XSL and XSLT

Outline

Glossary

context – as used in XSL, the tree level at which an instruction is executed; a single instruction will produce different results if executed in different contexts

declarative language – a computer language in which one specifies desired results rather than the procedures used to achieve those results (compare to *procedural language*)

engine – as used in XSL, a program that applies the declarations in an XSL file to the data in an XML file by performs the actions necessary to achieve the specified transformation

entity – a symbol in an XML file that begins with an ampersand and ends with a semi-colon; there are only five built-in entities in XSL, while there are many more in XML

filtering – the process of extracting selected data from an XML file that meet a set of specified criteria

match condition – a template attribute that specifies the engine state in which that template's instructions will be executed

mode – a further refinement of a match condition that allows differentiation between multiple templates with the same match condition

namespace – a collection of externally predefined words that are used in an XML document as element types and attribute names; the namespace is specified as a Uniform Resource Identifier, which for XSL is http://www.w3.org/1999/XSL/Transform

namespace prefix – a user-chosen name used to refer to the namespace to which an XML tag name belongs

node – as used in XSL, an object in an XML tree that has a zero or one parent and zero or more children; the abstract representation of an XML tag or attribute or text content

procedural language – a computer language in which one specifies the precise steps required to achieve a desired result (compare to *declarative language*)

processing instruction – a command in an XML file that specifies file attributes such as the version number or stylesheet link; a processing instruction is not part of the XML data

recursive descent – the process of visiting each node in a tree or subtree by recursively tracing the links to each node's child to traverse the entire tree structure

root element – the "top" element of a tree that has no parent node

style – an attribute of a visible element such as its color, size, font type, etc., or any other characteristic appropriate for some other type of element (compare to *transformation*)

stylesheet – a collection of specifications that define how information is presented; stylesheets can define how text data is rendered on a screen, how words are pronounced, etc.

template – a collection of instructions that specify the desired result of a transformation in a specific context and (optionally) mode

transformation – the process of generating a new form of data from another form (compare to *style*)

XSL Stylesheet Language (XSL) – an XML-structured technology that uses XSL transformations (XSLT), XPath addressing schemes, and (optionally) XSL formatting objects (XSL-FO) to process XML data

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 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 generating HTML from XML and converting disparate data into a common format.

INTRODUCTION: XML, XML EVERYWHERE, BUT NOT A DROP OF COMPATIBILITY

If we both store our date as XML, we should be compatible with each other, right? Unfortunately, no. The XML Recommendation specifies how XML documents are formed, but it does not specify how the data they contain should be organized. Consider, for example, the many valid ways to store a date:

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

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 loverly?” But as the auctioneer cum Mafioso chirped in *Mickey Blue Eyes*, “Fughedaboudit!”

The problem is not that people distain 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.

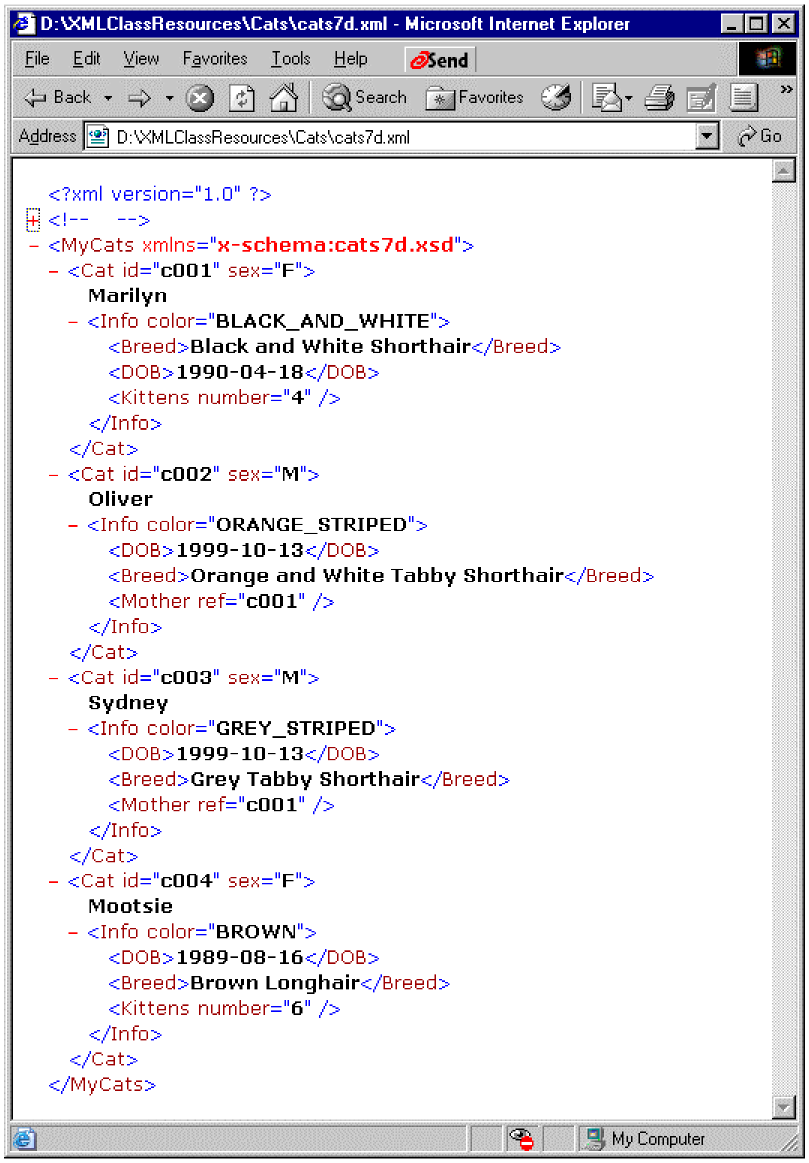
“But this is no big deal,” you say. “All one has to do is write a simple C++ or Java program to convert one data format to another.” I fear that such programs are not so simple as they may at first appear, and of course the more complex the conversion, the more complex the conversion program. XML data 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.

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 an outgrowth 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 DTDs. The XSL subset of XSLT 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.

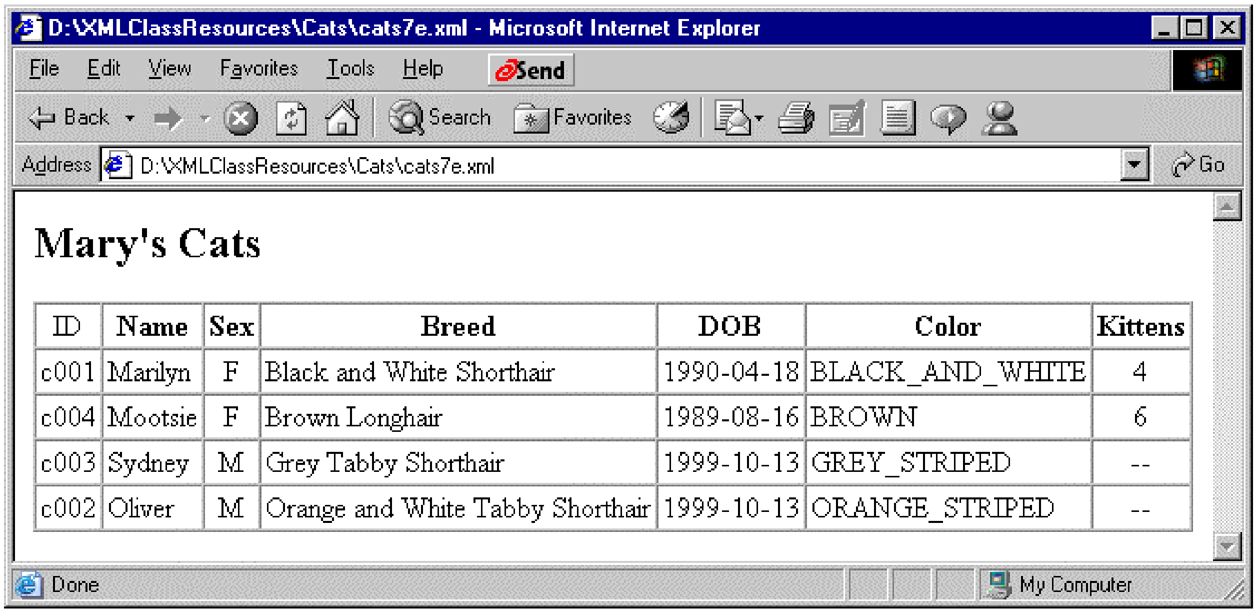

D:\XMLClassResources\Cats\cats7d.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help Send
Back Search Favorites
Address D:\XMLClassResources\Cats\cats7d.xml Go
My Computer

```
  <?xml version="1.0" ?>
+ <!--   -->
- <MyCats xmlns="x-schema:cats7d.xsd">
  - <Cat id="c001" sex="F">
      Marilyn
    - <Info color="BLACK_AND_WHITE">
        <Breed>Black and White Shorthair</Breed>
        <DOB>1990-04-18</DOB>
        <Kittens number="4" />
      </Info>
    </Cat>
  - <Cat id="c002" sex="M">
      Oliver
    - <Info color="ORANGE_STRIPED">
        <DOB>1999-10-13</DOB>
        <Breed>Orange and White Tabby Shorthair</Breed>
        <Mother ref="c001" />
      </Info>
    </Cat>
  - <Cat id="c003" sex="M">
      Sydney
    - <Info color="GREY_STRIPED">
        <DOB>1999-10-13</DOB>
        <Breed>Grey Tabby Shorthair</Breed>
        <Mother ref="c001" />
      </Info>
    </Cat>
  - <Cat id="c004" sex="F">
      Mootsie
    - <Info color="BROWN">
        <DOB>1989-08-16</DOB>
        <Breed>Brown Longhair</Breed>
        <Kittens number="6" />
      </Info>
    </Cat>
  </MyCats>
```

Fig. 1. The Internet Explorer default rendition of XML data.
(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.)

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).


D:\XMLClassResources\Cats\cats7e.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help Send
Back Search Favorites
Address D:\XMLClassResources\Cats\cats7e.xml Go
Done
My Computer

Mary's Cats

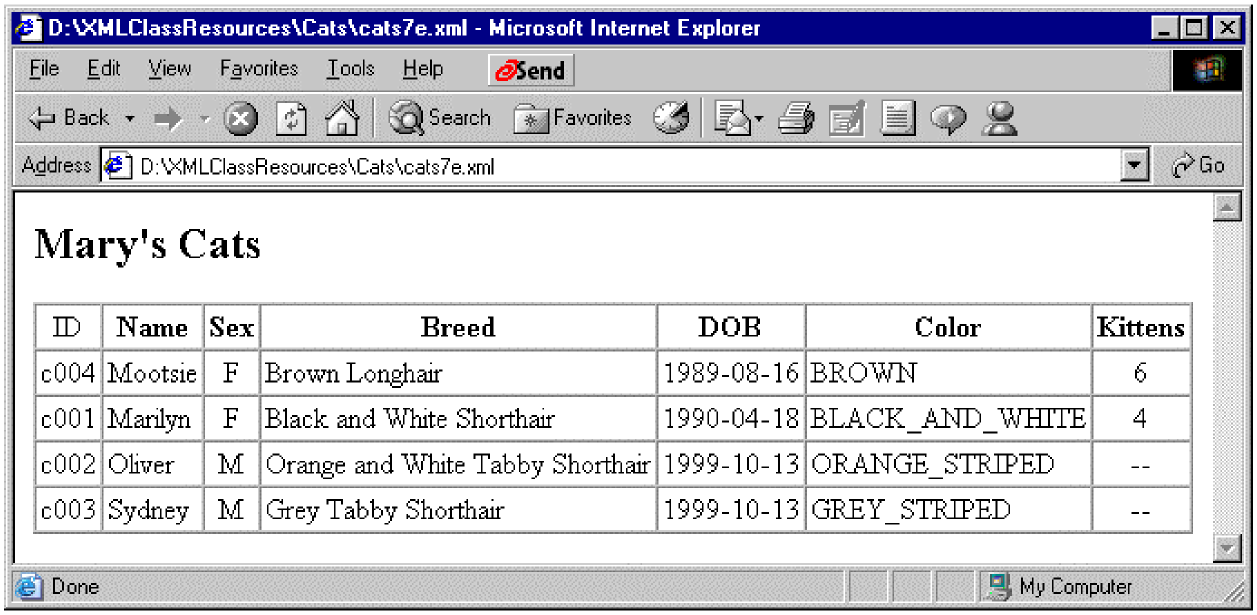
ID	Name	Sex	Breed	DOB	Color	Kittens
c001	Marilyn	F	Black and White Shorthair	1990-04-18	BLACK_AND_WHITE	4
c004	Mootsie	F	Brown Longhair	1989-08-16	BROWN	6
c003	Sydney	M	Grey Tabby Shorthair	1999-10-13	GREY_STRIPED	--
c002	Oliver	M	Orange and White Tabby Shorthair	1999-10-13	ORANGE_STRIPED	--

Fig. 2. **The same data formatted with XSL.**

Generating Reports from XML Data

If one looks carefully at Figures 1 and 2, one will notice that "Mootsie" appears as the fourth cat in Figure 1, yet is listed second in Figure 2. This is an indication that XSL can do more than just format data: it can also *rearrange* it. Rearrangement is another simple type of transformation [Cagle, 2000a]. XSL can also *filter* XML data, that is, extract selected subelements of individual records and, if so desired, juxtapose their data values with those of other selected subelements. Such "filtering" is similar to "selecting" specific 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 an order that makes sense for the purpose at hand.

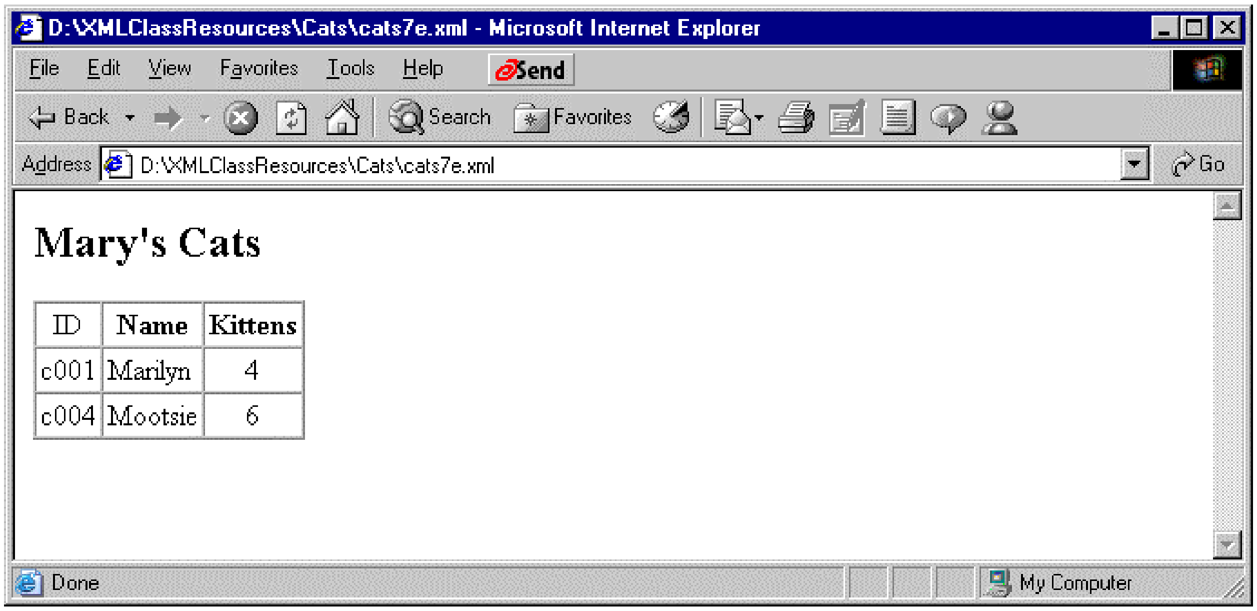
These types of basic transformations are the essence of generating reports, that is, presenting different views of the same data for different purposes or different audiences. Figure 3 shows the cats in order of their birth dates. Figure 4 shows only the two female cats and presents just their IDs, names,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


D:\XMLClassResources\Cats\cats7e.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help Send
Back Search Favorites
Address D:\XMLClassResources\Cats\cats7e.xml Go
Done My Computer

Mary's Cats

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).


D:\XMLClassResources\Cats\cats7e.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help Send
Back Search Favorites
Address D:\XMLClassResources\Cats\cats7e.xml Go
Done My Computer

Mary's Cats

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.

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 unambiguousness, dates expressed in ISO 8601 format have the distinct advantage of being alphabetically sortable. That is, a simple alphanumeric sort will put “2002-01-31” before

"2002-02-01," while the same sort would 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 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.

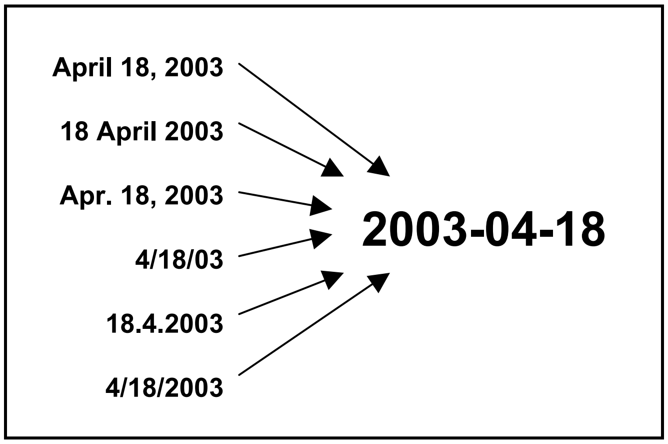


Fig. 5. Transforming dates from various disparate forms into a single, standard form: ISO 8601 format.

Communicating Between Applications

The first two sample XSL/XSLT 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).

Consider, for example, the relationships between various components in a multi-tiered 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. Accounting 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. Shipping may require yet another form to generate pick lists and mailing labels. Once again XSL might be used to perform the required transformation.

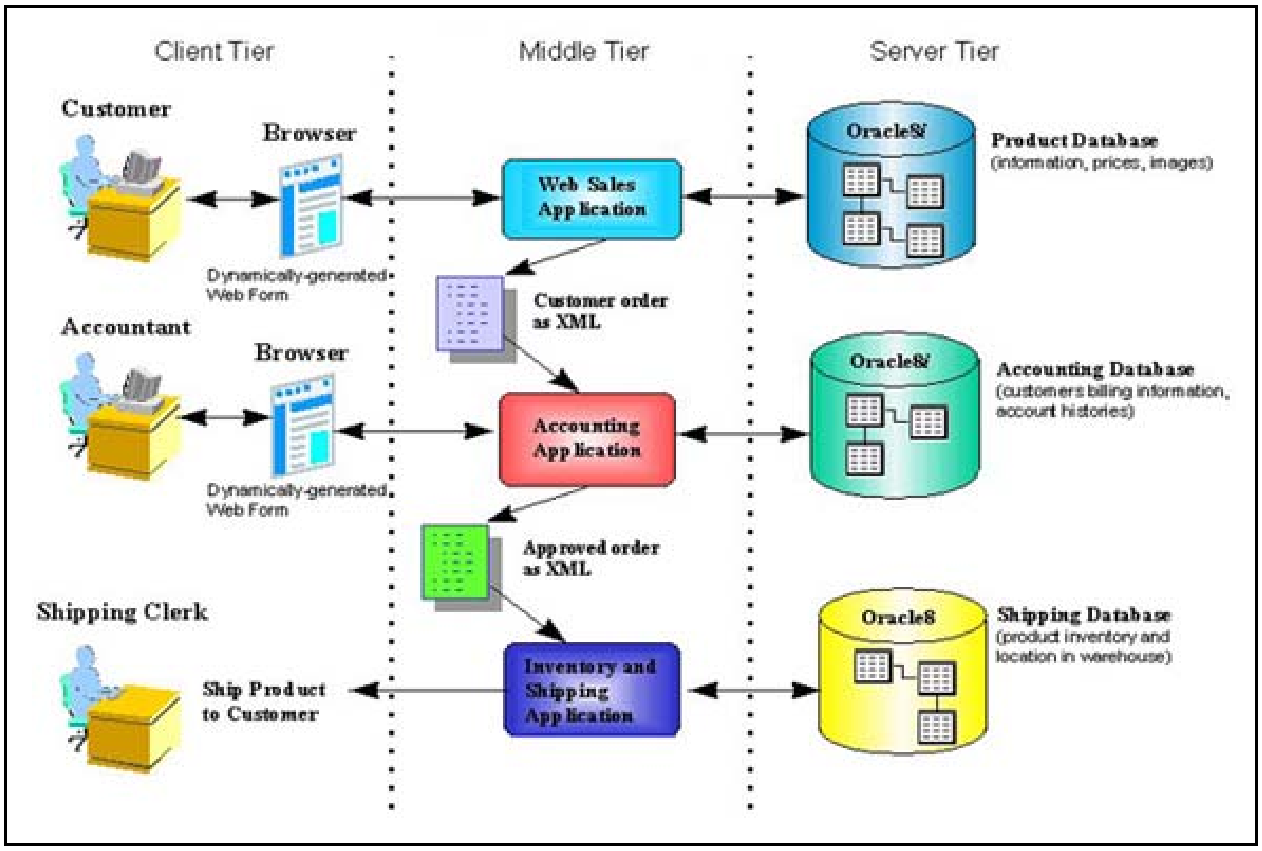
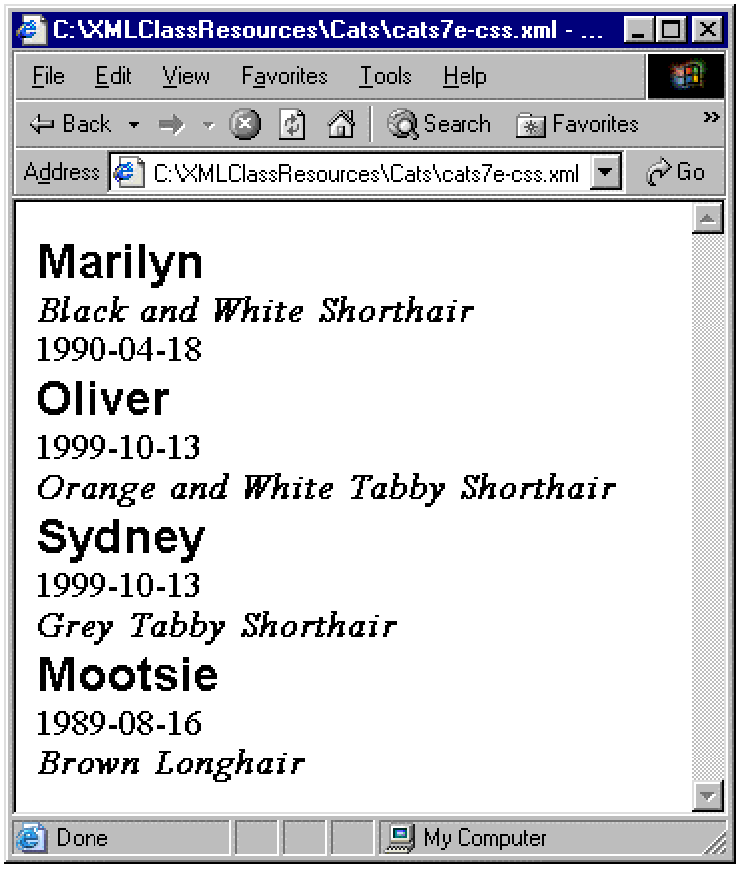

Client Tier
Middle Tier
Server Tier
Customer
Browser
Dynamically-generated Web Form
Accountant
Browser
Dynamically-generated Web Form
Shipping Clerk
Ship Product to Customer
Web Sales Application
Customer order as XML
Accounting Application
Approved order as XML
Inventory and Shipping Application
Oracle8i
Product Database (information, prices, images)
Oracle8i
Accounting Database (customers billing information, account histories)
Oracle8
Shipping Database (product inventory and location in warehouse)

Fig. 6. Work and data flow scenario involving XML data in various formats (Oracle, 1999).

WHAT XSL IS AND IS NOT

"Decorating" vs. "Transforming" (CSS vs. XSL)

At first glance, XSL may appear to be for XML what Cascading Stylesheets (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've demonstrated with XSL. For example, Figure 7 shows the Cats XML file with CSS attributes applied. However, Table 1 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.


C:\XMLClassResources\Cats\cats7e-css.xml - ...
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\XMLClassResources\Cats\cats7e-css.xml
Go
Done
My Computer

Marilyn
Black and White Shorthair
1990-04-18
Oliver
1999-10-13
Orange and White Tabby Shorthair
Sydney
1999-10-13
Grey Tabby Shorthair
Mootsie
1989-08-16
Brown Longhair

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

Table 1. 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 tex 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 latest version of XSL includes *formatting objects* (XSL-FO) which can indeed by used to render output and replace CSS. We do not have sufficient space in this article to do this topic justice, so we have chosen to focus on the basic structure of XSL and XSLT and leave exploration of XSL-FO to the reader. 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 comprehensive than, those found in CSS.

XSL as an XML Document

Any discussion of what XSL is and is not will reveal different opinions based on how one uses XSL (and XSLT). 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 <br> and <img> in an XSL file without their corresponding closing tags, </br> and </img>, which almost never appear in HTML. Shortcuts are allowed, like <br />. (The author has found that <br/> (without any spaces) is sometimes interpreted as <br><br> by browsers. <br /> (with a space between the r and the /) seems to be reliably interpreted as a single <br> tag.) One also cannot use entities like &nbsp; because XSL has only five predefined entities: &lt;, &gt;, &amp;, &apos;, and &quot;. (For &nbsp;, you can use &#160;.) Finally, if generating HTML, one must be more careful with whitespace characters, because these are significant in XML while typically insignificant in HTML.

The one thing that clearly distinguishes an XSL document from other XML documents is its *namespace*. The most recent XSL namespace is http://www.w3.org/1999/XSL/Transform (W3C, 2001), but one may see some older books and documents using a previous version, http://www.w3.org/TR/WD-xsl. Internet Explorer 5 (including version 5.5) only supports the older version unless one adds a plug-in. Internet Explorer 6.0 supports the more complete newer version, which more closely conforms to the W3C Recommendations.

The XSL Processing Model

Purists will state that XSL is a *stylesheet* language, not a *programming* language. Indeed, it would be quite cumbersome to do many general-purpose programming tasks in XSL or XSLT. 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), pattern matching, sorting, iteration, recursive descent, and numerous others.

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 multi-branching 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 XSL itself are the most common.

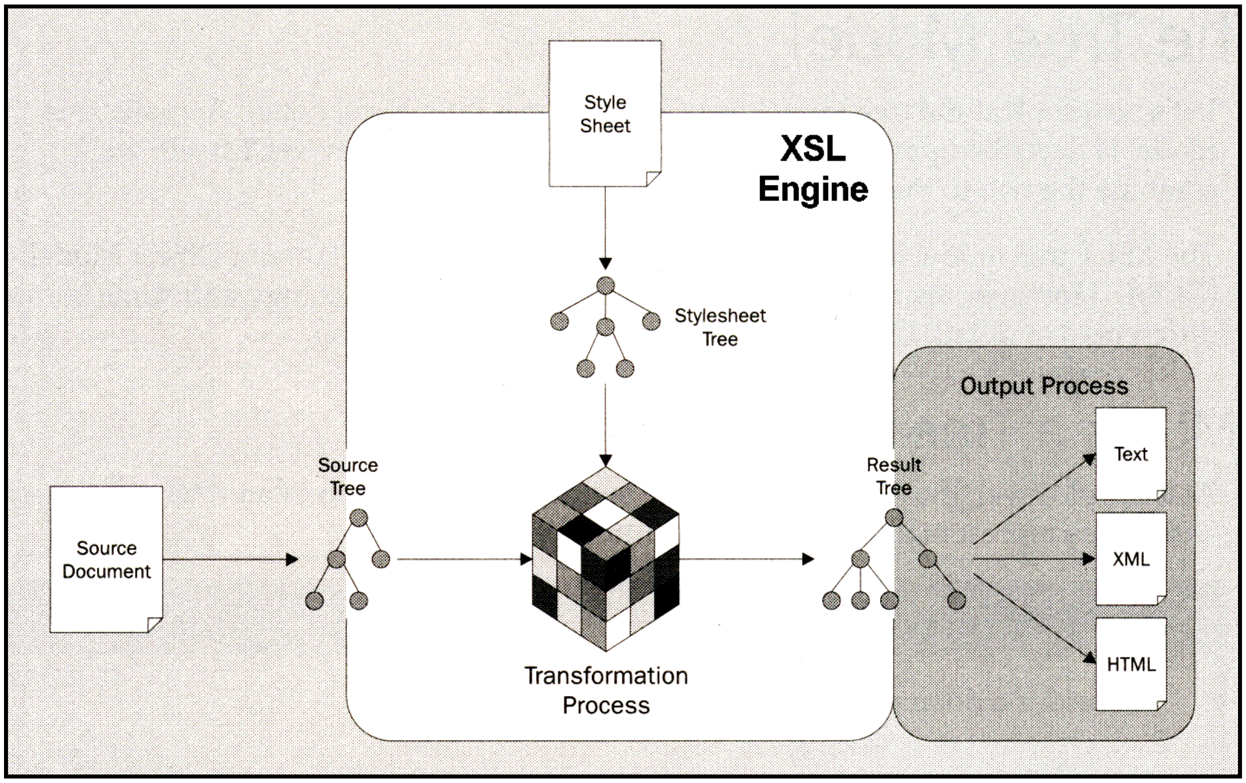


FIG. 8. **THE XSL PROCESSING MODEL (KAY, 2000, P. 49).**

XSL APPLICATION INFRASTRUCTURE

<u>The Minimal XSL Document</u>

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 xsl 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 which uses the most recent XSL namespace (as of April 2002) must contain the following three lines:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl" version="1.0">
    ...
</xsl:stylesheet>
```

XSL Templates

The abstract stylesheet nodes referred to on the previous page 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. 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.

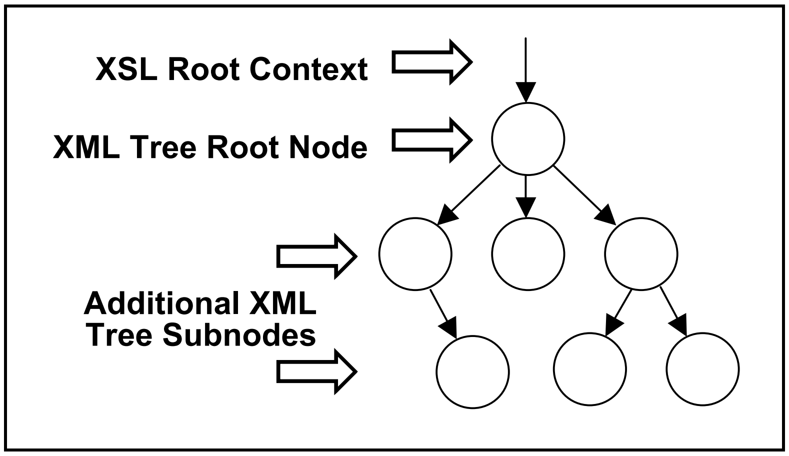


Fig. 9. The XSL Root Context vs. the XML Root Node.

Listing 1. File hello.xsl.

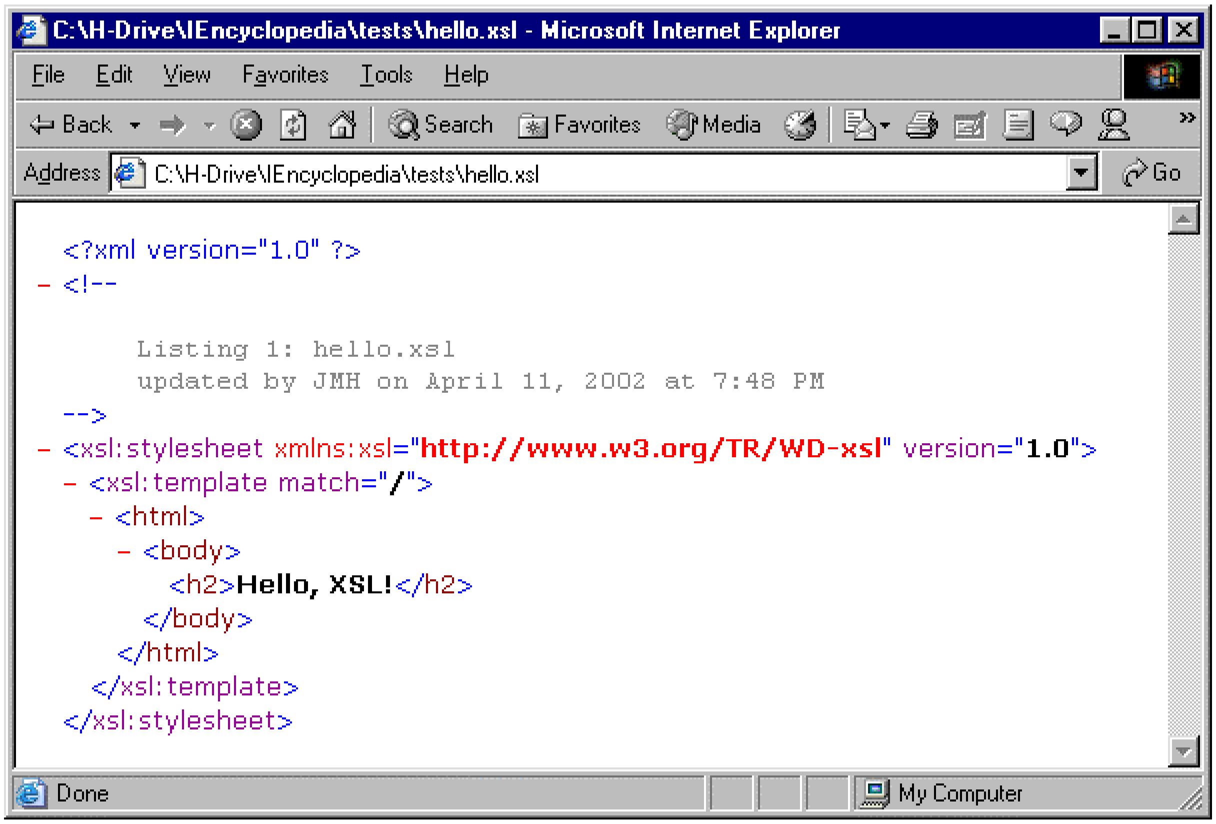
```
<?xml version="1.0" ?>
<!--
 hello.xsl - minimal XSL file
 updated by JMH on April 11, 2002 at 7:48 PM
-->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
         version="1.0">

```

```
<xsl:template match="/">
  <html>
    <body>
      <h2>
        Hello, XSL!
      </h2>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

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.


C:\H-Drive\IEncyclopedia\tests\hello.xsl - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\H-Drive\IEncyclopedia\tests\hello.xsl
Go
Done
My Computer

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

      Listing 1: hello.xsl
      updated by JMH on April 11, 2002 at 7:48 PM
  -->
- <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl" version="1.0">
  - <xsl:template match="/">
    - <html>
      - <body>
          <h2>Hello, XSL!</h2>
        </body>
      </html>
    </xsl:template>
  </xsl:stylesheet>
```

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

Choosing an XSL Engine

When it comes to choosing an XSL engine, there are many choices. We will follow the lead of most other books on this subject by using the XSL engine built into Internet Explorer (Microsoft, 2002), because that makes it easy to see our results. The XSL engine in Internet Explorer Version 6.0 conforms quite well to the latest World Wide Web XSLT Recommendation, but the one in previous versions of Internet Explorer does not. (There are plug-ins you can download from Microsoft to make those previous versions more W3C compliant.)

However, one should realize 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 (April 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'll work with the Internet Explorer engine, but the minimal Java Server Page code needed to apply an XSL file to an XML file using the Macromedia JRun Java Server (Macromedia, 2002) and the Apache Xalan-Java XSL engine is provided as an appendix.

<u>Applying an XSL File to an XML File</u>

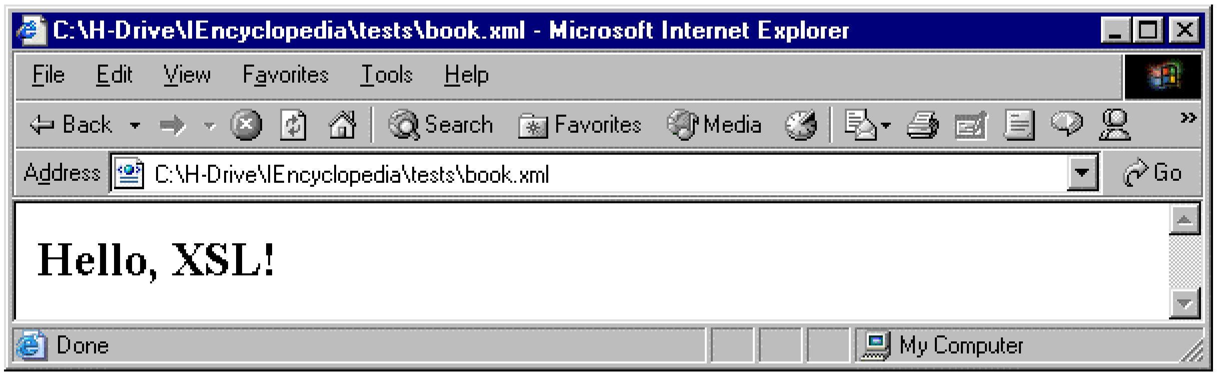
To link an XSL file to an XML file, one can include a processing instruction in the XML file that specifies the relative path to the XSL file to be applied:

```
<?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 6.0.2600 generates the output shown in Figure 11.

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

```
<?xml version="1.0"?>
<!--
  hello.xml - minimal version including xml-stylesheet processing
      instruction
  updated by JMH on April 11, 2002 at 09:10 PM
-->
<?xml-stylesheet type="text/xsl" href="hello.xsl" ?>
<hello>
</hello>
```


C:\H-Drive\IEncyclopedia\tests\book.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\H-Drive\IEncyclopedia\tests\book.xml
Go
Done
My Computer

Hello, XSL!

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

Inserting the xml-stylesheet processing instruction in an XML file as done 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 on 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.)

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

```
<html>
<!--
 Minimum Code for Applying an XSL file to an XML file on the
     Client Side using Internet Explorer Version 6.0.2600.0000
 updated by JMH on April 13, 2002 at 1:59 PM
-->
<body>
 <script type="text/javascript">
  var xmlDoc = new ActiveXObject( "Microsoft.XMLDOM" ) ;
   xmlDoc.async = false ;         // disable multithreading
   xmlDoc.load( "hello.xml" ) ;

   var xslStyleSheet = new ActiveXObject( "Microsoft.XMLDOM" ) ;
   xslStyleSheet.async = false ;   // disable multithreading
   xslStyleSheet.load( "hello.xsl" ) ;

   document.write( xmlDoc.transformNode( xslStyleSheet ) ) ;
  </script>
</body>
</html>
```

EXTRACTING XML DATA

We're 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. Of course, entire books have been written about XSL syntax and XPath, the language used to express the crucial *patterns* that appear in the match and test attributes of XSL elements (see, for example, the *XSLT Programmer's Reference* by Michael Kay, 2000). The intent of this article is considerably less ambitious. It strives only to give a feel for the types of things one can do with some of the basic XSL elements, how XSL patterns are 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 five of the chapters in this book 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. The transformations in the remainder of this article were all 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.

```xml
<?xml version="1.0"?>
<!--  book.xml - selected chapters and their authors
  updated by JMH on April 11, 2002 at 7:48 PM  -->
<book>
  <chapter title="Cascading Stylesheets (CSS)">
    <author last="Lawley" first="Elizabeth" middle="Lane" />
    <title>Assistant Professor</title>
    <affiliation>Rochester Institute of Technology
      <department>Information Technology</department>
      <e-mail>ell@mail.rit.edu</e-mail>
    </affiliation>
  </chapter>
  <chapter title="Java Server Pages (JSP)">
    <author last="Pratter" first="Frederick" />
    <title>Adjunct Instructor</title>
    <affiliation>University of Montana
      <department>Information Technology</department>
      <e-mail>pratter@cs.umt.edu</e-mail>
    </affiliation>
  </chapter>
  <chapter title="JavaScript">
    <author last="Roussos" first="Constantine" />
    <title>Professor</title>
    <affiliation>Lynchburg College
      <department>Computer Science</department>
      <e-mail>roussos@lynchburg.edu</e-mail>
    </affiliation>
  </chapter>
  <chapter title="Extensible Stylesheet Language (XSL/XSLT)">
    <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>
  <chapter title="Extensible Markup Language (XML)">
    <author last="Ulmer" first="John" />
    <title>Assistant Professor</title>
    <affiliation>Purdue University
      <department>Computer and Information Systems Technology</department>
      <e-mail>jjulmer@tech.purdue.edu</e-mail>
    </affiliation>
  </chapter>
</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*. 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 into the document, we refer to the nodes in the order in which they appear in the XML source tree, separating successive tree levels with slashes (/). This is basic XPath syntax. The pattern is easiest read 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 great-grandparent.

(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

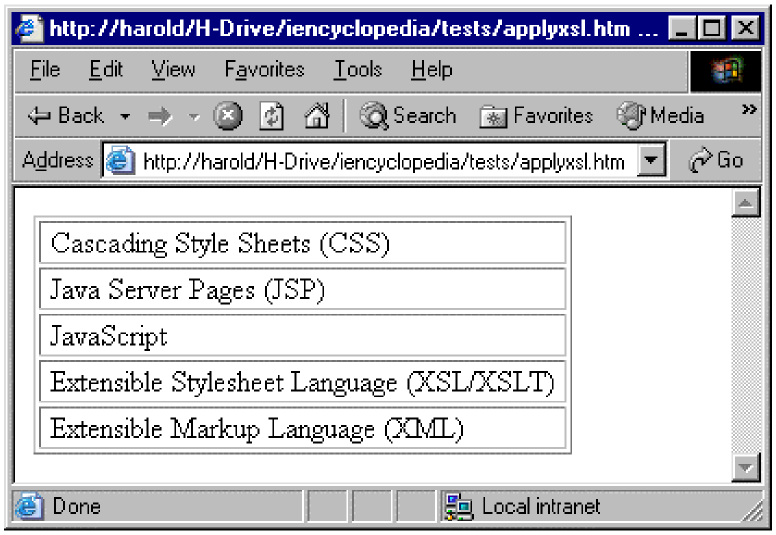
Iteration One way to extract sets of data is to use the xsl:for-each instruction. Like the xsl:value-of instruction, xsl:for-each has a select attribute, but this time the attribute is interpreted as a *node set expression* which selects all the XML data items that match its XPath expression. The xsl:for-each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

Consider the code in Listing 5. The xsl:for-each instruction appears at line 12, 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 results in 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 at 09:20 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:for-each select="book/chapter">
```

```
13          <tr>
14            <td>&#160;<xsl:value-of select="@title" />&#160;</td>
15          </tr>
16        </xsl:for-each>
17      </table>
18    </body>
19  </html>
20 </xsl:template>
21
22 </xsl:stylesheet>
```


http://harold/H-Drive/iencyclopedia/tests/applyxsl.htm ...
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://harold/H-Drive/iencyclopedia/tests/applyxsl.htm
Go
Done
Local intranet

Cascading Style Sheets (CSS)
Java Server Pages (JSP)
JavaScript
Extensible Stylesheet Language (XSL/XSLT)
Extensible Markup Language (XML)

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 (Listing 5 line 14) 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 which 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 as follows:

```
<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 following structure will be processed:

```
<chapter title="Extensible Stylesheet Language (XSL/XSLT)">
  <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="Basic XSL Constructs" />
</chapter>
```

<u>Filtering</u> 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. 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**.)

<u>Sorting Transformations</u>

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 takes 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.

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

Listing 6. Sorting data.

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

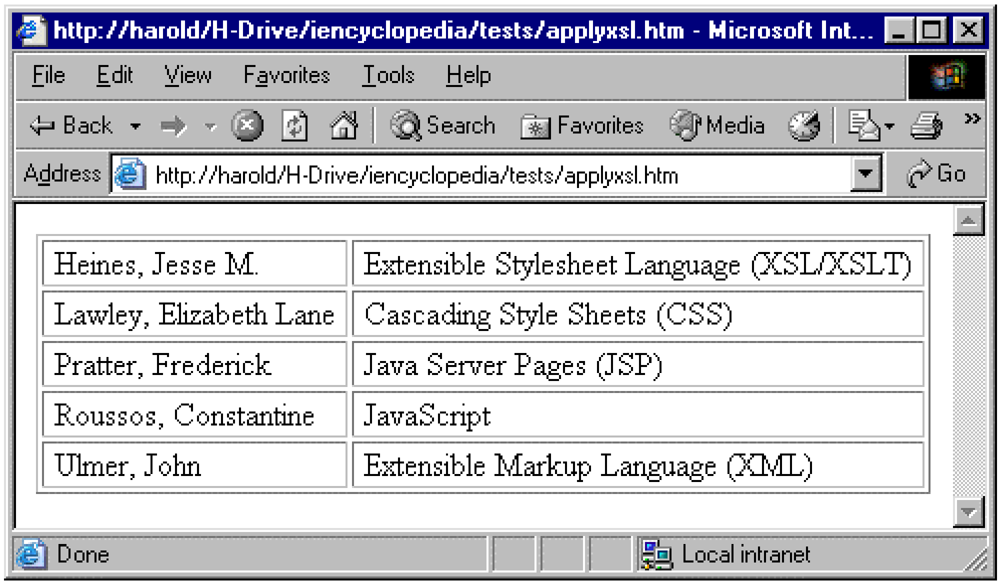
```
  book3.xsl
  updated by JMH on April 15, 2002 at 11:49 AM
-->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template match="/">
  <html>
    <body>
      <table border="1">
        <xsl:apply-templates select="book/chapter">
          <xsl:sort select="author/@last" data-type="text" order="ascending" />
          <xsl:sort select="author/@first" data-type="text" order="ascending" />
        </xsl:apply-templates>
      </table>
    </body>
  </html>
</xsl:template>

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

<xsl:template match="author">
  <td nowrap="">
    &#160;<xsl:value-of select="@last" />,
    <xsl:value-of select="@first" />&#32;
    <xsl:value-of select="@middle" />&#160;
  </td>
</xsl:template>

</xsl:stylesheet>
```


http://harold/H-Drive/iencyclopedia/tests/applyxsl.htm - Microsoft Int...
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://harold/H-Drive/iencyclopedia/tests/applyxsl.htm Go
Done
Local intranet

Heines, Jesse M.	Extensible Stylesheet Language (XSL/XSLT)
Lawley, Elizabeth Lane	Cascading Style Sheets (CSS)
Pratter, Frederick	Java Server Pages (JSP)
Roussos, Constantine	JavaScript
Ulmer, John	Extensible Markup Language (XML)

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 line 28-34.

(c) Line 29: Remember 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 the empty string: <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 &#160;, the XSL equivalent of &nbsp;, to get the extra aesthetically pleasing spaces before and after the text in each cell of our table. But if we use &#160; at the end of line 31, we get two spaces instead of one. Thus we have used &#32;, 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 over 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 which 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 one 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 example at the end of this article.

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 languages 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. The basic format of this instruction 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>
```

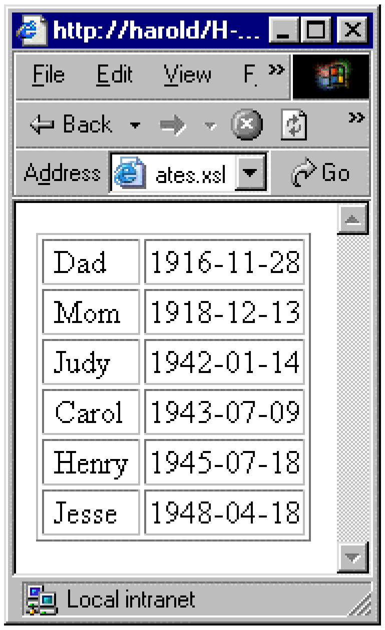
As you surely suspect by now, the Boolean expressions include XPath expressions, which can, in turn, include Boolean operators and numerous functions. The following sample xsl:choose construct outputs **Dear Ms. (*last name*):** if the current context node's first attribute contains the string "Elizabeth," 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 XML COMPATIBLE

To put everything together that we've seen, 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 (line 8), in subelements (lines 11-15), or as text (line 18). (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.) 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. Comments on specific XSL techniques follow the listings.



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.

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

```
<?xml version="1.0"?>
<!--
 birthdates.xml
 updated by JMH on April 15, 2002 at 7:39 PM
-->
<family>
 <person id="Dad">
  <birthdate month="November" day="28" year="1916" />
 </person>
 <person id="Mom">
  <birthdate>
   <month>December</month>
     <day>13</day>
     <year>1918</year>
  </birthdate>
 </person>
 <person id="Judy">
  <birthdate>1/14/42</birthdate>
 </person>
 <person id="Carol">
  <birthdate>
     <month>July</month>
     <day>9</day>
     <year>1943</year>
  </birthdate>
 </person>
```

```
  <person id="Henry">
    <birthdate month="July" day="18" year="1945" />
  </person>
  <person id="Jesse">
    <birthdate>4/18/48</birthdate>
  </person>
</family>
```

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

```
<?xml version="1.0" ?>
<!--
  birthdates.xsl
  updated by JMH on April 15, 2002 at 7:40 PM
-->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template match="/">
  <html>
    <body>
      <table border="1">
        <xsl:apply-templates select="family/person" />
      </table>
    </body>
  </html>
</xsl:template>

<xsl:template match="person">
  <tr>
    <td>&#160;<xsl:value-of select="@id" />&#160;</td>
    <td><xsl:apply-templates select="birthdate" /></td>
  </tr>
</xsl:template>

<xsl:template match="birthdate">
  <xsl:choose>

    <!-- handle case where date is in attributes -->
    <xsl:when test="@month">
      <xsl:value-of select="@year" />-<null />
      <xsl:call-template name="month">
        <xsl:with-param name="monthName" select="@month" />
      </xsl:call-template>-<null />
      <xsl:if test="@day &lt; 10">0</xsl:if>
      <xsl:value-of select="@day" />
    </xsl:when>

    <!-- handle case where date is in subelements -->
    <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       19<xsl:value-of
 84             select="substring-after(
 85                       substring-after( ./text(), '/' ), '/' )" />-<null />
 86       <xsl:if test="substring-before( ./text(), '/' ) &lt; 10">0</xsl:if>
 87       <xsl:value-of select="substring-before( ./text(), '/' )" />-<null />
 88       <xsl:if test="substring-before(
 89                       substring-after( ./text(), '/' ), '/' ) &lt; 10">0</xsl:if>
 90       <xsl:value-of
 91             select="substring-before( substring-after( ./text(), '/' ), '/' )" />
 92     </xsl:when>
 93
 94     <!-- handle default case -->
 95     <xsl:otherwise>
 96       unknown
 97     </xsl:otherwise>
 98
 99   </xsl:choose>
100 </xsl:template>
101
102 <xsl:template name="month">
103   <xsl:param name="monthName" />
104   <xsl:choose>
105     <xsl:when test="$monthName='January'">01</xsl:when>
106     <xsl:when test="$monthName='February'">02</xsl:when>
107     <xsl:when test="$monthName='March'">03</xsl:when>
108     <xsl:when test="$monthName='April'">04</xsl:when>
109     <xsl:when test="$monthName='May'">05</xsl:when>
110     <xsl:when test="$monthName='June'">06</xsl:when>
111     <xsl:when test="$monthName='July'">07</xsl:when>
112     <xsl:when test="$monthName='August'">08</xsl:when>
113     <xsl:when test="$monthName='September'">09</xsl:when>
114     <xsl:when test="$monthName='October'">10</xsl:when>
115     <xsl:when test="$monthName='November'">11</xsl:when>
116     <xsl:when test="$monthName='December'">12</xsl:when>
117   </xsl:choose>
118 </xsl:template>
120 </xsl:stylesheet>
```

The heart of this transformation is in the template that begins on line 58. This template is executed in the context 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 102 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 D/M/YY 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.

The template that begins at line 102 is essentially one large xsl:choose instruction. Line 103 accepts the monthName parameter passed to this template 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.

CONCLUSION: WHERE TO GO FROM HERE

Hundreds — if not thousands — of pages would of course be needed to cover all the features of XSL, XSLT, and XPath, 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 software to use XSL on your systems.

APPENDIX

Appendix Listing. Minimal Java Server Page code to apply an XSL file to an XML file using the Macromedia JRun Java Server and the Apache Xalan-Java XSL engine.

```
<html>
<!--
  Minimum Code for Applying an XSL file to an XML file on the Server Side
  adapted from xalan-j_2_3_1\samples\SimpleTransform\SimpleTransform.java
  download Xalan-Java from http://xml.apache.org/xalan-j free of charge
  updated by JMH on April 13, 2002 at 1:54 PM
-->
<head>
  <%! // Imported TraX classes %>
   <%@ page import="java.io.*, java.util.*, org.w3c.dom.*, javax.swing.*" %>
   <%@ page import="org.apache.xerces.parsers.DOMParser" %>
   <%@ page import="org.apache.xerces.parsers.SAXParser, org.xml.sax.*" %>
   <%@ page import="javax.xml.transform.TransformerFactory" %>
   <%@ page import="javax.xml.transform.Transformer" %>
   <%@ page import="javax.xml.transform.stream.StreamSource" %>
   <%@ page import="javax.xml.transform.stream.StreamResult" %>
   <%@ page import="javax.xml.transform.TransformerException" %>
   <%@ page import="javax.xml.transform.TransformerConfigurationException" %>

   <%! // Imported Java classes %>
   <%@ page import="java.io.FileOutputStream" %>
   <%@ page import="java.io.FileNotFoundException" %>
   <%@ page import="java.io.IOException" %>

 <%!
   /** Get the full server-side path for the running JSP.
    *  <p>This function is intended to compensate for what I believe is a bug in
    *  the UNIX/Linus implementation of the getRealPath() function in that it
    *  returns the path to the server rather than to the requested file.  This
    *  function performs correctly on all of the Windows, UNIX, and Linux systems
    *  to which I have access.</p>
    *  @param  request  the request object representing this page
    *  @return String containing the full server-side path or null if it fails
    */
   String getRequestPath( HttpServletRequest req )
   {
     /** full path to this JSP on the server side */
     String strFullFilePath = null ;

     /** name of the operating system running on the server side */
     String osName = System.getProperty( "os.name" ) ;

     // call the appropriate functions to get the real path to this JSP
     // depending on the server's operating system name

     // for Windows-based systems, try getRequestURI first, then the work-around
     // application.getRealPath( "." ) + req.getServletPath()
     if ( osName.equalsIgnoreCase( "Windows 2000" ) ||
          osName.equalsIgnoreCase( "Windows NT" )   ||
          osName.equalsIgnoreCase( "Windows 98" )   ||
          osName.equalsIgnoreCase( "Windows 95" ) )
     {
```

```
    try {
      strFullFilePath = application.getRealPath( req.getRequestURI() ) ;
      FileInputStream fis = new FileInputStream( strFullFilePath ) ;
    } catch ( FileNotFoundException fnfe1 ) {
      try {
        strFullFilePath = application.getRealPath( "." ) + req.getServletPath() ;
        FileInputStream fis = new FileInputStream( strFullFilePath ) ;
      } catch ( FileNotFoundException fnfe2 ) {
        return null ;
      }
    }
    return strFullFilePath.substring( 0, strFullFilePath.lastIndexOf( "\\" ) + 1 ) ;
  }

  // for UNIX-based systems, try the work-around application.getRealPath( "." ) +
  // req.getServletPath() first, then try getRequestURI
  else if ( osName.equalsIgnoreCase( "UNIX" )  ||
            osName.equalsIgnoreCase( "Linux" ) )
  {
    try {
      strFullFilePath = application.getRealPath( "." ) + req.getServletPath() ;
      FileInputStream fis = new FileInputStream( strFullFilePath ) ;
    } catch ( FileNotFoundException fnfe1 ) {
      try {
        strFullFilePath = application.getRealPath( req.getRequestURI() ) ;
        FileInputStream fis = new FileInputStream( strFullFilePath ) ;
      } catch ( FileNotFoundException fnfe2 ) {
        return null ;
      }
    }
    return strFullFilePath.substring( 0, strFullFilePath.lastIndexOf( "/" ) + 1 ) ;
  }

  // if neither works, unfortunately we have to return null
  return null ;
}


/** Apply an XSL file to an XML file and return the results as a String.
 *  @param  strXMLfile  the XML source file to process - String containing
 *                        full path on server
 *  @param  strXSLfile  the XSL source file to apply to the given XML file -
 *                        String containing full path on server
 *  @param  out         JspWriter for displaying helpful info to screen
 *  @return String containing the output of the transformation
 */
String ApplyXSL( String strXMLfile, String strXSLfile, JspWriter out )
 {
   // StringWriter is a child of java.io.Writer and can therefore be
  // used as an argument to a StreamResult constructor, which is
  // required by Transformer.transform().
  StringWriter swResult = new StringWriter() ;

  try {
   // Use the static TransformerFactory.newInstance() method to instantiate
   // a TransformerFactory. The javax.xml.transform.TransformerFactory
   // system property setting determines the actual class to instantiate -
   // org.apache.xalan.transformer.TransformerImpl.
   TransformerFactory tFactory = TransformerFactory.newInstance();

   // Use the TransformerFactory to instantiate a Transformer that will work
   // with the stylesheet you specify. This method call also processes the
   // stylesheet into a compiled Templates object.
   Transformer transformer =
     tFactory.newTransformer( new StreamSource( strXSLfile ) ) ;

   // Use the Transformer to apply the associated Templates object to an XML
   // document (foo.xml) and write the output to a file (foo.out).
   transformer.transform( new StreamSource( strXMLfile ),
              new StreamResult( swResult ) ) ;
   // return result
   return swResult.toString() ;
  } catch ( TransformerConfigurationException tfce ) {
   return tfce.toString() ;
  } catch ( TransformerException tfe ) {
   return tfe.toString() ;
  }
 }
%>
</head>

<body>
<%
 String strXMLfilename = "hello.xml" ;  // replace with your XML file name
 String strXSLfilename = "hello.xsl" ;  // replace with your XSL file name

 String strResult = ApplyXSL( getRequestPath( request ) + strXMLfilename,
               getRequestPath( request ) + strXSLfilename,
               out ) ;
 out.println( strResult ) ;
%>
</body>
</html>
```

Bibliography

Apache Software Foundation (2002). Xalan-Java. Available at http://xml.apache.org/xalan-j (date of access: April 18, 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. Birmingham, UK: Wrox Press.

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

Macromedia Corporation (2002). Macromedia JRun Server 3.1 documentation. Available at http://www.macromedia.com/software/jrun (date of access: April 18, 2002).

McNamara, Bruce, and Smaragdakis, Yannis (2000). Functional programming in C++. 2000 International Conference on Functional Programming, Montreal, Canada. Available at http://www.cc.gatech.edu/~yannis/fc++/fc++.main.pdf (date of access: April 18, 2002).

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/htm/xsl_ref_overview_1vad.asp (date of access: April 18, 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: April 18, 2002).

World Wide Web Consortium (1997). Date and time formats. "Note" available at http://www.w3.org/TR/NOTE-datetime (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: April 18, 2002).

World Wide Web Consortium (2001). XSL Transformations (XSLT). "Working Draft" available at http://www.w3.org/TR/xslt20 (date of access: April 18, 2002).