xsl:stylesheet>
```

When this empty stylesheet is applied to the XML file, the result just echoes all the data.

```
	<?xml version="1.0" encoding="UTF-8"?>

		Alice Lee
		alee@aol.com
		123-45-6789
		1983-7-15
```

If you just want one piece of data, say the name, you can use the following stylesheet:

```
	<?xml version="1.0" encoding="ISO-8859-1"?>
	<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
		<xsl:template match="address">
			<xsl:value-of select="name"/>
		</xsl:template>
	</xsl:stylesheet>
```

The result of this is just the heading followed by the name.

```
	<?xml version="1.0" encoding="UTF-8"?>
	Alice Lee
```

The stylesheet creates a *template* that is used to *match* data in the file. This one matches the address node and *selects* the *value-of* the name element. We can select other elements the same way.

A more useful stylesheet transforms the file into a web page. Any markup is allowed in the stylesheet, so long as it is well-formed. Since xhtml is, it can be added to the file. Here simple html tags are included in order to produce a complete html file.

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

		<xsl:template match="address">
			<html><head><title>Address Book</title></head>
```

```
                <body>
                    <xsl:value-of select="name"/>
                    <br/><xsl:value-of select="email"/>
                    <br/><xsl:value-of select="phone"/>
                    <br/><xsl:value-of select="birthday"/>
                </body>
                </html>
        </xsl:template>
    </xsl:stylesheet>
```

The result of this is the html file:

```
    <html>
    <head>
    <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Address Book</title>
    </head>
    <body>Alice Lee<br>alee@aol.com<br>123-45-6789<br>1983-7-15</body>
    </html>
```

Note that *white space* is not preserved in the html file. (White space consists of spaces, tabs, and end of line characters.) However, when displayed in a browser, the <br/> tag will place each value on a separate line. The data can also be put into a table by adding table tags, a list by using <ul> and <li>, etc. Empty tags such as <br> must be closed with a '/' in the stylesheet, but the transformer leaves out the extra slash in the resulting web page.

An Address Book Example

In a real address book you would have a number of names. The next file has three entries.

```
    <?xml version = "1.0" ?>
    <?xml-stylesheet type="application/xml" href="address.xsl"?>

    <address-book>
        <address>
            <name>Alice Lee</name>
            <email>alee@aol.com</email>
            <phone>123-45-6789</phone>
            <birthday>1983-07-15</birthday>
        </address>
        <address>
            <name>Barbara Smith</name>
            <email>bsmith@yahoo.com </email>
            <phone>234-56-7890 </phone>
            <birthday>1982-11-25</birthday>
        </address>
        <address>
            <name>Cathy Jones</name>
            <email>cjones@hotmail.com </email>
            <phone>345-67-8901 </phone>
```

```
		<birthday>1984-02-05</birthday>
	</address>
</address-book>
```

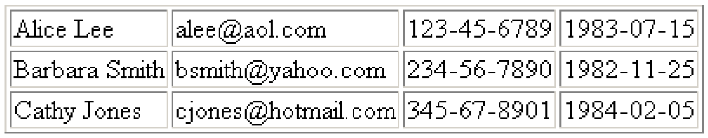
The stylesheet for this file has a section for the address book and a section for the entries.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
	<!-- Web [age section -->
	<xsl:template match="address-book">
		<html>
			<head><title>Address Book</title></head>
		<body>
			<table border="1">
				<xsl:apply-templates select="address"/>
			</table>
		</body>
		</html>
	</xsl:template>
	<!-- Data section -->
	<xsl:template match="address">
		<tr>
			<td><xsl:value-of select="name"/></td>
			<td><xsl:value-of select="email"/></td>
			<td><xsl:value-of select="phone"/></td>
			<td><xsl:value-of select="birthday"/></td>
		</tr>
	</xsl:template>
</xsl:stylesheet>
```

This stylesheet produces a table. The resulting web page is shown below.

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Address Book</title>
</head>
<body>
<table border="1">
<tr>
<td>Alice Lee</td><td>alee@aol.com</td><td>123-45-6789</td><td>1983-07-15</td>
</tr>
<tr>
<td>Barbara Smith</td><td>bsmith@yahoo.com </td><td>234-56-7890 </td><td>1982-11-25</td>
</tr>
<tr>
<td>Cathy Jones</td><td>cjones@hotmail.com </td><td>345-67-8901 </td><td>1984-02-05</td>
</tr>
</table>
</body>
</html>
```

The resulting table is shown below. It could be made more elaborate by adding cell padding, color, etc.



Alice Lee	alee@aol.com	123-45-6789	1983-07-15
Barbara Smith	bsmith@yahoo.com	234-56-7890	1982-11-25
Cathy Jones	cjones@hotmail.com	345-67-8901	1984-02-05

The Roster Example

An XSLT stylesheet can also handle attributes. Consider the roster example on pages 12 and 13.

```
<?xml version="1.0"?>
<?xml-stylesheet type="application/xml" href="roster.xsl"?>

<roster>
    <student>
        <name>Alice Lee</name>
        <midterm weight="40">85</midterm>
        <final weight="60">92</final>
    </student>
    <student>
        <name>Barbara Smith</name>
        <midterm weight="40">78</midterm>
        <final weight="60">84</final>
    </student>
    <student>
        <name>Cathy Jones</name>
        <midterm weight="40">82</midterm>
        <final weight="60">87</final>
    </student>
</roster>
```

Both the midterm and final tags contain attributes.

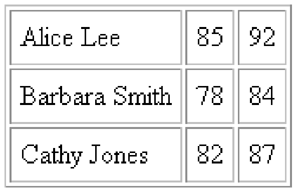
The following stylesheet can be used to put the data into a table ignoring the attributes.

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

    <!-- Web page section -->
    <xsl:template match="roster">
        <html>
            <head><title>Class Roster</title></head>
        <body>
            <table border="1" cellpadding="5">
                <xsl:apply-templates select="student"/>
            </table>
        </body>
        </html>
    </xsl:template>

    <!-- Data section -->
    <xsl:template match="student">
        <tr>
            <td><xsl:value-of select="name"/></td>
            <td><xsl:value-of select="midterm"/></td>
            <td><xsl:value-of select="final"/></td>
        </tr>
    </xsl:template>
 </xsl:stylesheet>
```

The table is shown below.



Alice Lee	85	92
Barbara Smith	78	84
Cathy Jones	82	87

XPath

XPath is the part of XSLT that is used to find places in the XML tree and then do something with them. An XPath command lets you travel either down the tree using the forward slash, '/', or back up the tree using a double slash, '//'. For example, if the current matched node is <student>, we can go down the tree to the weight attribute with midterm/weight. (Attributes are treated as children of the node they are in.) The root node is always denoted by a single slash, '/'. So if any XPath command begins with a slash, it starts at the root node.

A stylesheet can also do arithmetic and place the results in the output. An XSLT transformer reads +, -, and * the usual way, but uses *div* for divide and *mod* for the remainder (% in Java). There are also several functions available such as round (), which rounds the value to the nearest integer.

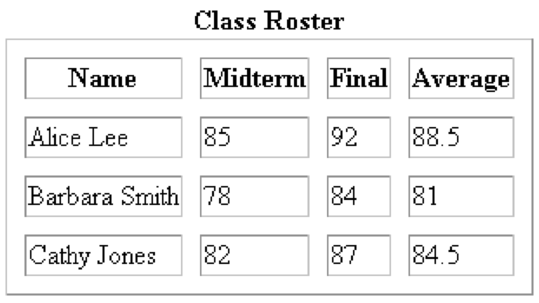
We can add a fourth column to our table that shows the average of the midterm and final grades. The stylesheet for this follows:

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

        <!-- Web page section -->
        <xsl:template match="roster">
            <html>
                <head><title>Class Roster</title></head>
            <body>
                <table border="1" cellspacing="10">
                    <caption><b>Class Roster</b></caption>
                    <tr>
                        <th>Name</th><th>Midterm</th><th>Final</th><th>Average</th>
                    </tr>
                    <xsl:apply-templates select="student"/>
                </table>
            </body>
            </html>
        </xsl:template>

        <!-- Student section -->
        <xsl:template match="student">
            <tr>
                <td><xsl:value-of select="name"/></td>
                <td><xsl:value-of select="midterm"/></td>
                <td><xsl:value-of select="final"/></td>
                <td><xsl:value-of select="(midterm+final)div 2"/></td>
            </tr>
        </xsl:template>
    </xsl:stylesheet>
```

In the fourth select in the student section, the stylesheet computes the average as "(midterm+final)div 2". Simple arithmetic like this can be placed anywhere in a stylesheet. The html table appears below.



Class Roster

Name	Midterm	Final	Average
Alice Lee	85	92	88.5
Barbara Smith	78	84	81
Cathy Jones	82	87	84.5

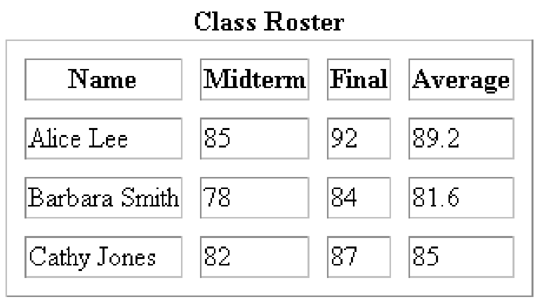
Using the Attributes

In the example above, we ignored the attribute, weight. But it is in there for a purpose. It indicates that for the final average, the midterm should be worth 40% and the final 60%. In a stylesheet attributes are denoted by placing an 'at' sign, '@', before them. So the weight attribute is shown as @weight.

But we cannot just replace the computation, "(midterm+final)div 2", by one containing the weights. We have to use XPath to get us from the <student> node to the weight nodes. These are children of the <midterm> and <final> nodes, respectively. So the computation above changes to

```
        <xsl:value-of select="(midterm/@weight*midterm+final/@weight*final)div 100"/>
```

The resulting web page table is shown below. Note that the averages are not the same as before.



Class Roster

Name	Midterm	Final	Average
Alice Lee	85	92	89.2
Barbara Smith	78	84	81.6
Cathy Jones	82	87	85

Java Support for Transformations

The Java 1.4[17] release contains full support for XSLT. The files are in the javax.xml.transform package. The transformation is done in an abstract class called Transform. The most important method in the class is the transform method. It has two parameters, an input source and an output destination.

Before using the transform method, you have to get a transformer factory object and with this a transformer. This is done with the following code.

```
    TransformerFactory tFactory = TransformerFactory.newInstance ();
    Transformer transformer = tFactory.newTransformer (new StreamSource (sourceFile));
```

Once you have a transformer, you can use it to transform the source.

```
    transformer.transform  (new StreamSource (sourceFile),
                                 new StreamResult (new FileOutputStream (outputFile)));
```

These methods throw several exceptions that either must be caught or thrown again.

The following Java program is a slight modification of the SimpleTransform example that comes with Xalan. Xalan has the following description: "Xalan-Java is an XSLT processor for transforming XML documents into HTML, text, or other XML document types."[18] It can be compiled with Java 1.4 and run with any XML file that includes a stylesheet processing instruction.

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import java.io.*;

/* Use the TraX interface to perform a transformation in the simplest manner possible. */
public class SimpleTransform
{
    public static void main(String[] args)
    {
        String filename = "";
        BufferedReader stdin = new BufferedReader (new InputStreamReader (System.in));
        try
        {
            System.out.print ("XML filename: ");
            filename = stdin.readLine ();
            TransformerFactory tFactory = TransformerFactory.newInstance ();
```

[17] Java versions can be found at http://java.sun.com/j2se/.
[18] Xalan is a product of the Apache Software Foundation. It can be found at http://xml.apache.org/xalan-j/.

```
                Transformer transformer = tFactory.newTransformer
                                        (new StreamSource (filename + ".xsl"));
                transformer.transform (new StreamSource (filename + ".xml"),
                                        new StreamResult (new FileOutputStream (filename + ".html")));
                System.out.println ("The result is in " + filename + ".html ");
            } catch (IOException e) {System.out.println ("IO Error");}
              catch (TransformerException e) {e.printStackTrace(System.err);}
        } // main
} // SimpleTransform
```

XML Parsers

There are two models for XML parsers, SAX (Simple API for XML) and DOM (Document Object Model). SAX is the simpler model. It uses callbacks to get the data from the parser. These are similar to listeners such as the MouseListener found in Java. DOM creates a complete parse tree and provides methods to find information in it. It is more complicated to write and to use.

If an XML document is well-formed, it defines a general ordered tree. A parser has to read through the document and store the tree information. And if the document has either a DTD or a schema, it must determine whether or not the document is valid. Parsers are not easy to construct, but several have been made available.

The parsers we will use are part of Xerces, from the Apache Software Foundation. The original versions were created in 1999 at IBM and subsequently donated to Apache. The open source community has since modified them. Using them requires several JAR files. These can be downloaded from Apache. There are also copies on my web site.[19]

The SAX Parser

SAX (Simple API for XML) uses a call back model.[20] As the parser reads through the document, it sends back information about what it finds. There are several built-in methods (similar to those that come with AWT listeners) that can be used to access this information. The interface that defines them follows:

```
public interface ContentHandler
{
        public void setDocumentLocator (Locator locator);
        public void startDocument () throws SAXException;
        public void endDocument () throws SAXException;
        public void startPrefixMapping (String prefix, String uri) throws SAXException;
        public void endPrefixMapping (String prefix) throws SAXException;
        public void startElement (String namespaceURI, String localName, String qualifiedName,
                                Attributes atts) throws SAXException;
        public void endElement (String namespaceURI, String localName, String qualifiedName)
                                throws SAXException;
        public void characters (char[] text, int start, int length) throws SAXException;
```

[19] See footnote 15.
[20] Elliotte Rusty Harold, *Processing XML with Java,* chapter 17, Addison Wesley, 2002.

```
    public void ignorableWhitespace (char[] text, int start, int length) throws SAXException;
    public void processingInstruction (String target, String data) throws SAXException;
    public void skippedEntity (String name)throws SAXException;
}
```

The important methods here are startDocument, endDocument, startElement, endElement, and characters. The others are needed for documents with namespaces (prefixes).

A specific class called a content handler has to be added to the parser, similar to the way we added listeners to buttons when using the AWT. The code for that follows, where ElementExtractor[21] is the name of the content handler class.

```
    // Get a content handler and add it to the parser.
    ContentHandler handler = new ElementExtractor ();
    parser.setContentHandler (handler);
```

The parser we will use is called SAXParser, one of the Xerces parsers. It goes into its own folder, with the name, org.apache.xerces.parsers.SAXParser. The parser itself is an instance of the XMLReader class. So the declaration of the parser is

```
    XMLReader parser =
        XMLReaderFactory.createXMLReader ("org.apache.xerces.parsers.SAXParser");
```

If we want the parser to validate the xml document using a DTD (document type definition), we have to add this feature to the parser. It is done by

```
    parser.setFeature ("http://xml.org/sax/features/validation", true);
```

The boolean, true, means that the feature will be active. So the parser will look for a link to the DTD and use it to validate the xml document.

The entire method, doParse follows:

```
// doParse gets a parser, adds a content handler, and parses the document.
public void doParse (String document)
{
    final String VALIDATION_FEATURE_ID = "http://xml.org/sax/features/validation";
    try
    {
        // Set the parser to validate using a DTD.
        XMLReader parser = XMLReaderFactory.createXMLReader
            ("org.apache.xerces.parsers.SAXParser");
        parser.setFeature (VALIDATION_FEATURE_ID, true);

        // Get a content handler and add it to the parser.
        ContentHandler handler = new ElementExtractor ();
        parser.setContentHandler (handler);

        // Parse the document.
        parser.parse(document);
    } catch (SAXException e)
    {   System.err.println("Warning: Parser does not support this validation feature .");}
```

[21] This was modeled after programs found in the book written by Elliotte Rusty Harold: *Processing XML with Java,* chapter 6, Addison Wesley, 2002.

```
	catch (Exception e) {System.err.println(e);}
} // doParse
```

Element Extractor

The main work is done by an inner class called ElementExtractor. It uses methods from the ContentHandler interface to extract information from the document. One of the methods in it is used to get the text data from between the tags. This uses the characters method in the interface.

```
// characters receives all the text in the xml document.
public void characters(char[] text, int start, int length) throws SAXException
{
	String word = "";
	word = word.copyValueOf (text, start, length);
	Node textNode = new Node ("text", word);
	tokens.add (textNode);
} // characters
```

The character data is stored in an array of characters called text. The method provides the start index and the length of the tag data. For example, if the xml document has

	<name>Alice Lee</name>

The method will include in the text array the characters in Alice Lee, tell where these start in the array, and how many (9) there are. These can then be copied into a string and stored in a node in a vector. The vector is called tokens, and the node is textNode. ElementExtractor has an inner class that can access data in the parser class.

SAX Parser Program.

```
package parsers;

import java.util.*;
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;

/*	SAXParser uses a SAX parser from the Apache Software Foundation. It parses the document and
stores the tags and text in a vector. When the end tag is read, it displays all the tags stored in the vector
on the screen.
*/
public class SAXParser
{
	public static void main (String [] args)
	{
		try
		{
			BufferedReader stdin = new BufferedReader (new InputStreamReader (System.in));
			System.out.println ("Enter XML document to parse: ");
			String xmlDoc = stdin.readLine ();
```

```
                ParseDocument parseDoc = new ParseDocument ();
                parseDoc.doParse (xmlDoc);
            } catch (IOException e) {System.out.println ("IO Exception");}
        } // main
} // SAXParser

/* ParseDocument parses the document and stores elements and text in a vector.  When the end tag is
read, it displays the contents of the vector on the screen.
*/
class ParseDocument
{
        private Vector tokens = new Vector (50);

        // doParse gets a parser, adds a content handler, and parses the document.
        public void doParse (String document)
        {
            final String VALIDATION_FEATURE_ID = "http://xml.org/sax/features/validation";
            try
            {
                // Set the parser to validate using a DTD.
                XMLReader parser = XMLReaderFactory.createXMLReader
                    ("org.apache.xerces.parsers.SAXParser");
                parser.setFeature (VALIDATION_FEATURE_ID, true);

                // Get a content handler and add it to the parser.
                ContentHandler handler = new ElementExtractor ();
                parser.setContentHandler (handler);

                // Parse the document.
                parser.parse(document);
            } catch (SAXException e)
            {   System.err.println ("Warning: Parser does not support validation feature .");}
            catch (Exception e) {System.err.println(e);}
        } // doParse

        // ElementExtractor is an inner class that receives call backs from the parser for text and elements.
        class ElementExtractor extends DefaultHandler
        {
            // characters receives all the text in the xml document.
            public void characters (char[] text, int start, int length) throws SAXException
            {

                String word = "";
                word = word.copyValueOf (text, start, length);
                Node textNode = new Node ("text", word);
                tokens.add (textNode);
            } // characters

            // startElement picks out each start element and stores it in the vector.
            public void startElement (String namespaceURI, String localName, String qName,
                                    Attributes atts) throws SAXException
            {
                Node startNode, attrNode;
                startNode = new Node ("startTag", localName);
                tokens.add (startNode);

                // Get the attributes in the tag and add them to the vector.
                for (int count = 0; count < atts.getLength (); count ++)
                {
                    String value = atts.getLocalName (count) + " = " + atts.getValue (count);
                    attrNode = new Node ("attr", value);
                    tokens.add (attrNode);
                }
            } // startElement

            // endElement picks out each end element and stores it in the vector.
            public void endElement (String namespaceURI, String localName, String qName)
                throws SAXException
            {
                Node endNode = new Node ("endTag", localName);
                tokens.add (endNode);
            } // endElement

            /*  endDocument reads the last tag in the document and then displays all the tokens in the
            vector on the screen.
            */
            public void endDocument() throws SAXException
            {
                for (int count = 0; count < tokens.size (); count ++)
                {
                    Node token = (Node)tokens.elementAt (count);
                    token.displayToken ();
                }
            } // endDocument
        } // ElementExtractor
} // ParseDocument

// The Node class is used to store each token returned by the parser.
class Node
{
    private String tagType, tagValue;

    Node (String type, String value)
    {
        tagType = type;
        tagValue = value;
    } // constructor

    protected String getType () {return tagType;}
    protected String getValue () {return tagValue;}
    protected void displayToken () {System.out.println (tagType + "  " + tagValue);}
} // Node
```

DOM Parsers

The Document Object Model[22] works on a different principle from the SAX Parser. The DOM Parser builds a complete parse tree with nodes for everything that can be included in an XML file. There are a number of these, but the most important ones are the document node, the element node and the text node. There are also special nodes for processing instructions, document declarations, comments, CDATA sections, and entities. Here we will just consider the first three.

The document node is used to store information about the entire document. It contains a pointer to the root node of the XML tree, but there are a number of other pointers in it as well. These include those for entities and system and public IDs. In order to begin processing the parse tree, you have to start with this node. The root node of the tree is obtained by

```
    Node node = document.getDocumentElement ();
```

The root node of the tree is an element node. It contains a name and a value, pointers to its first child and next sibling, and a pointer to a collection of attributes. The child nodes are used when you are doing a traversal of the tree, usually recursively. The name is the name in the start tag and usually the value is null. The attributes of the node are stored in a collection called a NamedNodeMap. They can be accessed by using an item method. The order may not be the same as that in the original document.

The text node is the simplest. It just contains all the text between the start and end tags. So the text for

```
    <name>Alice Lee</name>
```

is just 'Alice Lee'. Text nodes are always leaf nodes in the tree.

Each node contains a short value that indicates the type of the node. These all have constant values so that programs can refer to Node.DOCUMENT_NODE, Node.ELEMENT_NODE, and Node.TEXT_NODE, rather than having to remember the exact short value.

The first thing that a program that uses DOM has to do is to get a parser and parse the tree. The parser is usually configured to do either DTD or schema validation as well. This is handled by the following code.[23] It first gets the type of validation and the parser, adds the appropriate validation feature to the parser and then parses the tree.

The DOM Parser Program

```
package dom;

import java.io.*;
import org.w3c.dom.*;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;


/**
```

[22] Elliotte Rusty Harold, *Processing XML with Java,* chapter 9, Addison Wesley, 2002.
[23] This program was adapted from one provided by the Apache Software Foundation at http://xml.apache.org/xerces-j/.

```
 *    Adapted from code that comes with Xerces and was written by Andy Clark, IBM.
 *    The DomParser sets up a parser with either DTD or Schema validation.  It then calls DisplayTree
 *    to display the document on the console screen.
*/
public class DomParser
{
     // Validation feature id (http://xml.org/sax/features/validation).
     protected static final String
          Validation_Feature_Id = "http://xml.org/sax/features/validation";

     // Schema validation feature id (http://apache.org/xml/features/validation/schema).
     protected static final String
          Schema_Validation_Feature_Id = "http://apache.org/xml/features/validation/schema";

     // Default parser name.
     protected static final String Default_Parser_Name = "dom.wrappers.Xerces";

     // Main program.
     public static void main(String [] args)
     {
          // Request the validation option.
          System.out.println ("DTD and Schema Parser and Validator");
          BufferedReader stdin = new BufferedReader (new InputStreamReader (System.in));
          String xmlfile = "", option = "";
          try
          {
               // Get the validation option and file name.
               System.out.print ("Option: v for DTD or s for schema. ");
               option = stdin.readLine ();
               System.out.print ("XML File Name: ");
               xmlfile = stdin.readLine ();
          } catch (IOException e) {System.err.println ("No xml file name or option");}

          ParserWrapper parser = null;
          boolean DTDvalidation = false;
          boolean schemaValidation = false;

          if (option.equalsIgnoreCase ("v")) DTDvalidation = option.equals ("v");
          if (option.equalsIgnoreCase ("s")) schemaValidation = option.equals ("s");

          // Create parser.
          try
          {
               parser = (ParserWrapper) Class.forName (Default_Parser_Name).newInstance();
          }
          catch (Exception e) {System.err.println("Unable to instantiate parser");}

          // Set parser features.
          try
          {
               parser.setFeature (Validation_Feature_Id, DTDvalidation);
```

```
                parser.setFeature (Schema_Validation_Feature_Id, schemaValidation);
            } catch (SAXException e)
                {System.err.println ("Parser does not support this validation feature.");}

            try
            {
                // Parse the file.
                Document document = parser.parse (xmlfile);

                // Display the file on the screen.
                DisplayTree treeDisplay = new DisplayTree ();
                treeDisplay.display (document);
                System.out.println ();
            } catch (SAXParseException e) {System.err.println ("SAXParse error.");}
            catch (Exception e)
            {
                System.err.println ("Parse error occurred - "+e.getMessage());
                e.printStackTrace (System.err);
            }
        } // main
} // class DomParser
```

The parser above can be used in a number of different ways to get data from the XML file. It can be used alone in order to validate a file with either a DTD or schema. It can also be used to find a specific node in the tree. The following is taken from the program Write.java[24] that is included as an example with Xerces version 1.1. Write.java contains much more code and can be used with XML files that have processing instructions, CDATA sections, and comments.

The class, DisplayTree contains a number of methods, but the important one is the first one, display. It uses recursion to get the data from the tree. The recursion is a little unusual, since only start tags are members of the tree, not end tags. So to display an end tag in the correct place, it has to first display all the children of the node before tacking on the end tag. This is done by first visiting all the first child descendents of the node and then going back to their next siblings.

```
        Node child = node.getFirstChild();
        while (child != null)
        {
            display (child); // Recursive call.
            child = child.getNextSibling();
        }
```

The rest of the code is used to find the node type and then display the data in the node.

```
/**
 *    DisplayTree uses recursion to travel through the parse tree and display the document.
 *    The node types displayed are document, element and text.
 */
package dom;
import java.io.*;
```

[24] See footnote 15.

```
import org.w3c.dom.*;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import java.lang.reflect.Method;

public class DisplayTree
{
	// display writes the specified node and then uses recursion to visit the rest of the tree.
	public void display (Node node)
	{
		if (node == null) return; // End of recursion
		short type = node.getNodeType();
		displayNode (node, type);

		/* Get the first child of the node and then travel all the way to the left most leaf.  Pop up, go to
		the next sibling, and repeat.  This continues until the entire tree has been visited. */
		Node child = node.getFirstChild();
		while (child != null)
		{
			display (child); // Recursive call.
			child = child.getNextSibling();
		}

		// Add the end tag after returning from the recursion.
		if (type == Node.ELEMENT_NODE)
			System.out.print ("</" + node.getNodeName() + ">");
	} // display (Node)

	// displayNode checks for the node type and calls the method that displays that type.
	protected void displayNode (Node node, short type)
	{
		if (type == Node.DOCUMENT_NODE)	// xml declaration
		{
			displayDocNode (node);
			Document document = (Document) node;
			node = document.getDocumentElement ();
		}
		else
		if (type == Node.ELEMENT_NODE)		//start-tag
			displayElementNode (node);
		else
		if (type == Node.TEXT_NODE)		// text
			System.out.print (node.getNodeValue ());
	} // displayNode

	// displayDocNode is used to display some of the data in the document node.
	protected void displayDocNode (Node node)
	{
		Document document = (Document) node;
		String version = null;
		try
```

```
        {
            // Get the XML version
            Method getXMLVersion =
                document.getClass ().getMethod ("getXmlVersion", new Class[]{});
            // If Document class implements DOM L3, this method will exist.
            if (getXMLVersion != null)
                version = (String) getXMLVersion.invoke (document, null);
        } catch (Exception e) {}

        // Display the appropriate version declaration.
        if (version.equals("1.1"))
            {System.out.println ("<?xml version=\"1.1\" encoding=\"UTF-8\"?>");}
        else {System.out.println ("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");}
    } // displayDocNode

    // displayElementNode first displays the name of the node and then gets and displays the attributes.
    protected void displayElementNode (Node node)
    {
        System.out.print ("<" + node.getNodeName ());

        // Get the attributes from the node and display them in the tag.
        NamedNodeMap attrs = node.getAttributes ();
        for (int count = 0; count < attrs.getLength (); count++)
        {
            Attr attr = (Attr) attrs.item(count);
            System.out.print (" " + attr.getNodeName() + "=\"" + attr.getNodeValue () + "\"");
        }
        System.out.print ('>');
    } // displayElementNode
} // TreeDisplayClass
```

References

1. Clark, Andy, *Write.java*, Xerces version 1.1, 2003.
2. Elliotte Rusty Harold, *Processing XML with Java,* chapter 17, Addison Wesley, 2002.
3. Elliotte Rusty Harold and Scott Means, *XML In a Nutshell*, Third Edition, O'Reilly & Associates, Inc., 2004.
4. Raggett, Dave, A History of HTML, Chapter 2, Addison Wesley Longman, 1998, http://www.w3.org/People/Raggett/book4/ch02.html.
5. W3Schools Online Web Tutorials, http://www.w3schools.com.