

Internet Encyclopedia

XSL

Jesse M. Heines, Ed.D.
heines@cs.uml.edu

Dept. of Computer Science
University of Massachusetts Lowell
One University Avenue
Lowell, MA 01854 USA

Abstract

XSL — the Extensible Stylesheet Language — is an XML-based technology for transforming XML documents from one form to another. It uses a declarative programming paradigm and a specific XML namespace that gives programmers full access to all XML components — elements, attributes, and text — and the ability to manipulate them in ways that go far beyond the capabilities of Cascading Style Sheets (CSS). XSL can be used to control the rendition of XML data, selectively filter the data items selected for transformation, convert data from various incompatible forms into a single, standard form, or implement just about any other operation that one might want to perform on XML data without changing the original XML source. Various parts of XSL are now industry standards (known as World Wide Web Consortium “Recommendations”) and are therefore highly usable even in today’s ever-changing Web environment.

This article presents the basic concepts and techniques used in XSL. It provides a variety of examples of XSL Transformations (XSLT), XPath Expressions (the language used to refer to collections of XML nodes for processing by XSLT), and XSL Formatting Objects (XSL-FO).

*This article was published in The Internet Encyclopedia, edited by Hossein Bidgoli,
January 2004, John Wiley & Sons, Inc.*

XML, XML EVERYWHERE, BUT NOT A DROP OF COMPATIBILITY

If we both store our data as XML, we should be compatible with each other, right? Unfortunately, no. The XML Recommendation specifies how XML documents are formed, but not how the data they contain should be organized. Consider, for example, the many valid ways to store a date shown in Table 1. The simple solution, of course, is for everyone to agree to express dates in the same format. As the flower girl cum socialite mused in *My Fair Lady*, “Wouldn’t it be lovely?” But as the auctioneer cum Mafioso chirped in *Mickey Blue Eyes*, “Fughedaboutit!”

The problem is not that people disdain compatibility, it’s that XML allows each of us to wrap our data in any tags that make sense to us alone. There are no standards for tag names. So even though Company A may really want its inventory system to automatically communicate with Company B’s order processing system, and even though Company A’s sending program can generate XML output

and Company B’s receiving program can take XML input, the two may still not be able to communicate. Both may be Y2K compliant, but if Company A’s program represents the year number in an element (`<year>2002</year>`) and Company B’s program expects an attribute on a date tag (`year="2002"`), well, “never the twain shall meet.” The analogy here is that if you ask for a “hoagie” in a town where such sandwiches are known as “grinders” or “heroes” or “subs,” you’ll very likely go hungry.

“But this is no big deal,” you say. “All one has to do is write a simple C++ or Java program to convert one data format to another.” I fear that such programs are not as simple as they may at first appear, and of course the more complex the conversion, the more complex the conversion program. XML data is fairly easy to create and maintain, while C++ and Java programs are not. It would be nice if we had an XML-based solution to convert XML data from one format to another. This is precisely what XSLT — XML Stylesheet Language *Transformation* — is for.

Table 1. Many valid ways to store a date in XML.

<code><date>February 26, 2002</date></code>	<code><!-- American standard --></code>
<code><date>26 February 2002</date></code>	<code><!-- European standard --></code>
<code><date month="February" day="26" year="2002" /></code>	<code><!-- no ambiguity --></code>
<code><date></code>	<code><!-- reasonable alternative --></code>
<code><month>Feb</month></code>	<code><!-- which also eliminates --></code>
<code><day>26</day></code>	<code><!-- ambiguity but now uses --></code>
<code><year>2002</year></code>	<code><!-- an abbreviation for the --></code>
<code></date></code>	<code><!-- month name --></code>
<code><date>2/26/02</date></code>	<code><!-- common American abbreviation --></code>
<code><date>26.2.2002</date></code>	<code><!-- common European abbreviation --></code>
<code><date>2002-02-26</date></code>	<code><!-- ISO 8601 format, standard used --></code>
	<code><!-- in XML Schemas --></code>
<code><date>37313</date></code>	<code><!-- serial number --></code>

A “stylesheet language” may seem a strange moniker for a protocol that converts one data format to another, but the name is historical. XSLT is a component of XSL, the XML Stylesheet Language, which was originally conceived as a native XML replacement for CSS, just as the Schema was conceived as a native XML replacement for Document Type Definitions (DTDs). The XSLT part of XSL is an extremely powerful and efficient method for viewing XML data in a variety of formats. It does this by essentially *transforming* XML into HTML. A few examples will make this readily apparent.

Formatting XML-based Web Pages

If one brings up XML data in Internet Explorer, one sees a nice tree-structured rendition such as that in Figure 1. Microsoft has cleverly made this display interactive, allowing the user to expand and contract subtrees by clicking the + and – prefixes. Nice for developers, but rather awkward for regular folks who are interested in the data *per se*, not its XML representation.

A relatively few lines of XSL can transform this data into the user-friendly format shown in Figure 2. In this example, XSL was used to generate the entire HTML page being displayed, but one could just as easily use XSL to format just part of the page. Thus dynamic data stored in XML format can be integrated with dynamic or static data stored in other Web-friendly formats to produce pages that not only look good, but that also allow more sophisticated processing because the data has semantics (expressed in XML tags) rather than just formatting information (expressed in HTML tags).

[Readers who want to dive right into code will find listings for this and the examples in Figures 3 and 4 below in the Appendix. Also note that the Internet Explorer XSL Engine will *not* apply XSL formatting to an XML file that contains a Schema reference on the root node!]

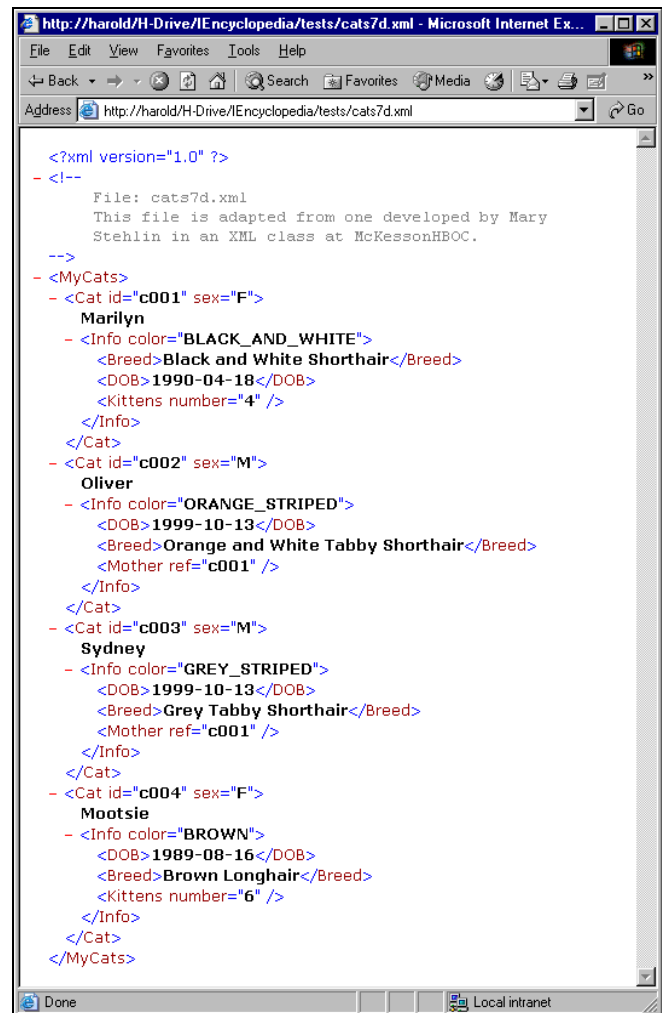


Fig. 1. The Internet Explorer XML data default rendition. This file is adapted from one developed by Mary Stehlin in an XML class taught by the author at McKessonHBOC, Inc., in Alpharetta, GA, May 2000.

Id	Name	Sex	Breed	Dob	Color	Kittens
c001	Marilyn	F	Black and White Shorthair	1990-04-18	BLACK_AND_WHITE	4
c004	Mootsie	F	Brown Longhair	1989-08-16	BROWN	6
c002	Oliver	M	Orange and White Tabby Shorthair	1999-10-13	ORANGE_STRIPED	--
c003	Sydney	M	Grey Tabby Shorthair	1999-10-13	GREY_STRIPED	--

Fig. 2. The same data formatted with XSL.

Generating Reports from XML Data

Look carefully at Figures 1 and 2 and notice that “Mootsie” appears as the fourth cat in Figure 1, yet is second in Figure 2. This is an indication that XSL can do more than just format data: it can *rearrange* it, too. Rearrangement is another simple type of transformation [Cagle, 2000a]. XSL can also *filter* XML data by extracting selected elements whose values match a given *pattern* and, if so desired, juxtapose their data values with those of other selected elements. Such “filtering” is similar to “selecting” data from relational databases with SQL queries that incorporate WHERE clauses. Furthermore, XSL can *sort* data on the values of any XML element, allowing data to be presented in a logical order for the purpose at hand.

These types of basic transformations are the essence of generating reports, that is, presenting different views of the same data for different purposes or different audiences. Figure 3 shows the cats in order of their birth dates. Figure 4 shows only the two female cats and presents just their IDs, names, and number of kittens. Each of these reports was generated from the same XML file by applying a different XSL file to its contents.

Id	Name	Sex	Breed	Dob	Color	Kittens
c004	Mootsie	F	Brown Longhair	1989-08-16	BROWN	6
c001	Marilyn	F	Black and White Shorthair	1990-04-18	BLACK_AND_WHITE	4
c002	Oliver	M	Orange and White Tabby Shorthair	1999-10-13	ORANGE_STRIPED	--
c003	Sydney	M	Grey Tabby Shorthair	1999-10-13	GREY_STRIPED	--

Fig. 3. Cat data sorted by DOB (Date Of Birth).

Resolving Differences Between Disparate XML Data

The DOB data in the Cats XML file is stored in ISO 8601 format, an international standard that defines a method for writing dates and times unambiguously (W3C, 1997). The date part of the standard specifies that a four-digit year is followed by a two-digit month (with a leading zero if necessary) and a two-digit day (again possibly with a

leading zero). Hyphens separate the three data items, making YYYY-MM-DD the common abbreviation for this format. In addition to begin unambiguous, dates expressed in ISO 8601 format have the distinct advantage of being alphabetically sortable. That is, a simple alphanumeric sort will correctly put “2002-01-31” before “2002-02-01,” while the same sort would incorrectly put “January 31, 2002” after “February 1, 2002” and “31.01.2002” after “01.02.2002.”

When a programmer is confronted with the need to compare dates stored in the many formats shown at the beginning of this article, one approach is to convert all dates to ISO 8601 format and then do the required comparison (see Figure 5). XSLT provides the capability to do this and similar conversions with relatively few lines of code and surprising execution speed. (Code to do this is presented in the final example of this article.) Using XSLT for this task allows a programmer to stay within the XML paradigm, eliminating the need to write functions in other, more general-purpose languages.

Id	Name	Kittens
c001	Marilyn	4
c004	Mootsie	6

Fig. 4. Cat data filtered to show only selected data in selected elements of the two female cats.

Communicating Between Applications

The first two sample XSL applications discussed above transform XML into HTML. The third transforms data from one form of XML to another. XSL can actually be used to transform XML data into virtually any text-based format, thus making it an invaluable partner to XML in situations where one application must exchange data with another (Cagle, 2000b).

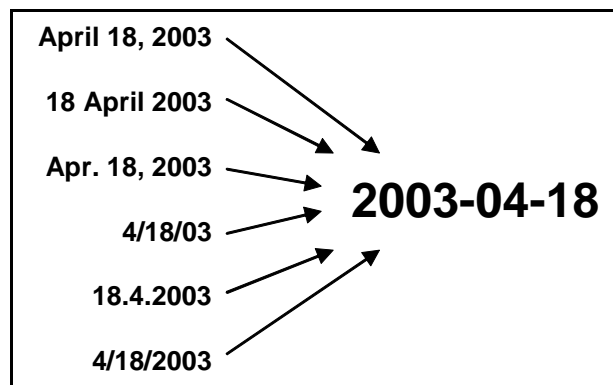


Fig. 5. Transforming dates from various disparate forms into a single, standard form: ISO 8601 format.

Consider, for example, the relationships between various components in a multitiered Web application (see Figure 6). A customer might browse product descriptions and prices on-line using HTML generated from XML. When the customer places the order, that same XML might be processed by another XSL file to generate an SQL query that determines product availability. The accounting department will need the same information in a form compatible with its invoice-generating program, and XSL might provide that in a comma-delimited data stream. The shipping department may require yet another form to generate pick lists and mailing labels. Once again, XSL might be used to perform the required transformation.

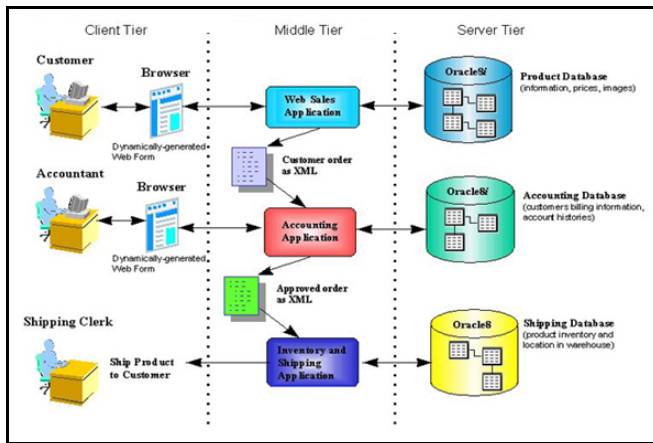


Fig. 6. Work and data flow scenario involving XML data in various formats (Oracle, 1999).
Copyright © 1999, 2001, Oracle Corporation.
All rights reserved. Used with permission.

WHAT XSL IS AND IS NOT
“Decorating” vs. “Transforming” (CSS vs. XSL)

At first glance, XSL may appear to be for XML what CSS are for HTML. Alternatively, a cursory comparison of CSS and XSL might lead one to reasonably — but erroneously — conclude that XSL is a “better” CSS. Indeed, one can apply CSS to an XML document to achieve some of the capabilities we demonstrated with XSL. For example, Figure 7 shows the Cats XML file with CSS attributes applied. However, Table 2 explains the real differences between CSS and XSL. Thus, XSL is *not* “a better CSS.” It is really a different technology with capabilities for manipulating XML data, while CSS is still as valuable as ever for controlling the appearance of generated HTML elements.

XSL as an XML Document

Any discussion of what XSL is and is not will reveal different opinions based on how one uses this technology. However, one issue is not open to debate: XSL structure. An XSL document is first and foremost an XML document, and as such it must conform to all the syntactic and semantic

rules for XML documents. This means that if one includes HTML constructs in an XSL file (a common practice), they must be well-formed. All start tags must have end tags or be self-closing (end with /> rather than just >). Thus one cannot use HTML tags such as
 and in an XSL file without their corresponding end tags, </br> and , which almost never appear in standard HTML. Shortcuts are allowed, like
. [
 is sometimes interpreted as

 by browsers, but
 (with a space) seems to be reliably interpreted as a single
 tag.] One also cannot use entity references like , because XSL has only five entity references: <, >, &, ', and ". (For , use .) Finally, if generating HTML, one must be more careful with white space characters, because these are significant in XML while typically insignificant in HTML. These types of issues become moot if one works in XHTML, which is a version of HTML that conforms to XML standards, thus requiring all documents to be “well-formed” in the XML sense.

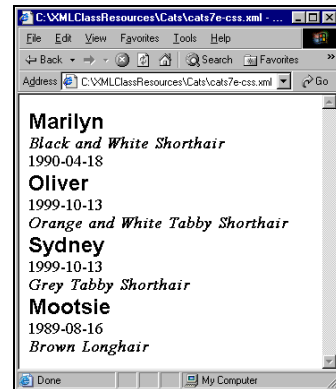


Fig. 7. Cats XML file linked to a CSS file.

Table 2. Capabilities of CSS vs. XSL

CSS	XSL
CSS controls the formatting properties of elements, their so-called “decorations”	XSL “transforms” an XML tree into a new tree
CSS cannot reorder elements, generate text, or perform calculations	XSL can do all of these
CSS cannot access XML attributes	XSL has full control over XML attributes, just as it does over elements
CSS can only render a node’s value once	XSL can use a node’s value as many times as desired and in as many contexts as needed

The characteristic that clearly distinguishes XSL documents from all other XML documents is their *namespace*. The most recent XSL namespace is <http://www.w3.org/1999/XSL/Transform> (W3C, 1999), but one may see older books and papers using a previous version, <http://www.w3.org/TR/WD-xs1>. Internet Explorer 5 (including version 5.5) only supports the older version unless one adds a plug-in. Internet Explorer 6.0 supports the newer version, which is more complete and more closely conforms to the W3C Recommendations. The many various versions of all browsers, including Netscape and Mozilla, provide widely varying levels of support for XSL.

The XSL Processing Model

Purists will state that XSL is a *pattern-matching* language, not a *programming* language. Indeed, it would be quite cumbersome to do many general-purpose programming tasks in XSL. There are, however, a number of programming constructs built into XSL, such as *if* constructs, *choose* constructs (analogous to *switch* or *select* in other languages), *sorting*, *iteration*, *recursive descent*, and numerous others as well as *pattern matching*.

The overall strategy, however, is *declarative* rather than procedural or functional. This means that you “specify how you want the result to look rather than saying how it should be transformed” (Martin *et al.*, 2000, p. 375). An *XSL engine* — software that applies an XSL file to an XML file — first loads an XML source document and an XSL stylesheet into memory (see Figure 8). Internally, each of these documents is represented as a multibranching tree.

The XSL engine then begins processing the stylesheet tree at its root node. The output specifications in this node may cause other stylesheet nodes to be applied to the source

tree in turn. Those may be applied to the whole source tree, selected subtrees, or collections of source tree nodes. Each stylesheet node specifies how the transformation result for some part of the source tree is to look. As shown in Figure 8, applying stylesheet specifications to a source tree results in the XSL engine generating a *result tree*. That result tree can then be output in any of a number of formats, of which text, HTML, and XML itself are the most common.

XSL APPLICATION INFRASTRUCTURE

The Minimal XSL Document

As mentioned above, an XSL document is first and foremost an XML document. Thus all XSL document files begin with the standard XML *processing instruction*:

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

(At present there is only one version of XML, so the version number is always 1.0.)

The *root element* of an XSL document is always *stylesheet*. This element name comes from the XSL namespace, so it must be preceded by a *namespace prefix* (W3C, 1999) followed by a colon. By convention, most people use *xs1* as the namespace prefix, declared with an *xmlns* attribute on this root element (see below). In addition, the stylesheet element requires a *version* attribute which (like XML itself) is currently 1.0. Thus the minimum well-formed and valid XSL document that uses the most recent XSL namespace (as of July 2002) must contain the following three lines:

```
<?xml version="1.0" ?>
<xs1:stylesheet version="1.0"
  xmlns:xs1=
    "http://www.w3.org/1999/XSL/Transform">
  ...
</xs1:stylesheet>
```

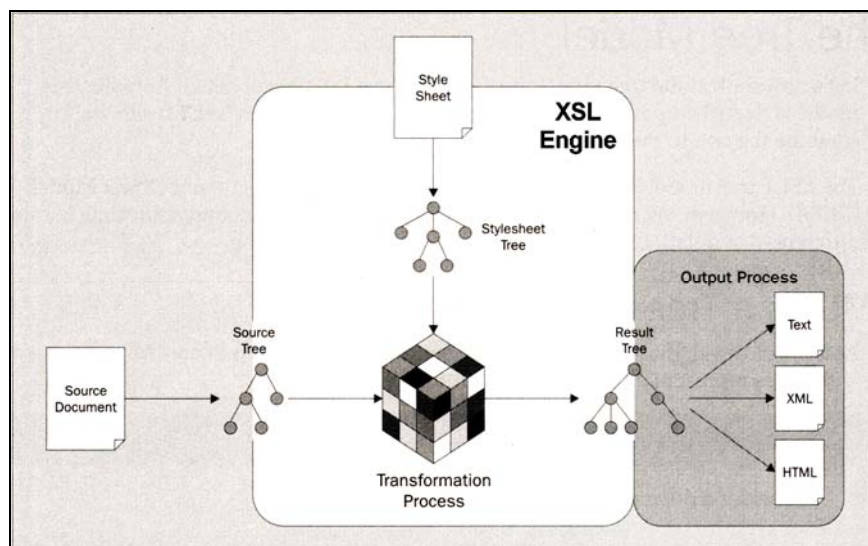


Fig. 8. The XSL Processing Model (Kay, 2000, p. 49).

Copyright © 2000, Wrox Press, Ltd. All rights reserved. Used with permission.

XSL Templates

The abstract stylesheet nodes referred to in the discussion of the XSL processing model above are specified using XSL *templates*, which are introduced by the `xsl:template` tag. These can be thought of as pieces of output that get generated — or further specifications that get processed — when the XSL engine state causes those templates to be applied. Each template is differentiated by a *match condition* that identifies the *context* in which it is applied. (Templates with identical match conditions may optionally be differentiated by *modes*.)

The XSL engine starts its processing with the template whose match condition specifies the XSL document's root context. The `template` tag for this node is:

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

It is important to realize that the XSL document root context is *not* the XML document root node. One should think of the XSL root context as an abstract context *just above* the XML root node, as represented in Figure 9.

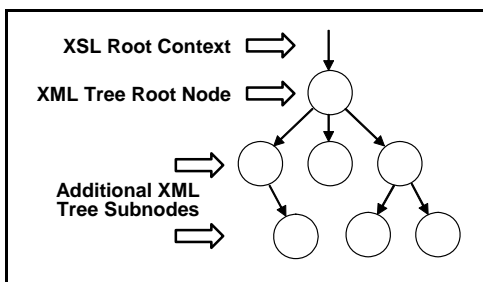


Fig. 9. The XSL Root Context vs. the XML Root Node.

This template is equivalent to the main function in a C or C++ or Java program: it is where processing begins. The output specified here is typically code that “frames” output that will be generated by other templates. For an XSL document designed to generate HTML, this means that this “main” template typically contains at least the opening and closing `<html>` and `</html>` tags, most of the output for the head section, and the body tags. A simple XSL file structured in this way is shown in Listing 1.

If you try to bring up this file in Internet Explorer Version 6.0, you will get the XML tree display shown in Figure 10. This is because an XSL file *is an XML file*, and without *applying* it to an XML file via an XSL engine, it is no different than any other XML file.

Choosing an XSL Engine

When it comes to selecting an XSL engine, there are many choices. We will follow the lead of most books on this subject by using the XSL engine built into Internet Explorer 6.0 (Microsoft, 2002), because that makes it easy to see our results. As mentioned earlier, the IE 6.0 XSL

Listing 1. File hello.xsl.

```
1 <?xml version="1.0" ?>
2 <!--
3   hello.xsl - minimal XSL file
4   updated by JMH on April 11, 2002
5 -->
6 <xsl:stylesheet version="1.0"
7   xmlns:xsl=
8     "http://www.w3.org/1999/XSL/Transform"
9 <xsl:template match="/">
10 <html>
11 <body>
12 <h2>
13   Hello, XSL!
14 </h2>
15 </body>
16 </html>
17 </xsl:template>
18
19 </xsl:stylesheet>
```

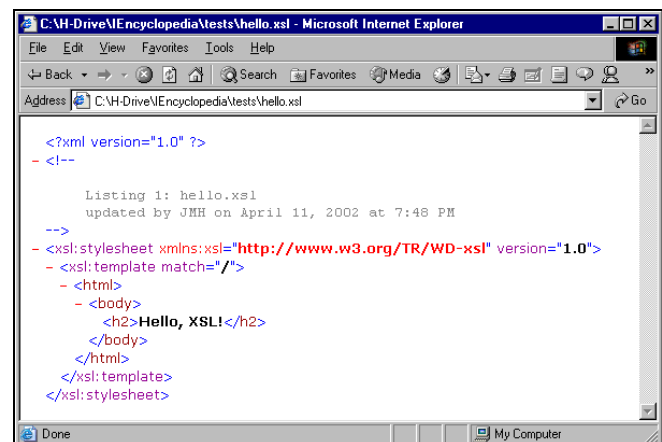


Fig. 10. File hello.xsl displayed as XML.

engine conforms well to the latest World Wide Web XSL Recommendation, but previous versions do not.

One should realize, however, that there are numerous other quality XSL engines available and several ways to link an XML file to an XSL file using these engines. The choice of which engine and which technique to use depends largely on where one is using XSL (on the client side, the server side, or in a stand-alone application) and the format of one's XML and XSL documents (files on disk or data structures in memory). We cannot explore all the possibilities in this article, but it is worth mentioning that the differences in browser versions as of the time of this writing (July 2002) make applying XSL on the client side extremely unreliable.

Most of today's XSL processing that generates HTML Web pages is therefore done on the server side. A popular XSL engine that integrates very smoothly with Java Web servers is the Xalan-Java engine, which is available free of charge from the Apache Software Foundation (Apache, 2002). In this article we will work with the client-side Internet Explorer engine, but the appendix provides listings

of small programs to apply XSL files to XML files on the server-side. These programs include a small, self-contained Java Server Page that demonstrates using hard-coded XML and XSL file names and an analogous Java Servlet that processes XML and XSL file names supplied by an HTML form.

Applying an XSL File to an XML File

To link an XSL file to an XML file, one can include a processing instruction in the XML file that specifies the relative path to the XSL file to be applied:

```
<?xml-stylesheet type="text/xsl"
  href="hello.xsl" ?>
```

Since the simple XSL file in Listing 1 makes no reference to the elements in any XML file that may link to it, we can create a minimal XML file that includes only this processing instruction and some arbitrary root tag required to make the XML document well-formed. Bringing up the XML file shown in Listing 2 in Internet Explorer Version 6.0 generates the output shown in Figure 11.

Listing 2. Minimal XML file that links to an XSL file.

```
1 <?xml version="1.0"?>
2 <!--
3   hello.xml - minimal version including
4   xml-stylesheet processing instruction
5   updated by JMH on April 11, 2002
6 -->
7 <?xml-stylesheet type="text/xsl"
8   href="hello.xsl" ?>
9 <hello>
10 </hello>
```

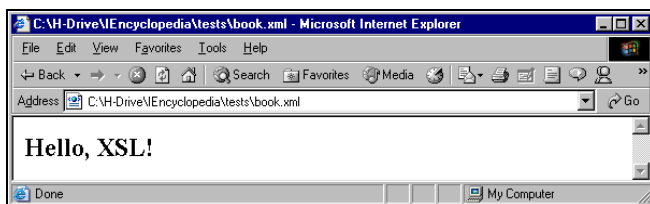


Fig. 11. File `hello.xsl` applied to XML file `hello.xml`.

Inserting the `xml-stylesheet` processing instruction in an XML file as at line 7 in Listing 2 is equivalent to hard-coding the link. Microsoft JScript (their extended version of JavaScript) provides the ability to load XML and XSL files dynamically and apply one to the other under program control. This technique allows XML data styled with XSL to be seamlessly integrated into HTML pages. The minimal code to accomplish this is shown in Listing 3. (One could, of course, read the file names into variables and apply them dynamically, but they are shown hard-coded in Listing 3 for simplicity.) Of course, JScript is only one way to apply XSL to be applied to XML dynamically. The appendix includes code to do so through a Java Server Page and a

Java Servlet, and other techniques are possible as well.

Listing 3. Minimal JScript code to apply an XSL file to an XML file.

```
1 <html>
2 <!--
3   Minimum Code for Applying an XSL file to
4   an XML file on the Client Side using
5   Internet Explorer Version 6.0
6   updated by JMH on April 13, 2002
7 -->
8 <body>
9   <script type="text/javascript">
10    var xmlDoc = new ActiveXObject(
11      "Microsoft.XMLDOM" );
12    xmlDoc.async = false ;
13      // disable multithreading
14    xmlDoc.load( "hello.xml" );
15
16    var xslStyleSheet = new ActiveXObject(
17      "Microsoft.XMLDOM" );
18    xslStyleSheet.async = false ;
19    xslStyleSheet.load( "hello.xsl" );
20
21    document.write( xmlDoc.transformNode(
22      xslStyleSheet ) );
23  </script>
24 </body>
25 </html>
```

EXTRACTING XML DATA

We're now ready to explore the code that gives XSL technology its real power. This code centers around the `xsl:template` element we have already seen, but with considerably more complexity. A large portion of that complexity resides in XPath, the language used to express the crucial *patterns* that appear in the match and test attributes of XSL elements. Like XSLT, XPath is an integral part of XSL. Of course, entire books have been written about XSLT and XPath (see, for example, the *XSLT Programmer's Reference* by Michael Kay, 2000), but the scope of this article is considerably less ambitious. It strives only to give a feel for the types of things one can do with the basic XSLT elements, how XPath is used to apply those elements to selected parts of the XML tree, and how one should think of an XSL document as a whole.

We'll use the XML file in Listing 4 for the examples in this section. This file contains information on the titles and authors of four of the chapters in the *Internet Encyclopedia* using a variety of XML structures. These structures allow us to demonstrate how XSL can address each piece of data by applying different XSL files to this XML document. Most of the transformations in the remainder of this article were generated under program control using the minimal JScript code shown previously in Listing 3, which explains why there is no `xml-stylesheet` processing instruction hard-coded in the `book.xml` file shown below.

Listing 4. XML file for use in examples in this section.

```

1  <?xml version="1.0"?>
2  <!--
3   book.xml - selected chapters and their authors
4   updated by JMH on July 17, 2002
5   -->
6  <book>
7   <chapter title="Java Server Pages (JSP)">
8     <author last="Pratter" first="Frederick" />
9     <title>Adjunct Instructor</title>
10    <affiliation>University of Montana
11      <department>Information Technology</department>
12      <e-mail>pratter@cs.umt.edu</e-mail>
13    </affiliation>
14  </chapter>
15  <chapter title="JavaScript">
16    <author last="Roussos" first="Constantine" />
17    <title>Professor</title>
18    <affiliation>Lynchburg College
19      <department>Computer Science</department>
20      <e-mail>roussos@lynchburg.edu</e-mail>
21    </affiliation>
22  </chapter>
23  <chapter title="Extensible Stylesheet Language (XSL)">
24    <author last="Heines" first="Jesse" middle="M." />
25    <title>Associate Professor</title>
26    <affiliation>University of Massachusetts Lowell
27      <department>Computer Science</department>
28      <e-mail>heines@cs.uml.edu</e-mail>
29    </affiliation>
30  <chapter title="XML, XML Everywhere, But Not a Drop of Compatibility" />
31  <chapter title="What XSL Is and Is Not" />
32  <chapter title="XSL Application Infrastructure" />
33  <chapter title="Extracting XML Data" />
34  </chapter>
35  <chapter title="Extensible Markup Language (XML)">
36    <author last="Ulmer" first="John" />
37    <title>Assistant Professor</title>
38    <affiliation>Purdue University
39      <department>Computer and Information Systems Technology</department>
40      <e-mail>jjulmer@tech.purdue.edu</e-mail>
41    </affiliation>
42  </chapter>
43 </book>

```

Extracting Single Data Items*Extracting Data Stored in Element Nodes*

The main way to extract data from an XML document is via the `xsl:value-of` element, and the heart of that element is the `select` attribute which specifies the data to be extracted. The value of the `select` attribute is an XSL *pattern*. The syntax of that pattern is an XPath *expression*, which defines a *collection* of XML nodes for processing by XSLT. Let's replace line 13 in Listing 1 with:

```

<xsl:value-of
  select="book/chapter/affiliation/department" />

```

The result is the single string:

Information Technology

To understand why, consider the following points:

- (a) This `xsl:value` instruction is being executed within the `xsl:template` element whose `match` attribute is `"/`. Therefore, as shown previously in

Figure 9, the context of this template is the XSL root context, just above the XML root node.

- (b) To “descend into” the XML document, we must therefore first reference the XML root node, `book`.
- (c) To descend further, we refer to the nodes in the order in which they appear in the XML source tree, separating successive tree levels with forward slashes (`/`). This is basic XPath syntax. Reading the pattern from right to left: we are looking for a `department` node that has an `affiliation` node as its parent, a `chapter` node as its grandparent, and a `book` node as its greatgrandparent.
- (d) When, as in this case, the XSL pattern references a node whose only child is an unnamed text node (that is, an element whose DTD specification is `<!ELEMENT elementName (#PCDATA)>`), the `xsl:value-of` element returns the text stored in

that node. Thus in this case we get the text “Information Technology.”

- (e) Note that in this example the text of only the first node that matches the XPath specification in the `select` attribute is returned. (We will see how to reference groups of nodes a little later.)

Extracting Data Stored in Attribute Nodes

To extract data stored in attributes, we use the `@` sign:

```
<p><i>Chapter Title:</i>&#160;
  <xsl:value-of select="book/chapter/@title" />
</p>
```

Reading the pattern from right to left: we are looking for the text stored in an attribute node named `title` that is a child of a `chapter` node which is in turn a child of a `book` node. We’ve added some additional HTML code to this group of instructions to show further how XSL output can be wrapped in formatting text. The resultant output is:

Chapter Title: Cascading Stylesheets (CSS)

Extracting Text Data in Mixed Content Nodes

Look at the structure of the data stored inside the affiliation tags in Listing 4. This type of structure is called *mixed content* because it include both text and subelements. The DTD code for this structure is:

```
<!ELEMENT affiliation
  (#PCDATA|department|e-mail)*>
<!ELEMENT department (#PCDATA)>
<!ELEMENT e-mail (#PCDATA)>
```

If we try to extract the text data stored in the affiliation node that begins on line 10 in Listing 4 with the statement:

```
<xsl:value-of
  select="book/chapter/affiliation" />
```

The result is all of the text in all of the subelements:

**Rochester Institute of Technology Information
Technology ell@mail.rit.edu**

To get only the text at the first level of the affiliation node, we use another XPath feature called a *location path* (W3C, 2001). In this case we want to use the `text()` location path, which selects all the text node children of the context node. In the problem at hand, the context node is affiliation. So to get just its text, we use the statement:

```
<xsl:value-of
  select="book/chapter/affiliation/text()" />
```

This gives us just the text we desire:

Rochester Institute of Technology

We have now seen how to extract data from elements that contain only text, attribute values, and nodes that contain mixed content. These are the three most common situations for any *single* piece of data. Let’s now see how to extract sets of data.

Extracting Sets of Data Items

Iteration

One way to extract sets of data is to use the `xsl:for-each` instruction. Like the `xsl:value-of` instruction, `xsl:for-each` has a `select` attribute, but this time the attribute is interpreted as a *node set expression* that selects all the XML data items that match its XPath expression. The `xsl:for-each` instruction then applies the template between its start and end tags to each node in the set.

Consider the code in Listing 5. The `xsl:for-each` instruction appears at line 13, and its XPath specification (reading right to left) selects all of the chapter elements that are children of book elements. This listing also formats the XSL output as an HTML table, a common practice with tabular data. Applying the XSL file in Listing 5 to book.xml in Listing 4 yields the display shown in Figure 12.

Listing 5. XSL iteration with the `xsl:for-each` instruction.

```
1 <?xml version="1.0" ?>
2 <!--
3   book2.xsl
4   updated by JMH on April 15, 2002
5 -->
6 <xsl:stylesheet version="1.0"
7   xmlns:xsl=
8     "http://www.w3.org/1999/XSL/Transform">
9   <xsl:template match="/">
10    <html>
11      <body>
12        <table border="1">
13          <xsl:for-each
14            select="book/chapter">
15            <tr>
16              <td>&#160;<xsl:value-of
17                select="@title" />&#160;</td>
18            </tr>
19          </xsl:for-each>
20        </table>
21      </body>
22    </html>
23  </xsl:template>
24
25 </xsl:stylesheet>
```

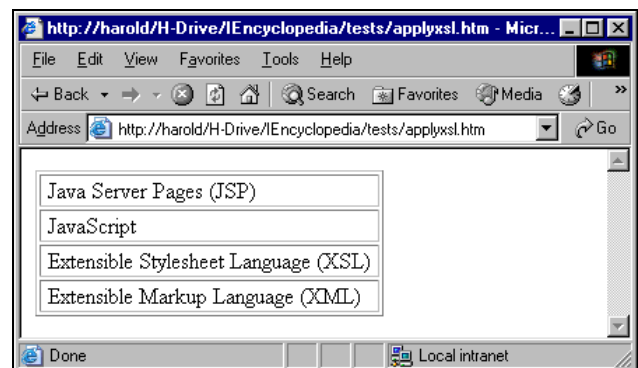


Fig. 12. Result of applying the XSL iteration construct in Listing 5 to the XML file in Listing 4.

This example also demonstrates the important concept of *node context changes*. Note that the XPath expression in the `xsl:value-of` instruction's `select` attribute (line 14 in Listing 5) is `"@title,"` not `"book/chapter/@title"` as in the example we looked at for extracting data stored in attribute nodes:

```
<p><i>Chapter Title:</i>&#160;
  <xsl:value-of select="book/chapter/@title"/>
</p>
```

The difference here is that the `xsl:for-each` instruction *changes the context* of the instructions inside its start and end tags to the node specified in its `select` attribute. Thus line 14 is executed in the context of a `book/chapter` node. Saying it another way, line 14 is executed on a `book/chapter subtree`. We extract the text in the `title` attribute by referring to the XPath *relative to* the current context. Since we're already at `book/chapter`, we only have to go down one more level to `@title`. Understanding context changes is crucial to understanding the preferred way of extracting sets of data items: using recursive descent.

Recursive Descent

There is nothing wrong with iteration, but it is generally thought of as a procedural construct. Since XSL is declarative by nature, creating additional templates and applying them under certain conditions is more in keeping with XSL's overall design philosophy. Templates can be applied recursively as one descends into the XML source tree. Thus this technique is a form of *recursive descent*.

The XSL instruction used to apply templates is aptly named `xsl:apply-templates`. Like the `xsl:for-each` instruction, `xsl:apply-templates` uses a `select` attribute to specify a set of nodes to which matching templates should be applied.

To change the code in Listing 5 from iteration to recursive descent, we first replace lines 12-16 with the single line:

```
<xsl:apply-templates select="book/chapter" />
```

We then define a new template:

```
<xsl:template match="chapter">
  <tr>
    <td>&#160;<xsl:value-of select="@title" />
      &#160;</td>
  </tr>
</xsl:template>
```

Note that the value of the `match` attribute is yet another XSL pattern, but also note that the context of this pattern is somewhat "free floating" and does not include the full XPath expression in the `xsl:apply-templates` `select` attribute. In this example, `match="chapter"` will cause this template to be called *whenever* the context is "chapter," which it is when we specifically descend into the XML document's `book` node and select all the chapter nodes. The output generated by the `xsl:apply-templates` and `xsl:template` instructions above is exactly the same as that

in Figure 12.

If chapters had subchapters that were also identified with chapter tags, we could do a true recursive descent into the source tree to find all the chapter nodes by changing the `xsl:apply-templates` instruction's `select` attribute:

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

Again reading right to left, this expression tells the XSL engine to select "all chapter nodes that are children of any other node." Since the selection is done recursively, all chapter nodes in the structure below would be processed:

```
<chapter
  title="Extensible Stylesheet Language (XSL)">
  <author last="Heines" first="Jesse"
    middle="M." />
  <title>Associate Professor</title>
  <affiliation>University of Massachusetts
    Lowell
    <department>Computer Science</department>
    <e-mail>heines@cs.uml.edu</e-mail>
  </affiliation>
  <chapter title="XML, XML Everywhere, But Not
    a Drop of Compatibility" />
  <chapter title="What XSL Is and Is Not" />
  <chapter title="XSL Application
    Infrastructure" />
  <chapter title="Extracting XML Data" />
</chapter>
```

Filtering

One last variation before we move on: the ability to simulate queries into the XML data by *filtering* the set of extracted data items. To do this, one adds a Boolean expression enclosed within square brackets to an XPath. (Such expressions are more precisely called *predicate expressions* and are an integral part of XPath.) For example:

```
<xsl:apply-templates
  select="book/chapter[not(author/@middle)]" />
```

returns the set of all chapter nodes that are children of `book` nodes and whose child `author` nodes do *not* include a `middle` attribute. (For our sample XML file, this statement would select the nodes for **Frederick Pratter**, **Constantine Roussos**, and **John Ulmer**.)

It is easy to see that such filters can quickly get very complex. The standard Boolean `=`, `!=`, `and`, `or`, and `not` operators exist in XPath, as well as numerous functions such as `contains` and `starts-with` for strings. When working with strings, remember that an XSL document is an XML document, so single quotes must be included inside double quotes, or vice versa, because there is no escape character like `"\"` in C/C++/Java.

This feature provides some of the capabilities of a `WHERE` clause in SQL queries. For example:

```
<xsl:apply-templates
  select="book/chapter[starts-with(
    author/@first,'J')]" />
```

returns the set of all chapter nodes that are children of `book` nodes, and where the value of the first name attribute of the

child author node begins with the letter J. (For our sample XML file, this statement would select the nodes for **Jesse Heines** and **John Ulmer**.)

Sorting Transformations

Once one knows how to refer to each type of data in an XML source using XPath expressions and iterate over sets of nodes or recurse into the XML tree, one has full access to the XML data and can use XSL to transform it in a myriad of ways. Let's look at sorting as an example. This is accomplished by adding `xsl:sort` instructions as children of either the `xsl:for-each` or `xsl:apply-templates` instructions.

The `xsl:sort` element has three main attributes:

- `select` specifies the data on which to sort
- `data-type` is typically either "text" or "number"
- `order` is either "ascending" or "descending"

If one includes multiple `xsl:sort` instructions, the first is taken as the primary sort key, the second as the secondary sort key, etc.

Listing 6. Sorting data.

```

1  <?xml version="1.0" ?>
2  <!--
3   book3.xsl
4   updated by JMH on April 15, 2002 at 11:49 AM
5  -->
6  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
7
8  <xsl:template match="/">
9    <html>
10   <body>
11     <table border="1">
12       <xsl:apply-templates select="book/chapter">
13         <xsl:sort select="author/@last" data-type="text" order="ascending" />
14         <xsl:sort select="author/@first" data-type="text" order="ascending" />
15       </xsl:apply-templates>
16     </table>
17   </body>
18 </html>
19 </xsl:template>
20
21 <xsl:template match="chapter">
22   <tr>
23     <xsl:apply-templates select="author" />
24     <td>&#160;<xsl:value-of select="@title" />&#160;</td>
25   </tr>
26 </xsl:template>
27
28 <xsl:template match="author">
29   <td nowrap="">
30     &#160;<xsl:value-of select="@last" />,
31     <xsl:value-of select="@first" />&#32;
32     <xsl:value-of select="@middle" />&#160;
33   </td>
34 </xsl:template>
35
36 </xsl:stylesheet>

```

The code in Listing 6 generates a table showing the five chapters sorted primarily on the authors' last names and secondarily on their first names. The output is shown in Figure 13.

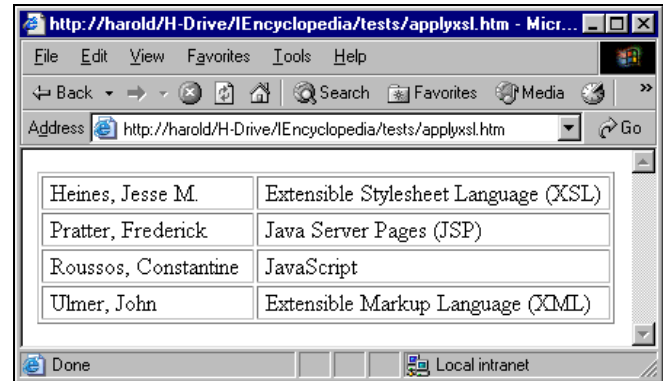


Fig. 13. Result of applying the XSL sort construct in Listing 6 to the XML file in Listing 4.

Note the following points about this code.

- (a) Lines 13 and 14: The XPath in the `xsl:sort` instructions' `select` attribute is relative to the context specified in the enclosing `xsl:apply-templates` instruction.
- (b) Line 23: We have chosen to create a new template to handle author elements. Once again, note the context in the `select` attribute at line 23 and how the context changes in the template at lines 28-34.
- (c) Line 29: Remember once again that an XSL document is an XML document, so we cannot use the standard HTML `<td nowrap>` construct because every attribute must have a value. Thus we have set the value to something. An empty string will do just fine: `<td nowrap="">`.
- (d) Lines 30-32: If we put nothing at the end of line 31, the authors' first and middle names will be run together. In line 24 we used ` `, the XSL equivalent of ` `, to get the extra aesthetically pleasing spaces before and after the text in each cell of our table. But if we use ` ` at the end of line 31, we get two spaces instead of one. Thus we have used ` `, which is the standard ASCII space character.

CONTROLLING XSL PROCESSING ENGINE FLOW

Calling Templates with Parameters

The `xsl:for-each` and `xsl:apply-templates` instructions certainly provide some degree of flow control within the XSL processing engine. Further control can be achieved with the `xsl:call-template` instruction, which allows XSL templates to be called by name just like standard subroutines. In a nutshell, the syntax is:

```
<xsl:call-template name="templateName" />
```

Rather than the `match` attribute we saw in templates called by `xsl:apply-templates`, templates called by name have a `name` attribute:

```
<xsl:template name="templateName">
  instructions to execute when this template is called
</xsl:template>
```

Another important difference between *applying* and *calling* templates is that in the former the node context changes, as we saw above, while in the latter it does not.

There is also a *parameter* construct that allows templates to be called with values that can be further used to control program flow. This construct can be used with `xsl:apply-templates`, but it is more commonly found with `xsl:call-template`:

```
<xsl:call-template name="templateName">
  <xsl:with-param name="parameterName"
    select="pattern" />
</xsl:call-template>
<xsl:template name="templateName">
  <xsl:param name="parameterName" />
```

```
<!-- name must match that used above -->
instructions to execute when this template is called,
parameter can be referenced using $parameterName
</xsl:template>
```

Examples of these constructs appear in the next section.

One can therefore see that even though no one would claim that XSL is a general-purpose programming language, it is a rich set of instructions that provides many of the basic features of a declarative programming language, coupled with exceptional abilities to manipulate XML data.

Conditional Execution

One of the basic flow control features in any language is the Boolean `if` construct that provides conditional instruction execution. XSL does indeed have an `xsl:if` instruction. Its basic format is:

```
<xsl:if test="Boolean expression">
  instructions to execute if the test of the enclosing
  xsl:if is true
</xsl:if>
```

This instruction is not as heavily used as one might expect, however, because it has no “else” clause. People therefore tend to use the XSL equivalent of the C/C++/Java `switch` statement: the `xsl:choose` instruction. Its basic format is:

```
<xsl:choose>
  <xsl:when test="Boolean expression">
    instructions to execute if the test of the enclosing
    xsl:when is true
  </xsl:when>
  ... any number of additional xsl:when
  elements may be included here ...
  <xsl:otherwise>
    instructions to execute if no other Boolean
    expressions in this xsl:choose are true
  </xsl:otherwise>
</xsl:choose>
```

As you surely suspect by now, the Boolean expressions include XPath expressions, which can, in turn, include Boolean operators and numerous functions. The following sample `xsl:choose` construct outputs **Dear Ms. (last name)**: if the current context node's first attribute contains the string “Elizabeth,” otherwise it outputs **Dear Mr. (last name)**:. Note the use of single and double quotes in the `test` attribute of the first `xsl:when` element.

```
<xsl:choose>
  <xsl:when test="@first='Elizabeth'">
    Dear Ms. <xsl:value-of select="@last" />
  </xsl:when>
  <xsl:otherwise>
    Dear Mr. <xsl:value-of select="@last" />
  </xsl:otherwise>
</xsl:choose>
```

One note of clarification to C/C++/Java programmers: each case (`xsl:when` or `xsl:otherwise`) is mutually exclusive. That is, each case ends with an implicit `break` statement (in C/C++/Java terms), and processing exits the

`xsl:choose` structure as soon as any case is completed. It is therefore critical that tests be sequenced from the most specific to the most general, with `xsl:otherwise` (if present) as the last case in the sequence.

MAKING DATA IN XML DOCUMENTS COMPATIBLE

To put everything together that we've seen so far, we return to the problem introduced at the beginning of this article: incompatibility of date formats. The XML file in Listing 7 has six person elements, each with a birth date specified in one of three different formats: in attributes (lines 8-9), in subelements (lines 13-15), or as text (line 19). (Having different date formats in the same XML file is certainly contrived for this demonstration, but it is common to have different formats in different files, as discussed at the beginning of this article.)

Listing 7. XML file with birth dates in various formats.

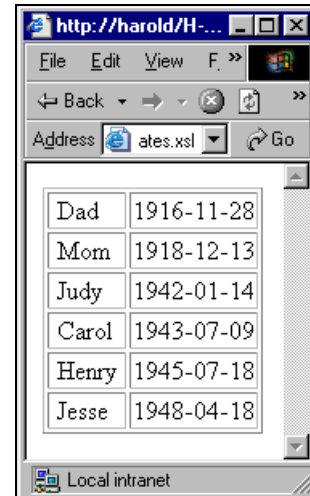
```

1 <?xml version="1.0"?>
2 <!--
3   birthdates.xml
4   updated by JMH on April 15, 2002
5 -->
6 <family>
7   <person id="Dad">
8     <birthdate month="November" day="28"
9       year="1916" />
10  </person>
11  <person id="Mom">
12    <birthdate>
13      <month>December</month>
14      <day>13</day>
15      <year>1918</year>
16    </birthdate>
17  </person>
18  <person id="Judy">
19    <birthdate>1/14/42</birthdate>
20  </person>
21  <person id="Carol">
22    <birthdate>
23      <month>July</month>
24      <day>9</day>
25      <year>1943</year>
26    </birthdate>
27  </person>
28  <person id="Henry">
29    <birthdate month="July" day="18"
30      year="1945" />
31  </person>
32  <person id="Jesse">
33    <birthdate>4/18/48</birthdate>
34  </person>
35 </family>

```

The XSL file in Listing 8 determines how the birth date for each person is stored and executes the required instructions to transform each into ISO 8601 format. The table resulting from the transformation is shown in Figure 14.

The heart of this transformation is in the template that begins on line 58. This template is executed in the context



Dad	1916-11-28
Mom	1918-12-13
Judy	1942-01-14
Carol	1943-07-09
Henry	1945-07-18
Jesse	1948-04-18

Fig. 14. Birthdates transformed to ISO 8601 format.

of a `birthdate` node, which exists for each person regardless of the birth date's format.

At line 62 we test for the existence of a `month` attribute associated with the `birthdate` node. If such an attribute exists, we assume that the birth date is stored in attributes and proceed accordingly. We extract the value of the year attribute and add it to the generated output at line 63. ISO 8601 format specifies that the year value be followed by a hyphen (YYYY-MM-DD). However, adding that hyphen at the end of line 63 generates an extra space in the output, so we follow it with a tag that we know HTML processors will ignore: `<null />`. This dummy tag is immediately followed by another tag at line 64, so no additional spaces are generated.

At line 64 we call the template named `month`, passing it parameter `monthName` with the value extracted from the `month` attribute (line 65). The template named `month` that begins at line 109 transforms month name strings into numbers to conform to ISO 8601 format. Line 67 tests the value of the `day` attribute to see if it is numerically less than 10. If so, this line outputs a 0 to add a leading 0 to the day number to conform to ISO 8601 format.

Line 72 tests for the existence of a `month` subelement in the `birthdate` context. If it exists, we assume that the birth date is stored in subelements and proceed accordingly. The code here is very similar to that in the previous case, except that we extract data from elements rather than from attributes, so no `@` signs appear in the `select` and `test` attributes of the various instructions.

Line 82 tests whether the text in a `birthdate` element contains a forward slash (`/`). If it does, we assume that the birth date is stored in the common American `month/day/year` format. We then use the XPath string functions `substring-before` and `substring-after` to isolate each of the numbers delineated by the slashes and transform them appropriately to conform to ISO 8601 format.

Listing 8. XSL file to display birth dates supplied in various formats in ISO 8601 format.

```

34 <?xml version="1.0" ?>
35 <!--
36   birthdates1.xsl
37   updated by JMH on April 15, 2002 at 7:40 PM
38 -->
39 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
40
41 <xsl:template match="/">
42   <html>
43     <body>
44       <table border="1">
45         <xsl:apply-templates select="family/person" />
46       </table>
47     </body>
48   </html>
49 </xsl:template>
50
51 <xsl:template match="person">
52   <tr>
53     <td>&#160;<xsl:value-of select="@id" />&#160;</td>
54     <td><xsl:apply-templates select="birthdate" /></td>
55   </tr>
56 </xsl:template>
57
58 <xsl:template match="birthdate">
59   <xsl:choose>
60
61     <!-- handle case where date is in attributes -->
62     <xsl:when test="@month">
63       <xsl:value-of select="@year" /><null />
64       <xsl:call-template name="month">
65         <xsl:with-param name="monthName" select="@month" />
66       </xsl:call-template><null />
67       <xsl:if test="@day &lt; 10">0</xsl:if>
68       <xsl:value-of select="@day" />
69     </xsl:when>
70
71     <!-- handle case where date is in subelements -->
72     <xsl:when test="month">
73       <xsl:value-of select="year" /><null />
74       <xsl:call-template name="month">
75         <xsl:with-param name="monthName" select="month" />
76       </xsl:call-template><null />
77       <xsl:if test="day &lt; 10">0</xsl:if>
78       <xsl:value-of select="day" />
79     </xsl:when>
80
81     <!-- handle case where date is in text -->
82     <xsl:when test="contains( ./text(), '/' )">
83       <!-- extract month, day, and year values as strings -->
84       <xsl:variable name="strMonth"
85         select="substring-before( ./text(), '/' )" />
86       <xsl:variable name="strDay"
87         select="substring-before( substring-after( ./text(), '/' ), '/' )" />
88       <xsl:variable name="strYear"
89         select="substring-after( substring-after( ./text(), '/' ), '/' )" />
90       <!-- output year, prefixing with "19" if it's expressed in two digits -->
91       <xsl:if test="string-length( $strYear ) = 2">19</xsl:if>
92       <xsl:value-of select="$strYear" /><null />
93       <!-- output month, prefixing with "0" if it's expressed in one digit -->
94       <xsl:if test="$strMonth &lt; 10">0</xsl:if>
95       <xsl:value-of select="number( $strMonth )" /><null />
96       <!-- output day, prefixing with "0" if it's expressed in one digit -->
97       <xsl:if test="$strDay &lt; 10">0</xsl:if>
98       <xsl:value-of select="number($strDay)" />
99     </xsl:when>
100

```

```

101     <!-- handle default case -->
102     <xsl:otherwise>
103         unknown
104     </xsl:otherwise>
105
106 </xsl:choose>
107 </xsl:template>
108
109 <xsl:template name="month">
110     <xsl:param name="monthName" />
111     <xsl:choose>
112         <xsl:when test="$monthName='January'">01</xsl:when>
113         <xsl:when test="$monthName='February'">02</xsl:when>
114         <xsl:when test="$monthName='March'">03</xsl:when>
115         <xsl:when test="$monthName='April'">04</xsl:when>
116         <xsl:when test="$monthName='May'">05</xsl:when>
117         <xsl:when test="$monthName='June'">06</xsl:when>
118         <xsl:when test="$monthName='July'">07</xsl:when>
119         <xsl:when test="$monthName='August'">08</xsl:when>
120         <xsl:when test="$monthName='September'">09</xsl:when>
121         <xsl:when test="$monthName='October'">10</xsl:when>
122         <xsl:when test="$monthName='November'">11</xsl:when>
123         <xsl:when test="$monthName='December'">12</xsl:when>
124     </xsl:choose>
125 </xsl:template>
126
127 </xsl:stylesheet>

```

The template that begins at line 109 is essentially one large `xsl:choose` instruction. Line 110 accepts the `monthName` parameter passed via the `xsl:with-param` instructions at lines 65 and 75. We then use that parameter in the `xsl:when` test attributes by preceding its name with a dollar sign (\$). We check for each of the twelve month names in turn and add the corresponding month number (with a leading 0 if necessary) to the output when one of the Boolean tests is true.

Thus, the disparate date formats are all made compatible with one another. Of course, one could add additional cases to the `birthdate` template that begins at line 58

to handle all of the variations presented at the beginning of this article. (Serial dates require computing a unique day number, where January 1, 1900 is defined as day 1.)

The previous example is fine for generating HTML output as we have been doing throughout this article, but many applications that use XML and XSL have no interest at all in generating HTML. Instead, they want to take an XML file in one format and convert it to an equivalent XML file another format. That is, they would want to convert the file in Listing 7 to a new file with all dates in the same format. The XSL file in Listing 9 does precisely this, generating the output in Listing 10.

Listing 9. XSL file to generate a new XML file with birth dates in various formats converted to ISO 8601 format.

```

1 <?xml version="1.0" ?>
2 <!--
3   birthdates2.xsl
4   updated by JMH on July 17, 2002 at 03:02 PM
5 -->
6 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
7
8 <xsl:output method="xml" version="1.0" indent="yes" />
9
10 <xsl:template match="/">
11     <xsl:comment> <!-- text within xsl:comment tags will appear in the output file -->
12         birthdates2.xml
13         generated from birthdates.xml by birthdates2.xsl
14     </xsl:comment>
15     <xsl:element name="family"> <!-- root element -->
16         <xsl:apply-templates select="family/person" />
17     </xsl:element>
18 </xsl:template>
19
20 <xsl:template match="person">
21     <xsl:text> <!-- go to a new line and add indent two spaces -->

```

```

22 </xsl:text>
23 <xsl:element name="person"> <!-- person subelement -->
24   <xsl:attribute name="id"> <!-- first attribute on the person element -->
25     <xsl:value-of select="@id" />
26   </xsl:attribute>
27   <xsl:attribute name="birthdate"> <!-- second attribute on the person element -->
28     <xsl:apply-templates select="birthdate" />
29   </xsl:attribute>
30 </xsl:element>
31 </xsl:template>
32
33 <xsl:template match="birthdate">
34   <xsl:choose>
35
36     <!-- handle case where date is in attributes -->
37     <xsl:when test="@month">
38       <xsl:value-of select="@year" />
39       <xsl:text>-</xsl:text> <!-- use this element to avoid whitespace problems -->
40       <xsl:call-template name="month">
41         <xsl:with-param name="monthName" select="@month" />
42       </xsl:call-template>
43       <xsl:text>-</xsl:text>
44       <xsl:if test="@day &lt; 10">
45         <xsl:text>0</xsl:text>
46       </xsl:if>
47       <xsl:value-of select="@day" />
48     </xsl:when>
49
50     <!-- handle case where date is in subelements -->
51     <xsl:when test="month">
52       <xsl:value-of select="year" />
53       <xsl:text>-</xsl:text>
54       <xsl:call-template name="month">
55         <xsl:with-param name="monthName" select="month" />
56       </xsl:call-template>
57       <xsl:text>-</xsl:text>
58       <xsl:if test="day &lt; 10">
59         <xsl:text>0</xsl:text>
60       </xsl:if>
61       <xsl:value-of select="day" />
62     </xsl:when>
63
64     <!-- handle case where date is in text -->
65     <xsl:when test="contains( ./text(), '/' )">
66       <!-- extract month, day, and year values as strings -->
67       <xsl:variable name="strMonth"
68         select="substring-before( ./text(), '/' )" />
69       <xsl:variable name="strDay"
70         select="substring-before( substring-after( ./text(), '/' ), '/' )" />
71       <xsl:variable name="strYear"
72         select="substring-after( substring-after( ./text(), '/' ), '/' )" />
73       <!-- output year, prefixing with "19" if it's expressed in two digits -->
74       <xsl:if test="string-length( $strYear ) = 2">19</xsl:if>
75       <xsl:value-of select="$strYear" />
76       <xsl:text>-</xsl:text>
77       <!-- output month, prefixing with "0" if it's expressed in one digit -->
78       <xsl:if test="$strMonth &lt; 10">0</xsl:if>
79       <xsl:value-of select="number( $strMonth )" />
80       <xsl:text>-</xsl:text>
81       <!-- output day, prefixing with "0" if it's expressed in one digit -->
82       <xsl:if test="$strDay &lt; 10">0</xsl:if>
83       <xsl:value-of select="number($strDay)" />
84     </xsl:when>
85
86     <!-- handle default case -->
87     <xsl:otherwise>
88       unknown
89     </xsl:otherwise>

```



```

90
91   </xsl:choose>
92 </xsl:template>
93
94 <xsl:template name="month">
95   <xsl:param name="monthName" />
96   <xsl:choose>
97     <xsl:when test="$monthName='January'">01</xsl:when>
98     <xsl:when test="$monthName='February'">02</xsl:when>
99     <xsl:when test="$monthName='March'">03</xsl:when>
100    <xsl:when test="$monthName='April'">04</xsl:when>
101    <xsl:when test="$monthName='May'">05</xsl:when>
102    <xsl:when test="$monthName='June'">06</xsl:when>
103    <xsl:when test="$monthName='July'">07</xsl:when>
104    <xsl:when test="$monthName='August'">08</xsl:when>
105    <xsl:when test="$monthName='September'">09</xsl:when>
106    <xsl:when test="$monthName='October'">10</xsl:when>
107    <xsl:when test="$monthName='November'">11</xsl:when>
108    <xsl:when test="$monthName='December'">12</xsl:when>
109   </xsl:choose>
110 </xsl:template>
111
112 </xsl:stylesheet>

```

Listing 10. XML file generated by the XSL file in Listing 9.

```

113 <?xml version="1.0" encoding="utf-8"?>
114 <!--
115   birthdates.xml
116   generated from birthdates.xml by birthdates2.xsl
117   -->
118 <family>
119   <person id="Dad" birthdate="1916-11-28"/>
120   <person id="Mom" birthdate="1918-12-13"/>
121   <person id="Judy" birthdate="1942-01-14"/>
122   <person id="Carol" birthdate="1943-07-09"/>
123   <person id="Henry" birthdate="1945-07-18"/>
124   <person id="Jesse" birthdate="1948-04-18"/>
125 </family>

```

XSL FORMATTING OBJECTS

The third component of XSL is the set of *formatting objects* (XSL-FO) that can be used to render output in sophisticated formats such as PostScript, Portable Document Format (PDF), and even Java (using the Abstract Windowing Toolkit for screen display). Use of these formatting objects is a direct extension of the concepts and techniques presented in this article, applied using the <http://www.w3.org/1999/XSL/Format> namespace. This namespace provides tags similar to, but considerably more sophisticated than, those found in CSS. To include the XSL-FO namespace in an XSL file, simply add it to the `xsl:stylesheet` tag:

```

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

```

The full set of XSL-FO tags and allowable attributes is truly immense. One “short reference” that merely lists them all is 26 pages long! Thus this article can only address the basic structure of an XSL file that uses the XSL-FO namespace. For detailed discussion of XSL-FO, please see the *W3C*

Recommendation for XSL Version 1.0 (W3C, 2001).

XSL formatting objects specify page layout using a hierarchical system of *pages*, *flows*, and *blocks*. Further refinements of the hierarchy include *rectangles*, *borders*, and *spaces*. Full discussion of these topics is beyond the scope of this article, but basically one may think of a simple printed document as having a *master layout* that is broken down into *page sequences* that contain *flows* that are comprised of *blocks*.

Formatting Static Text

As a first example, consider the code in Listing 11, which is adapted from one of the examples provided with the transformation engine used to render the examples in this section: the FOP Formatting Objects Processor from the Apache Group. (This software can be downloaded free of charge from <http://xml.apache.org/fop/>.) The adapted example renders the abstract for this article in Portable Document Format (PDF), which can be viewed in the popular Adobe Acrobat Reader as shown in Figure 15. (The

explanation of this code that follows is drawn largely from the comments embedded in the original example.)

We see from line 1 that this is an XML file, but it does not use the XSL Transformation engine. Therefore, only the XSL-FO namespace is specified on the `fo:root` element in line 9. (The next example will use both namespaces as discussed above.)

The `fo:root` element must contain one and only one `fo:layout-master-set` element (line 12), which in turn contains one or more page master elements that specify sets of *master layout* parameters. In this example, there is only one `fo:simple-page-master` element that defines the layout for all pages. If there were multiple page layouts, they would be differentiated by their *master-name* attributes (line 13).

Listing 11. Simple XSL-FO to render the abstract for this article.

```

1  <?xml version="1.0" encoding="utf-8"?>
2
3  <!--
4  j2a.fo, adapted from fop-0.20.3\docs\examples\fo\simple.fo
5  primary source: http://xml.apache.org/fop/index.html
6  updated by JMH on July 18, 2002 at 01:18 PM
7  -->
8
9  <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
10
11  <!-- master document layout -->
12  <fo:layout-master-set>
13    <fo:simple-page-master master-name="simple"
14      page-height="11.0in" margin-top="1in" margin-left="1.25in"
15      page-width="8.5in" margin-bottom="1in" margin-right="1.25in">
16      <fo:region-body margin-top="0in"/>
17      <fo:region-before extent="0.25in"/>
18      <fo:region-after extent="0.5in"/>
19    </fo:simple-page-master>
20  </fo:layout-master-set>
21
22  <!-- beginning of a page within a document -->
23  <fo:page-sequence master-reference="simple">
24
25    <!-- beginning of a flow within a page -->
26    <fo:flow flow-name="xsl-region-body">
27
28      <!-- title -->
29      <fo:block
30        font-size="18pt" line-height="24pt" background-color="black"
31        font-family="Times" space-after.optimum="24pt" color="white"
32        text-align="center" padding-top="0pt">
33        The Extensible Stylesheet Language (XSL)
34      </fo:block>
35
36      <!-- subtitle -->
37      <fo:block font-size="14pt" line-height="20pt"
38        font-family="Times" space-after.optimum="12pt"
39        font-weight="bold" padding-top="0pt">
40        Abstract
41      </fo:block>
42
43      <!-- first paragraph -->
44      <fo:block font-size="12pt" line-height="15pt" text-indent="25pt"
45        font-family="Times" space-after.optimum="12pt" text-align="justify">
46        XSL - the Extensible Stylesheet Language - is an XML-based technology for
47        transforming XML documents from one form to another. It uses a declarative
48        programming paradigm and a specific XML namespace that gives programmers full
49        access to all XML components - elements, attributes, and text - and the ability
50        to manipulate them in ways that go far beyond the capabilities of Cascading
51        Style Sheets (CSS). XSL can be used to control the rendition of XML data,
52        selectively filter the data items selected for transformation, convert data from
53        various incompatible forms into a single, standard form, or implement just about
54        any other operation that one might want to perform on XML data without changing
55        the original XML source. Various parts of XSL are now industry standards (known

```

```

56     as World Wide Web Consortium "Recommendations"), and are therefore highly usable
57     even in today's ever-changing Web environment.
58 </fo:block>
59
60     <!-- second paragraph -->
61     <fo:block font-size="12pt"      line-height="15pt"      text-indent="25pt"
62           font-family="Times"  space-after.optimum="12pt" text-align="justify">
63         This article presents the basic concepts and techniques used in XSL. It
64         provides a variety of examples of XSL Transformations (XSLT), XPath Expressions
65         (the language used to refer to collections of XML nodes for processing by XSLT),
66         and XSL Formatting Objects (XSL FO).
67     </fo:block>
68
69 </fo:flow>
70 </fo:page-sequence>
71 </fo:root>

```

A document's content is organized into one or more *page sequences*. Each `fo:page-sequence` element (line 23) has a `master-reference` attribute whose value corresponds to the `master-name` attribute of one of the previously defined page master elements (line 13).

Page sequences contain *flows*, which can be positioned in one of five regions: the page header or footer, the left or right margin, or the body. The `fo:flow` tag at line 26 begins the definition of a body flow.

Actual content is then contained within *blocks* that contain text and formatting instructions. There are four blocks in this example. The first one, which begins at line 29, displays the overall title centered in 18-point white type on a black background. (72 points = 1 inch.) The text for this block appears on line 33. The second block, which begins at line 37, displays a subtitle left-justified in bold 14-point type. The text for this block appears on line 40.

The format of the two paragraphs is defined at lines 44-45 and 61-62. The first line of each paragraph is indented 25 points and their text is justified (all text in all lines except the first is aligned at both the left and right margins).

Formatting Text from an XML Document

The real power of XSL formatting objects, of course, comes into play when they are combined with XSL transformations to format text read dynamically from XML files. The basic structure of the XSL file is the same as that shown for static text in the first example, but the XSL-FO tags are now embedded within XSLT tags that control flow and pull data from an XML file. That may sound more like

alphabet soup than computer programming, but the example that follows is intended to clarify how these technologies fit together.

This example uses the XML file in Listing 12, which is an excerpt from a glossary of terms used in this article. The corresponding XSL file in Listing 13.

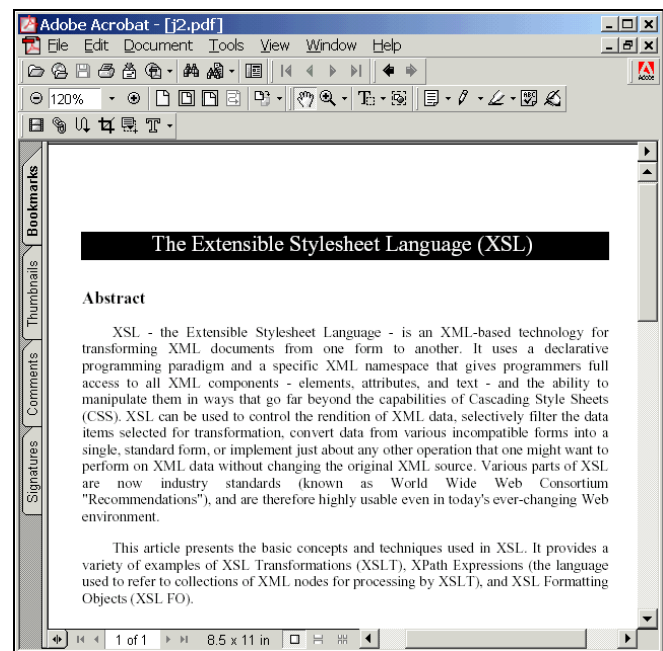


Fig. 15. PDF rendering of the XSL-FO file in Listing 11.

Listing 12. Excerpt from a glossary of terms used in this article encoded in XML.

```

1 <?xml version="1.0" ?>
2 <!--
3   jglossary.xml, adapted from fop-0.20.3\docs\examples\markers\glossary.xml
4   primary source: http://xml.apache.org/fop/index.html
5   Jesse M. Heines, UMass Lowell Computer Science, heines@cs.um1.edu
6   updated by JMH on July 17, 2002 at 10:04 PM
7   -->

```

```

8 <article>
9 <title>The Extensible Stylesheet Language (XSL)</title>
10 <section>
11 <title>Glossary</title>
12 <entry>
13 <term>context</term>
14 <definition>
15 as used in XSL, the tree level at which an instruction is executed; a single
16 instruction will produce different results if executed in different contexts
17 </definition>
18 </entry>
19 <entry>
20 <term>declarative language</term>
21 <definition>
22 a computer language in which one specifies desired results rather than the
23 procedures used to achieve those results (compare to pattern matching language
24 and procedural language)
25 </definition>
26 </entry>
27 <entry>
28 <term>engine</term>
29 <definition>
30 as used in XSL, a program that applies the declarations in an XSL file to the
31 data in an XML file by performs the actions necessary to achieve the specified
32 transformation
33 </definition>
34 </entry>
35 <entry>
36 <term>entity reference</term>
37 <definition>
38 a symbol in an XML file that begins with an ampersand and ends with a semi-
39 colon; there are only five built-in entities in XSL, while there are many more
40 in XML
41 </definition>
42 </entry>
43 <entry>
44 <term>eXtensible Stylesheet Language (XSL)</term>
45 <definition>
46 an XML-structured technology that uses XSL transformations (XSLT), XPath
47 addressing schemes, and (optionally) XSL formatting objects (XSL FO) to
48 transform XML data into other forms
49 </definition>
50 </entry>
51 <entry>
52 <term>filtering</term>
53 <definition>
54 the process of extracting selected data from an XML file that meet a set of
55 specified criteria
56 </definition>
57 </entry>
58 <entry>
59 <term>match condition</term>
60 <definition>
61 a template attribute that specifies the engine state in which that template's
62 instructions will be executed
63 </definition>
64 </entry>
65 <entry>
66 <term>mode</term>
67 <definition>
68 a further refinement of a match condition that allows differentiation between
69 multiple templates with the same match condition
70 </definition>
71 </entry>
72 </section>
73 </article>

```

Listing 13. XSL-FO code to render the glossary XML file.

```

74 <?xml version="1.0" encoding="utf-8"?>
75 <!--
76   jglossary.xml, adapted from fop-0.20.3\docs\examples\markers\glossary.xml
77   primary source: http://xml.apache.org/fop/index.html
78   Jesse M. Heines, UMass Lowell Computer Science, heines@cs.uml.edu
79   updated by JMH on July 18, 2002 at 08:55 AM
80 -->
81 <xsl:stylesheet version="1.0"
82   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
83   xmlns:fo="http://www.w3.org/1999/XSL/Format">
84
85 <xsl:template match="/">
86   <xsl:apply-templates select="article" />
87 </xsl:template>
88
89 <xsl:template match="article">
90   <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
91
92     <fo:layout-master-set>
93       <fo:simple-page-master master-name="all"
94         page-height="11.5in" margin-top="1in" margin-left="1in"
95         page-width="8.5in" margin-bottom="0.75in" margin-right="1in">
96         <fo:region-body margin-top="0in" margin-bottom="0in"/>
97         <fo:region-before extent="0in"/>
98         <fo:region-after extent="0.5in"/>
99       </fo:simple-page-master>
100     </fo:layout-master-set>
101
102     <fo:page-sequence master-reference="all">
103
104       <fo:flow flow-name="xsl-region-body">
105         <!-- article title -->
106         <fo:block
107           font-size="18pt" font-family="sans-serif" line-height="24pt"
108           color="white" text-align="center" padding-top="0pt"
109           background-color="black" space-after.optimum="24pt">
110           <xsl:value-of select="title" />
111         </fo:block>
112
113         <!-- process all sections -->
114         <xsl:apply-templates select="section" />
115
116       </fo:flow>
117     </fo:page-sequence>
118
119   </fo:root>
120 </xsl:template>
121
122 <!-- process a section -->
123 <xsl:template match="section">
124   <!-- section title -->
125   <fo:block
126     font-size="14pt" font-family="sans-serif" font-weight="bold"
127     line-height="20pt" space-after.optimum="12pt" padding-top="0pt">
128     <xsl:value-of select="title" />
129   </fo:block>
130
131   <!-- process a Glossary section -->
132   <xsl:if test="title='Glossary'">
133     <xsl:apply-templates select="entry" />
134   </xsl:if>
135 </xsl:template>
136
137 <!-- process a glossary entry -->
138 <xsl:template match="entry">
139   <fo:block
140     text-align="start" font-size="12pt" text-indent="-0.5in"

```

```

141     margin-left="0.5in" font-family="Times" space-after.optimum="12pt">
142     <xsl:apply-templates select="term"/>
143 </fo:block>
144 </xsl:template>
145
146 <!-- process a term -->
147 <xsl:template match="term">
148   <fo:inline font-weight="bold" >
149     <xsl:value-of select="." />
150     <xsl:apply-templates select="../definition" />
151   </fo:inline>
152 </xsl:template>
153
154 <!-- process a definition -->
155 <xsl:template match="definition">
156   <fo:inline font-weight="normal">
157     - <xsl:value-of select="." />
158   </fo:inline>
159 </xsl:template>
160
161 </xsl:stylesheet>

```

First, we see the presence of both namespaces defined in the `xsl:stylesheet` element at line 81-83. Lines 85-87 are the standard top level “main” template which simply passes processing on to the XML document’s root element, `article`.

The template that matches XML element `article` begins at line 89. It contains the `fo:root` element (line 90) and defines the *master layout* using an `fo:layout-master-set` element (lines 92-100). The *page sequence* begins at line 102. It starts a *flow* at line 104 and creates the first *block* at lines 106-111. Note, however, the one major difference between this example and the previous one: line 110 does not contain static text. Rather, line 110 inserts the text contained in the `title` child element of the XML `article` element using an `xsl:value-of` instruction. Line 114 then descends into the XML tree by applying the appropriate template that matches all the `section` child elements.

That template begins at line 123. It creates a new block at lines 125-129. Here again, at line 128, we use an `xsl:value-of` instruction to pull the text content to be rendered from the XML file rather than hard-coding it as static text in the XSL file. Note, however, that the `title` element here refers to a child of the XML `section` element, which is not the same as the `title` referred to at line 111. Refer back to lines 9 and 11 of the XML file in Listing 12 to see that `title` elements appear at two different levels. The recursive descent structure of the XSL processing engine insures that we are referred to the right element at each level.

Next, controlled by the `xsl:if` instruction at line 132 to make sure that the section we’re currently processing is a glossary, the `xsl:apply-templates` instruction at line 133 initiates processing of all glossary entries. Here’s where the real fun begins!

Each entry element in the XML file has two child elements: `term` and `definition` (see lines 13 and 14 in Listing 12). Before displaying each term, the entry template begins a *block* (line 139 in Listing 13), which in this case is essentially a paragraph. Thus a line break occurs and the formatting instructions for that block are applied. All of the blocks in this layout have `space-after.optimum` set to 12 points (1/6 of an inch), essentially leaving a blank line between glossary entries. Line 142 then initiates processing of the entry element’s `term` child element.

In the `term` template we see an `fo:inline` tag at line 148. This tag begins the definition of a section of text similar to a block, but `inline` indicates that no line break is to occur. The `fo:inline` tag is analogous to an HTML `` tag: it delineates a section that has its own formatting but that is not separated from its surrounding elements by white space. Here the font weight is set to bold, and then the `xsl:value-of` instruction is used to insert the text of the current XML `term` element.

Line 150 continues processing by applying the `definition` template to the current `term`’s sibling `definition` element. In that template (lines 155-159), the `fo:inline` tag is used to return the font weight to normal (non-bold), and then a hyphen is rendered followed by the content of the XML `definition` element.

In addition to the finely tuned formatting control provided by XSL-FO tags and attributes, XSL-FO engines are now beginning to appear that can render those formats in a variety of ways. Figures 16, 17, and 18 show the glossary excerpt we have been discussing rendered in PDF, PostScript, and Java AWT formats, respectively. All of these renderings were created from the same XML and XSL files using the FOP processing engine. To produce the different outputs, only the output specification parameter passed to the FOP program was changed.

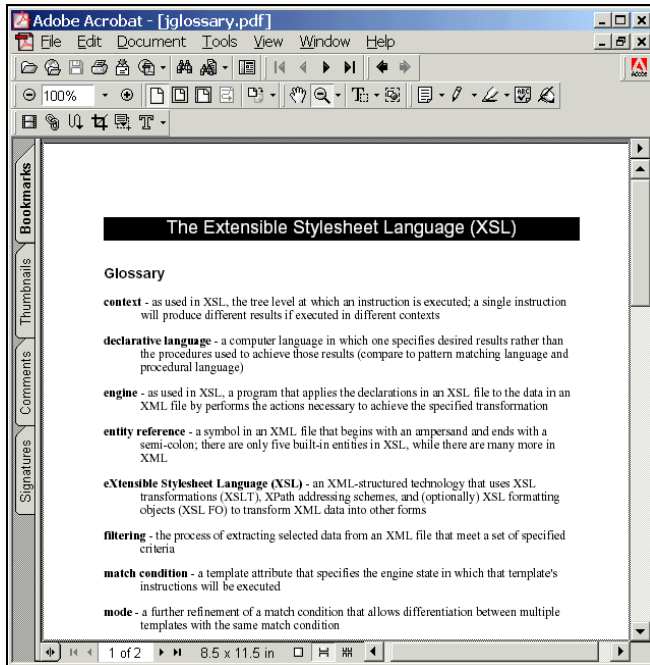


Fig. 16. Glossary transformed to PDF format using XSL-FO.

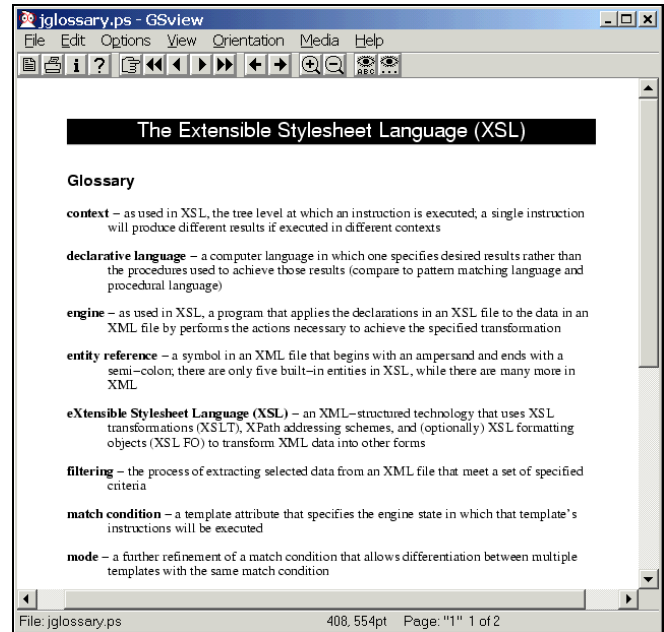


Fig. 17. Glossary transformed to PostScript format using XSL-FO.

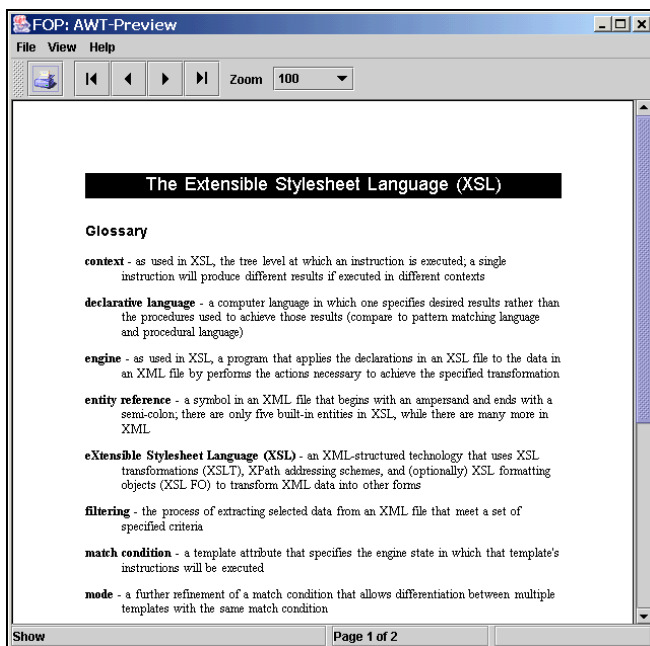


Fig. 18. Glossary transformed to Java AWT format using XSL-FO.

WHERE TO GO FROM HERE

Hundreds — if not thousands — of pages would of course be needed to cover all the features of XSL and its XSLT, XPath, and XSL-FO subcomponents, but hopefully this article has given you enough information to grasp the essence of these powerful Web technologies. The list of references cited in the Bibliography provides pointers to further reference material and Web sites where you can not only learn about XSL, but also download the software discussed in this article to use XSL on your own systems.

BIBLIOGRAPHY

Apache Software Foundation (2002). Xalan-Java. Available at <http://xml.apache.org/xalan-j> (date of access: July 16, 2002).

Cagle, Kurt (2000a). Transform your data with XSL. *XML Magazine* 1(1), 76-80, Winter 1999/2000.

Cagle, Kurt (2000b). ArchitectureX: Designing for XML. *XML Magazine* 1(2), 22-28, Spring 2000.

Kay, Michael (2000). *XSLT Programmer's Guide*, First Edition, ISBN 186-100-3129. (A second edition, ISBN 186-100-5067, is now available.) Birmingham, UK: Wrox Press, Ltd.

Martin, Didier (and twelve other authors) (2000). *Professional XML*. Birmingham, UK: Wrox Press, Ltd.

Microsoft Corporation (2002). Microsoft XML Core Services (MSXML) 4.0 - XSLT Reference. Available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/xmlsdk/html/xsl_ref_overview_1vad.asp (date of access: July 16, 2002).

Oracle Corporation (1999, updated 2001). Using XML

in Oracle database applications: exchanging business data among applications. White paper available at http://technet.oracle.com/tech/xml/info/htdocs/otnwp/xml_data_exchange.htm (date of access: July 16, 2002).

World Wide Web Consortium (1997). Date and time formats. “Note” available at <http://www.w3.org/TR/NOTE-datetime.html> (date of access: April 18, 2002).

World Wide Web Consortium (1999). Namespaces in XML. “Recommendation” available at <http://www.w3.org/TR/REC-xml-names> (date of access: July 16, 2002).

World Wide Web Consortium (1999). XSL Transformations (XSLT), Version 1.0. “Recommendation” available at <http://www.w3.org/TR/xslt> (date of access: July 16, 2002).

World Wide Web Consortium (2001). Extensible Stylesheet Language (XSL), Version 1.0. “Recommendation” available at <http://www.w3.org/TR/2001/REC-xsl-20011015> (date of access: July 16, 2002).

World Wide Web Consortium (2002). XSL Transformations (XSLT), Version 2.0. “Working Draft” available at <http://www.w3.org/TR/xslt20> (date of access: July 16, 2002).

REFERENCE WEB SITES

(in alphabetical order)

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/xmlsdk30/html/xmrefxsltreference.asp> – comprehensive and very well organized reference material on all

aspects of XSLT and XPath

<http://www.dpawson.co.uk/xsl> – extremely comprehensive list of XSL FAQs with extensive answers and examples

<http://www.jenitennison.com/xslt/index.html> – wonderful tutorials and documentation on many advanced applications of XSLT and XPath

<http://www.mulberrytech.com/xsl> – wealth of XSL reference material including wonderful “quick reference” sheets for XML, XSLT, and XPath; also the parent of the phenomenally active XSL-List Open Forum at <http://www.mulberrytech.com/xsl/xsl-list/index.html>

<http://www.netcrucible.com/xslt/msxml-faq.htm> – *unofficial* answers to MSXML XSLT frequently asked questions

<http://www.oasis-open.org/cover/xsl.html> – a comprehensive list of online references for XSL, XSLT, XPath, and related standards

<http://www.w3.org/Style/XSL> – World Wide Web Consortium home page for all XSL-related documents

<http://www.w3.org/TR/xpath> – World Wide Web Consortium Recommendation for the XML Path Language, XPath

<http://www.w3.org/TR/xslt> – World Wide Web Consortium Recommendation for XSL Transformations, XSLT

<http://www.xslt.com> – links to a wealth of tools and tutorials on XSLT

<http://xml.apache.org/fop> – home page for the Apache Group’s FOP Formatting Objects Processor, a print formatter driven by XSL formatting objects

APPENDIX LISTINGS

Appendix Listing 1. XSL code to generate the output in Figure 2 from the XML in Figure 1.

```

1  <?xml version='1.0'?>
2  <!--
3   File: cats7e.xsl
4   updated by JMH on July 17, 2002 at 12:19 PM
5  -->
6  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
7   version="1.0">
8
9  <xsl:template match="/">
10 <html>
11 <body>
12
13 <h2>Mary's Cats</h2>
14
15 <table border="1" cellpadding="2" cellspacing="0">
16 <thead>
17 <td align="center"><b>Id</b></td>
18 <td align="center"><b>Name</b></td>
19 <td align="center"><b>Sex</b></td>
20 <td align="center"><b>Breed</b></td>
21 <td align="center"><b>Dob</b></td>
22 <td align="center"><b>Color</b></td>
23 <td align="center"><b>Kittens</b></td>
24 </thead>
25 <tbody>
26 <xsl:for-each select="MyCats/Cat">
27 <xsl:sort select="./text()" order="ascending" />
28 <tr>

```



```

29         <td><xsl:value-of select="@id"/></td>
30         <td><xsl:value-of select="./text()"/></td>
31         <td align="center">
32             <xsl:value-of select="@sex"/>
33         </td>
34         <td><xsl:value-of select="Info/Breed"/></td>
35         <td><xsl:value-of select="Info/DOB"/></td>
36         <td><xsl:value-of select="Info/@color"/></td>
37         <td align="center">
38             <xsl:choose>
39                 <xsl:when test="Info/Kittens/@number">
40                     <xsl:value-of select="Info/Kittens/@number"/>
41                 </xsl:when>
42                 <xsl:otherwise>
43                     --
44                 </xsl:otherwise>
45             </xsl:choose>
46         </td>
47     </tr>
48 </xsl:for-each>
49 </tbody>
50 </table>
51
52 </body>
53 </html>
54 </xsl:template>
55
56 </xsl:stylesheet>

```

Appendix Listing 2. XSL code to generate the output in Figure 3 from the XML in Figure 1.

```

1  <?xml version='1.0'?>
2  <!--
3   File: cats7e-3.xsl
4   updated by JMH on July 17, 2002 at 12:19 PM
5  -->
6  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
7   version="1.0">
8
9  <xsl:template match="/">
10 <html>
11 <body>
12
13 <h2>Mary's Cats</h2>
14
15 <table border="1" cellpadding="2" cellspacing="0">
16 <thead>
17 <td align="center"><b>Id</b></td>
18 <td align="center"><b>Name</b></td>
19 <td align="center"><b>Sex</b></td>
20 <td align="center"><b>Breed</b></td>
21 <td align="center"><b>Dob</b></td>
22 <td align="center"><b>Color</b></td>
23 <td align="center"><b>Kittens</b></td>
24 </thead>
25 <tbody>
26 <xsl:apply-templates select="MyCats/Cat">
27 <xsl:sort select="Info/DOB" order="ascending" />
28 </xsl:apply-templates>
29 </tbody>
30 </table>
31
32 </body>
33 </html>
34 </xsl:template>
35
36 <xsl:template match="Cat">

```

```

37 <tr>
38 <td><xsl:value-of select="@id"/></td>
39 <td><xsl:value-of select="."/text()"/></td>
40 <td align="center">
41 <xsl:value-of select="@sex"/>
42 </td>
43 <td><xsl:value-of select="Info/Breed"/></td>
44 <td><xsl:value-of select="Info/DOB"/></td>
45 <td><xsl:value-of select="Info/@color"/></td>
46 <td align="center">
47 <xsl:choose>
48 <xsl:when test="Info/Kittens/@number">
49 <xsl:value-of select="Info/Kittens/@number"/>
50 </xsl:when>
51 <xsl:otherwise>
52 --
53 </xsl:otherwise>
54 </xsl:choose>
55 </td>
56 </tr>
57 </xsl:template>
58
59 </xsl:stylesheet>

```

Appendix Listing 3. XSL code to generate the output in Figure 4 from the XML in Figure 1.

```

1 <?xml version='1.0'?>
2 <!--
3 File: cats7e-4.xsl
4 updated by JMH on July 17, 2002 at 01:48 PM
5 -->
6 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
7 version="1.0">
8
9 <xsl:template match="/">
10 <html>
11 <body>
12
13 <h2>Mary's Cats</h2>
14
15 <table border="1" cellpadding="2" cellspacing="0">
16 <thead>
17 <td align="center"><b>Id</b></td>
18 <td align="center"><b>Name</b></td>
19 <td align="center"><b>Kittens</b></td>
20 </thead>
21 <tbody>
22 <xsl:apply-templates select="MyCats/Cat[Info/Kittens/@number]" />
23 </tbody>
24 </table>
25
26 </body>
27 </html>
28 </xsl:template>
29
30 <xsl:template match="Cat">
31 <tr>
32 <td><xsl:value-of select="@id"/></td>
33 <td><xsl:value-of select="."/text()"/></td>
34 <td align="center">
35 <xsl:value-of select="Info/Kittens/@number"/>
36 </td>
37 </tr>
38 </xsl:template>
39
40 </xsl:stylesheet>

```

Appendix Listing 4. Java Server Page code to apply an XSL file to an XML file using the Apache Xalan-Java XSL engine.

```

1 <html>
2 <!--
3 Code to Apply an XSL File to an XML File on the Server Side
4 adapted from xalan-j_2_3_1\samples\SimpleTransform\SimpleTransform.java
5 download Xalan-Java from http://xml.apache.org/xalan-j free of charge
6 updated by JMH on July 16, 2002 at 07:23 PM
7 -->
8 <head>
9 <%@ page import="java.io.*" %> <!-- for StringWriter -->
10 <%@ page import="javax.xml.transform.*, javax.xml.transform.stream.*" %>
11
12 <%!
13 /** Apply an XSL file to an XML file and return the results as a String.
14 * @param strXMLfile String containing full URL to the XML file
15 * @param strXSLfile String containing full URL to the XSL file
16 * @return String containing the output of the transformation
17 */
18 String ApplyXSL( String strXMLfile, String strXSLfile )
19 {
20 // StringWriter is a child of java.io.Writer and can therefore be
21 // used as an argument to a StreamResult constructor, which is
22 // required by Transformer.transform().
23 StringWriter swResult = new StringWriter() ;
24
25 try {
26 // Use the static TransformerFactory.newInstance() method to instantiate
27 // a TransformerFactory. The javax.xml.transform.TransformerFactory
28 // system property setting determines the actual class to instantiate --
29 // org.apache.xalan.transformer.TransformerImpl.
30
31 TransformerFactory tFactory = TransformerFactory.newInstance();
32
33 // Use the TransformerFactory to instantiate a Transformer that will work
34 // with the stylesheet you specify. This method call also processes the
35 // stylesheet into a compiled Templates object.
36
37 Transformer transformer =
38     tFactory.newTransformer( new StreamSource( strXSLfile ) ) ;
39
40 // Use the Transformer to apply the associated Templates object to an XML
41 // document (foo.xml) and write the output to a file (foo.out).
42
43 transformer.transform( new StreamSource( strXMLfile ),
44                       new StreamResult( swResult ) ) ;
45
46 // Return the result.
47 return swResult.toString() ;
48
49 } catch ( TransformerConfigurationException tfce ) {
50     return tfce.toString() ;
51
52 } catch ( TransformerException tfe ) {
53     return tfe.toString() ;
54 }
55 }
56 %>
57 </head>
58
59 <body>
60 <%
61 // This code assumes that the XML and XSL files reside in the same directory
62 // as this JSP. Those file names are hard-code here, but they could be passed
63 // as parameters from an HTML form or via some other such technique.
64 String strXMLfilename = "hello.xml" ; // replace with your XML file name
65 String strXSLfilename = "hello.xsl" ; // replace with your XSL file name
66
67 // We need to construct a full path to the XML and XSL files, including the

```

```

68 // "http://" protocol specification, server name, and server port number (if
69 // it's not the default of 80). The following code accomplishes this.
70 String strFullPath = "http://" + request.getServerName() ;
71 if ( request.getServerPort() != 80 )
72     strFullPath += ":" + request.getServerPort() ;
73
74 // We then add the full path to this JSP and finally strip off this JSP's file
75 // name and extension that follow the last forward slash (/).
76 strFullPath += request.getRequestURI() ;
77 strFullPath = strFullPath.substring( 0, strFullPath.lastIndexOf( "/" ) + 1 ) ;
78
79 // We append the XML and XSL file names to the full path to pass them to our
80 // ApplyXSL method, which applies the XSL file to the XML file and returns the
81 // result as a string. Printing that String shows the result in the browser.
82 out.println( ApplyXSL( strFullPath + strXMLfilename,
83                      strFullPath + strXSLfilename ) ) ;
84 %>
85 </body>
86 </html>

```

Appendix Listing 5. Java Servlet to apply an XSL file to an XML file using the Apache Xalan-Java XSL engine.

```

1  /*
2  * ApplyXSLServlet.java
3  *
4  * Created using Forte for Java 4, Community Edition, on July 17, 2002, 10:30 AM
5  */
6
7  package InternetEncyclopedia.XSLDemos ;
8
9  import javax.servlet.*;
10 import javax.servlet.http.*;
11
12 import java.io.* ;    // for StringWriter and PrintWriter
13 import javax.xml.transform.* ;
14     // for Transformer, TransformerFactory, TransformerException,
15     // and TransformerConfigurationException
16 import javax.xml.transform.stream.* ;
17     // for StreamSource and StreamResult
18
19 /**
20 * This servlet applies an XSL file to an XML file and displays the results.
21 * @author Jesse M. Heines
22 * @version 1.0
23 */
24 public class ApplyXSLServlet extends HttpServlet
25 {
26     /** Apply an XSL file to an XML file and return the results as a String.
27     * @param strXMLfile String containing full URL to the XML file
28     * @param strXSLfile String containing full URL to the XSL file
29     * @return String containing the output of the transformation
30     */
31     private String ApplyXSL( String strXMLfile, String strXSLfile )
32     {
33         // StringWriter is a child of java.io.Writer and can therefore be
34         // used as an argument to a StreamResult constructor, which is
35         // required by Transformer.transform().
36         StringWriter swResult = new StringWriter() ;
37
38         try {
39             // Use the static TransformerFactory.newInstance() method to instantiate
40             // a TransformerFactory. The javax.xml.transform.TransformerFactory
41             // system property setting determines the actual class to instantiate --
42             // org.apache.xalan.transformer.TransformerImpl.
43
44             TransformerFactory tFactory = TransformerFactory.newInstance();
45

```

```

46     // Use the TransformerFactory to instantiate a Transformer that will work
47     // with the stylesheet you specify. This method call also processes the
48     // stylesheet into a compiled Templates object.
49
50     Transformer transformer =
51         tFactory.newTransformer( new StreamSource( strXSLfile ) ) ;
52
53     // Use the Transformer to apply the associated Templates object to an XML
54     // document (foo.xml) and write the output to a file (foo.out).
55
56     transformer.transform( new StreamSource( strXMLfile ),
57                           new StreamResult( swResult ) ) ;
58
59     // Return the result.
60     return swResult.toString() ;
61
62     } catch ( TransformerConfigurationException tfce ) {
63         return tfce.toString() ;
64
65     } catch ( TransformerException tfe ) {
66         return tfe.toString() ;
67     }
68 }
69
70 /** Display an error message for a missing field.
71  * @param strErrorMsg String containing error message to show
72  * @return String containing the output of the transformation
73  */
74 private void ShowErrorMessage( PrintWriter out, String strErrorMsg )
75 {
76     out.println( "<p><font color='red'>" + strErrorMsg + "</font></p>" );
77     out.println( "<p>Please press your browser's BACK button and try again.</p>" );
78 }
79
80 /** Initializes the servlet. (supplied by Forte for Java 4)
81  */
82 public void init(ServletConfig config) throws ServletException
83 {
84     super.init(config);
85 }
86
87 /** Destroys the servlet. (supplied by Forte)
88  */
89 public void destroy()
90 { }
91
92 /** Processes requests for both HTTP <code>GET</code> and <code>POST</code> methods.
93  * @param request servlet request
94  * @param response servlet response
95  */
96 protected void processRequest(
97     HttpServletRequest request, HttpServletResponse response )
98     throws ServletException, java.io.IOException
99 {
100     // This code assumes that the XML and XSL files reside in the same directory.
101     String strXMLpath = request.getParameter( "XMLpath" ) ;
102     // XML file name passed from an HTML form
103     String strXMLfilename = request.getParameter( "XMLfilename" ) ;
104     // XSL file name passed from an HTML form
105     String strXSLfilename = request.getParameter( "XSLfilename" ) ;
106
107     response.setContentType("text/html");
108     java.io.PrintWriter out = response.getWriter();
109
110     out.println( "<html>" ) ;
111     out.println( "<head>" ) ;
112     out.println( " <title>Apply XSL Servlet</title>" ) ;
113     out.println( "</head>" ) ;

```

```

114
115     out.println( "<body>" ) ;
116
117     if ( ( strXMLpath == null ) || strXMLpath.equals( "" ) ) {
118         ShowErrorMessage( out, "No path supplied for XML and XSL files." ) ;
119     } else if ( ( strXMLfilename == null ) || strXMLfilename.equals( "" ) ) {
120         ShowErrorMessage( out, "No name supplied for your XML file." ) ;
121     } else if ( ( strXSLfilename == null ) || strXSLfilename.equals( "" ) ) {
122         ShowErrorMessage( out, "No name supplied for your XSL file." ) ;
123     } else {
124         out.println( ApplyXSL( strXMLpath + strXMLfilename,
125                               strXMLpath + strXSLfilename ) ) ;
126     }
127
128     out.println( "</body>" ) ;
129     out.println( "</html>" ) ;
130
131     out.close();
132 }
133
134 /** Handles the HTTP <code>GET</code> method. (supplied by Forte for Java 4)
135  * @param request servlet request
136  * @param response servlet response
137  */
138 protected void doGet(HttpServletRequest request, HttpServletResponse response)
139 throws ServletException, java.io.IOException
140 {
141     processRequest(request, response);
142 }
143
144 /** Handles the HTTP <code>POST</code> method. (supplied by Forte for Java 4)
145  * @param request servlet request
146  * @param response servlet response
147  */
148 protected void doPost(HttpServletRequest request, HttpServletResponse response)
149 throws ServletException, java.io.IOException
150 {
151     processRequest(request, response);
152 }
153
154 /** Returns a short description of the servlet. (supplied by Forte for Java 4)
155  */
156 public String getServletInfo()
157 {
158     return "This small servlet applies an XSL file to an XML file and displays the " +
159           "results.";
160 }
161 }

```

Appendix Listing 6. HTML form to supply parameters to the Apply XSL Java Servlet.

```

1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
2 <!--
3   ApplyXSLForm.htm: to provide parameters for the ApplyXSLForm servlet
4   Jesse M. Heines, UMass Lowell Computer Science, heines@cs.uml.edu
5   updated by JMH on July 17, 2002 at 10:54 AM
6 -->
7 <html>
8   <head>
9     <title>Apply XSL Form</title>
10    <script type="text/javascript">
11      function init( frm ) // called when body is loaded to initialize form
12      {
13        // set default path to the location of this form
14        var strPathName = "http://" + location.host ;
15        strPathName += location.pathname.substring(
16          0, location.pathname.lastIndexOf( "/" ) + 1 ) ;

```

```
17     frm.XMLpath.value = strPathName ; // set default path in first form field
18     frm.XMLfilename.focus() ; // set focus to second form field
19     }
20 </script>
21 </head>
22 <body onload="init( frm )">
23 <p>Please fill in the fields below to specify the parameters needed to
24 apply an XSL file to an XML file.</p>
25 <p><i>Notes:</i></p>
26 <ul>
27 <li>This form expects your XML and XSL files to be in the same directory.</li>
28 <li>You may edit the full path field, but your entry must be of the form shown
29 initially.</li>
30 </ul>
31 <br />
32 <form name="frm" action="/servlet/InternetEncyclopedia.XSLDemos.ApplyXSLServlet">
33 <table name="tbl">
34 <tr>
35 <td align="right">Path to XML and XSL files:</td>
36 <td><input name="XMLpath" size="60"/></td>
37 </tr>
38 <tr>
39 <td align="right">XML file name:</td>
40 <td><input name="XMLfilename" size="60"/></td>
41 </tr>
42 <tr>
43 <td align="right">XSL file name:</td>
44 <td><input name="XSLfilename" size="60"/></td>
45 </tr>
46 <tr>
47 <td align="right"></td>
48 <td align="right">
49 <input type="submit" value="Submit Entries" />
50 <input type="reset" value="Clear Fields" />
51 </td>
52 </tr>
53 </table>
54 </form>
55 </body>
56 </html>
```