# Introduction to XML

410-XML-3.2.0

# Table of Contents

# Introduction to XML

## Overview

XML (e**X**tensible **M**arkup **L**anguage) has been touted as the language which will replace all other Web tools, making HTML, databases, and much more obsolete. Although it will not do that, it has huge advantages over current Web technologies in flexibility, portability, and data awareness. The key to these abilities is a designer's freedom to design XML tags that accurately describe the structure and content of his or her data.

## The Development of XML

XML, like HTML, grew out of the SGML language. Like HTML, it is simplified so that users have a predefined set of rules to follow. Like SGML, designers write their own tags and tag structure to accurately define their data. And unique to XML, the World Wide Web Consortium (*http://www.w3.org*) has developed rigorous rules to define well-formed XML, ensuring that XML data will be readily available to any application that needs it.

Although SGML has been in existence since 1970, and HTML since 1990, XML was not proposed until 1996. It was designed to attain the following goals (*http://www.w3.org/TR/WD-xml-961114.html#sec1.1)*:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness is of minimal importance.

Two of these features deserve special attention, as they have had a major impact on the way XML Web documents are prepared.

- First, that XML documents be able to support a wide variety of applications. This indicates that XML should not include display information, as that information will necessarily be specific to a particular platform. A well-written XML document will be readable through the Web, by cell phones, by databases, and by many other platforms.

- Second, that XML should be straightforwardly usable over the Internet. This specification, in conjunction with that XML be required to be platform-independent, mandates that XML for the Web be transmitted along with other "helper" files which will specify how browsers should treat the XML data.

## Exercise 0: Downloading and Extracting the Files for today's class

You will each have your own copy of the files for today's class. Working with your instructor, please do the following:

- Download the class files from *http://www.westlake.com/classes/materials/xml.exe* to your **desktop**.
- Extract the self-extracting file by double-clicking on the **xml.exe** icon. You should extract the files to your desktop.
- You'll see an **XML** folder appear on your desktop. Open it up and see what's inside.

Note that you have a folder named **Exercise Files**, and a folder named **Demos**. Explore around a bit to familiarize yourself.

## An XML Example

Before we go further into the nature of XML, let's look at a sample document. Doing so will allow us to discuss the specifics of XML syntax more knowledgeably. This example is in **demos > syntax > simple_example.xml**:

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

<client>
   <name>
      <firstname>John</firstname>
      <lastname>Doe</lastname>
   </name>
   <address type="home">
      <street_address>123 School St.</street_address>
      <city>Ithaca</city>
      <state>NY</state>
      <zip>14850</zip>
   </address>
   <age>26</age>
</client>
```

As you can see from the above example, XML describes the **logical** structure of data. XML tags (other than the initial language declaration) have no inherent meaning, but instead can be applied flexibly as the designer chooses. What those meanings are, and how they are translated into visual results, is determined by separate sections of the XML application. What you see above is commonly termed an **XML Datasheet**. To turn this into an XML application will require the inclusion of a couple of additional files: a Document Type Definition (DTD) or Schema, and a stylesheet, written in either Cascading Style Sheets (CSS) or Extensible Stylesheet Language (XSL). We will examine these components in upcoming sections.

## Why XML?

Even before you get started, it is worth thinking briefly about what sorts of applications find XML useful. We will see examples of these sorts of applications throughout the course. These are:

### When you need to output the same data in multiple formats

As you saw above, XML holds the data of a document without specifying how it is displayed. At first, this may seem like a detriment, and you do need some sort of stylesheet to transform the raw data into something appropriate for the end display mechanism (such as HTML for a browser, formatted text for a fax or mailing, and plain text for sending out as an email). Storing the document in XML means that you only have to write the data once, and if you rewrite your document, you can just re-apply different stylesheets and have the end results automatically generated. What's more, if you standardize your document types (such as press releases, for example) on a specific set of tags your stylesheets will be reusable without modification.

### When you need to exchange data between otherwise incompatible systems, applications, or organizations

For example, consider the case of a manufacturer. They routinely need to subcontract out to produce various parts for inclusion into their products. In the pre-XML world, they had to send out a document specifying what they needed in a bid to produce these products, and might easily receive the information back in various formats: paper documents, faxes, emails, attachments, and spreadsheets, at least. A person would have to then enter this information into some application so that the different bids could be compared. The added layer of data entry adds time, expense, and error to the process.

With XML the same manufacturer can send out their requirements in the form of a Document Type Definition (DTD) or Schema. Both of these specify a particular tag structure that is legal for a given application. Once a subcontractor receives the structure that they are expected to produce, they can tune their applications to produce XML as specified by the DTD.

### When you need to store structured data, but a database is not appropriate or necessary

For many applications, particularly applications where data is being generated in multiple locations with questionable connectivity, it can be difficult or undesirable to have team members connecting directly to the main database for a project. XML allows the opportunity to store structured information in a format that can be easily imported into a database, or evaluated without one.

## When you want multiple different views of information, presented dynamically

We will learn more about this capability later in the course. For now, consider the process you need to go through to re-sort a display in a traditional database driven application. Imagine that you wish to see a display of information sorted by last name rather than zip code. You have to follow the following process:

1. Browser makes a new HTTP request to the server
2. Server processes the request with a middleware application
3. Middleware application builds a new SQL query for the database
4. Middleware application creates a connection to the database, and passes it the SQL query
5. Database receives the query and retrieves the recordset
6. Database returns the recordset to the middleware application
7. Middleware application generates the HTML from the recordset, and sends it via HTTP to the browser
8. Browser displays the information

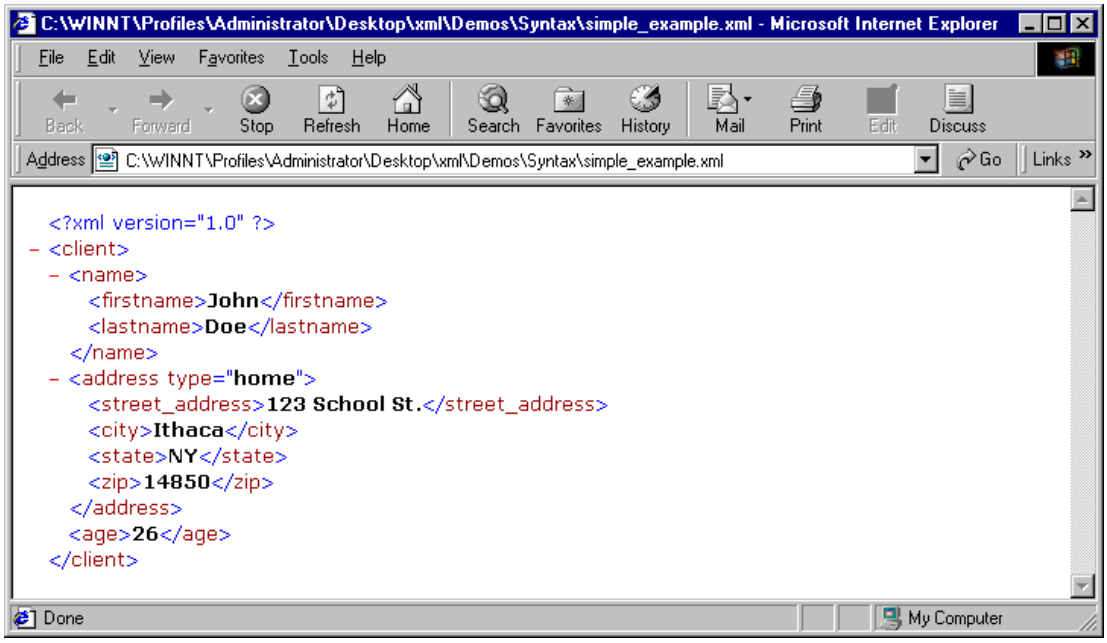
Whew! That's a lot of steps for one application, and requires your server and your database to do extra work for reformatting. However, if you can provide your data in XML form to an XML-capable browser, the process is simplified to:

1. Browser uses JavaScript to modify XSL stylesheet
2. JavaScript reapplies (now modified) XSL stylesheet to XML data
3. JavaScript places the generated HTML content into the page

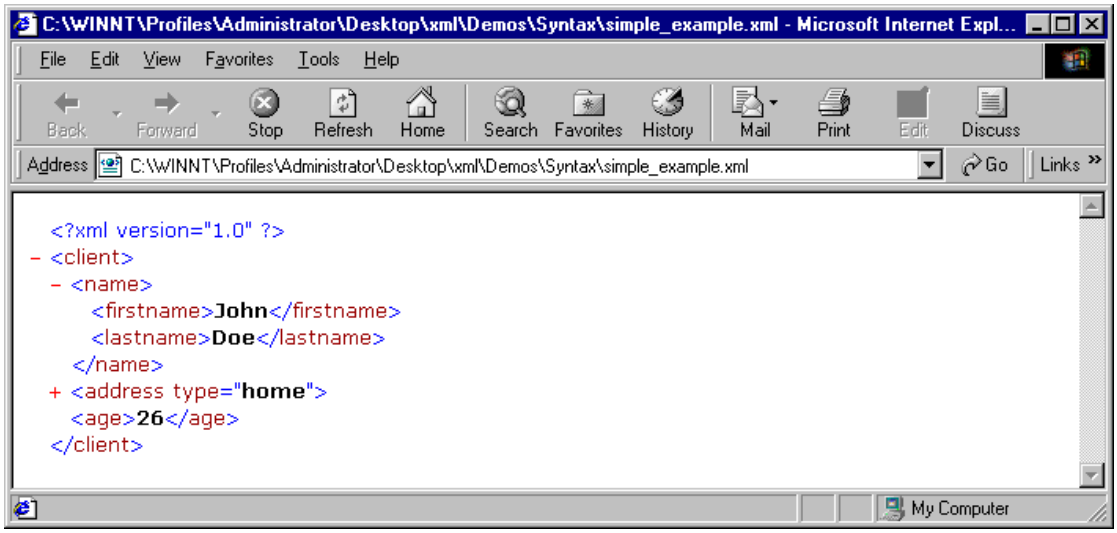
As you can see, not only is the process a much shorter one, but it also is entirely processed by the browser's processor. By taking advantage of efficiency savings like these, you can make applications that are extraordinarily responsive.

# XML and Browser Compatibility

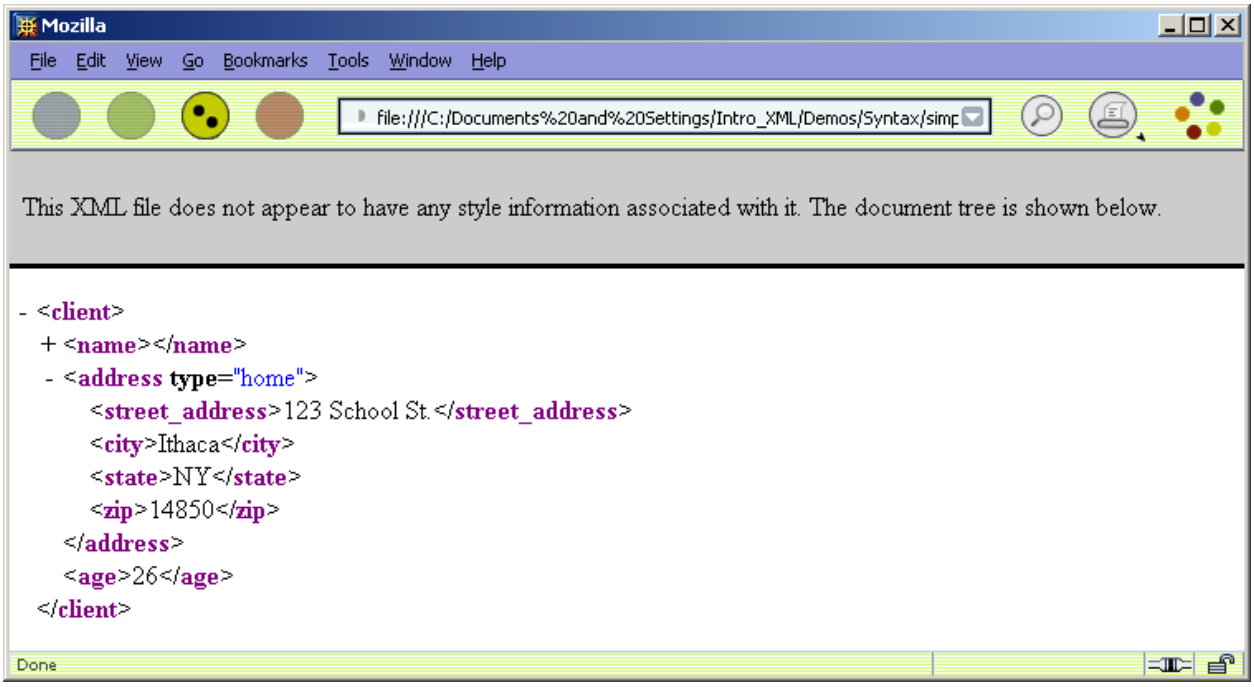
Over the past several years, several browsers have developed the capability to deal with XML documents. These include Microsoft Internet Explorer 5+, Netscape 6+, Mozilla 1+, and recent versions of Opera. Different versions of these browsers have different levels of support. To be sure of solid XML support, you (and those viewing your site) should be using IE 5.5 with Microsoft's 3.0 plugin ( a free download) installed, IE 6, Netscape 7, or Mozilla. IE5+ and Mozilla 1.2.1 + has implemented most of the standards proposed for XML by the World Wide Web consortium, and can act as stand-alone XML viewers. We will make use of this capability many times during this course. For example, the code you saw in the previous section, viewed in IE 6, would look as follows:



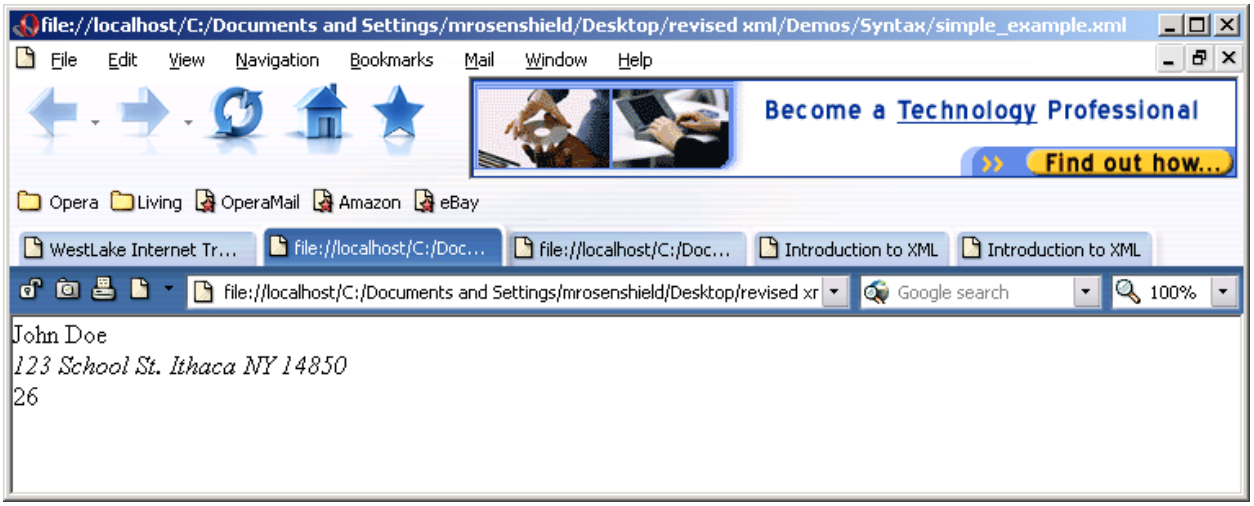

Note that Internet Explorer understands the heirarchichal structure of the tags, and actually allows you to expand or contract the display to show the depth of data you choose. For example, clicking on the small "**-**" next to the **address** tag produces the following display:



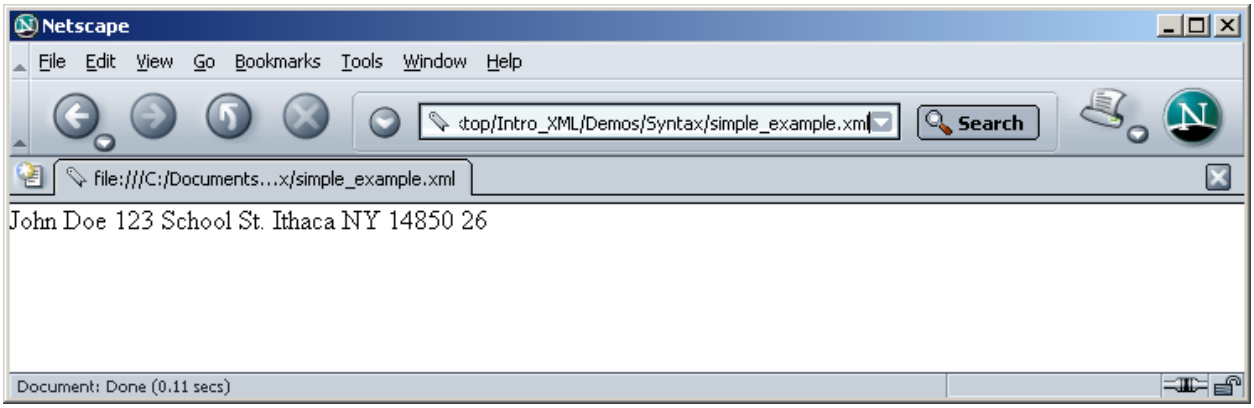

Mozilla has implemented a very similar display style:



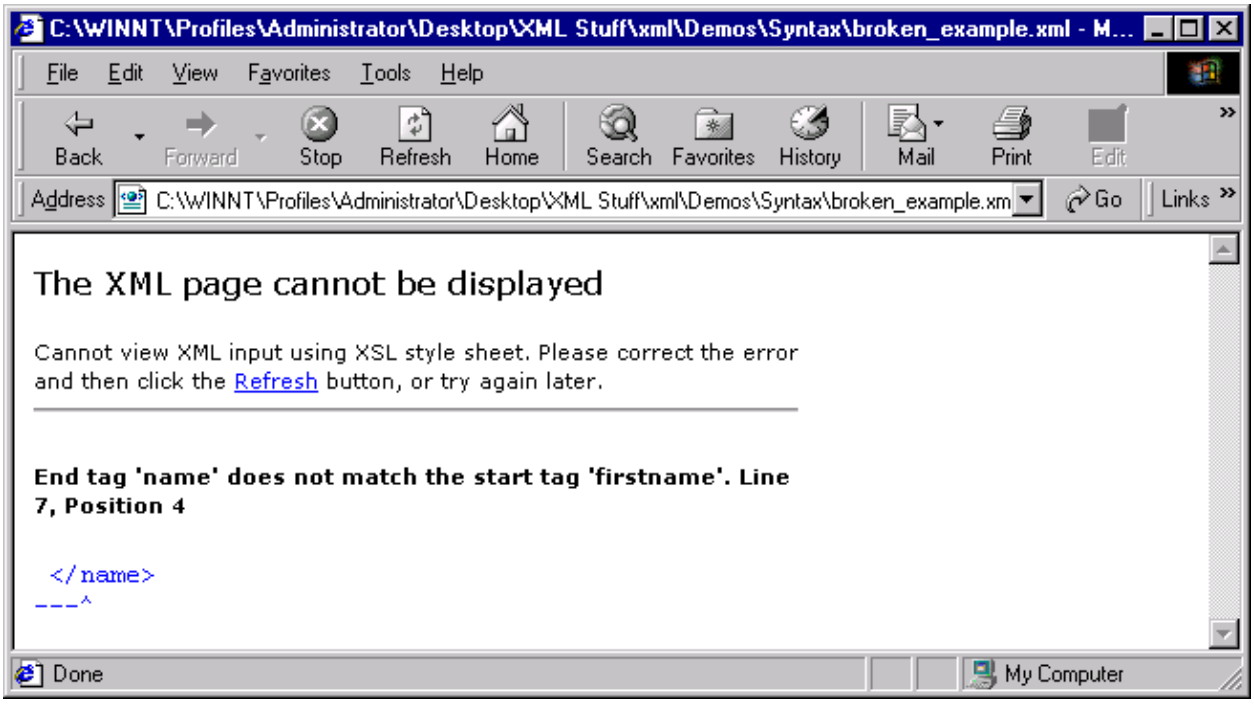

Other browsers do not have the same capability. Here is the same page in Opera 7:



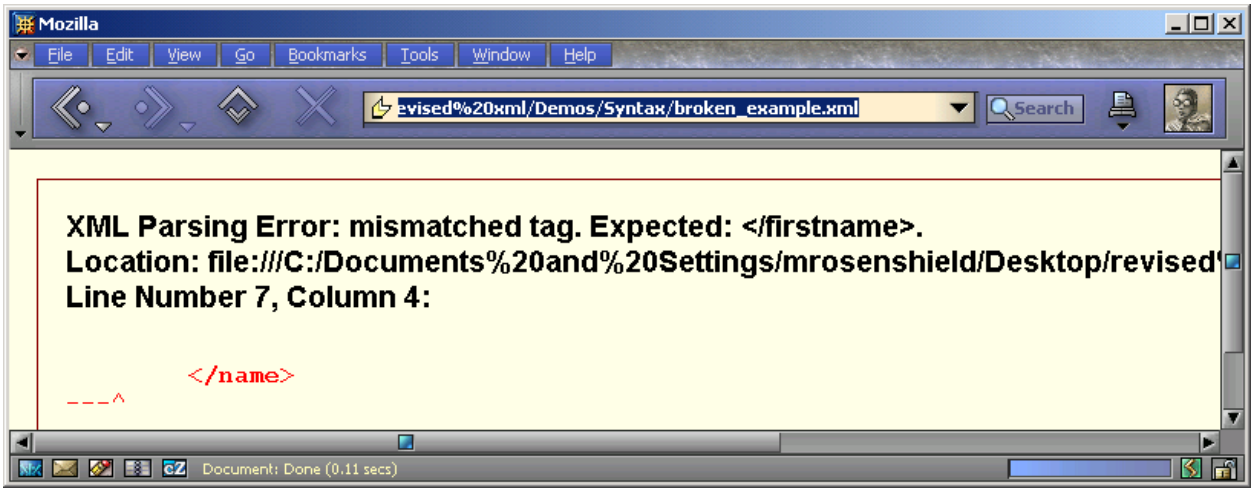

And Netscape 7.0:



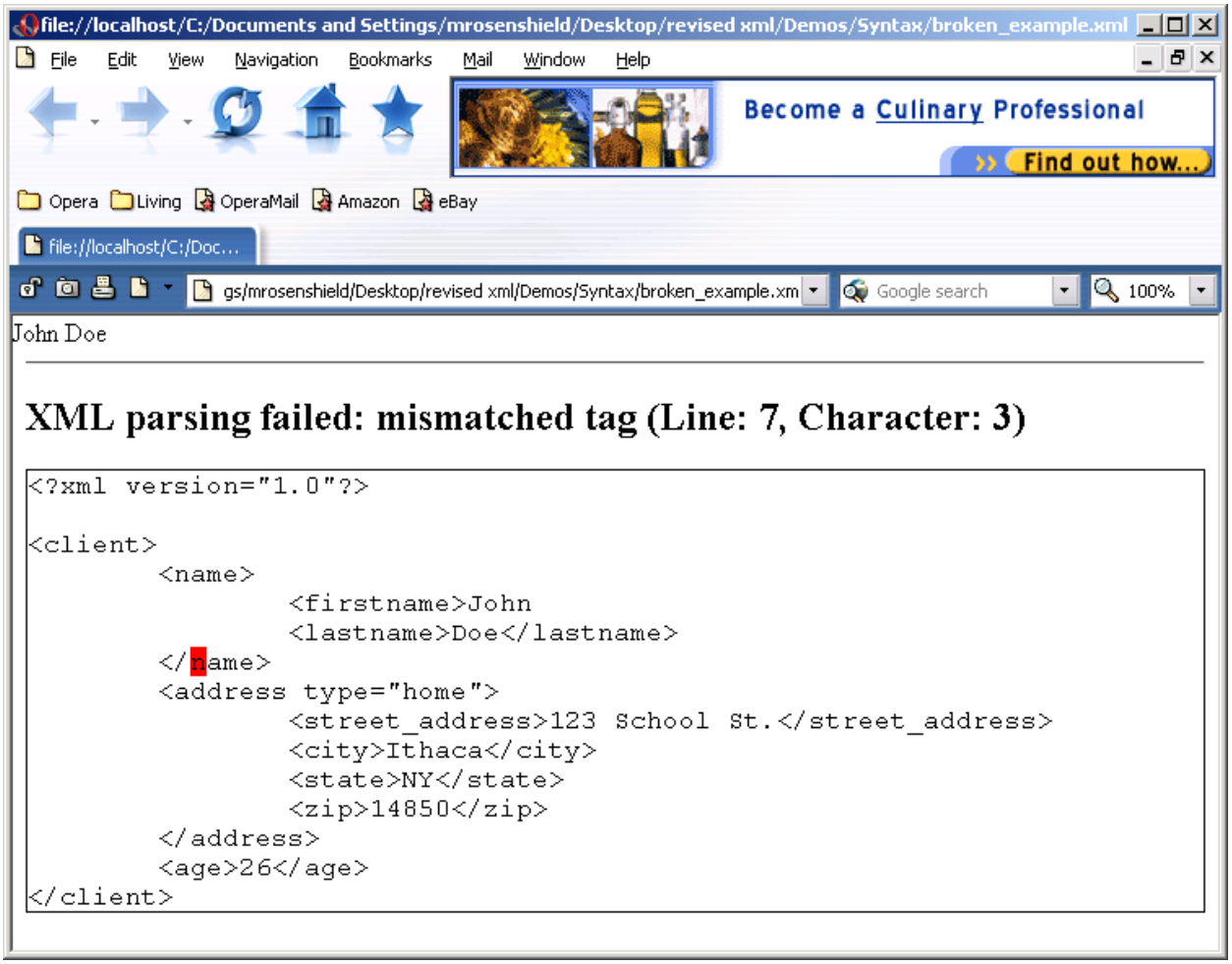

If Internet Explorer cannot parse the XML code (because it breaks some syntax rule or does not conform to a given document type) it produces an error message (see **demos > syntax > broken_example.xml**):




Here is the same page in Mozilla:




And Opera 7:




## Microsoft and XML Extensions

Microsoft has commited iteslf publicly to supporting the XML standards, as published by the W3C. In addition, they have been at the forefront of the **XML Working Group**, which proposes, evaluates, and publishes additions or changes to the XML standards. This effort has meant that Microsoft has been quite aggressive in adopting XML standards, and in producing tools with the capability of working with XML.

However, there areas where Microsoft's compliance differs from the published standards. The most important is the dynamic manipulation of XML data via the XML DOM. This is done on the server side with ColdFusion, ASP, JSP, Perl, or PHP (see WestLake's courses: *Integrating ASP and XML*, and *Integrating JSP and XML*) or on the client side with JavaScript. Before the W3C issued final standards on how this should be accomplished, and in an effort to provide developers with tools, Microsoft instituted a few extensions to the XML specifications. Most of these extensions have been presented to the World Wide Web Consortium for inclusion into the next draft of the XML specifications, although there is no guarantee that they will be accepted. Many of them are extremely useful and we will see some of them near the end of this course.

The vast majority of this course is standards compliant and platform-independent. Where we use anything that is browser-specific (particularly in the dynamic XSL transformation section at the very end of the course) we will note the applications as such, and discuss possible alternatives.

## Using MSXML

MSXML Microsoft's XML parser, schema validator, and XSLT processor bundled into one package. It's mainly used for client – side transformations, where the XML datasheet and the stylesheet are both delivered to the browser (such as Internet Explorer), then the browser carries out the transformations and displays the results.

If you currently have Internet Explorer 6+ installed on your machine, then you will have a version of MSXML3 (which you can install in replace mode to make it work with IE 5 and above). You cannot use MSXML4 to get automatic transformations with IE; you must use MSXML3. To detect which XSXML version you have installed you might want to visit Chris Bayes's site at http://www.bayes.co.uk./xml/index.xml?/xml/main.xml.

## Why Use a Browser to Display XML?

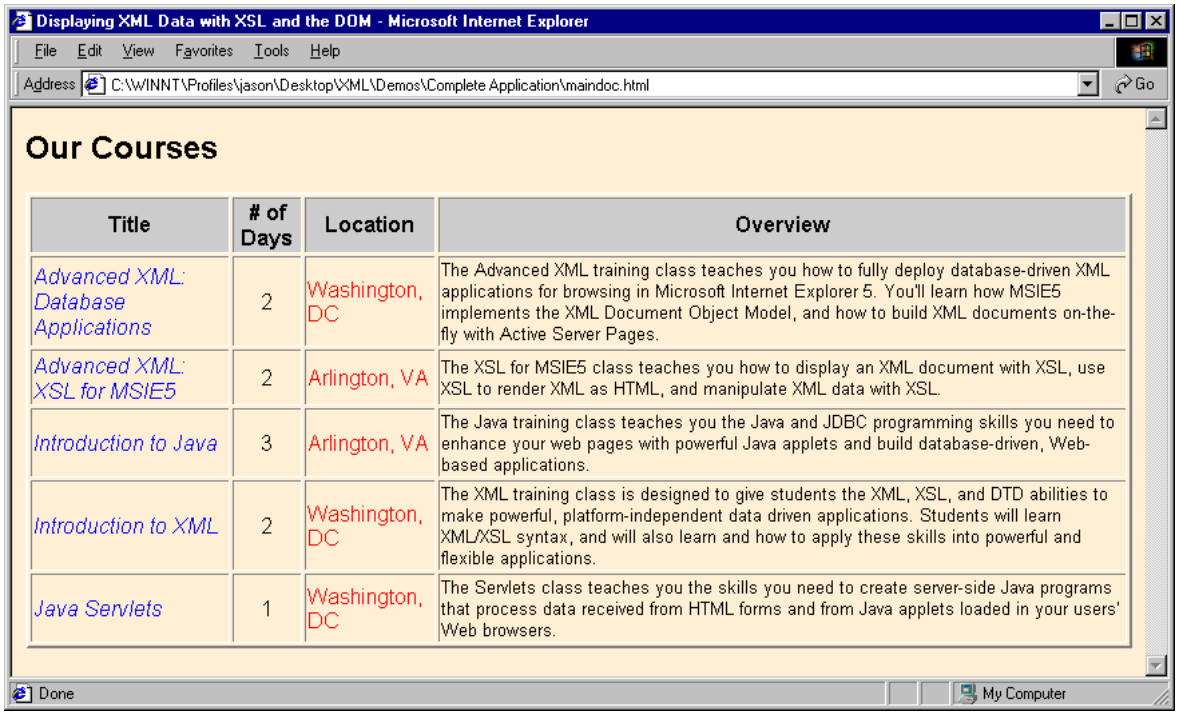
Even a brief glance at the previous "Why XML" section should make it clear that the majority of XML applications are run on a server. So, why do we teach this course using a browser?

1. Browers are free and widespread.
2. The support for XML in the Microsoft and Netscape/Mozilla browsers is very well developed, and provide error messages if you make a mistake in coding your XML files.
3. Professional developers writing applications that will eventually run on a server often use a browser as a lightweight environment for testing their applications because of its ease of use.

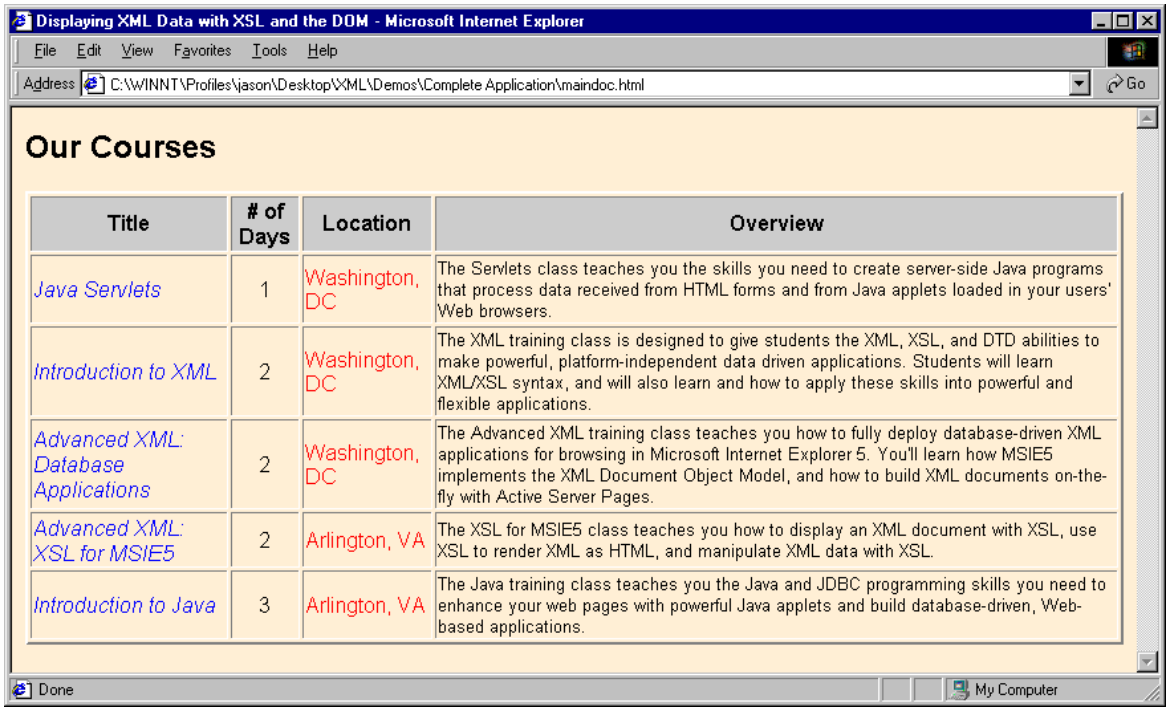
Essentially, a browser makes a perfect lightweight development environment for XML.

# Examining an XML Application

The easiest way to get a feel for the different components of an XML application is to take a look at a complete one. So, let's look at our first application. In your **XML** folder, open **Demos.** You will see a list of files, as well as a few other folders. Please open the folder **Complete Application**, and open the file **maindoc.html** in Internet Explorer 5+. You should see the following display:




At first glance, it looks like a normal tabular display of HTML data, with a little formatting and coloring thrown in. However, try clicking on the different columns. If you click on the "Location" column, the table resorts by location. If you pick the "# of Days" column, it will be resorted by class length:

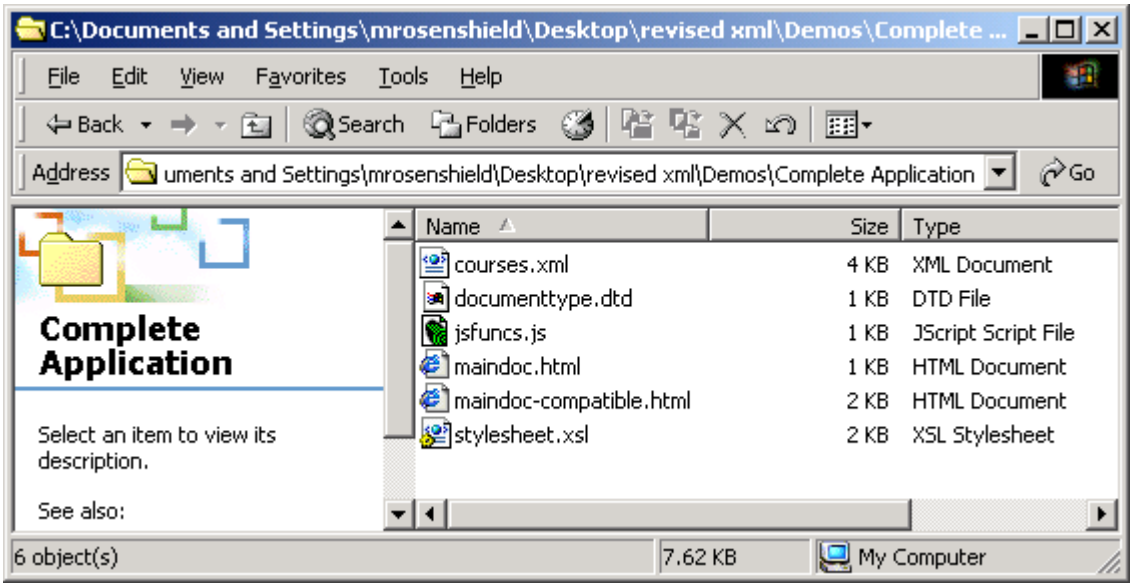



So, there's obviously something beyond a simple HTML display going on here,. If you view the source of **maindoc.html**, you will see further evidence:

```
<html>
<head>
   <title>Displaying XML Data with XSL and the DOM</title>
<xml id="courselist" src="courses.xml"></xml>
<xml id="coursedisplay" src="stylesheet.xsl"></xml>
<script language="JavaScript" src="jsfuncs.js"></script>
</head>

<body onLoad=
"document.getElementById('dataTarget').innerHTML =
courselist.transformNode(coursedisplay)"
bgcolor="papayawhip">

<div id="dataTarget"></div>

</body>
</html>
```

## The Components of an XML Application

As you can see, there are links to three external documents in the above code. These, along with the HTML, provide the main blocks of an XML application. We'll examine each of these briefly below, and then in more depth later. To begin, take a look at the **Complete Application** folder:




The components of a Web-based XML application are (generally) as follows:

- **An HTML File** (maindoc.html): The HTML file acts as a wrapper for calling the other components of an XML application, and can also set certain display defaults (such as the background color in the above example). It will also specify areas of the page (normally with ** ** and **** tags) where XML data is to be displayed. Maindoc.html uses Microsoft-specific tags, whereas the other html file – maindoc-compatible.html – has code that will work in IE, Netscape and Mozilla.

- **An XML File**: The XML file contains the data, organized hierarchically and structured internally. In IE (and only IE – this tag is a Microsoft extension) it can be loaded into the browser with the **<xml>** tag:

  ```
  <xml id="courselist" src="courses.xml"></xml>
  ```

  XML can also be dynamically generated by a server-side process such as ASP, ColdFusion, JSP, PHP, or Perl.

- **An XSL or CSS File:** You will need a stylesheet to format the data for display in the browser. There are two options, not mutually exclusive. **C**ascading **S**tyle **S**heets (CSS) can determine the display characteristics of data in a browser. So, you can

specify (for example) that your course names be **green**, your day numbers be **bold**, and everything be displayed in the font **T**ahoma.

However, you will often want to build more complex structures to display your data. These might include HTML tables, lists, borders, and more, in addition to text formatting. e**X**tensible **S**tylesheet **L**anguage (XSL) can generate HTML code, access CSS pages, and even generate other XML code! XSL files are actually XML code, with scripted extensions that give them significant power and flexibility. We will spend several exercises exploring the capabilities of XSL.

- **A DTD or Schema File:** Most XML applications require a **D**ocument **T**ype **D**efinition (DTD) or a Schema to be specified for the XML data. This DTD/Schema specifies the acceptable structure of the XML, such as the nested hierarchy, the attributes of tags, if any, and their acceptable values and data types. Both DTDs and Schemas can be applied externally, allowing several different data sheets to be validated against the same document specifics.

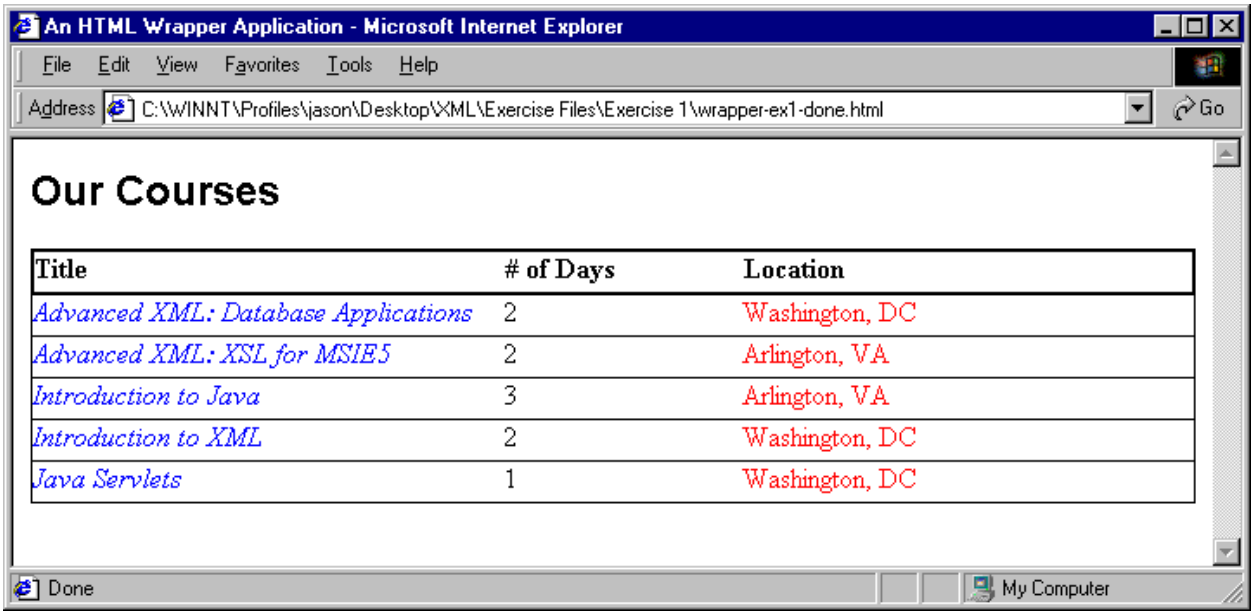
- **A JavaScript File:** (optional) Many of the most appealing features of XML (such as dynamic resorting, tree structure, and visibility changes) are accessed through XML/XSL interaction with JavaScript. It is often most convenient and efficient to write JavaScript functions as a separate **.js** file, and then make that file available to as many different applications as necessary. Of course, it is also possible to specify JavaScript in an internal `<SCRIPT>...</SCRIPT>` block, or, for simple applications, directly in-line.

We will spend significant time examining each of these components a little later in the course. As a first step, you will turn a collection of XML/XSL/DTD/CSS/JS components into an XML application by finishing the HTML wrapper.

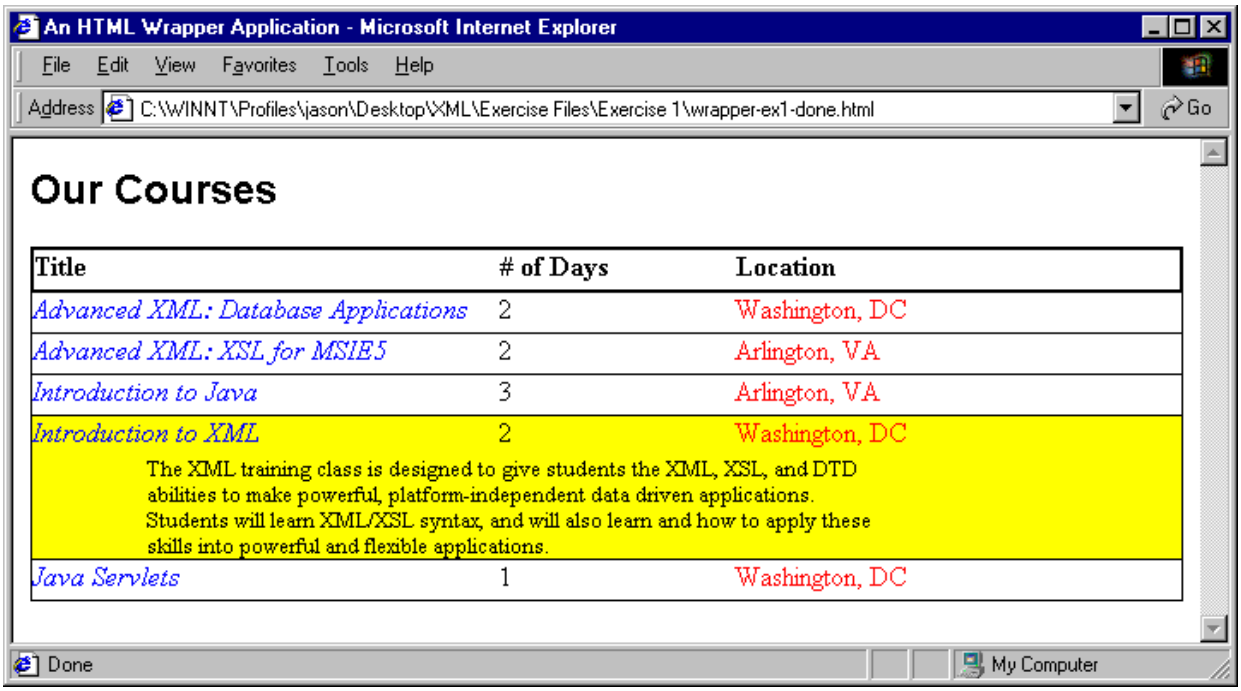
## Exercise 1: Completing an HTML "Wrapper"

In this exercise, you will be completing the HTML front-end to an XML application. The XML, XSL, and JS files are already completed for you (if you're interested, there is a DTD, but it's internal to the XML file).

When you have completed the exercise, the resulting page should look as follows:

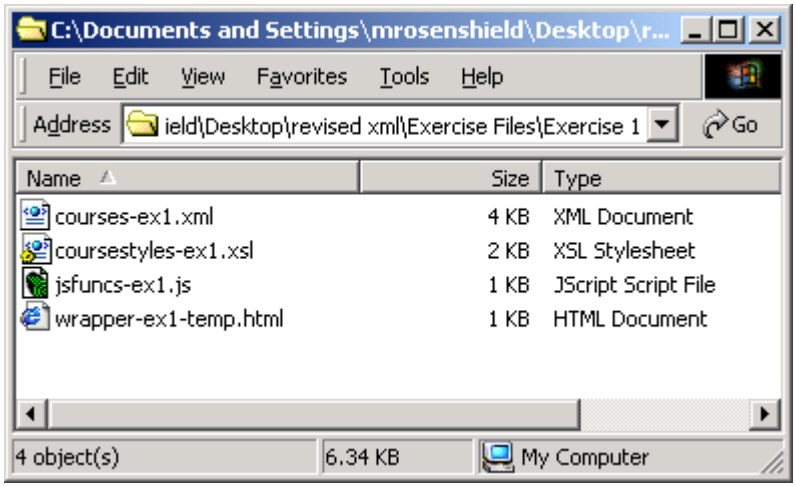



You should also be able to reorder the listing based on the different fields, trigger a mouse-over driven highlight, and make visible a display of additional data by clicking on the course title or number of days. For example, clicking on "Introduction to XML" would produce a display like this:




To complete this exercise:

1. Take a look at the files in **Exercise Files > Exercise 1**. You should see the following files:




2. Open **wrapper-ex1-temp.html** in **Notepad** (or some other text editor). You should see the following code:

```
<html>
<head>
   <title>An HTML Wrapper Application</title>
```

```
<!-- INSERT AN <xml> TAG TO LINK TO THE XML DATA.
     THE ELEMENT id SHOULD BE "courselist" AND THE SOURCE
     SHOULD BE "courses-ex1.xml" -->
<!-- INSERT AN <xml> TAG TO LINK TO THE XSL STYLESHEET.
     THE ELEMENT id SHOULD BE "coursedisplay" AND THE
     SOURCE SHOULD BE "coursestyles-ex1.xsl" -->
<!-- INSERT A <script> TAG TO LINK TO A LIBRARY OF
     JAVASCRIPT FUNCTIONS.  THE SCRIPT language SHOULD
     BE "JavaScript" AND THE SOURCE SHOULD BE "jsfuncs-ex1.js" -->

</head>

<body onLoad="document.getElementById('dataTarget').innerHTML =
courselist.transformNode(coursedisplay)">

<div id="dataTarget"></div>

</body>
</html>
```

3. Please complete the sections in **bold**. When you're done, save the file back into the same directory.

4. Open **wrapper-ex1-temp.html** in IE.

5. If you need to make corrections, edit your page in Notepad, save it, then reload it in the browser.

## If you are done early…

- Open **courses-ex1.xml** in Internet Explorer, and spend some time exploring how IE displays XML data. Try opening it in some other browsers, such as Netscape 4, 6, or 7, Mozilla, or Opera.

- Open up **courses-ex1.xml** in Notepad, and take a look at the internal structure of the tags. Also, take a look at the document type definition in the head of the page. What happens if you add an XML tag?

- Open **wrapper-ex1-temp.html** in Netscape. What do you see? Try it in Netscape 7 or Mozilla.

## A Possible Solution to Exercise 1…

As stored in **Exercise Files > Solutions > Exercise 01 > wrapper-ex1-done.html**:

```
<html>
<head>
   <title>An HTML Wrapper Application</title>

<xml id="courselist" src="courses-ex1.xml"></xml>
<xml id="coursedisplay" src="coursestyles-ex1.xsl"></xml>
<script language="JavaScript" src="jsfuncs-ex1.js"></script>

</head>

<body onLoad="document.getElementById('dataTarget').innerHTML =
courselist.transformNode(coursedisplay)">

<div id="dataTarget"></div>

</body>
</html>
```

# XML Syntax

XML syntax is generally very intuitive, although you may find some of the rules of XML a bit more picky than the html rules you are used to. However, if you understand the purposes behind the rules, they will quickly become second nature. To begin, let's take another look at the example we saw earlier (**Demos > syntax > simple_example.xml**):

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

<client>
   <name>
      <firstname>John</firstname>
      <lastname>Doe</lastname>
   </name>
   <address type="home">
      <street_address>123 School St.</street_address>
      <city>Ithaca</city>
      <state>NY</state>
      <zip>14850</zip>
   </address>
   <age>26</age>
</client>
```

The example illustrates several important rules of XML, explained below. But, before we begin, let's settle on some terminology (a complete glossary is included in **Appendix C**):

- **Tag:** An individual string of text that determines the opening or closing component of an element. In the XML example above, an example of a tag would be **address**. Tags are defined by a pair of opening and closing angle-brackets, and so can include any attributes. Note that this does not include either closing tags or any content enclosed by that tag: **<address>** is an "opening tag" and **</address>** is a "closing tag".

- **Attribute:** A name and value pair contained in an element's opening tag (in the above example, **type="home"** would be an attribute).

- **Element:** The complete object, as defined by the opening and closing tags, any attributes, and any content, including sub-elements and text. In the above example, **client** would be an element, which would contain the sub-elements **name**, **address**, and **age**, and all their contents. A synonym for element is **node**.

# XML Logical Structure

## Each XML document must begin with a language declaration

The initial line of any XML file must be the language declaration. The **<?** … **?>** brackets indicate processing instructions, so that the line is interpreted by the XML parser instead of being treated as a generic, user-defined tag. In addition, it is strongly encouraged that the author also indicates the version of XML she is using (in this case **version="1.0"**). The version declaration is an example of an XML attribute, which will be discussed more thoroughly below.

Finally, an **important note**: XML tags are case sensitive. In the language declaration, the tag must be written in lower case. The XML parser is not able to understand a tag written **<?XML VERSION="1.0"?>.**

## Each XML document must have a document (root) element, which normally contains other child elements

In our previous example, the **document or root element** name is **client**. Normally, XML datasheets will include several elements (i.e. a list of 50 clients). However, as all markup languages (including XML) must have a single root, we specify a root-level tag to contain all these child elements. So, a more complete example might look as below (**demos > syntax > simple_example_2.xml**). Particularly note the new **clients** element, which contains three individual **client** elements):

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

<clients>
   <client>
      <name>
         <firstname>John</firstname>
         <lastname>Doe</lastname>
      </name>
      <address type="home">
         <street_address>123 School St.</street_address>
         <city>Ithaca</city>
         <state>NY</state>
         <zip>14850</zip>
      </address>
      <age>26</age>
   </client>
   <client>
      <name>
         <firstname>Jane</firstname>
         <lastname>Doe</lastname>
      </name>
      <address type="home">
         <street_address>1 Wall St.</street_address>
```

```
            <city>New York</city>
            <state>NY</state>
            <zip>10014</zip>
         </address>
         <age>22</age>
      </client>
      <client>
         <name>
            <firstname>Mary</firstname>
            <lastname>Doe</lastname>
         </name>
         <address type="business">
            <street_address>4000 Technology Way</street_address>
            <city>Palo Alto</city>
            <state>CA</state>
            <zip>94301</zip>
         </address>
         <age>73</age>
      </client>
   </clients>
```

Note that if you're interested, the file is available as **simple_example_2.xml** in your **Demos** folder.

## XML Physical Structure

The general rules of XML syntax are similar to those of HTML, but more rigorous. The distinctions from HTML fall into five main categories:

### Case Sensitivity

In HTML, tags are not case-sensitive. You might equally write the following possibilities:

```
<TITLE>My New Page</TITLE>
<TITLE>My New Page</title>
<title>My New Page</TITLE>
<title>My New Page</title>
<TiTlE>My New Page</tItLe>
```

However, in XML, the case of closing tags must match the case of the opening tags. So, only the first and the fourth of the above examples would be valid as XML:

```
<TITLE>My New Page</TITLE>
<title>My New Page</title>
```

Case is particularly important in the **processing instructions** (like the `<?xml version="1.0"?>` tag we saw above), where only one case is acceptable.

## Required Closing Tags

In HTML, many tags have developed with optional closing elements (such as the </LI>, </P>, </TD>, and other tags). In XML, all tags must have closing elements. When generating HTML code (as XSLs often do) that will require scripting the optional closing tags.

## New Syntax for "empty elements"

In HTML, there are many tags (such as the IMG and HR tags) that have no end tag, and no logical meaning for one. In XML, such tags must be indicated with a forward slash (/) before the ending angle bracket:

```
<emptyelement attr="value" />
```

## Tags must be nested properly

In HTML, nesting rules are relatively relaxed for most tags. So, you might reasonably write the following code examples:

```
<b>This sentence is really <i>important</b> and interesting</i>.
<b><i>This sentence is very heavily emphasized</b></i>.
```

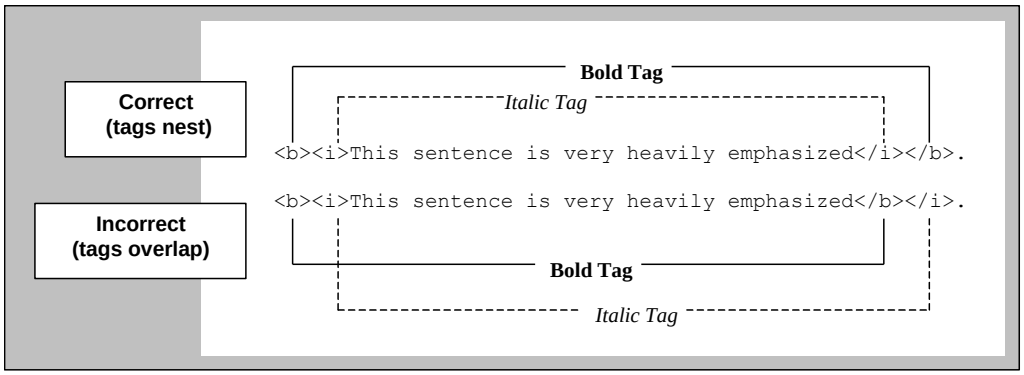
to produce the resulting:

**This sentence is really *important* *and interesting*.**
***This sentence is very heavily emphasized*.**

Neither of the above examples is well-formed, as nested tags are not completed before their parents. Both would produce errors in XML. To correct them, one could write:

```
<b>This sentence is really</b> <i><b>important</b> and interesting</i>.
<b><i>This sentence is very heavily emphasized</i></b>.
```

Take a look at the below diagram:




## Attribute Values must be enclosed properly in single or double quotes

In HTML, you can generally get away without enclosing attribute values in quotes, except in cases where you have spaces or punctuation in the value. So, in HTML, the following tag is legal:

```
<img src=mypicture.gif name=myphoto height=80 width=150>
```

However, the above tag would not be valid in XML. In fact, all values, even numeric ones, must be enclosed in quotes. We will examine attributes in greater depth after the next exercise.

## XML Comments

Comments in XML are identical to comments in HTML. So, the following page includes a commented note:

```
<?xml version="1.0"?>
<!-- Last modified March 23rd -->
<clientlist>
   <client>
.
[etc.]
```

## Character References

XML uses Unicode to represent characters. Unicode is a standard that assigns numerical values to characters in almost every language under the sun, as well as symbols and mathematical notations. Using Unicode means that XML supports internationaliztion fairly easily. Almost all Unicode characters can be included in your XML document if you use a **character reference**. A character reference looks a bit like an entity – starts with an ampersand (&) and ends with a semicolon (;) – but it has a hash (#) right after the ampersand. The hash is followed by the number of character in Unicode, either as a decimal or in hexidecimal if the number starts with an x.

If you wish to include non-standard characters in your document, such as copyrights (©) or Mathematical symbols (Σ) you must use their Unicode numbering system. Similarly, if you want to use a reserved character such as the angle-bracket (<)or the ampersand (&), you must use the ASCII number. We have included a list of most commonly used symbols and non-English alphabetic characters in **Appendix E**.

## XML Logic: Designing Datasheets

It can often be difficult to decide how to mark up your data in XML. For example, consider the following information, representing an address:

John Doe (age 26)
123 School St.
Ithaca, NY 14850

In addition to the previous example (**demos > syntax > simple_example.xml**, and referenced on p. 17), you might imagine data marked up either less or more precisely. We have prepared examples of each, to illustrate the markup process. First the less precise example (**demos > syntax > broad_example.xml**):

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

<client>
   <name>John Doe</name>
   <address type="home">123 School St., Ithaca, NY, 14850</address>
   <age>26</age>
</client>
```

In the above example, we have collapsed the internal nesting of the elements **name** and **address**, producing more complex strings of text for each. By doing so, we haven't lost any textual data, but our data is less usable. We can no longer easily use individual components of name or address, as, for instance, to sort by last name would now be very difficult. Similarly, we would have a very hard time selecting only clients from New York, or even recognizing them as such.

In our next example, we have broken out our data even more precisely in the original example (**demos > syntax > narrow_example.xml**):

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

<client>
   <name>
      <firstname>John</firstname>
      <lastname>Doe</lastname>
   </name>
   <address type="home">
      <street_address>
         <number>123</number>
         <street_name>School</street_name>
```

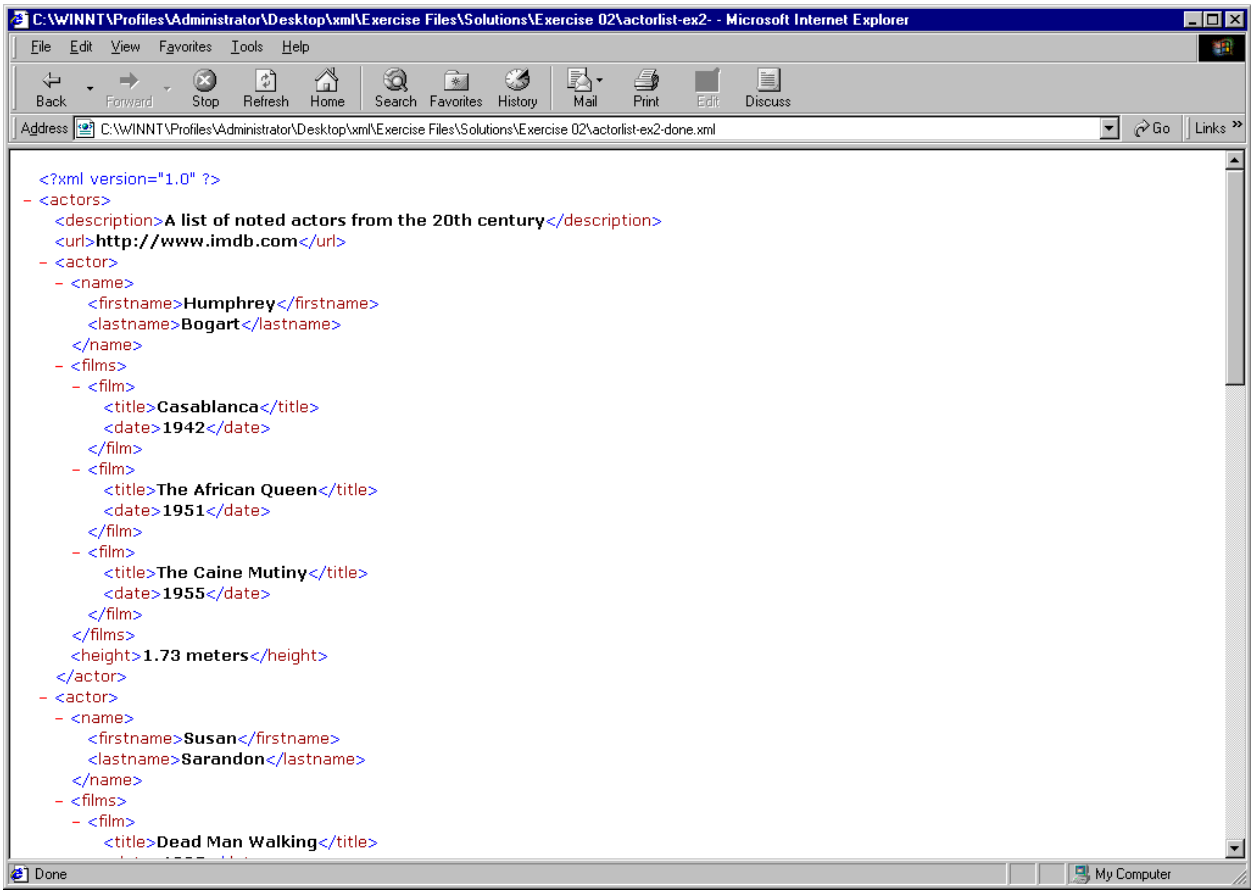
```
            <road_type>St.</road_type>
         </street_address>
         <city>Ithaca</city>
         <state>NY</state>
         <zip>14850</zip>
      </address>
      <age>26</age>
   </client>
```

In our above example, we separate out the **street_address** element into constituent components. At first glance, it appears to be fine. However, what would you do with an address like "**RR1, Box 160**", or one with an apartment number? When you find that you have to do contortions to make your data fit your structure, you have probably subdivided too much.

A good rule of thumb is to *subdivide your data into logically discrete, but not arbitrary, elements*. Once you find yourself making arbitrary distinctions, it's time to stop. This may seem very fuzzy at first, but you'll soon get comfortable with the distinctions.

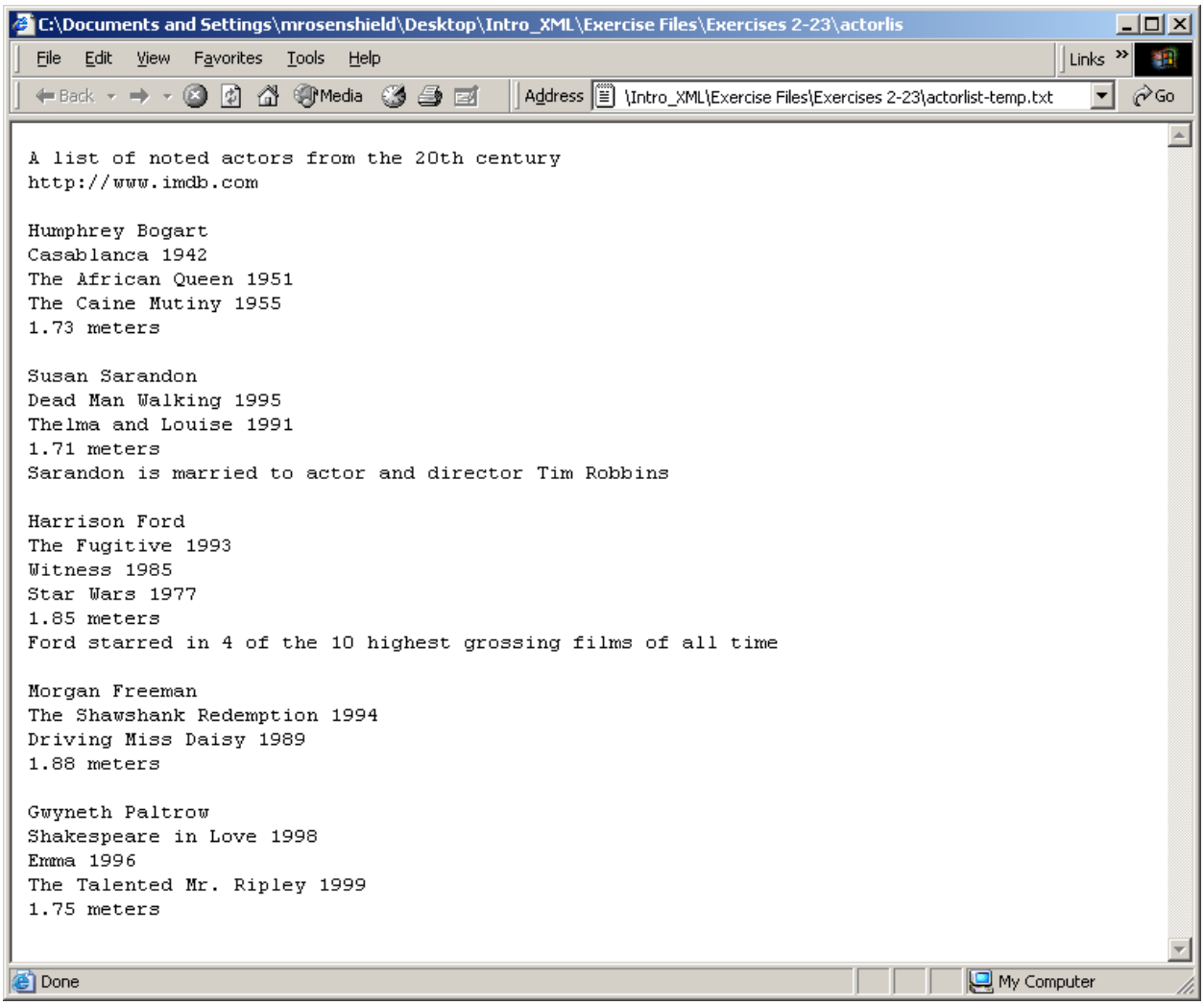
## Exercise 2: Building a well-formed XML document from text data

In this exercise, you will be taking a raw table of text data, and building an XML document out of it. When you are done, you should be able to view the document in Internet Explorer. It should look as follows (note that we are only seeing part of the data; the rest extends off the edge of the page):



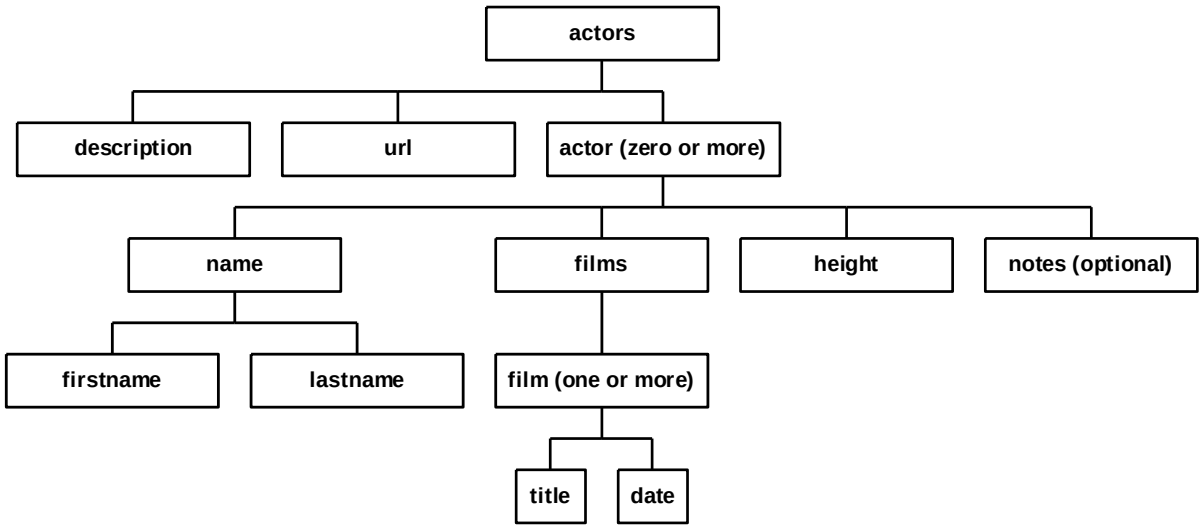

To complete this exercise:

1. Open the file **XML > Exercise Files > Exercises-Main > actorlist-temp.txt** in Notepad. You should see the following:




2. Please turn the text into an XML document. Remember that the first line will have to be the language declaration. The element hierarchy is depicted in the following diagram:

### Hierarchy of actorlist file: element and sub-elements




3. When you are done, save your file as **actorlist.xml** inside your **Exercises-Main** directory, and open it in Internet Explorer.
4. If you get an error message, go back into Notepad, edit your file, and reload.
5. Once you have it working, navigate up and down the hierarchy, making sure all your data is present.

## If you are done early…

- Add a listing for your favorite actor.
- Add notes data for a few of the other names in the list.

## A Possible Solution to Exercise 2

As contained in **actorlist-ex2-done.xml**:

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

<actors>
   <description>A list of noted actors from the 20th
century</description>
   <url>http://www.imdb.com</url>
   <actor>
      <name>
         <firstname>Humphrey</firstname>
         <lastname>Bogart</lastname>
      </name>
      <films>
         <film>
            <title>Casablanca</title>
            <date>1942</date>
         </film>
         <film>
            <title>The African Queen</title>
            <date>1951</date>
         </film>
         <film>
            <title>The Caine Mutiny</title>
            <date>1955</date>
         </film>
      </films>
      <height>1.73 meters</height>
   </actor>
   <actor>
      <name>
         <firstname>Susan</firstname>
         <lastname>Sarandon</lastname>
      </name>
      <films>
         <film>
            <title>Dead Man Walking</title>
            <date>1995</date>
         </film>
         <film>
            <title>Thelma and Louise</title>
            <date>1991</date>
         </film>
      </films>
      <height>1.71 meters</height>
      <notes>Sarandon is married to actor and director Tim
Robbins</notes>
   </actor>
   <actor>
      <name>
```

```
         <firstname>Harrison</firstname>
         <lastname>Ford</lastname>
      </name>
      <films>
         <film>
            <title>The Fugitive</title>
            <date>1993</date>
         </film>
         <film>
            <title>Witness</title>
            <date>1985</date>
         </film>
         <film>
            <title>Star Wars</title>
            <date>1977</date>
         </film>
      </films>
      <height>1.85 meters</height>
      <notes>Ford starred in 4 of the 10 highest grossing films of all
time</notes>
   </actor>
   <actor>
      <name>
         <firstname>Morgan</firstname>
         <lastname>Freeman</lastname>
      </name>
      <films>
         <film>
            <title>The Shawshank Redemption</title>
            <date>1994</date>
         </film>
         <film>
            <title>Driving Miss Daisy</title>
            <date>1989</date>
         </film>
      </films>
      <height>1.88 meters</height>
   </actor>
   <actor>
      <name>
         <firstname>Gwyneth</firstname>
         <lastname>Paltrow</lastname>
      </name>
      <films>
         <film>
            <title>Shakespeare in Love</title>
            <date>1998</date>
         </film>
         <film>
            <title>Emma</title>
            <date>1996</date>
         </film>
         <film>
            <title>The Talented Mr. Ripley</title>
            <date>1999</date>
```

```
            </film>
        </films>
        <height>1.75 meters</height>
    </actor>
</actors>
```

## XML Attributes Revisited

XML attributes work similarly to HTML attributes, in that they extend the functionality of the tag that contains them. However, they are subject to a few additional syntax rules. First, XML attribute values must be surrounded by quotes. In HTML, quotes around the attribute values are generally optional, except when they contain spaces. So, the following is a valid HTML tag:

```
<img src=myfile.gif name=myimage height=80 width=150>
```

However, the above tag would not be valid in XML. All values, including numeric values, must be contained in either single or double quotes (they must match, of course). If the data is to be interpreted as numeric, it will be parsed as such in the processing script.

Attributes that may not require a value are considered to be **Boolean attributes**. For example, in HTML, the **noshade** attribute of the the <hr> tag doesn't have a value when written as follows:

```
<hr noshade />
```

In order to make this a well – formed XML element you must assign this attribute a value. To assign a value to a Boolean attribute you simple assign the attribute a value that is equal to its name, such as:

```
<hr noshade = "noshade" />
```

## Why Use Attributes?

In XML, there is little distinction made between values in attributes and values contained in sub-elements. So, you might equally well mark up your data in either of the following two ways. First, as sub-tags:

```
<car>
   <type>4-door</type>
   <color>green</color>
   <make>Ford</make>
   <model>Taurus</model>
</car>
```

or, with attributes:

```
<car type="4-door">
   <color>green</color>
   <make>Ford</make>
   <model>Taurus</model>
</car>
```

In general, you will better be able to describe your data in nested elements. So you will usually use sub-elements. There are several exceptions however.

**First**, using attributes often allows you to get a strictly numeric value as your text. For example:

<weight>**10 kg**</weight>

will force you to do some complicated string manipulation should you want to extract a pure numeric value out of your weight element, whereas

<weight **units="kg"**>**10**</weight>

will allow you to natively treat the data as numeric (and, incidentally, will allow you easier access to the values of your units as well).

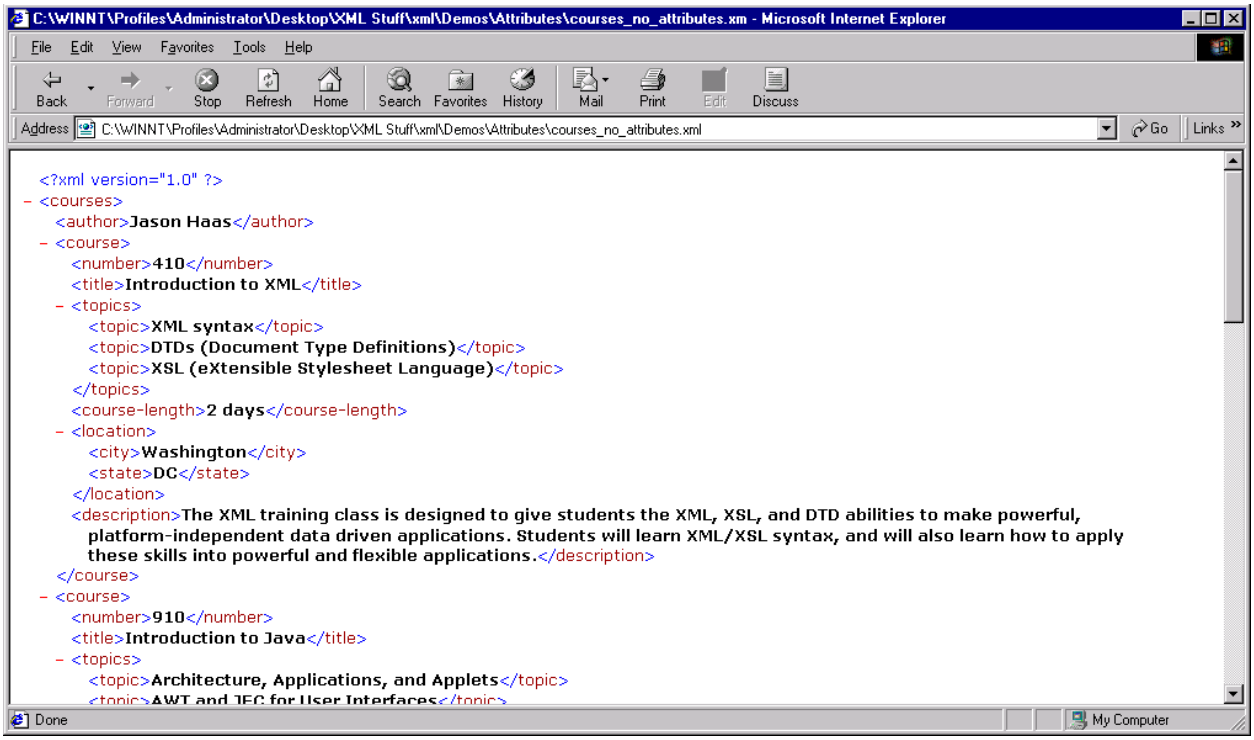
**Second**, if you are using DTDs, you have more options for validating data in attributes than you do in tag values. You will want to use attributes in the following situations:

1. When you have a strictly limited list of possible values for a particular attribute (for example, only "yes" or "no" legally acceptable).

2. When you want to be able to specify a default value.

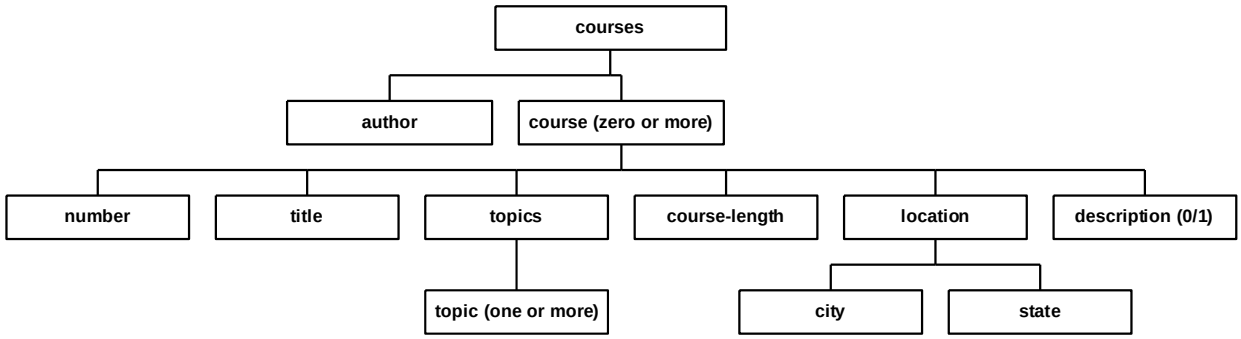
## An Introduction to Our Demo Application

We've prepared an XML application that will follow the same development steps that you do with your exercises. In each section, we will take a look at another step in the development of this application. Please first take a look at the application without any attributes (in **Demos > Attributes > courses_no_attributes.xml**):




Here is the XML hierarchy for the case study:

**Hierarchy of Our Case Study: the courses.xml Series**




Take a look at the code for one **course** element below. Certain portions (which make good candidates for attribute values) are highlighted in **bold**:

```
<course>
   <number>410</number>
   <title>Introduction to XML</title>
   <topics>
      <topic>XML syntax</topic>
      <topic>DTDs (Document Type Definitions)</topic>
      <topic>XSL (eXtensible Stylesheet Language)</topic>
   </topics>
```

```
      <course-length>2 days</course-length>
      <location>
         <city>Washington</city>
         <state>DC</state>
      </location>
      <description>The XML training class is designed to give
students the XML, XSL, and DTD abilities to make powerful,
platform-independent data driven applications.  Students will learn
XML/XSL syntax, and will also learn how to apply these skills into
powerful and flexible applications.</description>
   </course>
```

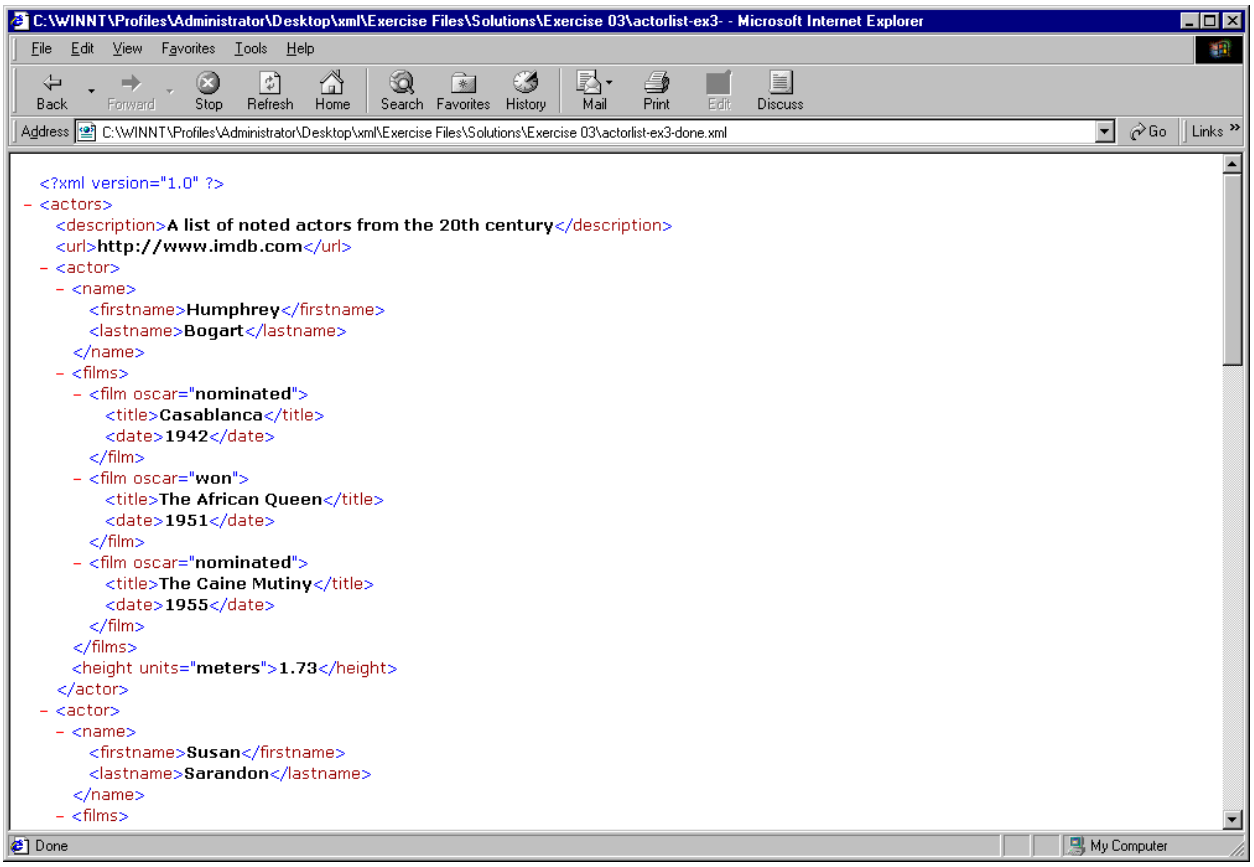
Compare the above code to the code below, which includes attributes to provide better information about each course element, or simplifies the data for some sort of sorting:

```
   <course subject="XML">
      <number>410</number>
      <title>Introduction to XML</title>
      <topics>
         <topic>XML syntax</topic>
         <topic>DTDs (Document Type Definitions)</topic>
         <topic>XSL (eXtensible Stylesheet Language)</topic>
      </topics>
      <course-length scale="days">2</course-length>
      <location>
         <city>Washington</city>
         <state>DC</state>
      </location>
      <description>The XML training class is designed to give
students the XML, XSL, and DTD abilities to make powerful,
platform-independent data driven applications.  Students will learn
XML/XSL syntax, and will also learn how to apply these skills into
powerful and flexible applications.</description>
   </course>
```

In the first case, adding a course **type** attribute will allow us to sort or select on a particular course type, while in the second case, specifying the time **scale** in an attribute will allow us to treat the value of **course-length** numerically. (Note: the above code can be found in **Demos > Attributes > courses_with_attributes.xml**).

## Exercise 3: Adding attributes to your XML Datasheet

In this exercise, you will be adding attributes to two tags in your copy of **actorlist.xml**. When you are done, if you view the document in Internet Explorer, it should look as follows (note again that we are only seeing part of the data; the rest extends off the edge of the page):




To complete this exercise:

1. Re-open the file **actorlist.xml** from your **Exercises-Main** folder in Notepad.

2. Add the attribute "oscar" to your XML **film** tag. The possible values are "won", "nominated", and "no". If you are interested in being factually correct, the films in which the relevant actors won Oscars were:

    - The African Queen
    - Dead Man Walking
    - Shakespeare in Love

    The films for which the actors were nominated (but did not win) were:

- Casablanca
- The Caine Mutiny
- Thelma and Louise
- Witness
- The Shawshank Redemption
- Driving Miss Daisy

The films for which the actors were not nominated were:

- The Fugitive
- Star Wars
- Emma
- The Talented Mr. Ripley

3. Please add the attribute "units" to your XML **height** tag, and correct the tag's value so that it is numeric.

4. When you are done, save your file again as **actorlist.xml** and open it in Internet Explorer. If you get an error message, go back into Notepad, edit your file, and reload.

## If you are done early…

- Replace your correctly matched quotes with mismatched (or missing) ones. What is the error that is generated?

## A Possible Solution to Exercise 3

As contained in **actorlist-ex3-done.xml**:

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

<actors>
   <description>A list of noted actors from the 20th
century</description>
   <url>http://www.imdb.com</url>
   <actor>
      <name>
         <firstname>Humphrey</firstname>
         <lastname>Bogart</lastname>
      </name>
      <films>
         <film oscar="nominated">
            <title>Casablanca</title>
            <date>1942</date>
         </film>
         <film oscar="won">
            <title>The African Queen</title>
            <date>1951</date>
         </film>
         <film oscar="nominated">
            <title>The Caine Mutiny</title>
            <date>1955</date>
         </film>
      </films>
      <height units="meters">1.73</height>
   </actor>
   <actor>
      <name>
         <firstname>Susan</firstname>
         <lastname>Sarandon</lastname>
      </name>
      <films>
         <film oscar="won">
            <title>Dead Man Walking</title>
            <date>1995</date>
         </film>
         <film oscar="nominated">
            <title>Thelma and Louise</title>
            <date>1991</date>
         </film>
      </films>
      <height units="meters">1.71</height>
      <notes>Sarandon is married to actor and director Tim
Robbins</notes>
   </actor>
   <actor>
      <name>
```

```
            <firstname>Harrison</firstname>
            <lastname>Ford</lastname>
         </name>
         <films>
            <film oscar="no">
               <title>The Fugitive</title>
               <date>1993</date>
            </film>
            <film oscar="nominated">
               <title>Witness</title>
               <date>1985</date>
            </film>
            <film oscar="no">
               <title>Star Wars</title>
               <date>1977</date>
            </film>
         </films>
         <height units="meters">1.85</height>
         <notes>Ford starred in 4 of the 10 highest grossing films of all
time</notes>
      </actor>
      <actor>
         <name>
            <firstname>Morgan</firstname>
            <lastname>Freeman</lastname>
         </name>
         <films>
            <film oscar="nominated">
               <title>The Shawshank Redemption</title>
               <date>1994</date>
            </film>
            <film oscar="nominated">
               <title>Driving Miss Daisy</title>
               <date>1989</date>
            </film>
         </films>
         <height units="meters">1.88</height>
      </actor>
      <actor>
         <name>
            <firstname>Gwyneth</firstname>
            <lastname>Paltrow</lastname>
         </name>
         <films>
            <film oscar="won">
               <title>Shakespeare in Love</title>
               <date>1998</date>
            </film>
            <film oscar="no">
               <title>Emma</title>
               <date>1996</date>
            </film>
            <film oscar="no">
               <title>The Talented Mr. Ripley</title>
               <date>1999</date>
```

```
         </film>
      </films>
      <height units="meters">1.75</height>
   </actor>
</actors>
```

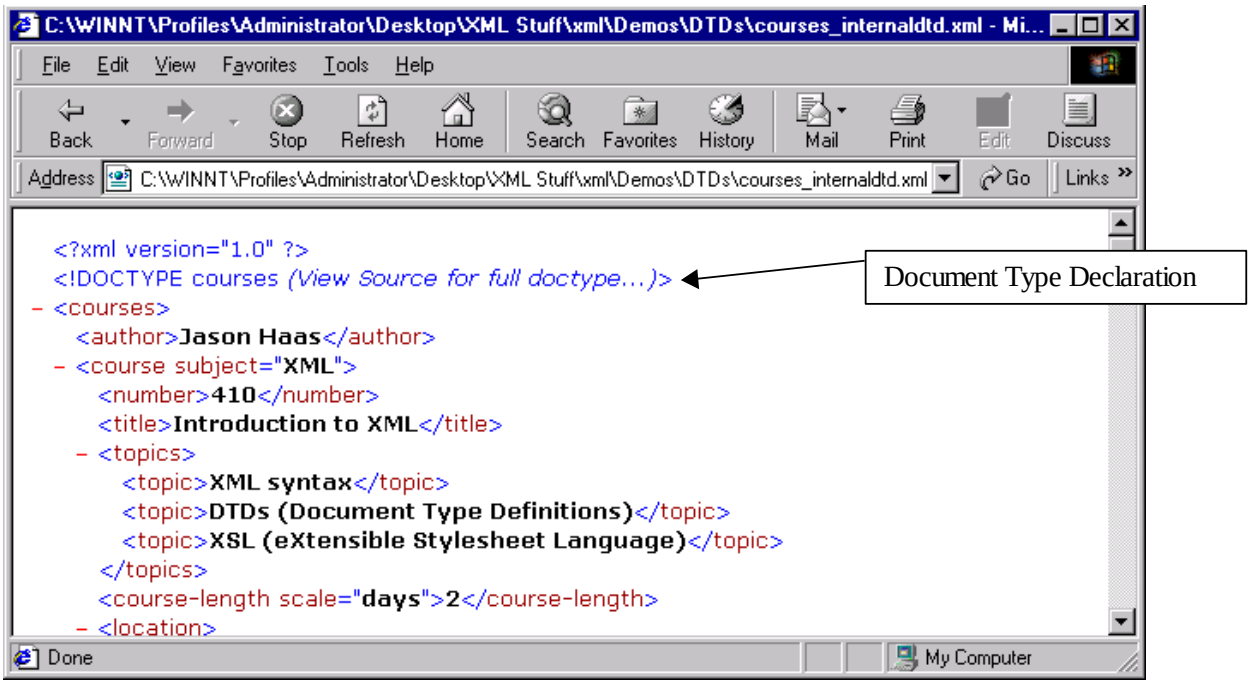
# Document Type Definitions (DTDs)

Although you can have a well-formed XML datasheet without pre-declaring its structure, self-validating is more useful. You wouldn't send out a text document without spell-checking it, so why shouldn't you validate your XML data as well? You make XML data self-validating by including specifications for the document type. For example, you may want users to specify the **oscar** value as a tag, rather than an attribute (and, either way, you are going to want to be clear what the appropriate structure is). A document type definition (DTD) is the specification of the structure and relationships that are valid for a given XML application. This information can be included directly at the beginning of the XML datasheet, or (more commonly) as an external document, where it is accessible by many different applications.

DTDs are particularly important for applications that use XML to exchange data between otherwise incompatible systems, applications, or organizations. Different applications can produce XML in a specified format, and applications can be developed that expect the data in the format specified. DTDs are used both to describe the specifications and to enforce them.

## Example: A Basic DTD

Take a look at the next section from our case study, found in **Demos > DTDs > courses_internaldtd.xml**. When parsed, you will see one new line at the top of your document:




You may have seen a document type declaration at the beginning of some HTML documents. It is suggested that HTML authors include one at the beginning of the pages they author, in order

to identify the version of HTML that they have used. This is the DTD declaration for an HTML 4.0 file:

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

However, as the meaning of most tags is relatively constant, the document type declaration is little more than a comment on HTML version. In XML, document type declarations are crucial, as the meaning of tags is determined by the author each time a new page is designed. Let's take a look at the document type definition code from the above example:

```
<!DOCTYPE courses [
<!ELEMENT courses (author, course*)>
   <!ELEMENT author (#PCDATA)>
   <!ELEMENT course (number, title, topics, course-length, location, description?)>
      <!ATTLIST course subject CDATA #REQUIRED>
      <!ELEMENT number (#PCDATA)>
      <!ELEMENT title (#PCDATA)>
      <!ELEMENT topics (topic+)>
         <!ELEMENT topic (#PCDATA)>
      <!ELEMENT course-length (#PCDATA)>
         <!ATTLIST course-length scale (days|hours|weeks) "days">
      <!ELEMENT location (city, state)>
         <!ELEMENT city (#PCDATA)>
         <!ELEMENT state (#PCDATA)>
      <!ELEMENT description (#PCDATA)>
]>
```

- The DOCTYPE tag, and the square brackets, must surround an internal document type definition. The syntax for the DOCTYPE tag is a bit different for an external DTD. These differences will be covered in the next section.

- In either internal or external DTDs, the DOCTYPE name must be the same as the top-level element name (in the above case, **courses**).

- Each XML element must have an ELEMENT tag. This tag will contain either the sub-elements or the data type in parentheses.

- Tags with attributes must have an ATTLIST tag specifying the possible attributes. This tag can also specify the legal values and default value for the attribute.

- There is a quantity indicating syntax specific to DTDs to specify sub-element quantities. The table below contains a summary of DTD pattern matching:

| Symbol | Meaning |
|---|---|
| + | The element must appear one or more times |
| * | The element can appear zero or more times |
| ? | The element can appear zero or one time |
| | Element must appear exactly once |
| \| | Separates element in a list of possible values |

- The DTD tags, including DOCTYPE, ELEMENT, and ATTLIST are all case-sensitive, and must be in upper case.

If the syntax seems a little strange, let's analyze an example of each of the four pattern-matching symbols.

## The * Symbol (Matching Zero or More Elements)

```
<!ELEMENT courses (author, course*)>
```

In the above example, we are thinking ahead to where our document type definition will be external. We want to be able to match a **courses** list in many possible circumstances. We know that most course lists will have multiple **course** elements, and we decided that you can have a valid courses list with no **course** elements (imagine a list of courses for a trainer who is on sabbatical leave), so we specify *zero or more* **course** sub-elements for each **courses** element.

## The + Symbol (Matching One or More Elements)

```
<!ELEMENT topics (topic+)>
```

Each course contains a **topics** element, which must contain at least one **topic**. However, it is perfectly acceptable for a course to contain more than one topic. So, we wish to keep out any course with no topics (would you want to take such a course?) but allow any course with *one or more* **topic** elements.

## The ? Symbol (Matching Zero or One Elements)

```
<!ELEMENT course (number, title, topics, course-length, location, description?)>
```

Each course must contain **number**, **title**, **topics**, **course-length**, and **location**. In addition, some courses will have a **description**. To indicate an *optional* description, we follow the element name with a question mark.

## The | Symbol (Separating Possible Elements in a List)

```
<!ATTLIST course-length scale (days|hours|weeks) "days">
```

The **course-length** element has an attribute **scale**. We wish to specify acceptable values for the scale attribute, and have decided that valid options are days, hours, and weeks (so, no courses measured in centuries or minutes). To specify a list of possible options, we follow the attribute name with a list of the options, in parentheses, separated by the pipe (vertical line) symbol. If you're looking for it on your keyboard, it's **shift-**.

## Default Values, #REQUIRED and #IMPLIED

In your DTD, you may provide a default value for an attribute. If the XML datasheet does not specify a value, the default will be used. In the above example, we are specifying "days" as the default value. If you choose not to give a default, you must specify either that the attribute is required (#REQUIRED) or optional (#IMPLIED). A third option, #FIXED, is rarely used.

## CDATA and PCDATA

When you specify a node that contains text content (either tag content or attribute values) you must specify a data type. Your two choices are **CDATA** (short for character data) and **PCDATA** (short for parsed character data). The main distinction between the two is that data defined as PCDATA will be parsed for character equivalents such as < and & but cannot contain any child elements. CDATA will not allow these kinds of characters.

Effectively, this means that **CDATA** is the only commonly accepted data type for attributes. For node data, **PCDATA** is much more common.

## Mixed Content

What happens when a node contains PCDATA as well as subnodes? Do we have to run into the woods screaming in terror? We could, but it's not necessary. Instead we could validate such a "mixed content" node as follows:

```
<!ELEMENT nodeName (#PCDATA|subnodeName1|subnodeName2)*>
```

This code says that nodeName has subnodeName1, subnodeName2 as subnodes and that there may be text interspersed between them. There are a couple of important things to note here. First, PCDATA must be the first item in the list of subnodes (text inside an element is actually considered a subnode of that element). Second, you can no longer specify how many instances of the subnodes are allowed: any number is now fair game.

## Empty Elements

In the event you need to define the use of an element that has no content, that element will be declared as an empty element. For example, HTML declares the <BR/>, <HR/>, and <IMG/> as empty element because they contain no information other than attributes. To declare empty elements in your DTD use the following syntax:

```
<!ELEMENT nodeName EMPTY>
```

## Comments in DTDs

DTDs can contain comments, just like the rest of an XML document. These comments cannot appear inside a declaration, but they can appear outside one. Comments are often used to organize the DTD in different parts, to document the allowed content of particular elements, and to further explain what an element is. For example, a typical element declaration might look like the following:

```
<!-- This is comment about the following element -->
<!ELEMENT nodeName EMPTY>
```

### Specifying Multiple ttributes with ATTLIST

What if you want an element to have multiple attributes? Here's the syntax:

```
<!ATTLIST course subject CDATA #REQUIRED comments CDATA #IMPLIED>
```

This code would require each course node to have a subject attribute, and would allow an optional attribute called comments.

## Validating Against your DTDs

Unfortunately, the W3C standards do not declare what should happen if data does not validate successfully against its DTD. Therefore, individual implementations have varied. In Internet Explorer, although it will parse the DTD for syntax problems, default values, and the like, DTD patterns are not *evaluated against the XML datasheet* by default. It is validated if the XML data is being accessed by an outside application (such as an HTML page). Short of building an extraneous HTML wrapper page just to check for validation against a DTD, you have three options for checking your XML:

1. You can use a DOM method **validate()** to check an XML datasheet against its DTD. Although the DOM is outside the scope of this course, we have included a utility file named **wrapper_validate.html** (which references a JavaScript function) that will use the DOM to check any XML document against its DTD.

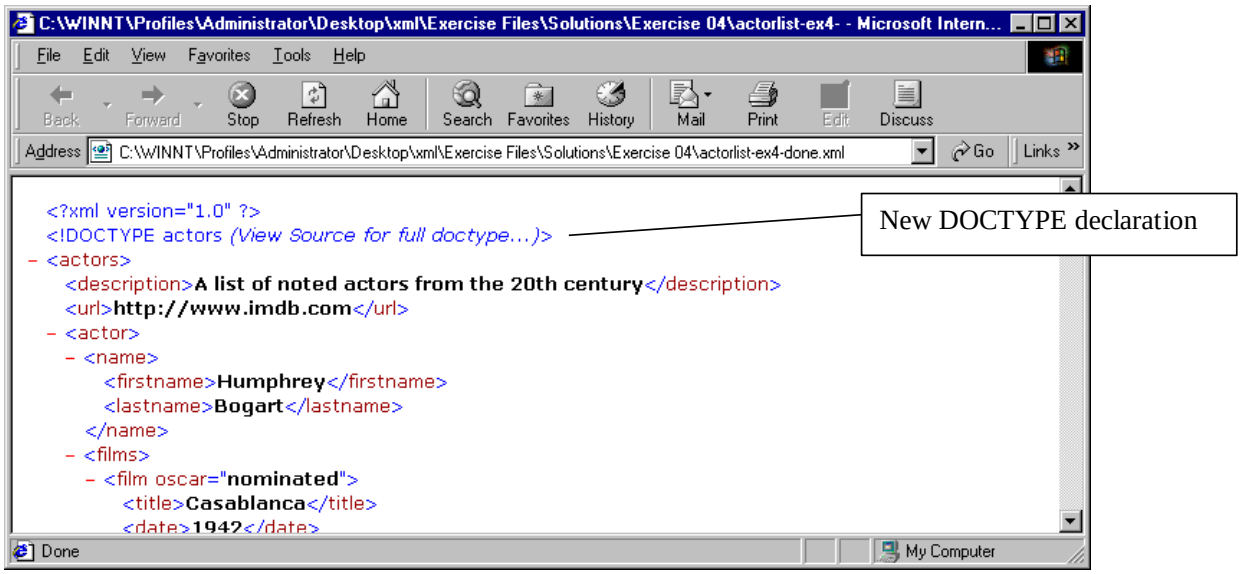
2. You can use one of many DTD validation utilities available publicly on the Web. Two of the best that we have found are:

    - **Brown University Scholarly Technology Group**
      (*http://www.stg.brown.edu/service/xmlvalid/*)

    - **Richard Tobin's XML well-formedness checker and validator**
      (*http://www.cogsci.ed.ac.uk/%7Erichard/xml-check.html)*

## Exercise 4: Adding an Internal DTD to your XML Datasheet

In this exercise, you will be adding a document type definition to **actorlist.xml**. When you are done, if you view the document in Internet Explorer, you should see the following extra line in your code:




To complete this exercise:

1. Re-open the file **actorlist.xml** from your **XML** folder in Notepad.
2. Please add a document type declaration to your code, according to the following specifications:
    - The DOCTYPE name should be **actors**
    - You should accept a **description**, a **url**, and zero or more **actor** sub-elements for **actors.**
    - The **actor** element should have the required sub-elements **name**, **films**, and **height**, and the optional sub-element **notes**.
    - The **name** element should have the required sub-elements **firstname** and **lastname**.
    - The **films** element should have the one or more **film** elements as children.
    - The **film** element should have the attribute **oscar**, with the possible values "won", "nominated", and "no". The default value should be "no".
    - The **film** element should have the required child elements **title** and **date**

- The **height** element should have the attribute **units**, which should have the possible legal values of **meters**, **cm**, and **feet**. It should be required.

- All elements should take the text type **#PCDATA**.

3. When you are done, save your file again as **actorlist.xml** and open it in Internet Explorer.

4. If you get an error message, go back into Notepad, edit your file, and reload.

5. Once you have it working, open the file **Exercise Files > Exercises-Main > wrapper_validate.html**. If you receive the results "Document Validates OK" then you've been successful. If not, fix the error that is described. Experiment with seeing what kinds of errors this validator catches.

## If you are done early…

- What happens now if a user enters "WON" or "NO" as values for oscar? What could you do to keep the values above from producing errors?

- Try adding a "mixed content" node. See the text of this section for more on mixed content nodes.

- If you are familiar with JavaScript, look at the file *validateXML.js* to see if you can figure out how the validation was done.

- Try out some of the validation websites mentioned earlier.

- Download the xml right-click utility from Microsoft and try out its validation capabilities. To download it, go to:

  *http://msdn.microsoft.com/xml*

  *Microsoft often changes the architecture of their website, so we cannot give you the exact URL. At the above url, look for an xml downloads link. There should be a link there for "Internet Explorer Tools for Validating XML and Viewing XSLT Output."*

  Once you have it downloaded, go to the folder where the files were stored (if you haven't specified otherwise, this will be C:\IEXMLTLS). Right-click on the first file ending in the .inf extension and choose install from the resulting pop-up menu. Repeat this for any other files with that extension. Now restart your browser, open an xml file with a DTD reference, right-click, and choose "Validate XML".

## A Possible Solution to Exercise 4

As contained in **actorlist-ex4-done.xml**:

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

<!DOCTYPE actors [
<!ELEMENT actors (description, url, actor*)>
   <!ELEMENT description (#PCDATA)>
   <!ELEMENT url (#PCDATA)>
   <!ELEMENT actor (name, films, height, notes?)>
      <!ELEMENT name (firstname, lastname)>
         <!ELEMENT firstname (#PCDATA)>
         <!ELEMENT lastname (#PCDATA)>
      <!ELEMENT films (film+)>
         <!ELEMENT film (title, date)>
            <!ATTLIST film oscar (nominated|won|no) "no">
            <!ELEMENT title (#PCDATA)>
            <!ELEMENT date (#PCDATA)>
      <!ELEMENT height (#PCDATA)>
         <!ATTLIST height units (meters|cm|feet) #REQUIRED>
      <!ELEMENT notes (#PCDATA)>
]>

<actors>
   <description>A list of noted actors from the 20th
century</description>
   <url>http://www.imdb.com</url>
   <actor>
      <name>
         <firstname>Humphrey</firstname>
         <lastname>Bogart</lastname>
      </name>
      <films>
         <film oscar="nominated">
            <title>Casablanca</title>
            <date>1942</date>
         </film>
         <film oscar="won">
            <title>The African Queen</title>
            <date>1951</date>
         </film>
         <film oscar="nominated">
            <title>The Caine Mutiny</title>
            <date>1955</date>
         </film>
      </films>
      <height units="meters">1.73</height>
   </actor>
   <actor>
      <name>
         <firstname>Susan</firstname>
```

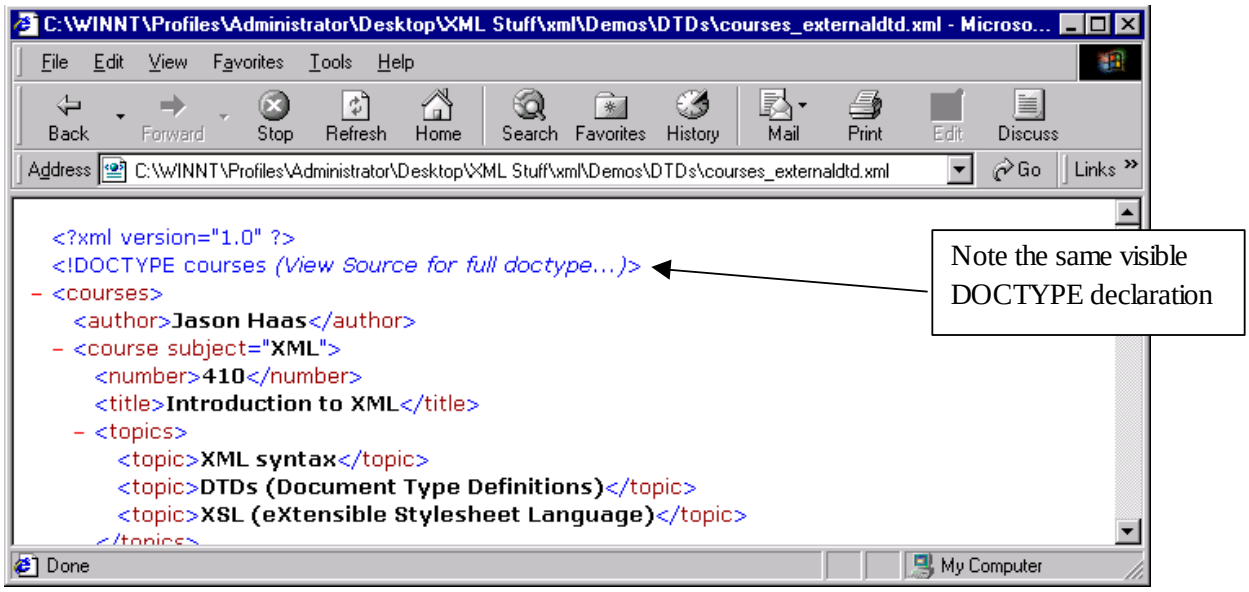
```
         <lastname>Sarandon</lastname>
      </name>
      <films>
         <film oscar="won">
            <title>Dead Man Walking</title>
            <date>1995</date>
         </film>
         <film oscar="nominated">
            <title>Thelma and Louise</title>
            <date>1991</date>
         </film>
      </films>
      <height units="meters">1.71</height>
      <notes>Sarandon is married to actor and director Tim
Robbins</notes>
   </actor>
   <actor>
      <name>
         <firstname>Harrison</firstname>
         <lastname>Ford</lastname>
      </name>
      <films>
         <film oscar="no">
            <title>The Fugitive</title>
            <date>1993</date>
         </film>
         <film oscar="nominated">
            <title>Witness</title>
            <date>1985</date>
         </film>
         <film oscar="no">
            <title>Star Wars</title>
            <date>1977</date>
         </film>
      </films>
      <height units="meters">1.85</height>
      <notes>Ford starred in 4 of the 10 highest grossing films of all
time</notes>
   </actor>
   <actor>
      <name>
         <firstname>Morgan</firstname>
         <lastname>Freeman</lastname>
      </name>
      <films>
         <film oscar="nominated">
            <title>The Shawshank Redemption</title>
            <date>1994</date>
         </film>
         <film oscar="nominated">
            <title>Driving Miss Daisy</title>
            <date>1989</date>
         </film>
      </films>
      <height units="meters">1.88</height>
```

```
   </actor>
   <actor>
      <name>
         <firstname>Gwyneth</firstname>
         <lastname>Paltrow</lastname>
      </name>
      <films>
         <film oscar="won">
            <title>Shakespeare in Love</title>
            <date>1998</date>
         </film>
         <film oscar="no">
            <title>Emma</title>
            <date>1996</date>
         </film>
         <film oscar="no">
            <title>The Talented Mr. Ripley</title>
            <date>1999</date>
         </film>
      </films>
      <height units="meters">1.75</height>
   </actor>
</actors>
```

## External DTDs

DTDs can be (and more typically, are) specified externally, where they can be accessed by many XML datasheets, or distributed to different people or applications to ensure uniformity. Using external DTDs is an excellent, scalable way of ensuring that individual datasheets (which can be produced by many different people and applications) all follow the necessary structure.

The tag to link to an external DTD is relatively simple. Again, we're taking a look at the ongoing demo (this example can be found in **Demos > DTDs > courses_externaldtd.xml**):




As you can see above, the browser treats an external DTD the same as an internal DTD, as what it references and parses is the DOCTYPE name. The actual code structure is below:

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

<!DOCTYPE courses SYSTEM "courses.dtd">

<courses>
   <author>Jason Haas</author>
   <course subject="XML">
      <title>Introduction to XML</title>
      <topics>
         <topic>XML syntax</topic>
.
.
[code deleted]
.
.
```

Inside the DTD itself, the code is unchanged (except that there is no DOCTYPE declaration, as the DOCTYPE itself is declared in the XML document). From **Demos > DTDs > courses.dtd**:

```
<!ELEMENT courses (author, course*)>
   <!ELEMENT author (#PCDATA)>
   <!ELEMENT course (number, title, topics, course-length, location,
description?)>
   <!ATTLIST course subject CDATA #REQUIRED>
   <!ELEMENT number (#PCDATA)>
   <!ELEMENT title (#PCDATA)>
   <!ELEMENT topics (topic+)>
      <!ELEMENT topic (#PCDATA)>
   <!ELEMENT course-length (#PCDATA)>
   <!ATTLIST course-length scale (days|hours|weeks) "days">
   <!ELEMENT location (city, state)>
      <!ELEMENT city (#PCDATA)>
      <!ELEMENT state (#PCDATA)>
   <!ELEMENT description (#PCDATA)>
```

As you can see, there is no special punctuation necessary to begin a DTD. It is not an XML file, and needs (in fact, can take) no `<?xml?>` language declaration. It also does not need to be contained inside the square brackets of the DOCTYPE declaration.

One caution: you will still need to make sure that your DOCTYPE name matches the name of your primary element. It can be harder to notice with an external DTD that your names don't match, and you will need to pay special care that they do.

## Public vs. System DTDs

You have two options for external DTDs. Their syntax is given below:

```
<!DOCTYPE docname SYSTEM "path_to_DTD_file">
<!DOCTYPE docname PUBLIC "public_identifier" "path_to_DTD_file">
```

An external SYSTEM DTD allows you to specify the document type name, and a file (or path to a file) that will serve as the DTD. A PUBLIC DTD allows you to specify an internal or external location or library of DTDs. If the file cannot be found there (or the library cannot be located) the file or path is used. A good example of a public DTD is the HTML DTD mentioned previously:

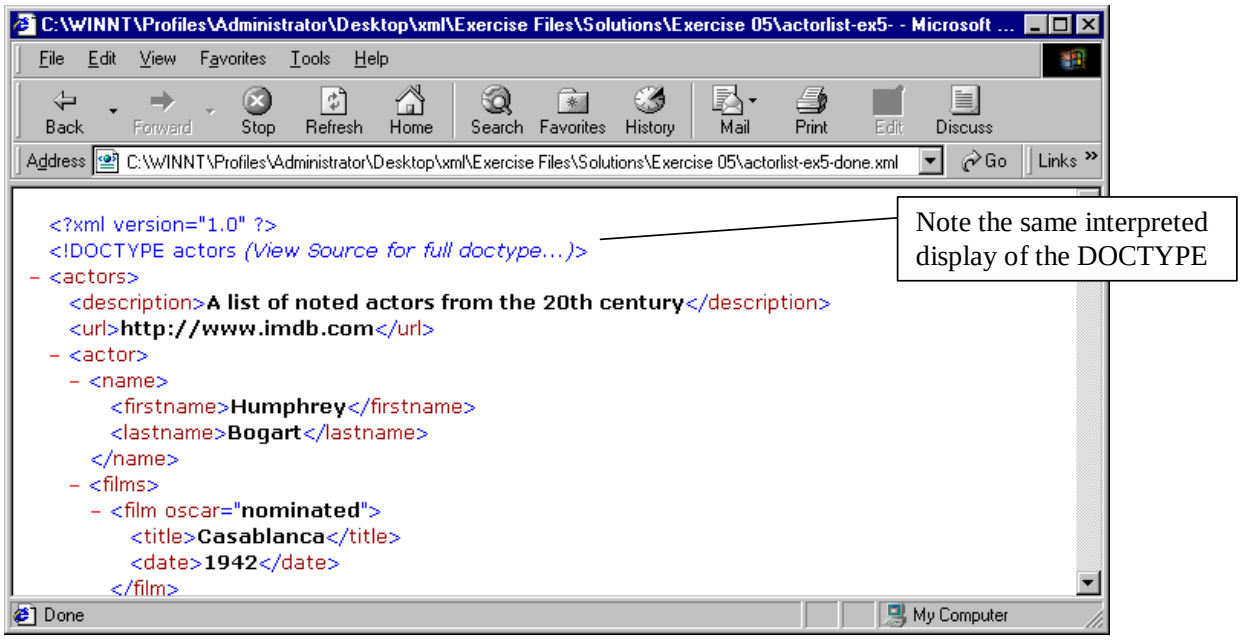
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
```

The more commonly used of the two is SYSTEM, as the library of existing XML document types is currently limited.

Note that there are no operators in a DOCTYPE tag (no equal signs, for example). The tag is parsed as individual elements, separated by a space. Be careful to maintain your spacing in DTD statements.

## Exercise 5: Creating an External DTD, and linking it to your XML Datasheet

In this exercise, you will be moving the DTD you created in the previous exercise to an external file, and then modifying your XML datasheet to link to it as a SYSTEM DTD. When you are done, the display should be no different from what it was at the end of the previous exercise:




To complete this exercise:

1. Re-open the file **actorlist.xml** from your **XML** folder in Notepad.
2. Open a blank document in Notepad, and transfer the DTD code from **actorlist.xml** to the new file.
3. Save the new file as **actordtd.dtd**.
4. Modify **actorlist.xml** so that it contains a link to **actordtd.dtd**. The link should be in the form of a SYSTEM DTD.
5. When you are done, save your XML file again as **actorlist.xml** and open it in Internet Explorer.
6. If you get an error message, go back into Notepad, edit your file, and reload.
7. Once you have it working, check the utility **wrapper_validate.html**, and make sure that the validation is successful.

## A Possible Solution to Exercise 5

As contained in **actorlist-ex5-done.xml**:

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

<!DOCTYPE actors SYSTEM "actordtd-ex5-done.dtd">

<actors>
   <description>A list of noted actors from the 20th
century</description>
   <url>http://www.imdb.com</url>
   <actor>
      <name>
         <firstname>Humphrey</firstname>
         <lastname>Bogart</lastname>
      </name>
      <films>
         <film oscar="nominated">
            <title>Casablanca</title>
            <date>1942</date>
         </film>
         <film oscar="won">
            <title>The African Queen</title>
            <date>1951</date>
         </film>
         <film oscar="nominated">
            <title>The Caine Mutiny</title>
            <date>1955</date>
         </film>
      </films>
      <height units="meters">1.73</height>
   </actor>
   <actor>
      <name>
         <firstname>Susan</firstname>
         <lastname>Sarandon</lastname>
      </name>
      <films>
         <film oscar="won">
            <title>Dead Man Walking</title>
            <date>1995</date>
         </film>
         <film oscar="nominated">
            <title>Thelma and Louise</title>
            <date>1991</date>
         </film>
      </films>
      <height units="meters">1.71</height>
      <notes>Sarandon is married to actor and director Tim
Robbins</notes>
   </actor>
   <actor>
      <name>
```

```
         <firstname>Harrison</firstname>
         <lastname>Ford</lastname>
      </name>
      <films>
         <film oscar="no">
            <title>The Fugitive</title>
            <date>1993</date>
         </film>
         <film oscar="nominated">
            <title>Witness</title>
            <date>1985</date>
         </film>
         <film oscar="no">
            <title>Star Wars</title>
            <date>1977</date>
         </film>
      </films>
      <height units="meters">1.85</height>
      <notes>Ford starred in 4 of the 10 highest grossing films of all
time</notes>
   </actor>
   <actor>
      <name>
         <firstname>Morgan</firstname>
         <lastname>Freeman</lastname>
      </name>
      <films>
         <film oscar="nominated">
            <title>The Shawshank Redemption</title>
            <date>1994</date>
         </film>
         <film oscar="nominated">
            <title>Driving Miss Daisy</title>
            <date>1989</date>
         </film>
      </films>
      <height units="meters">1.88</height>
   </actor>
   <actor>
      <name>
         <firstname>Gwyneth</firstname>
         <lastname>Paltrow</lastname>
      </name>
      <films>
         <film oscar="won">
            <title>Shakespeare in Love</title>
            <date>1998</date>
         </film>
         <film oscar="no">
            <title>Emma</title>
            <date>1996</date>
         </film>
         <film oscar="no">
            <title>The Talented Mr. Ripley</title>
            <date>1999</date>
```

```
         </film>
      </films>
      <height units="meters">1.75</height>
   </actor>
</actors>
```

As contained in **actordtd-ex5-done.dtd**:

```
<!ELEMENT actors (description, url, actor*)>
   <!ELEMENT description (#PCDATA)>
   <!ELEMENT url (#PCDATA)>
   <!ELEMENT actor (name, films, height, notes?)>
      <!ELEMENT name (firstname, lastname)>
         <!ELEMENT firstname (#PCDATA)>
         <!ELEMENT lastname (#PCDATA)>
      <!ELEMENT films (film+)>
         <!ELEMENT film (title, date)>
            <!ATTLIST film oscar (nominated|won|no) "no">
            <!ELEMENT title (#PCDATA)>
            <!ELEMENT date (#PCDATA)>
      <!ELEMENT height (#PCDATA)>
         <!ATTLIST height units (meters|cm|feet) #REQUIRED>
      <!ELEMENT notes (#PCDATA)>
```

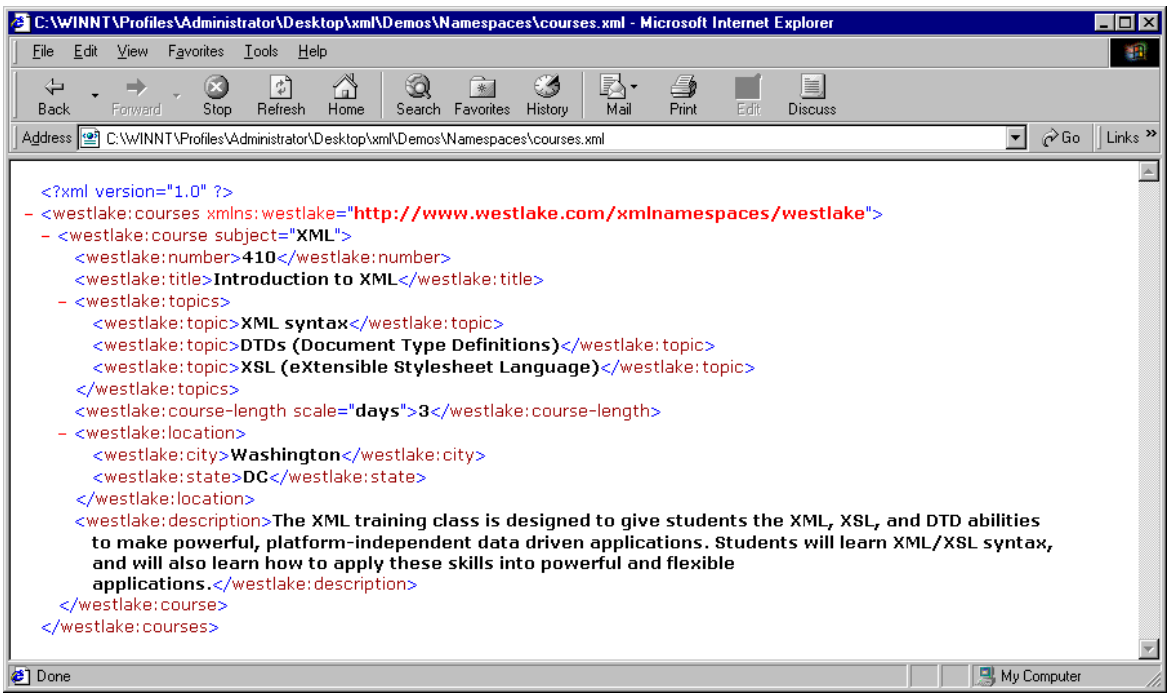
# XML Namespaces

XML tags are endlessly flexible. Tags have whatever meanings their authors define for them. This flexibility is a key piece of the appeal of XML. However, at the same time, it is often useful (even essential) to be able to create tags with specified meanings, in the interest of functionality. For example, in HTML, when a browser encounters a <LINK> tag, it knows to load an external CSS stylesheet into browser memory. We don't want to have recreate the wheel every time we want to make text bold, so we can use the already-defined <b> tag, which happens to belong to a namespace that looks like this:

```
http://www.w3.org/1999/xhtml
```

The inventors of XML recognized the need to allow for the creation of tag subsets that have fixed meaning, while still allowing users maximum flexibility to design the structure that suits their application. The compromise between flexibility and consistency produced the proposal for **XML Namespaces**.

Namespaces are used to reference pre-defined tag subsets, usually with fixed meanings. For example, if WestLake were to create an XML subset to describe its courses (and to distinguish its definitions from other vendors' courses) it might define the prefix **westlake**. It could then write code like (**Demos > Namespaces > courses.xml**):




To declare a namespace, you write an attribute to your root element that begins with the prefix **xmlns** (XML Namespace). Then, you append, after a colon (:) the name of the tag subset which you are referencing, in this case for the **westlake** namespace:

```
xmlns:westlake="http://www.westlake.com/xmlnamespaces/westlake"
```

The value of the attribute is a URI. All namespaces are uniquely defined using a URI, which serves several purposes. First, and most importantly, as URIs are unique, it eliminates the possibility that tag definitions might contradict (so, for example, you don't have to worry that someone else is already using your tag to mean something different). Second, the URI generally indicates the creator of the tag subset, should others want to use or further develop the vocabulary. Finally, a URI provides a natural location for documentation on the tags' meanings (although there is no requirement that there actually be documentation in the location specified; in fact, **applications do not go onto the Internet to reference the URI at any time**. The software that is parsing the document (in our case, the browser) must be internally configured to understand the tags referenced by the namespace).

Once you've declared the namespace, you can then make use of the tags defined in that namespace. In the previous example, several tags use the **westlake** namespace:

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

<westlake:courses xmlns:westlake="http://www.westlake.com/xmlnamespaces/westlake">

   <westlake:course subject="XML">
      <westlake:number>410</westlake:number>
      <westlake:title>Introduction to XML</westlake:title>
      <westlake:topics>
         <westlake:topic>XML syntax</westlake:topic>
         <westlake:topic>DTDs (Document Type Definitions)</westlake:topic>
         <westlake:topic>XSL (eXtensible Stylesheet Language)</westlake:topic>
      </westlake:topics>
      <westlake:course-length scale="days">3</westlake:course-length>
      <westlake:location>
         <westlake:city>Washington</westlake:city>
         <westlake:state>DC</westlake:state>
      </westlake:location>
      <westlake:description>The XML training class is designed to give
students the XML, XSL, and DTD abilities to make powerful,
platform-independent data driven applications.  Students will learn
XML/XSL syntax, and will also learn how to apply these skills into
powerful and flexible applications.</westlake:description>
   </westlake:course>

</westlake:courses>
```

Who decides what the tags within the **westlake** namespace mean? WestLake, of course. Such is the beauty of namespaces: they allow a company, organization, or industry to declare tags to have certain fixed meanings without impacting what tags other companies, industries, or organizations can use. A Web-development company might have a very different meaning for the tag table than would a furniture manufacturer. With namespaces, these companies can share information, even within a single document, without confusion.

# XML Schemas

DTDs have some important weaknesses. You probably recognize them even from having built a couple of DTDs in the previous chapter, but it is worth reviewing them. The principal weaknesses are:

1. **DTDs do not allow data typing**. So, you can only specify that an element contains #PCDATA… not that that data should contain an integer, or a date, or a Boolean value.

2. **DTDs only allow rough specification of quantities**. So, you can say that an element must appear once… or zero or one times… or zero or more times… or one or more times. However, you cannot specify that it must appear 1 to 10 times… or 5 or more times… or zero to two times.

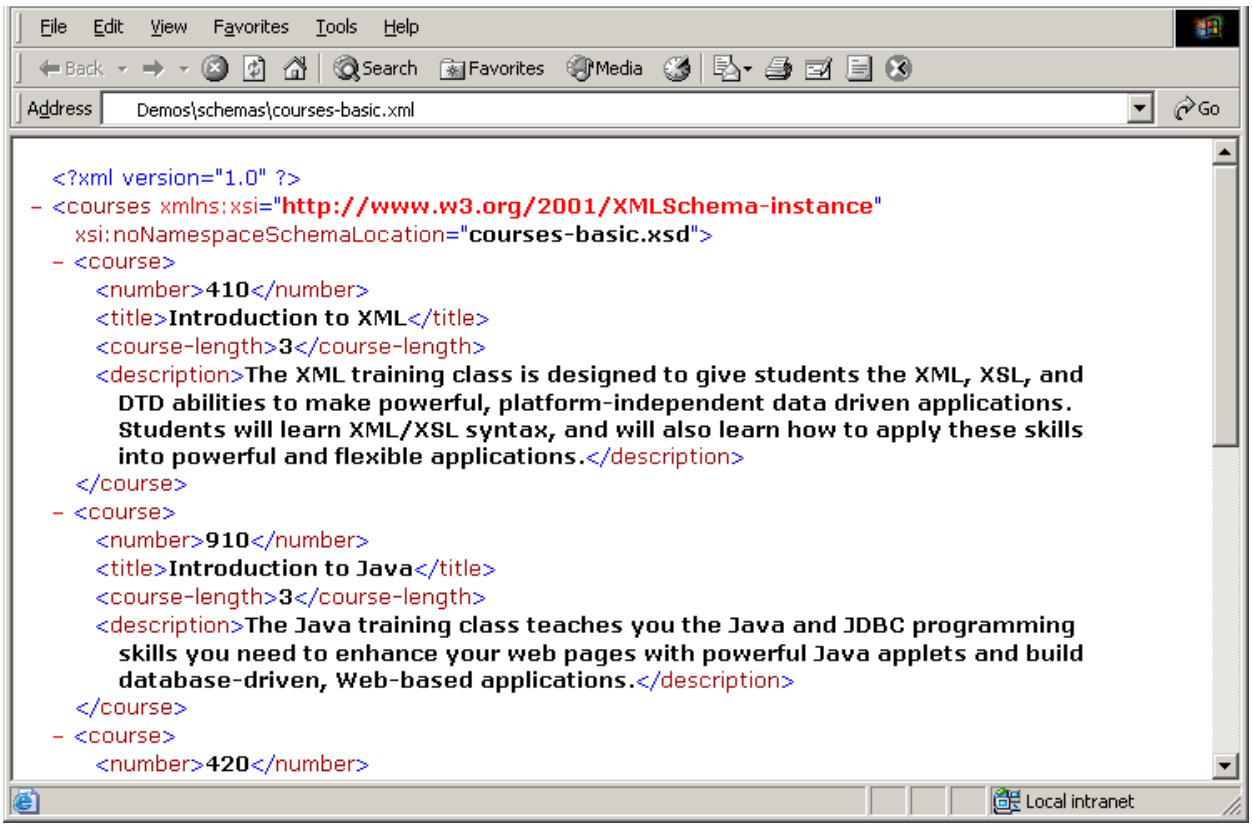
3. **DTDs do not allow you to specify a list of options or a default for element values**. While you have relatively robust options for specifying attribute values with DTDs, you do not have the same power with elements.

4. **DTDs are not written in XML-compatible syntax**. While, once you've learned it, DTD syntax is not that complex, it is still strange when evaluated by XML standards. One of the working theories of the W3C is that all XML-family languages should themselves be XML-compatible.

You may wonder why DTDs were even adopted by the W3C. The principal advantage that DTDs had was that they were an already-settled standard, which was acceptable to the various members of the XML Working Group while they worked out something better.

## Referencing An XML Schema

XML Schema documents are files with an .**xsd** extension. They serve a similar (though much more robust) function to the DTD: to specify the contents and structure of an XML document. We'll look at a complete Schema document, and then break the creation of the schema (which can be rather daunting at first) into several stages. We will be using essentially the same XML data as before (with our course list). A simplified version of the file, complete with its call to its schema, can be found in **demos > schemas > courses-basic.xml**:




You will notice two attributes for your top-level **courses** tag. The first (`xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`) defines a namespace that documents the meanings of a subset of XML tags. We prefix these tags with **xsi** to differentiate them from other tags in this file. The second (`xsi:noNamespaceSchemaLocation="schema.xsd"`) uses one of those tags to specify the location of the schema document. This is our first real-world encounter with a **namespace declaration**. The namespace used here was developed by the W3C to provide for the functionality of referencing schemas and a few other things that we might want to add to an "instance" document (i.e. a particular implementation of a schema).

## The "xsi" Namespace

The **xsi** namespace stands for XML Schema Instance. An *instance* is an XML document that should conform to a particular schema.

Tags within this namespace are used to reference the schema appropriate to an XML document. The namespace declaration is below:

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

In this case, we are saying that we will prefix any tags from this namespace with the letters "xsi". You may use some other string of letters to identify the above namespace. For instance, the following would be perfectly valid:

```
xmlns:apple="http://www.w3.org/2001/XMLSchema-instance"
```

Of course, this would mean that all tags that are part of the http://www.w3.org/2001/XMLSchema-instance namespace have to be referenced with the prefix apple. In practice, everyone uses "xsi" and there's no good reason not to do so.

Once you've declared the namespace, you can then make use of the tags defined in that namespace. In our example, the second attribute in the root node is an attribute within the XML Schema Instance namespace that specifies the location of the schema document.

```
xsi:noNamespaceSchemaLocation="schema.xsd"
```

**noNamespaceSchemaLocation** means that the schema to which we are referring does not declare its elements to be part of a particular namespace. For more on namespaces and advanced schema topics, see WestLake Internet Training's *Introduction to XML Schema* course.

## An XML Schema Document

The schema file is located in the same directory, and is named **courses-basic.xsd**. Its code is below, and we've highlighted the top-level **xsd:schema** tag in **bold**. Note that the root tag itself contains the call to a namespace, in this case the **xsd** (XML Schema Document) namespace (again, you can use any prefix you like – many people use xs rather than xsd), which will be used for all the tags in the schema:

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

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:annotation>
   <xsd:documentation>This schema is designed to validate lists of
courses taught by WestLake Internet Training</xsd:documentation>
</xsd:annotation>

<xsd:element name="courses">
   <xsd:complexType>
   <xsd:sequence>
   <xsd:element name="course" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="number" type="xsd:integer" />
            <xsd:element name="title" type="xsd:string" />
            <xsd:element name="course-length" type="xsd:integer" />
            <xsd:element name="description" minOccurs="0" />
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   </xsd:sequence>
```

```
   </xsd:complexType>
</xsd:element>
</xsd:schema>
```

You will notice that within this (relatively long) code, there are declarations of elements (**xsd:element**) as well as element types (**xsd:complexType**). These are just some of the facets of a schema that we can declare. We will introduce more in the upcoming sections.

## Beginning a Schema Document

The initial stages of a schema document should define the basic elements of the XML data that you're working with, as well as set up the structure of the XSD document itself. Take another look at the xml and schema files we've already mentioned (**courses-basic.xml** and **courses-basic.xsd**). You'll see that we first start by declaring the schema:

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

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
.
.
.
</xsd:schema>
```

You will note that as schemas are valid XML documents, they must begin with the XML language declaration. Following that, their top-level tag must be **xsd:schema**, complete with namespace declaration. Note that the namespace declaration is case-sensitive.

### xsd:annotation and xsd:documentation

Schemas can contain annotation at the beginning of the document and inside just about any other schema element. They are used to provide descriptions of various parts of the schema or of the schema itself, and can be used to make the file easier to read or even to build help files and more thorough documentation. This is done with the two tags **xsd:annotation** and **xsd:documentation**.

```
<xsd:annotation>
   <xsd:documentation>This schema is designed to validate lists of
courses taught by WestLake Internet Training</xsd:documentation>
</xsd:annotation>
```

The documentation tag provides a human-legible description or note. There is another child of <xsd:annotation> called <xsd:appInfo>. This tag is used to provide information to a consuming application.

With these basics in mind, let's look at the element declarations in our simple schema (**Demos\schemas\courses-basic.xsd**) and on the previous page.

## xsd:element

This is used to declare that an element should appear. It also stipulates what the name of the element should be. You may also use the minOccurs and maxOccurs attributes of the xsd:element declaration to specify how many times an element may appear. The default value for both attributes is 1. You may change either one to any integer (as long as the min is less than the max), or to the text "unbounded" if you want to allow an unlimited number of occurrences of a particular node.

## Complex and Simple Types

All elements must be declared as simple or complex. An element is "complex" if it contains subnodes and/or attributes. If this is the case, you must declare it as **xsd:complexType**. There are several possible options after you have declared it as such:

- **<xsd:all>:** the order of sub elements does not matter, but sub-elements may not occur more than once;
- **<xsd:sequence>**: the order of sub elements must be the same in the instance document as in the schema. You may specify whatever number of times you want to allow those elements to appear;
- **<xsd:choice>**: there are more than one possible combinations of subnodes that can be contained in the element;
- **<xsd:simpleContent>**: the element has no subnodes, but it has attributes.

If it is a **simple type** (i.e. it does not contain subnodes or attributes), you can add a type attribute to declare what type of data it must contain. There are 44 such types built into XML Schema.

```
<xsd:element name="number" type="xsd:integer" />
<xsd:element name="title" type="xsd:string" />
```

The other data types are located at the following url as of the writing of this book: http://www.w3.org/TR/xmlschema-2/#built-in-datatypes. If you don't find the types there, go to http://www.w3.org/XML/Schema, and look for the schema specifications. Follow the links to the data types section. Here are some common types:

| Data Type | Description |
|---|---|
| xsd:string | Any string |
| xsd:boolean | Values of *true* or *false (* OR *1 or 0)* |
| xsd:anyURI | Any valid URL or other URI reference |
| xsd:decimal | Any decimal number, positive or negative (including whole numbers) |
| xsd:float | Any valid floating point number |

| | |
|---|---|
| xsd:integer | Any valid positive or negative integer |
| xsd:date | Any date, in the format *YYYY-MM-DD* |
| xsd:time | Any time, in the format *HH:MM:SS*, where HH is the 24-hour time or HH:MM:SS-05:00 where the –05:00 is the timezone offset. |

Often the 44 built-in types will not validate your data adequately. We will deal with that issue in an upcoming section.

## Mixed Content

There is a way to allow content to be included inside of nodes that contain sub-nodes. In other words, the following is valid xml:

```
<letter>
   <paragraph>I read your letter concerning giving me
<amount>1000000</amount> <monetary_unit>dollars</monetary_unit>. I
am interested. Please send a <payment-form>check</payment-form>.
   </paragraph>
   <signoff>Love,<signoff>
   <author>George</author>
</letter>
```

How would you validate `<paragraph>`? It contains text *and* subnodes. Use

```
<xsd:element name="paragraph">
   <xsd:complexType mixed="true">
      <xsd:all>
         <xsd:element name="amount" type="xsd:integer" />
         <xsd:element name="monetary_unit" type="xsd:string" />
         <xsd:element name="payment-form" type="xsd:string" />
      </xsd:all>
   </xsd:complexType>
</xsd:element>
```

## Validating Against A Schema

We have provided an html page with a javascript function that allows you to choose an xml file and validate it with your schema. It is called wrapper-validate-schema.html and there's a copy in your **Demos/schemas** folder as well as in your **Exercises/Exercises-Schemas** folder. NOTE: You must enter the URL of your XML file, NOT the schema document. Your XML file has a reference to the schema built in.

This page is designed to work with Internet Explorer. However, version 6 and lower of IE don't come with a W3C-schema aware parser. If you open the file and you don't have the right parser, you will receive an error message telling you to download it. It's free!

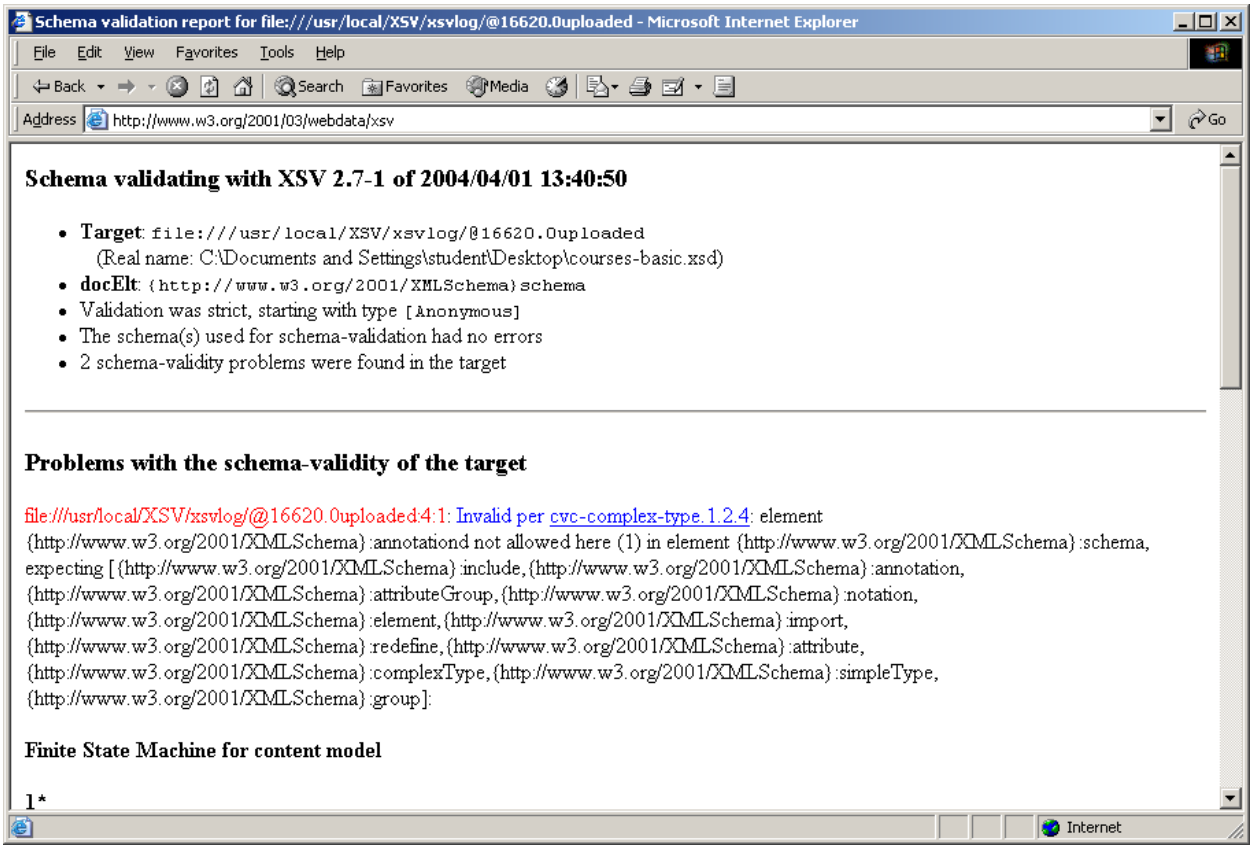
If it gives you an error message telling you that the root node (i.e. courses, actors, or whatever your root node is named) is used but not declared, that generally means you have an error in your schema. That raises the question:

## How Can I Check My Schema?

The first thing you can do is open your schema in a browser. This will show you any well-formedness errors right away.

If your schema is well-formed and you're still getting errors, you probably have a syntax error. Syntax errors can be tricky to find. One easy way to find them is by going to the W3C's XML Schema Validator tool, which they call XSV. Currently, it's at http://www.w3.org/2001/03/webdata/xsv, but url's can change. You may want to click on the XMLSchema Link from the main W3C web page, then look for tools, then XSV. Or just do a search on XSV from the main page of their site.

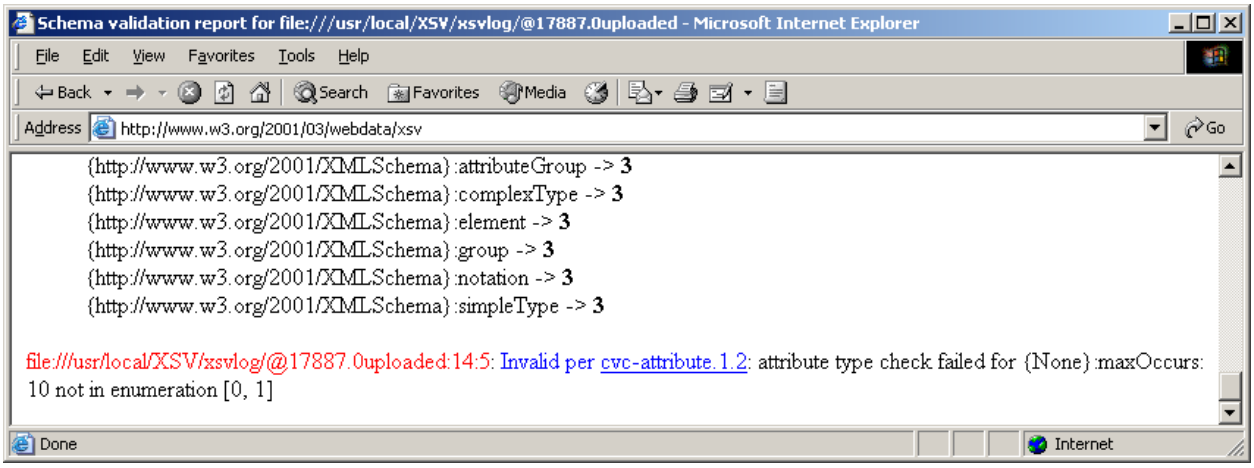
There is a version you can download and run off the command line. Or you can use the online version which gives you web-form access. You simply upload your schema (or provide an http link to it if it is publicly accessible). It will give you a result that looks something like this:




The last line of the first section says the uploaded schema has two errors. In the *Problems with the schema-validity of the target* section, the first list tells we where the error is. In this case, it's line 4

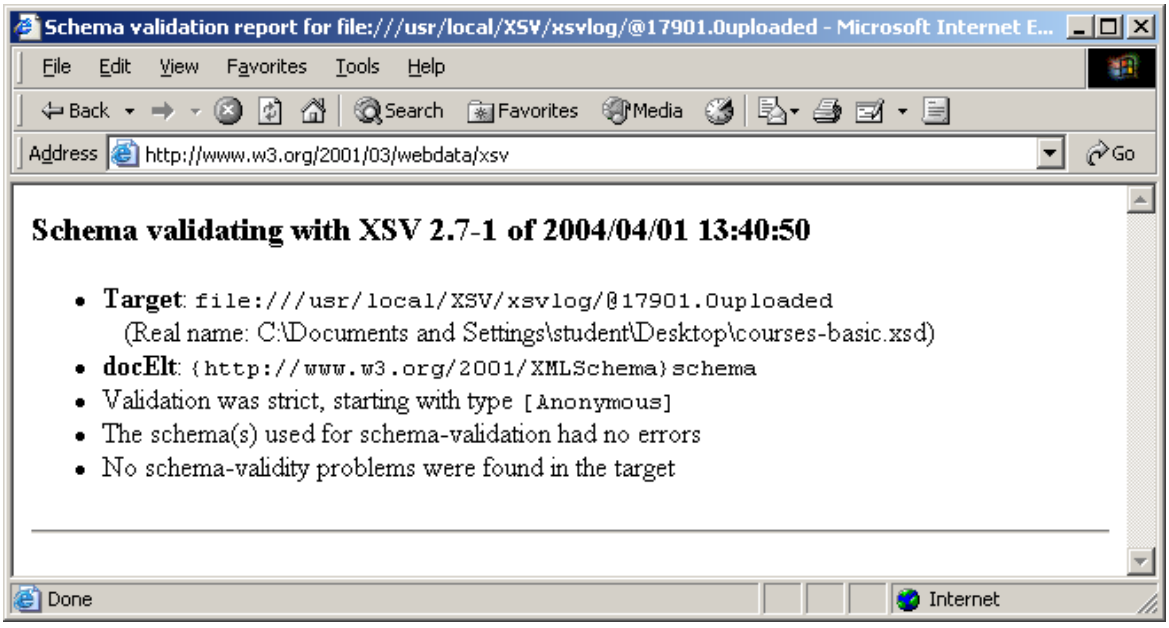
(see where it says: file:///usr/local/XSV/xsvlog/@16620.0uploaded:**4**:1). It's telling us that the element "annotationd" is not allowed. It lists the permissible elements in this context afterwards.

Scrolling down reveals the second error:




This error tells us that on line 14, we used an invalid value for the attribute maxOccurs. The schema specification says we can only use values 0 or 1, but our schema said 10.

If we fix these errors, we get:



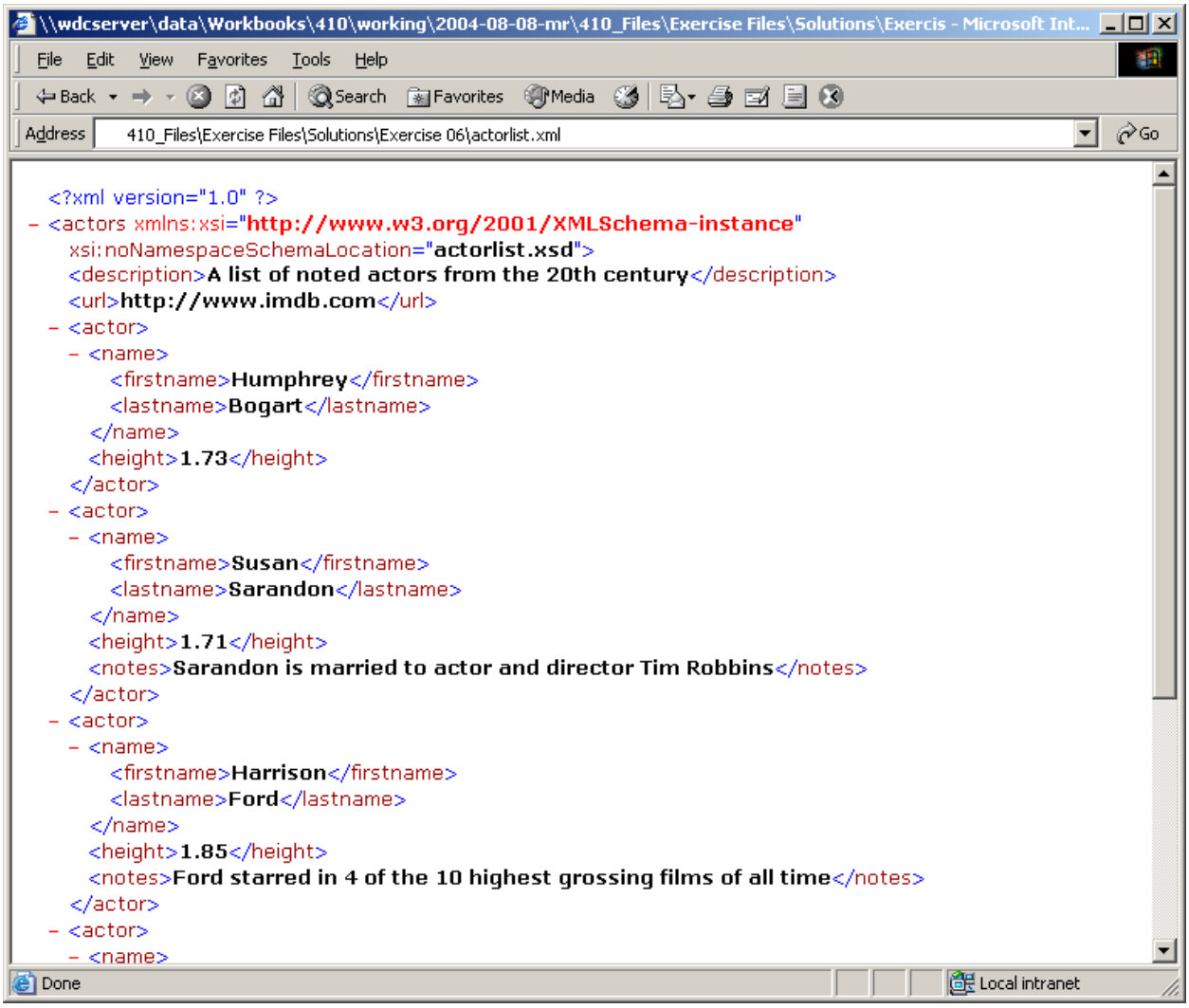

There are some further errors in logic and naming that can occur, but since they aren't syntax errors, the schema checker won't find them. If the W3C says your schema is valid but things still don't work, you have to either hunt and peck for the answer or download one of the many

software programs that allow you to work with schemas, such as XML Spy, Turbo XML, XML Writer, and many more. Some of these programs allow free trials.

## Exercise 6: Beginning an XML Schema Document

In this exercise, you will be beginning an XML Schema document to describe your list of actors. We have provided a simplified actorlist for this first exercise.

The exercise will involve adding the appropriate call to your schema from your XML data file (your "instance" document), and then creating a schema that matches. When you are done, your XML document should look as follows (note the namespace declaration and schema location at the top of the XML datasheet):




To complete this exercise, please do the following:

1. Open the **Exercises-Schemas\Basic** folder. Open the file **actorlist.xml** in a text editor. Notice that the node structure is simplified. We have removed the films and all attributes. Don't worry, we'll add them later.

2. Add the appropriate namespace declaration and reference to a schema file named **actorlist.xsd**.

3. Open the file **actorlist.xsd** from the same folder. You'll notice that the XML declaration and the root schema node (including the namespace) are already there.

4. In that file, begin your Schema. Remember to include the following:

    - An **xsd:annotation** and **xsd:documentation** set that describe the function of your Schema file.

    - **xsd:element** declarations for each of your elements, including their **name**s.

    - You'll need to declare the objects as complexType elements if they have subnodes or attributes.

    - For those nodes that just contain data, add the **type** attribute:

        - **xsd:string** will work for **firstname**, **lastname** and **description**;

        - **xsd:decimal** is appropriate for the **height** node;

        - **xsd:anyURI** is the best type for the **url** element.

5. Save your file.

6. When you are done, open **Exercise Files\Exercises-Schemas\wrapper_validate_schema.html** in Internet Explorer. Browse to the file **Exercise Files\Exercises-Schemas\Basic\actorlist.xml**. If you get an error telling you you don't have the right parser, go to Microsoft's website and download and install it.

7. If you find an error, correct it, save, and re-test.

8. You may need to consult the section entitled *Validating Against a Schema* (see above) for some help in debugging your schema.

## If you are done early...

- Experiment with xsd:choice and xsd:all.

- Experiment with some other data types.

## A Possible Solution to Exercise 6

As contained in **actorlist.xml**:

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

<actors xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="actorlist.xsd">
   <description>A list of noted actors from the 20th
century</description>
   <url>http://www.imdb.com</url>
   <actor>
      <name>
         <firstname>Humphrey</firstname>
         <lastname>Bogart</lastname>
      </name>
      <films>
         <film>
            <title>Casablanca</title>
            <date>1942</date>
         </film>
         <film>
            <title>The African Queen</title>
            <date>1951</date>
         </film>
         <film>
            <title>The Caine Mutiny</title>
            <date>1955</date>
         </film>
      </films>
      <height>1.73</height>
   </actor>
   .
   .
   [other actor elements omitted]
   .
   .
</actors>
```

And as contained in **actorlist.xsd**:

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

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:annotation>
      <xsd:documentation>This schema is designed to validate lists of
actors and their films</xsd:documentation>
   </xsd:annotation>

   <xsd:element name="actors">
     <xsd:complexType>
      <xsd:sequence>
```

```
          <xsd:element name="description" type="xsd:string" />
          <xsd:element name="url" type="xsd:anyURI" />
          <xsd:element name="actor" maxOccurs="unbounded">
            <xsd:complexType>
             <xsd:sequence>
              <xsd:element name="name">
                  <xsd:complexType>
                    <xsd:sequence>
                       <xsd:element name="firstname" type="xsd:string" />
                       <xsd:element name="lastname" type="xsd:string" />
                    </xsd:sequence>
                  </xsd:complexType>
              </xsd:element>
              <xsd:element name="height" type="xsd:decimal" />
              <xsd:element name="notes" minOccurs="0" type="xsd:string" />
             </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

## Declaring Attributes

Your schema can also allow you to specify attributes. This is done in one of two ways, depending on whether the attribute is attached to a node that has subnodes or one that only has text.

### Adding an attribute to a node with subnodes

It's pretty easy to add an attribute to a node that has subnodes. Let us suppose that you want a node in your XML document to look like this:

```
<course subject="XML">
  <number>410</number>
  <title>Introduction to XML</title>
  <course-length>3</course-length>
</course>
```

We already know what to do with the course, number, title and course-length nodes. But what about that subject attribute? Here you go:

```
<xsd:element name="course" minOccurs="0" maxOccurs="unbounded">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="number" type="xsd:integer" />
         <xsd:element name="title" type="xsd:string" />
         <xsd:element name="course-length" type="xsd:string" />
      </xsd:sequence>
      <xsd:attribute name="subject" type="xsd:string" />
   </xsd:complexType>
</xsd:element>
```

The attribute gets sandwiched between the closing sequence element and the closing complexType element. This makes sense if you think about the definition of a complexType: a node with subnodes and/or attributes. So the attribute is part of what makes course complex, but it's not part of the sequence of subnodes.

### Adding an attribute to a node with no subnodes

What if we wanted to tweak the XML code above to add a "scale" attribute to the course-length node mentioned above:

```
<course subject="XML">
  <number>410</number>
  <title>Introduction to XML</title>
  <course-length scale="days">3</course-length>
</course>
```

<course-length> is now a complexType, but it doesn't make sense to use xsd:all, xsd:sequence or xsd:choice, since there are no subnodes. So we declare it a complexType with **simpleContent** (this code can be found in **Demos/schemas/ courses-basic-attribute.xsd**):

```
<xsd:element name="course" minOccurs="0" maxOccurs="unbounded">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="number" type="xsd:integer" />
         <xsd:element name="title" type="xsd:string" />
         <xsd:element name="course-length">
            <xsd:complexType>
               <xsd:simpleContent>
                  <xsd:extension base="xsd:integer">
                     <xsd:attribute name="scale" type="xsd:string" />
                  </xsd:extension>
               <xsd:simpleContent>
            <xsd:complexType>
         </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="subject" type="xsd:string" />
   </xsd:complexType>
</xsd:element>
```

The xsd:extension element declares the content of the course-length node itself. The xsd:attribute declares the content of the scale attribute.

## Other attribute stuff

Attributes can take several other qualifiers. The ***default*** attribute allows you to specify a default value if the XML value does not include one.

```
<xsd:attribute name="scale" type="xsd:string" default="days" />
```

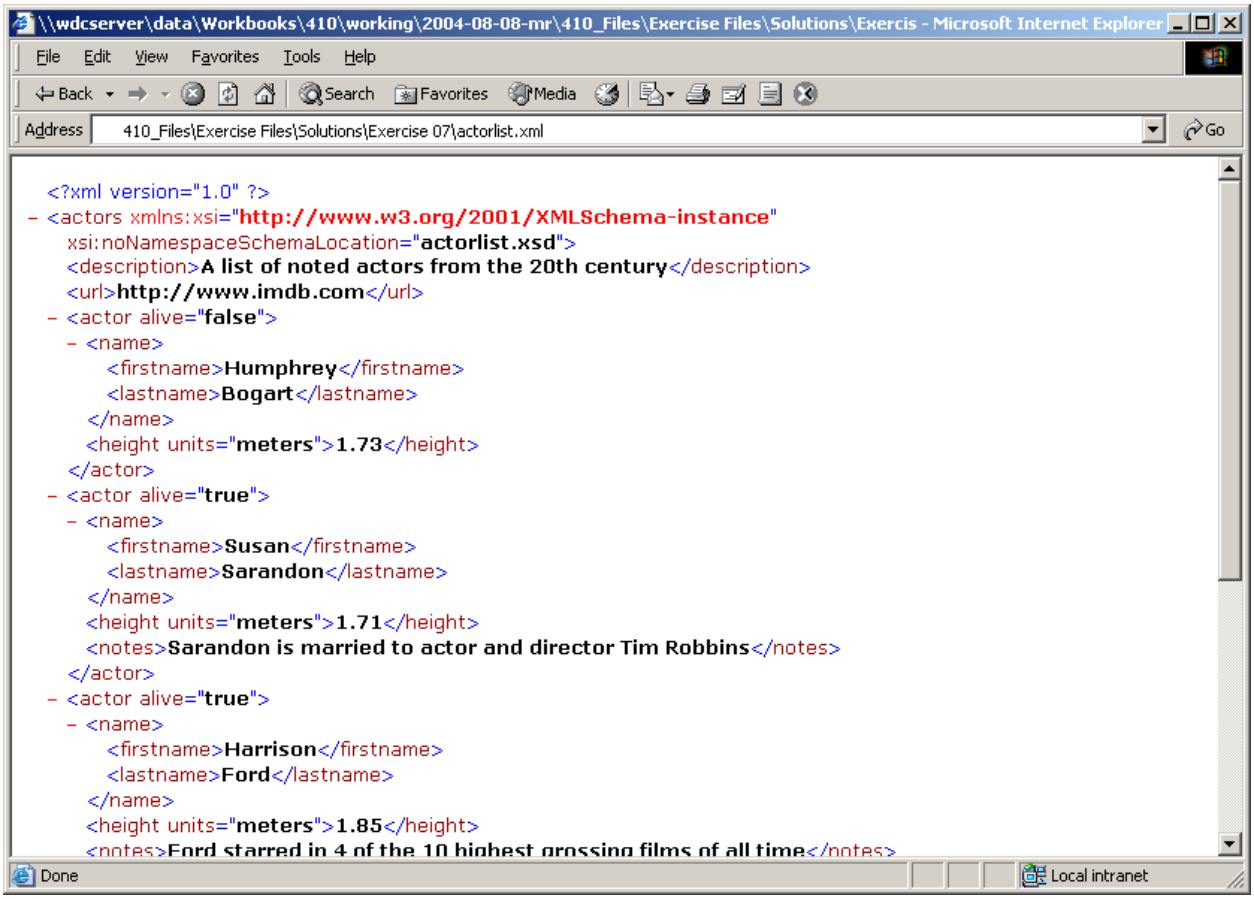
The ***use*** attribute allows you to specify whether the attribute is **optional, prohibited,** or **required:**

```
<xsd:attribute name="scale" type="xsd:string" use="required" />
```

It's exercise time.

## Exercise 7: Adding Attributes to Your Schema

In this exercise, you will be improving your XSD to specify the attributes in the XML instance. You will add the attribute **alive** to all your actor nodes (with possible values of true or false, i.e. **xsd:boolean**), and the attribute **units** to all the height nodes.




To complete this exercise, please do the following:

1. Open **Exercise Files\Exercises-Schemas\Attributes\actorlist.xsd**.

2. Add code to allow for the boolean (true/false, 1/0) value alive on the actor nodes. Make this attribute required.

3. Add code to allow for the string value **units** on the height nodes. It should have a default value of "meters" and the height node should contain a decimal.

4. Save your file, and run **Exercise Files\Exercises-Schemas\Attributes\actorlist.xml** through the validator located at **Exercise Files\Exercises-Schemas\wrapper_validate_schema.html**. If you find errors, fix them, save, and re-test. If you don't find errors, see if the schema checker finds invalid values (for example, it should

not let you put a string of text in the height node, or a value of "alive and kicking" in the alive attribute).

## If you are done early...

- Try adding another attribute somewhere. Perhaps you could make an optional attribute on name specifying whether this is the actor's given name at birth.

## A Possible Solution to Exercise 7

As contained in **actorlist.xsd**:

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

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:annotation>
      <xsd:documentation>This schema is designed to validate lists of
actors and their films</xsd:documentation>
   </xsd:annotation>

   <xsd:element name="actors">
     <xsd:complexType>
       <xsd:sequence>
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="url" type="xsd:anyURI" />
      <xsd:element name="actor" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
           <xsd:element name="name">
            <xsd:complexType>
             <xsd:sequence>
              <xsd:element name="firstname" type="xsd:string" />
              <xsd:element name="lastname" type="xsd:string" />
             </xsd:sequence>
            </xsd:complexType>
           </xsd:element>
           <xsd:element name="height">
            <xsd:complexType>
             <xsd:simpleContent>
              <xsd:extension base="xsd:decimal">
                <xsd:attribute name="units" default="meters"
type="xsd:string" />
              </xsd:extension>
             </xsd:simpleContent>
            </xsd:complexType>
           </xsd:element>
           <xsd:element name="notes" minOccurs="0" type="xsd:string" />
         </xsd:sequence>
         <xsd:attribute name="alive" use="required" type="xsd:boolean" />
        </xsd:complexType>
       </xsd:element>
      </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

## Using References

Our schema is starting to get a little unwieldy, even though the instance it is describing has a fairly simple structure. Luckily, the folks at the W3C have made our life a little easier with something called references. References allow us to break our schema into bite-sized chunks. The pieces of code then call one another so it all comes out in the right order. Take a look at our code from the previous demonstration, this time redone using references (see **Demos/schemas/courses-basic-attribute-ref.xsd**).

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

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:annotation>
   <xsd:documentation>This schema is designed to validate lists of
courses taught by WestLake Internet Training</xsd:documentation>
</xsd:annotation>

<xsd:element name="courses">
   <xsd:complexType>
     <xsd:sequence>
      <xsd:element ref="course" minOccurs="0" maxOccurs="unbounded" />
     </xsd:sequence>
   </xsd:complexType>
</xsd:element>

<xsd:element name="course">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="number" type="xsd:integer" />
         <xsd:element name="title" type="xsd:string" />
         <xsd:element ref="course-length" />
         <xsd:element name="description" minOccurs="0" />
      </xsd:sequence>
      <xsd:attribute name="subject" type="xsd:string" />
   </xsd:complexType>
</xsd:element>

<xsd:element name="course-length">
   <xsd:complexType>
     <xsd:simpleContent>
      <xsd:extension base="xsd:integer">
         <xsd:attribute name="scale" type="xsd:string" />
      </xsd:extension>
     </xsd:simpleContent>
   </xsd:complexType>
</xsd:element>

</xsd:schema>
```

OK, OK. So it's still not exactly beach reading. But it does allow us to avoid some of that nasty nesting we did in the last exercise. Each *element or attribute* declaration can take a ref attribute in

place of the name attribute. This then points to a stand-alone declaration of an element or attribute by that name. These stand-alone element declarations are technically known as "global" declarations whereas the nested element declarations are known as "local" declarations.

In the example above, the course element tag *refers* to an element declaration by the same name. The same is true of the course-length reference. Now our code is broken down into manageable pieces. This will be especially important as our schema gets more complex. For our full course list (code is below, as well as in **Demos/schemas/courses-complete-ref.xsd)** you will see that it is long, but each piece is relatively simple:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation>This schema is designed to validate lists of courses
taught by WestLake Internet Training</xsd:documentation>
  </xsd:annotation>
  <xsd:element name="courses">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="author" type="xsd:string"/>
        <xsd:element ref="course" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="course">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="number" type="xsd:integer" />
        <xsd:element name="title" type="xsd:string"/>
        <xsd:element ref="topics"/>
        <xsd:element ref="course-length"/>
        <xsd:element ref="location"/>
        <xsd:element name="description" minOccurs="0" type="xsd:string" />
      </xsd:sequence>
      <xsd:attribute name="subject" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="topics">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="topic" type="xsd:string" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="course-length">
    <xsd:complexType>
      <xsd:simpleContent>
        <xsd:extension base="xsd:integer">
          <xsd:attribute name="scale" default="days" type="xsd:string" />
```

```
        </xsd:extension>
      </xsd:simpleContent>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="location">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="state" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

</xsd:schema>
```

Again, any nodes that will present complex structure are deferred until later, when we can define them by themselves, while simple nodes like "title" can be taken care of inline.

As you can see, we reduce the complexity of our document. Instead of constantly having to nest more and more layers inside of each other, we can go just a few levels deep. We then refer to a schema fragment somewhere else and move on.

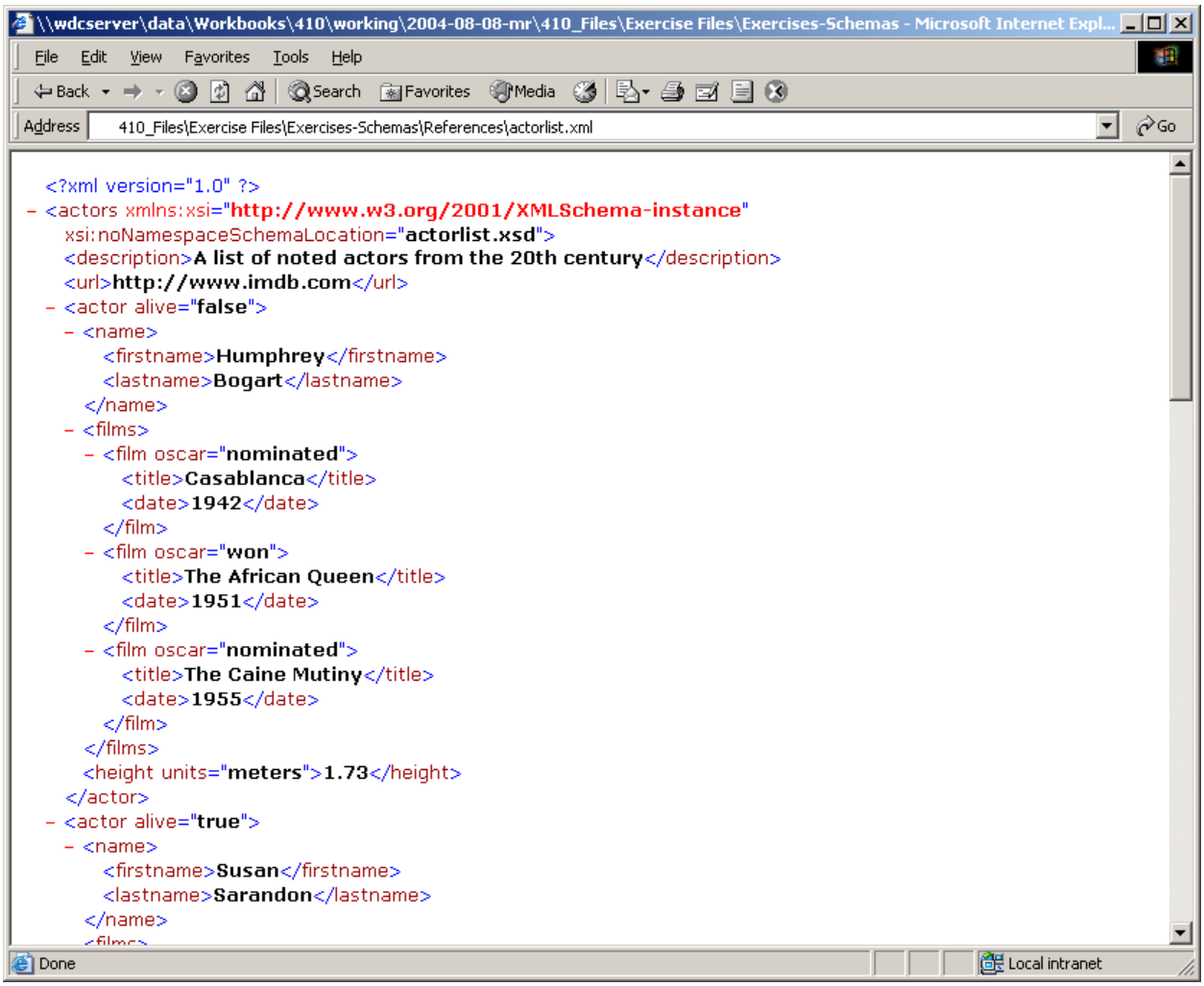
Furthermore, there is a way to include xml fragments from one schema file into another. So you can see that you might be able to build a library of common validation elements and simply include them. For more on including files and XML libraries, look into WestLake's *Advanced XML and XSL* class and *Introduction to XML Schema* classes. But for now, just reducing the complexity of our file is well worth it.

## maxOccurs, minOccurs, type

When an element takes a ref attribute, you need to be aware of how that affects the attributes it can take. You'll notice in the code above that the minOccurs and maxOccurs attributes still appear inside the **<xsd:element>** tag that refers to the full declaration. It is illegal, however, to include a **type** attribute on an element that has the **ref** attribute. The type should be defined in the declaration to which you are referring.

## Exercise 8: Using References to Simplify Your Schema

In this exercise, you will be improving your schema document by adding references. This will make it much easier to read and work with, which will be especially useful since we are also going to be adding some more nodes to our file. And in the next exercise we are going to add even more complexity to our schemas, so it behooves us to clean things up sooner rather than later.




To complete this exercise, please do the following:

1. Open **actorlist.xml** from your **Exercise Files\Exercises-Schemas\References** folder. You will notice that each actor now has a films node, which in turn has several film subnodes, which in turn have title and date subnodes.

2. Open **actorlist.xsd** from your **Exercise Files\Exercises-Schemas\References** folder. You will see that it looks like your completed schema from the last exercise.

3. Create references and corresponding standalone element declarations to simplify your code. You should create references for any node that is not a simple type: **actors**, **actor**, **name**, and **height**. We have also made **notes**, **oscar**, and **units** into references, since they will start getting more complex in the next exercise. This will mostly be an exercise in copying and pasting.

4. Now add a reference to **films** (as a subnode of actor) and a corresponding element declaration.

5. The **films** declaration should include a reference to a **film** element declaration.

6. The **film** declaration will contain local definitions of **title** and **date**, as well as a declaration of an **oscar** attribute, which should take a string.

7. Save your file.

8. Validate.

9. If the validation is successful, put an error into your xml data sheet to see if the validation page catches it.

## If you are done early...

- That was enough, wasn't it?

## A Possible Solution to Exercise 8

As contained in **actorlist.xsd**:

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

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:annotation>
      <xsd:documentation>This schema is designed to validate lists of
actors and their films</xsd:documentation>
   </xsd:annotation>
   <xsd:element name="actors">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="description" type="xsd:string"/>
            <xsd:element name="url" type="xsd:anyURI"/>
            <xsd:element ref="actor" maxOccurs="unbounded"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="actor">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="name"/>
            <xsd:element ref="films"/>
            <xsd:element ref="height"/>
            <xsd:element ref="notes" minOccurs="0"/>
         </xsd:sequence>
         <xsd:attribute name="alive" type="xsd:boolean" use="required" />
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="name">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="firstname" type="xsd:string"/>
            <xsd:element name="lastname" type="xsd:string"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="films">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="film" maxOccurs="unbounded"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="film">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="title" type="xsd:string"/>
            <xsd:element name="date" type="xsd:integer"/>
         </xsd:sequence>
```

```
            <xsd:attribute ref="oscar"/>
         </xsd:complexType>
      </xsd:element>
      <xsd:attribute name="oscar" default="no" type="xsd:string" />
      <xsd:element name="height">
         <xsd:complexType>
            <xsd:simpleContent>
               <xsd:extension base="xsd:decimal">
                  <xsd:attribute ref="units"/>
               </xsd:extension>
            </xsd:simpleContent>
         </xsd:complexType>
      </xsd:element>
      <xsd:attribute name="units" type="xsd:string" />
      <xsd:element name="notes" type="xsd:string" />
</xsd:schema>
```

## Restricting Content with Schemas

In addition to specifying data types and document structure with your schemas, XML Schema provides "facets," which allow very precise control over the content of nodes and attributes. The types of things that you can specify include the following:

- Minimum values, maximum values, and formats of numbers
- Enumerated options (such as you could with attributes in DTDs)
- Specific patterns, using regular expression syntax
- Length of strings (i.e. 500 characters, more than 3 characters, fewer than 3 caracters, between 2 and 3 characters, etc.)

Take a look at the next installment of our demo (**Demos\schemas\courses-complete-ref-restriction.xsd**), with new simpleType content restrictions highlighted in **bold**:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:annotation>
      <xsd:documentation>This schema is designed to validate lists of
courses taught by WestLake Internet Training</xsd:documentation>
   </xsd:annotation>
   <xsd:element name="courses">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="author" type="xsd:string"/>
            <xsd:element ref="course" maxOccurs="unbounded"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="course">
      <xsd:complexType>
         <xsd:all>
            <xsd:element ref="number"/>
            <xsd:element name="title" type="xsd:string"/>
            <xsd:element ref="topics"/>
            <xsd:element ref="course-length"/>
            <xsd:element ref="location"/>
            <xsd:element ref="description" minOccurs="0"/>
         </xsd:all>
         <xsd:attribute name="subject" type="xsd:string"/>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="number">
      <xsd:simpleType>
         <xsd:restriction base="xsd:integer">
            <xsd:minInclusive value="100"/>
            <xsd:maxInclusive value="999"/>
         </xsd:restriction>
```

```
      </xsd:simpleType>
   </xsd:element>
   <xsd:element name="topics">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="topic" type="xsd:string"
maxOccurs="unbounded"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="course-length">
      <xsd:complexType>
         <xsd:simpleContent>
            <xsd:extension base="xsd:integer">
               <xsd:attribute ref="scale" />
            </xsd:extension>
         </xsd:simpleContent>
      </xsd:complexType>
   </xsd:element>
   <xsd:attribute name="scale" default="days">
      <xsd:simpleType>
         <xsd:restriction base="xsd:string">
            <xsd:enumeration value="days"/>
            <xsd:enumeration value="hours"/>
            <xsd:enumeration value="weeks"/>
         </xsd:restriction>
      </xsd:simpleType>
   </xsd:attribute>
   <xsd:element name="location">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="city" type="xsd:string"/>
            <xsd:element ref="state" />
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="state">
      <xsd:simpleType>
         <xsd:restriction base="xsd:string">
            <xsd:pattern value="[A-Z]{2}"/>
         </xsd:restriction>
      </xsd:simpleType>
   </xsd:element>
   <xsd:element name="description">
      <xsd:simpleType>
         <xsd:restriction base="xsd:string">
            <xsd:maxLength value="500"/>
         </xsd:restriction>
      </xsd:simpleType>
   </xsd:element>
</xsd:schema>
```

## Declaring restrictions with xsd:simpleType and xsd:restriction

When you want to specify something more specific than a data type for a leaf element, you need to explicitly declare that element as a simpleType. This is done using the tag **<xsd:simpleType>**. Within that simpleType declaration, you specify the base data type for your content using **xsd:restriction** (and its required attribute **base**).

The application that is checking the document against the schema will first check to see if the data matches the data type specified in the **xsd:restriction** tag's **base** attribute. If the data is valid according to the data type, it goes on to check it against the facet(s) you have specified.

Within the **xsd:restriction** block, you have a number of "facets" for specifying your content. The table below is the full list of W3C XML Schema options. All these facets take a "value" attribute which you use to specify how you want to constrain the simpleType.

| Schema Element | Description |
|---|---|
| xsd:minInclusive<br>xsd:maxInclusive<br>xsd:minExclusive<br>xsd:maxExclusive | Specifies the minimum or maximum value for elements of any numeric, date, or time data type. They may be used independently (so, a number with only an *xsd:minInclusive* attribute would have no maximum value) |
| xsd:enumeration | Specifies a legal value for the given element for any data type except *xsd:boolean*. |
| xsd:maxLength<br>xsd:minLength<br>xsd:length | Specifies the minimum or maximum number of characters of any string data type. Note that *xsd:length* specifies an exact string length required. |
| xsd:fractionDigits | Specifies the maximum number of decimal places in a decimal type |
| xsd:totalDigits | Specifies the maximum number of digits in a decimal type |
| xsd:pattern | Specifies a pattern using standard Perl/JavaScript regular expression syntax. Note that the pattern must always match the complete element content. Regular expressions are outside the scope of this course, but if you are interested, you may want to look at WestLake's JavaScript or Perl classes. |
| xsd:whitespace | *collapse*: trim leading and trailing whitespaces. Then replace multiple whitespaces with one space.<br>*replace*: tabs, line feeds, and carriage returns replaced by a single space.<br>*preserve*: leave whitespace as is. |

So a typical node will look like this:

```
<xsd:element name="number">
   <xsd:simpleType>
      <xsd:restriction base="xsd:integer">
         <xsd:minInclusive value="100" />
         <xsd:maxExclusive value="1000" />
      </xsd:restriction>
```

```
   </xsd:simpleType>
</xsd:element>
```

## Specifying Default Values

If you want to specify a default value for an element or attribute, you can do so, whether or not you otherwise restrict its content. Then, if the element/attribute is omitted in the XML, the default value will be used. Defaults are set using the **default** attribute of the **xsd:element** or **xsd:attribute** tags:

```
<xsd:attribute name="scale" default="days">
   <xsd:simpleType>
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="days" />
         <xsd:enumeration value="hours" />
         <xsd:enumeration value="weeks" />
      </xsd:restriction>
   </xsd:simpleType>
</xsd:attribute>
```

## Exercise 9: Restricting the Content of Elements and Attributes

In this exercise you will refine your schema so that it specifies very precisely what content is allowed in various nodes.

1. Open **Exercise Files\Exercises-Schemas\SimpleTypes\actorlist.xsd** in your favorite text editor.
2. Restrict the acceptable values of the oscar attribute to "won", "nominated" or "no".
3. Restrict the acceptable values of the units attribute to "meters", "feet" or "cm".
4. Limit the length of the notes element to 500 characters.

### If you are done early…

You may have noticed that, given the syntax you've learned thus far, it is not possible to use one of the facets on a text node with attributes. There simply isn't enough time in this course to cover everything! But here is how you would specify that a node like height should take a decimal that does not go to more than two decimal places. The code is in **Solutions/Exercise 09/actorlist-extra.xsd**.

```
<xsd:element name="height">
   <xsd:complexType>
      <xsd:simpleContent>
         <xsd:extension base="heightType">
            <xsd:attribute ref="units"/>
         </xsd:extension>
      </xsd:simpleContent>
   </xsd:complexType>
</xsd:element>
<xsd:simpleType name="heightType">
   <xsd:restriction base="xsd:decimal">
      <xsd:fractionDigits value="2" />
   </xsd:restriction>
</xsd:simpleType>
```

## A Possible Solution to Exercise 9

As contained in **actorlist.xsd**:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:annotation>
      <xsd:documentation>This schema is designed to validate lists of
actors and their films</xsd:documentation>
   </xsd:annotation>
   <xsd:element name="actors">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="description" type="xsd:string"/>
            <xsd:element name="url" type="xsd:anyURI"/>
            <xsd:element ref="actor" maxOccurs="unbounded"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="actor">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="name"/>
            <xsd:element ref="films"/>
            <xsd:element ref="height"/>
            <xsd:element ref="notes" minOccurs="0"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="name">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="firstname" type="xsd:string"/>
            <xsd:element name="lastname" type="xsd:string"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="films">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="film" maxOccurs="unbounded"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="film">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="title" type="xsd:string"/>
            <xsd:element name="date" type="xsd:integer"/>
         </xsd:sequence>
         <xsd:attribute ref="oscar"/>
      </xsd:complexType>
```

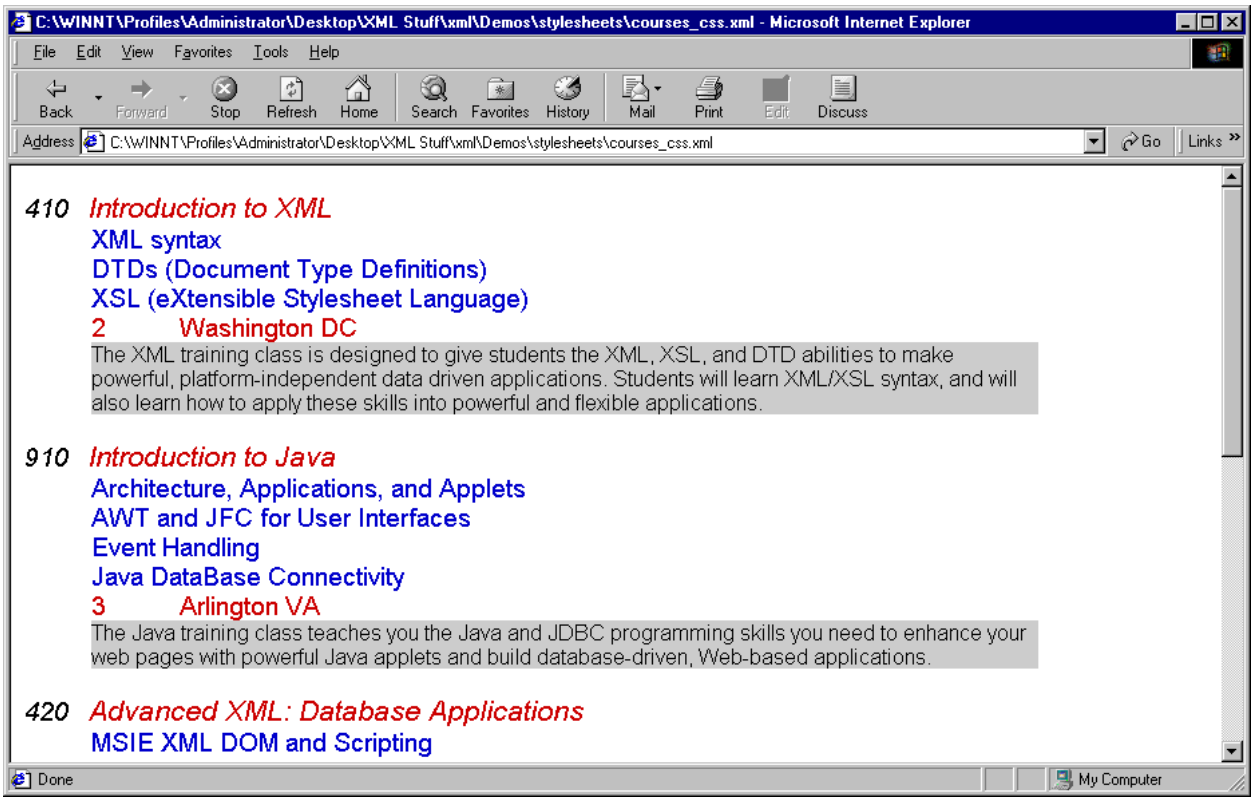
```
    </xsd:element>
    <xsd:attribute name="oscar" default="no">
       <xsd:simpleType>
          <xsd:restriction base="xsd:string">
             <xsd:enumeration value="no"/>
             <xsd:enumeration value="nominated"/>
             <xsd:enumeration value="won"/>
          </xsd:restriction>
       </xsd:simpleType>
    </xsd:attribute>
    <xsd:element name="height">
       <xsd:complexType>
          <xsd:simpleContent>
             <xsd:extension base="xsd:decimal">
                <xsd:attribute ref="units"/>
             </xsd:extension>
          </xsd:simpleContent>
       </xsd:complexType>
    </xsd:element>
    <xsd:attribute name="units">
       <xsd:simpleType>
          <xsd:restriction base="xsd:string">
             <xsd:enumeration value="meters"/>
             <xsd:enumeration value="cm"/>
             <xsd:enumeration value="feet"/>
          </xsd:restriction>
       </xsd:simpleType>
    </xsd:attribute>
    <xsd:element name="notes">
       <xsd:simpleType>
          <xsd:restriction base="xsd:string">
             <xsd:maxLength value="500"/>
          </xsd:restriction>
       </xsd:simpleType>
    </xsd:element>
</xsd:schema>
```

# Using Cascading Style Sheets (CSS) to Present XML Data

At the moment, we are viewing our XML data through the built-in stylesheet of the XML parser in Internet Explorer 5. The browser is not just displaying the code; you can tell that it is interpreting the display by its use of color coding and nested structure. The little pluses and minuses are the visual representation of that structure.

Of course, in the applications that you design, you will probably want to display your XML data differently than the default output in the browser. By using Cascading Style Sheets (CSS) you can dictate rules of display for your XML data.

To get us started, take a look at the next installment of our ongoing demo. The file can be found in **demos > stylesheets > courses_css.xml**:




You will note that you no longer see the code (even though you are viewing the XML file directly). Instead, you see an interpreted display. The course numbers are in italics. The course titles are red and italic. The course topics are blue. The description is smaller, with a gray background. All the sub-elements are indented from the course number. In addition, there are no tags in sight. Instead, the data has been parsed for display according to a custom stylesheet.

The code for the courses_css.xml is below, with the new line in **bold**:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="courses.css"?>

<!DOCTYPE courses SYSTEM "courses.dtd">

<courses>
   <author>Jason Haas</author>
   <course subject="XML">
      <number>410</number>
      <title>Introduction to XML</title>
      <topics>
         <topic>XML syntax</topic>
         <topic>DTDs (Document Type Definitions)</topic>
         <topic>XSL (eXtensible Stylesheet Language)</topic>
      </topics>
      <course-length scale="days">2</course-length>
      <location>
         <city>Washington</city>
         <state>DC</state>
      </location>
      <description>The XML training class is designed to give students
the XML, XSL, and DTD abilities to make powerful,
platform-independent data driven applications.  Students will learn
XML/XSL syntax, and will also learn how to apply these skills into
powerful and flexible applications.</description>
   </course>

.
[other courses deleted]
.

</courses>
```

To apply a stylesheet to an XML document, you need to add an **<?xml-stylesheet?>** tag. The tag is case sensitive, and requires two attributes. The first is the **type**, which is generally either **text/css** or **text/xsl**. The second is **href**, which specifies the path to the stylesheet. We will look first at CSS stylesheets, and a little later at the more powerful XSL stylesheets.

The code for the stylesheet referenced as **courses.css** is below:

```
author      {display:none}

course      {display: block;
            font-family: Arial;
            font-size: 14pt;
            margin-top: 20px}

number      {font-style:italic}

title       {color: #cc0000;
            font-style: italic;
```

```
            font-size: 16pt;
            position: relative;
            left: 10px}

topics      {position: relative;
            left: 50px;
            color: #0000cc}
topic       {display: block}

course-length  {color: #cc0000;
            position: relative;
            left: 50px}

location    {color: #cc0000;
            position: relative;
            left: 100px}
city        {}
state       {}

description {display: block;
            background-color: #cccccc;
            font-size: 12pt;
            position: relative;
            left: 50px;
            width: 80%}
```

## A Brief Review of CSS Rules

Cascading Style Sheet declarations are rules that a designer makes, which override the default display of some bit of Web content. When used with HTML, CSS is an extension of HTML display properties, and can be applied either to individual tags or as **classes**, which can then be applied flexibly, where the designer chooses. With XML, as with HTML, CSS can specify display characteristics for individual XML tags. However, since XML tags have no default display characteristics, CSS becomes the sole determinant of structure and style.

Cascading Style Sheet declarations are organized in the form **rule:declaration**, with separate rule-declaration pairs separated by semicolons. Spacing and case are irrelevant in the rules and declarations, although with XML, the case of the CSS tag must match the case of the XML tag. A full review of CSS properties and acceptable values is contained in **Appendix D**, but it's worth reviewing some of the most important for XML purposes below.

### Display

Possible display values are **block**, **inline** (the default), **list-item**, and **none**. **Block** display treats the data as an entire object, forces the content to begin on a new line, and allows you to provide formatting information such as margins and indent. **Inline** data is treated in the context of the surrounding page elements (and so will appear on the same line as any previous content). **List-item** data is given bullets, as in HTML lists (though it's not yet supported in IE). Finally, items with the display value **none** are not displayed at all.

## Position

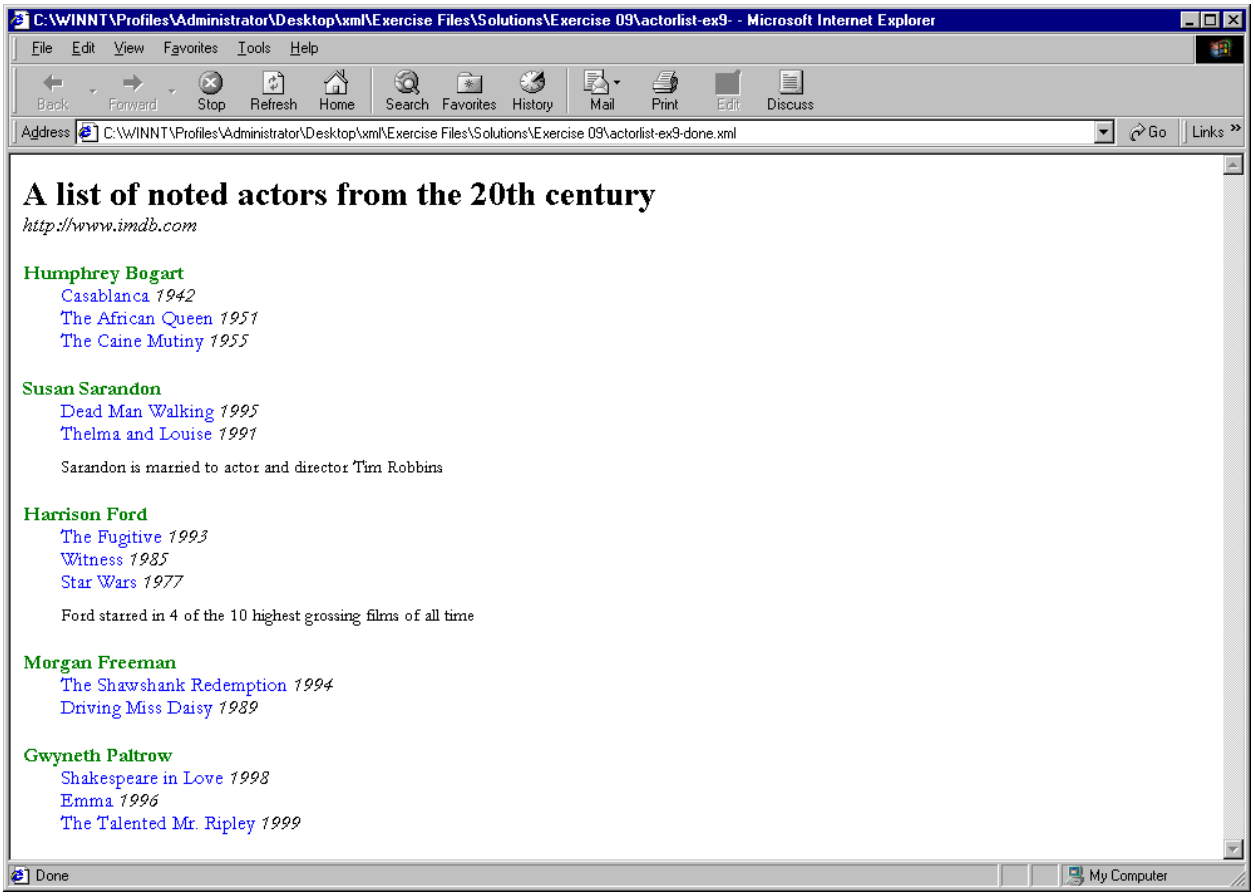
Possible values are **absolute**, **relative**, and **static** (the default). **Static** position treats positioning as it would be treated in HTML. So, any section of data follows the previous, and extra spaces are ignored. **Absolute** positioning is determined with respect to the upper left-hand corner of the screen. While crucial for Dynamic HTML, it is of limited value with XML, as your data will be iterative. In other words, you will likely have more than one actor, more than one film, more than one name. So, positioning all actors absolutely will probably cause them to overlap. **Relative** positioning is the most useful for XML. With it, you can specify that a section of data display a certain number of pixels from the static (default) position, using **top** and **left** values.

## Text-Indent

For block-level elements, you can specify a text-indent in pixels. Text-indent acts like relative positioning along the horizontal axis, and is often syntactically simpler to implement.

# Exercise 10: Displaying XML Data with a CSS Stylesheet

In this exercise, you will be building a CSS datasheet to display the XML data you have been building over the previous exercises. When you have completed the exercise, your page should look as follows:




To complete this exercise:

1. Open a blank document in Notepad.
2. In this document, build a custom stylesheet to simulate the display above. The specifications are as follows:
    - The **description** element should be 20pt, bold, and block.
    - The **url** element should be italic.
    - All **actor** elements should be block level and Garamond font, with a 20px margin at the top.

- All **name** elements should be green and bold.
- **Film** elements should be blue, on lines by themselves, and indented 30px, with their **date** black and italic next to them.
- **Note** elements should be smaller, indented 30px, and have a 10px margin at the top.
- **Height** should not be displayed.

3. Save that document in your **XML** folder as **styles.css**.
4. Re-open the file **actorlist.xml** from your **XML** folder in Notepad.
5. Add a line to **actorlist.xml** linking to the **styles.css** stylesheet.
6. Save **actorlist.xml**, and test it in Internet Explorer.
7. If you get an error message, go back into Notepad, edit your file, and reload.
8. Once you have it working, view the source of your file, and take a look at the differences from before.

## If you are done early…

- Add a background color to your notes field, or for some other elements in your display.
- Restructure your display so that each actor's movies appear on one line, below the actor's name.
- Now restructure the display so that each actor's movies appear on the same line as the actor's name.

## A Possible Solution to Exercise 10

As contained in **actorlist-ex10-done.xml**:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="styles-ex10-done.css"?>

<!DOCTYPE actors SYSTEM "actordtd-done.dtd">

<actors>
   <description>A list of noted actors from the 20th
century</description>
   <url>http://www.imdb.com</url>
   <actor>
      <name>
         <firstname>Humphrey</firstname>
         <lastname>Bogart</lastname>
      </name>
      <films>
         <film oscar="nominated">
            <title>Casablanca</title>
            <date>1942</date>
         </film>
         <film oscar="won">
            <title>The African Queen</title>
            <date>1951</date>
         </film>
         <film oscar="nominated">
            <title>The Caine Mutiny</title>
            <date>1955</date>
         </film>
      </films>
      <height units="meters">1.73</height>
   </actor>
   .
   .
   [other actors omitted]
   .
   .
</actors>
```

and as found in **styles-ex10-done.css**:

```
description {display:block; font-size:20pt; font-weight:bold}
url {font-style:italic}
actor {display:block; font-family:garamond; margin-top:20px}
name {font-weight:bold; color:green}
film {display:block; text-indent:30px}
title {color:blue}
date {font-style:italic}
height {display:none}
notes {display:block; text-indent:30px; margin-top:10px; font-size:90%}
```

# Extensible Stylesheet Language (XSL)

In this section, we will address many of the shortcomings of XML display with CSS, by looking at a language developed specifically for translating XML data into other languages. Chief among the weaknesses of CSS are:

- Data can't be sorted. The order in which an XML document is written is the order in which it will be displayed with CSS
- You can't output content that is not in the XML document. So, if you want to output markup (such as a table, or a horizontal rule) you can't. Similarly, you cannot display any additional text, characters, or tags.
- You can't display information in attribute values.
- You can't treat elements differently based on what their content is; only on what their names are. So, if you wanted films which had won Oscars to be displayed differently than films that hadn't, you couldn't.

Enter XSL. XSL was developed by the W3C under a similar process to the one with which they developed schemas. Although CSS was adequate for basic views of XML data, it was clear that there would need to be a more robust stylesheet language to produce output from XML data.

## XSL, XSLT, and XSLFO

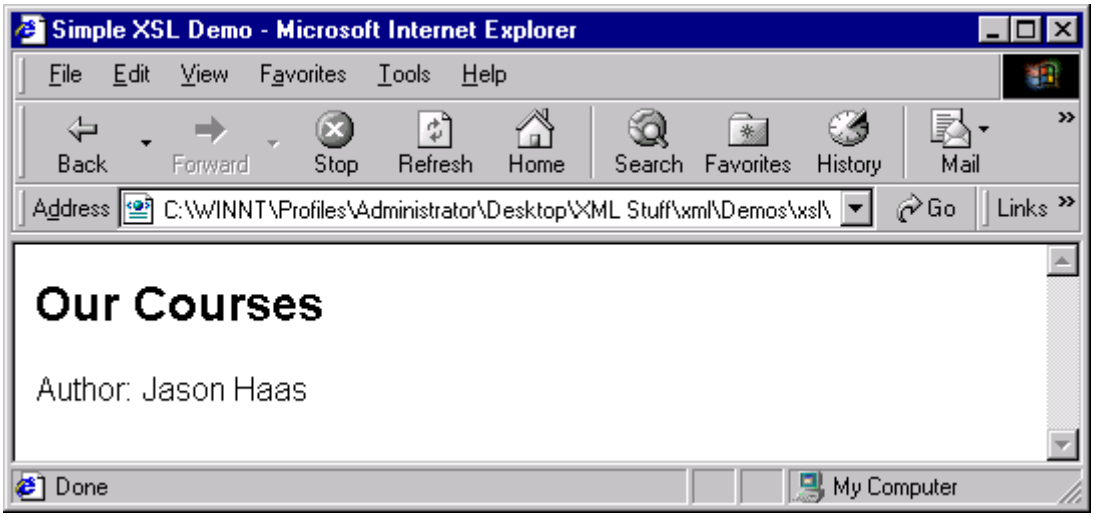
You may have heard all three of the above acronyms. XSL is really two languages, XSLT (for XSL Transformations) and XSLFO (for XSL Formatting Objects). Of the two, XSLT is by far the more widely used, and in speech, XSL and XSLT are often used interchangeably.

- **XSLFO** is a stylesheet language designed to allow very precise formatting and display for XML data, similar in many ways to CSS. XSLFO is not supported by browsers, and is most commonly used in custom applications to format and display output in non-English languages.
- **XSLT** is a language designed by the W3C to translate XML data into other formats, including HTML, WML, text, and even new XML. It has undergone several series of proposals and revisions, and the current namespace used for XSLT is `xmlns:xsl="http://www.w3.org/1999/XSL/Transform"`.

As we are focusing on XML Web applications, the language that XSL will transform XML into will typically be HTML. However, there is no restriction on the output language of an XSL stylesheet, and translations into other languages (and even other XML structures) are becoming increasingly common.

## XSL Basics: Linking to an XSL Stylesheet

As with CSS stylesheets, you can link an XSL stylesheet directly to an XML document. As a first example, take a look at the next installment of our ongoing demo (**xml > demos > xsl > courses_simple.xml**):




What you see is a simple interpreted display of some of the data contained in the XML courses datasheet that we saw earlier. Take a look at the code for **courses_simple.xml**:

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

<!DOCTYPE courses SYSTEM "courses_with_attributes.dtd">

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

<courses>
   <author>Jason Haas</author>
   <course subject="XML">
      <number>410</number>
      <title>Introduction to XML</title>
      <topics>
         <topic>XML syntax</topic>
         <topic>DTDs (Document Type Definitions)</topic>
         <topic>XSL (eXtensible Stylesheet Language)</topic>
      </topics>
      <course-length scale="days">2</course-length>
      <location>
         <city>Washington</city>
         <state>DC</state>
      </location>
      <description>The XML training class is designed to give
students the XML, XSL, and DTD abilities to make powerful,
platform-independent data driven applications.  Students will learn
XML/XSL syntax, and will also learn how to apply these skills into
powerful and flexible applications.</description>
```

```
   </course>

.
[other courses omitted]
.

</courses>
```

You should notice two things: a new call to a stylesheet named **courses_simple.xsl**, and the author element (which will be displayed by the XSL stylesheet). The only syntactic difference between a link to a CSS stylesheet and an XSL stylesheet is the value of the **type** attribute, which must in the case of an XSL stylesheet be "**text/xsl**".

## Examining an XSL Stylesheet

XSL works by selecting out node elements from an XML datasheet by use of a filter. This filter searches the datasheet for specific patterns, and returns the matching nodes. The content of these nodes can then be included in the syntax of whatever language you are transforming the XML data into. XSL has adopted a pattern-matching syntax similar to Windows URL pathing syntax, which will be discussed in more detail below. The code for the **courses_simple.xsl** is below:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
         <head>
            <title>Simple XSL Demo</title>
         </head>
         <body>
            <div style="font-size:12pt;font-family:Arial">
            <h2>Our Courses</h2>
               Author: <xsl:value-of select="courses/author" />
            </div>
         </body>
      </html>
   </xsl:template>
</xsl:stylesheet>
```

It is important to remember that an XSL stylesheet is an XML document, and must conform to XML rules of syntax. A quick recap:

- Tags are case-sensitive, and nearly all lower-case.
- All tags must have matching end tags, even HTML tags with optional closing pairs. For "empty elements", the correct end syntax must be observed.
- Tags must be correctly nested.
- Attributes must be enclosed in matching quotes.

As XSL is a part of the XML language, the initial line of any XSL stylesheet should be a language declaration, with the version included:

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

## The xsl:stylesheet tag and the namespace declaration

The XSL stylesheet is declared by the **xsl:stylesheet** tag. This tag will enclose the entire content of the XSL stylesheet. **xsl:stylesheet** tags should also contain a namespace declaration (in this case **xmlns:xsl="http://www.w3.org/1999/XSL/Transform"**) and a version attribute (currently **version="1.0"**. All XSL parsers (including Internet Explorer) require that your namespace look exactly as in the example above.

## The xsl:template tag

Enclosed inside the xsl:stylesheet tag will be some number of **xsl:template** blocks, from one (in a very simple XSL stylesheet, such as the one above) to many (when you are selecting multiple elements of your datasheet). Each **xsl:template** requires a **match** attribute, which specifies the pattern or node location that must be matched to have the enclosed formatting applied. In nearly every example, you will have a template that matches the root node (signified by the single forward-slash). This node, which appears only once in any XML document, allows the stylesheet to output the required elements that must also only appear once in the output (such as the required HTML document tags).

In our example, the main **xsl:template** element is:

```
<xsl:template match="/">
   <html>
      <head>
         <title>Simple XSL Demo</title>
      </head>
      <body>
         <div style="font-size:12pt;font-family:Arial">
         <h2>Our Courses</h2>
            Author: <xsl:value-of select="courses/author" />
         </div>
      </body>
   </html>
</xsl:template>
```

We'll talk more about pattern matching below, but for now, suffice to explain that the forward-slash (/) matches the document root, and will only ever have one match. For that node, we specify HTML formatting, including the required beginning and ending HTML tags, a page title, and a page body containing one heading and an **xsl:value-of** tag.

## The xsl:value-of tag

When you want to read out the contents of an XML element, you will use the **xsl:value-of** tag. This tag takes a required **select** attribute, inside which you specify the node to be output. Note that nodes are generally referenced relatively. So, from the root, we are specifying that the path to the element we want to output is as follows:

```
<xsl:value-of select="courses/author" />
```

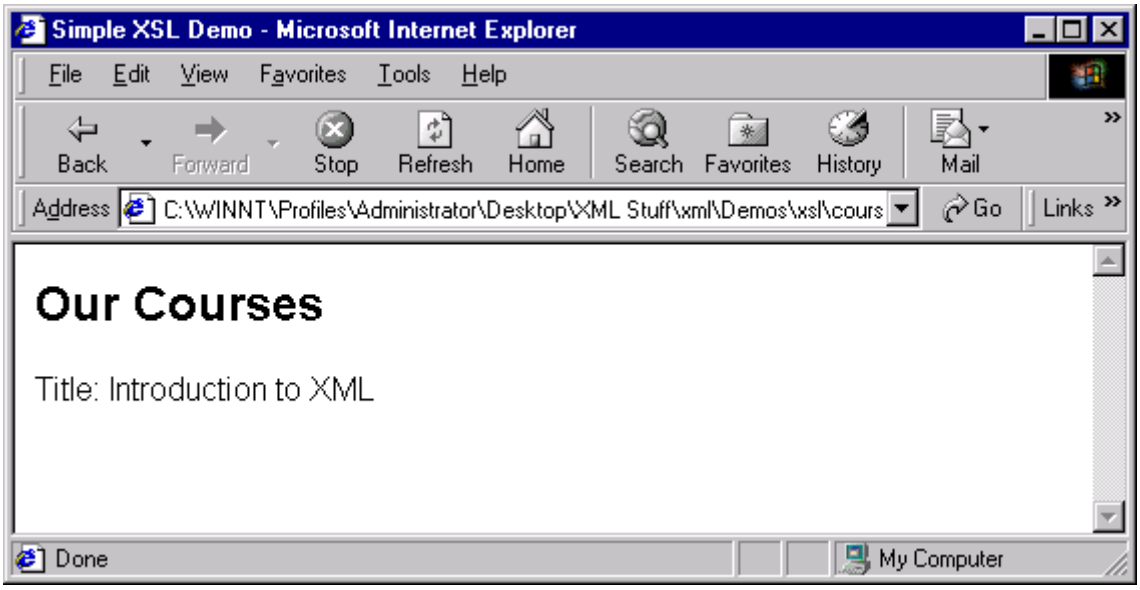
Finally, note that the **xsl:value-of** element is empty.

## Referencing iterating nodes

What would happen if you were to replace the above value-of tag with the following?:

```
<xsl:value-of select="courses/course/title" />
```

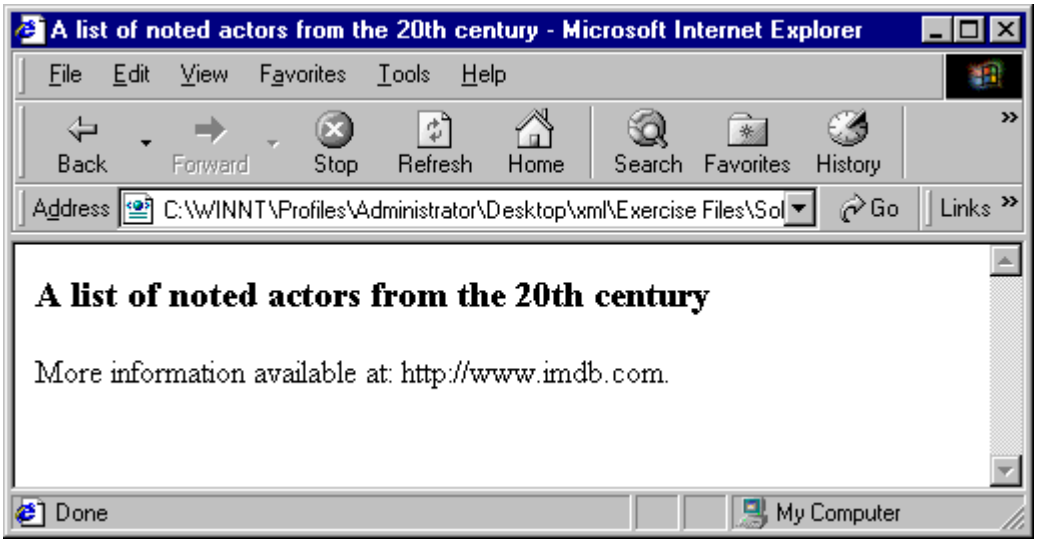
You can see the output at **demos > xsl > courses_title.xml**:




You should note that although there are 5 different title elements in the XML, only the first is displayed. We will see how to reference and output iterative content in the next section.

## Exercise 11: Beginning an XSL Stylesheet

In this exercise, you will be beginning an XSL stylesheet that will transform your XML data into HTML for display. When you are done with this first exercise, you should see:




To complete this exercise, please do the following:

1. Re-open **actorlist.xml** in Notepad.
2. Add a line to link it to the XSL stylesheet **actorxsl.xsl**.
3. Open a blank document in Notepad.
4. Create your XSL stylesheet. It should follow the following characteristics:
    - Be sure to include the required namespace declaration and version number
    - Also include the required HTML tags such as <html>, <head>, etc.
    - Your page's title should be the value of the **description** element
    - You should select and display the **description** element inside <h3> tags
    - Inside <p> tags, display the text "More information available at: " and print out the **url** element from your datasheet.
5. Save your file as **actorxsl.xsl**.
6. Open **actorlist.xml** in Internet Explorer.
7. If you get an error message, go back into Notepad, edit your file, and reload.

## If you are done early...

- Open **actorxsl.xsl** in Internet Explorer. What do you notice?
- Add some tags around the text you've printed out to make it more visually appealing. Perhaps some color?
- Display some information about the first actor.

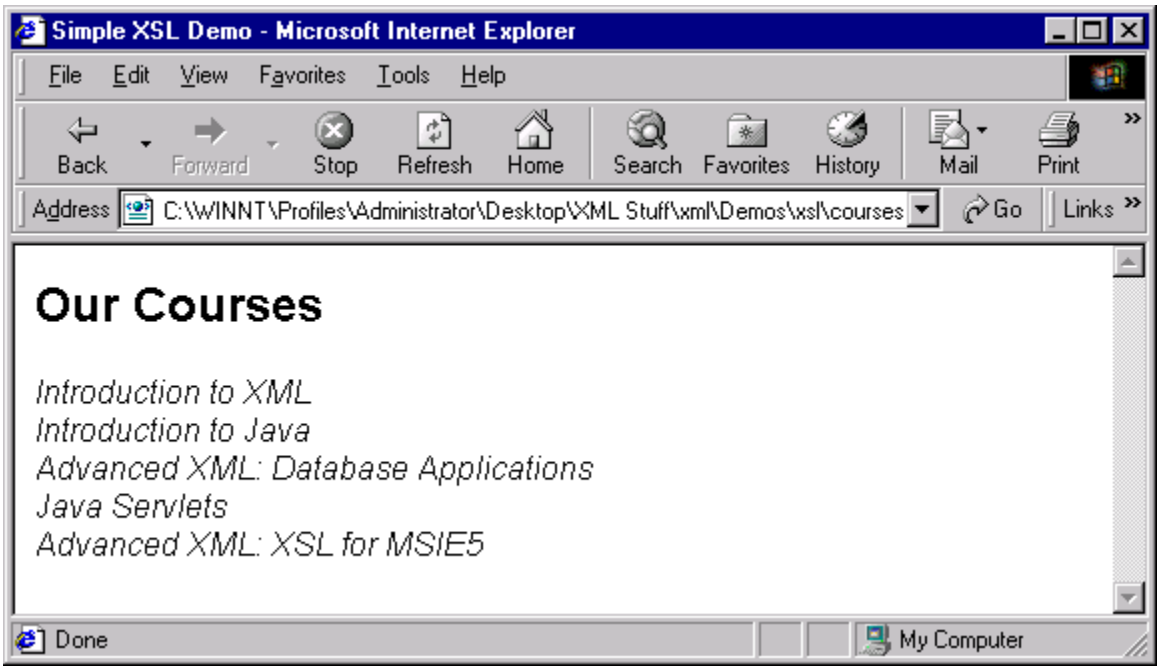
## A Possible Solution to Exercise 11

As contained in **actorxsl-ex11-done.xsl**:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
         <head>
            <title><xsl:value-of select="actors/description" /></title>
         </head>
         <body>
         <h3><xsl:value-of select="actors/description" /></h3>
         <p>
            More information available at:
            <xsl:value-of select="actors/url" />.
         </p>
         </body>
      </html>
   </xsl:template>
</xsl:stylesheet>
```

## xsl:apply-templates and Iterative Content

When you wish to reference nodes that appear more than once (usually to generate output using them), you include an **xsl:apply-templates** tag inside your root-level xsl:template element. This tag selects out a series of nodes, and calls a matching template to generate output appropriately. To get a sense of how it works, take a look at the next demo in our series (**demos > xsl > courses_list.xml**):




You will note that in addition to the heading "Our courses" there is a list of all the course title elements from the XML datasheet. The XSL code used to produce this display is below. Note particularly the sections in **bold** (**demos > xsl > courses_list.xsl**):

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
      <head>
         <title>Simple XSL Demo</title>
      </head>
      <body>
         <div style="font-size:12pt;font-family:Arial">
         <h2>Our Courses</h2>
         <xsl:apply-templates select="courses/course/title" />
         </div>
      </body>
      </html>
   </xsl:template>
```

```
      <xsl:template match="title">
         <div style="font-style:italic"><xsl:value-of select="." /></div>
      </xsl:template>

   </xsl:stylesheet>
```

As you can see from the above code, **xsl:apply-templates** is an empty XML element that specifies in its select attribute what nodes to apply templates to. The pattern matching syntax will be discussed below, but the **select="courses/course/title"** attribute follows the XML object hierarchy from the root node (which, remember, can be found in the template's match attribute) down to the **title** element we wish to display. The above expression will select all **title** elements. Finally, once you specify an **xsl:apply-templates** tag, you must also include an **xsl:template** tag to match each element you have selected.

## A child xsl:template element

In our example, we have specified that we are interested in selecting all title elements from our datasheet. The final **xsl:template** block on the page specifies how they should be treated:

```
<xsl:template match="title">
   <div style="font-style:italic"><xsl:value-of select="." /></div>
</xsl:template>
```

We have chosen that each course title should be displayed on a separate line, in italic font. The **xsl:template match="title"** tag will be applied once for each match. The **xsl:value-of** tag is going to print out the current node (the dot refers to the currently selected node)

## XPath: the XSL Node Matching Syntax

There are 6 characters that can be used with XSL node matching. These are part of the XML standard **XPath**, which has been absorbed by XSL at the mechanism for identifying nodes based on their relationships, names, or contents. The common XPath relationships are:

| Character | Description |
|---|---|
| / | The root node (if at the beginning of the path) or the child of the preceding node (if between elements). An example would be **`/courses/course/title`**. |
| // | Recursive descent from a given location. When at the beginning of the path, recursive descent begins at the root node. So, **`//title`** matches all title elements anywhere in the hierarchy. |
| .. | Parent of a given element. Used in expressions such as **`title[../course-length='2']`**, which would select all title elements with a sibling element course-length with a value of '2'. |
| * | Wildcard operator. So, **`//course/*`** would select all child elements of all course elements. |
| . | The current element operator. So, **`./*`** would select all child elements of the current element. |
| @ | Attribute indicator. So, **`//course-length/@scale`** would match the scale attribute of the course-length element. |

### Choosing Between Equivalent XPath Expressions

It is often possible to specify the same list of XML nodes in multiple ways using XPath. For example, the **xsl:apply-templates** element from the above example

```
<xsl:apply-templates select="courses/course/title" />
```

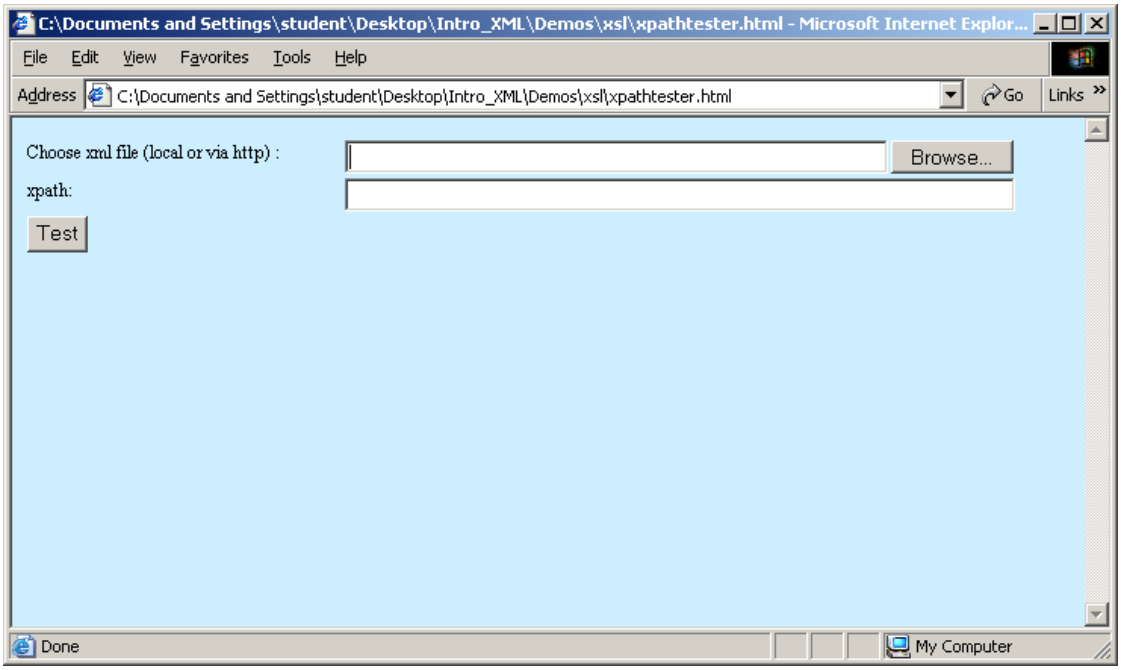
could have been written:

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

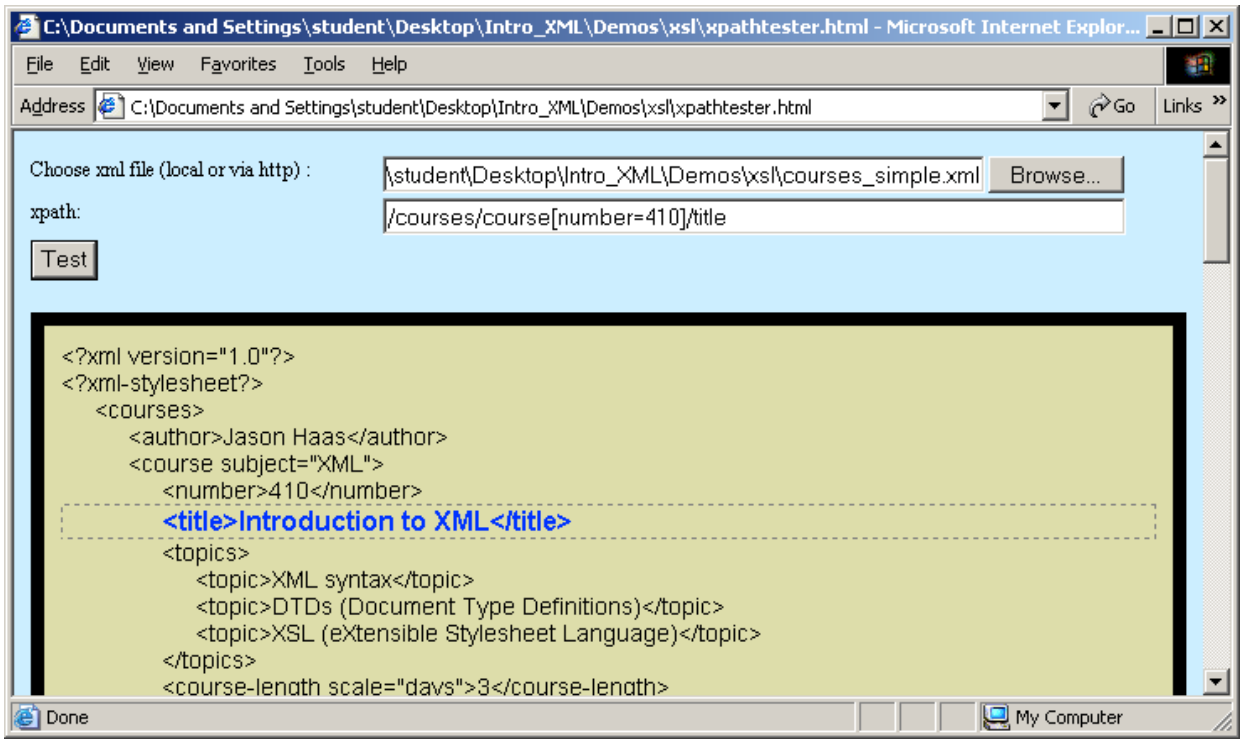
So, how do you choose? When in doubt, choose the XPath expression that forces the XSL processor to examine the fewest possible nodes in order to return the list you request. Be particularly careful if you use the recursive search operator (//). This operator, though easy to write, forces your XSL processor to examine every node in your document to see if its name is **title**. On a small stylesheet processing a small datasheet, you won't notice the difference. But, for a busy server evaluating complex XML data, you will see your performance suffer.

### An XPath testing tool

Take a few moments to explore the file **demos/xsl/xpathtester.html.** A screen shot is below:

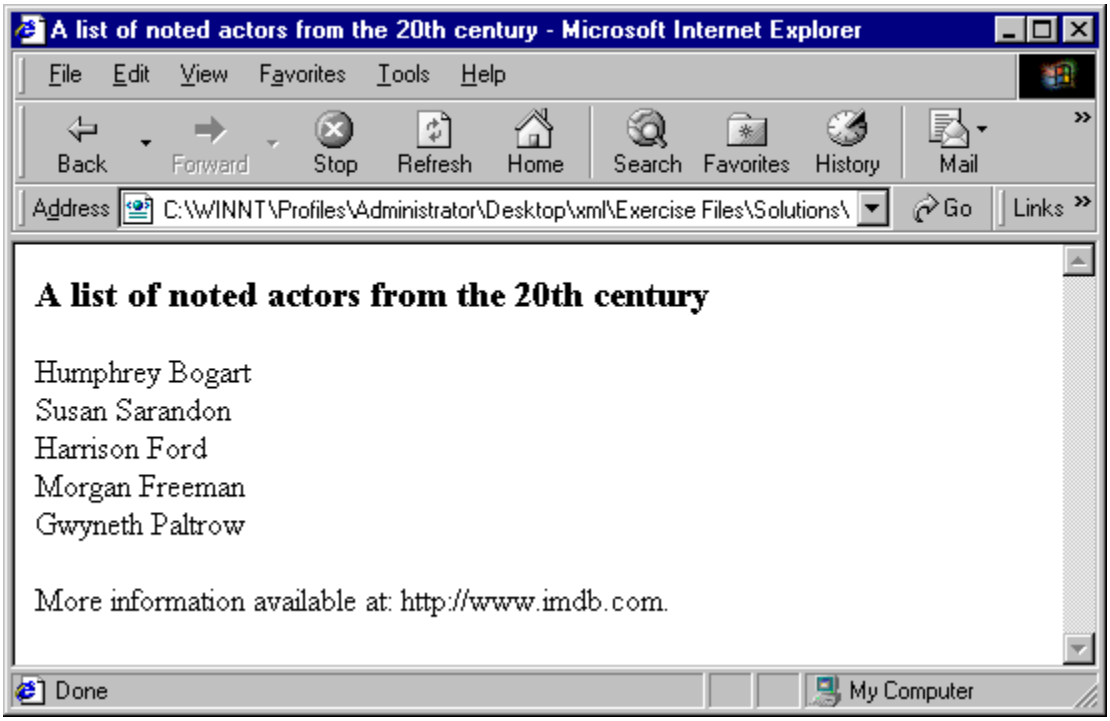



Enter the url of an XML file (local or external). Then enter an XPath in the input field. Click the Test button, and any matching nodes should be highlighted:



## Exercise 12: Displaying Iterative Data with Your XSL Stylesheet

In this exercise, you will be improve your XSL stylesheet to display your actors' names in addition to the basic information you displayed in the previous exercise. When you have completed the exercise, your display should look like:




To complete this exercise:

1. Re-open **actorxsl.xsl** in Notepad.
2. Add an **xsl:apply-templates** element to your stylesheet. That element should select all **name** elements.
3. Add a new **xsl:template** element that matches name, and specify that each **name** element be displayed in a DIV, which will place each listing on a separate line.
4. Save your file.
5. Open **actorlist.xml** in Internet Explorer.
6. If you get an error message, go back into Notepad, edit your file, and reload.

## If you are done early…

- Reformat the display so that it is in the format **Lastname**, Firstname. Hint: you may need more than one **xsl:value-of** tag to accomplish this.
- Change the list from a straight list into a bulleted list.
- Add some other elements from **actorlist.xml** to your display, formatting each so that they appear as appropriate.

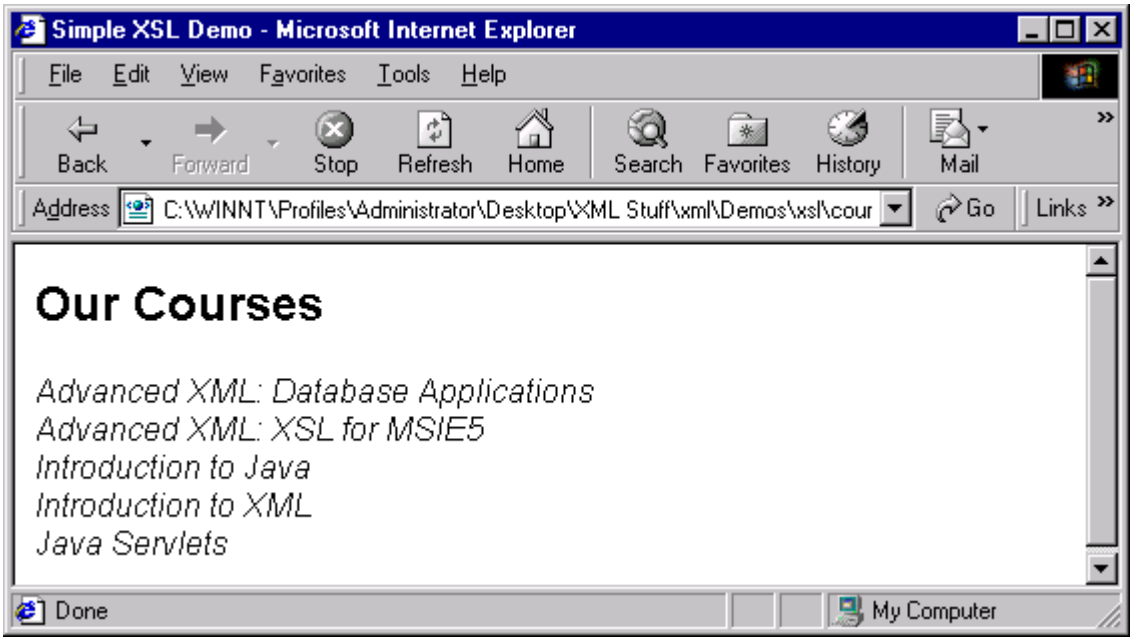
## A Possible Solution to Exercise 12

And as contained in **actorxsl-ex12-done.xsl**:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
         <head>
            <title><xsl:value-of select="actors/description" /></title>
         </head>
         <body>
         <h3><xsl:value-of select="actors/description" /></h3>
         <xsl:apply-templates select="actors/actor/name" />
         <p>
            More information available at:
            <xsl:value-of select="actors/url" />.
         </p>
         </body>
      </html>
   </xsl:template>
   <xsl:template match="name">
      <div><xsl:value-of select="." /></div>
   </xsl:template>
</xsl:stylesheet>
```

## Using xsl:sort to re-sort your display

You may have noticed that your list of actors (or course titles) was in the same order in which the XML datasheet was written. Although this may be sufficient, more often you will want to apply an order to your output. Luckily, XSL makes this easy. To get a sense of how it's done, take a look at the file **demos > xsl > courses_sort.xml**:




Note that now, instead of the default authored order, the list of titles is ordered alphabetically. The XSL code used to produce this is below (**demos > xsl > courses_sort.xsl**):

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
         <head>
            <title>Simple XSL Demo</title>
         </head>
         <body>
            <div style="font-size:12pt;font-family:Arial">
            <h2>Our Courses</h2>
            <xsl:apply-templates select="courses/course/title">
               <xsl:sort select="." />
            </xsl:apply-templates>
            </div>
         </body>
      </html>
   </xsl:template>

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

```
        <div style="font-style:italic"><xsl:value-of select="." /></div>
      </xsl:template>

</xsl:stylesheet>
```

Using the **xsl:sort** tag, you specify the criteria on which a node list should be sorted. If you want to sort by multiple criteria, simply specify a second (or third, or fourth…) **xsl:sort** tag. Their priority will be determined by their order.

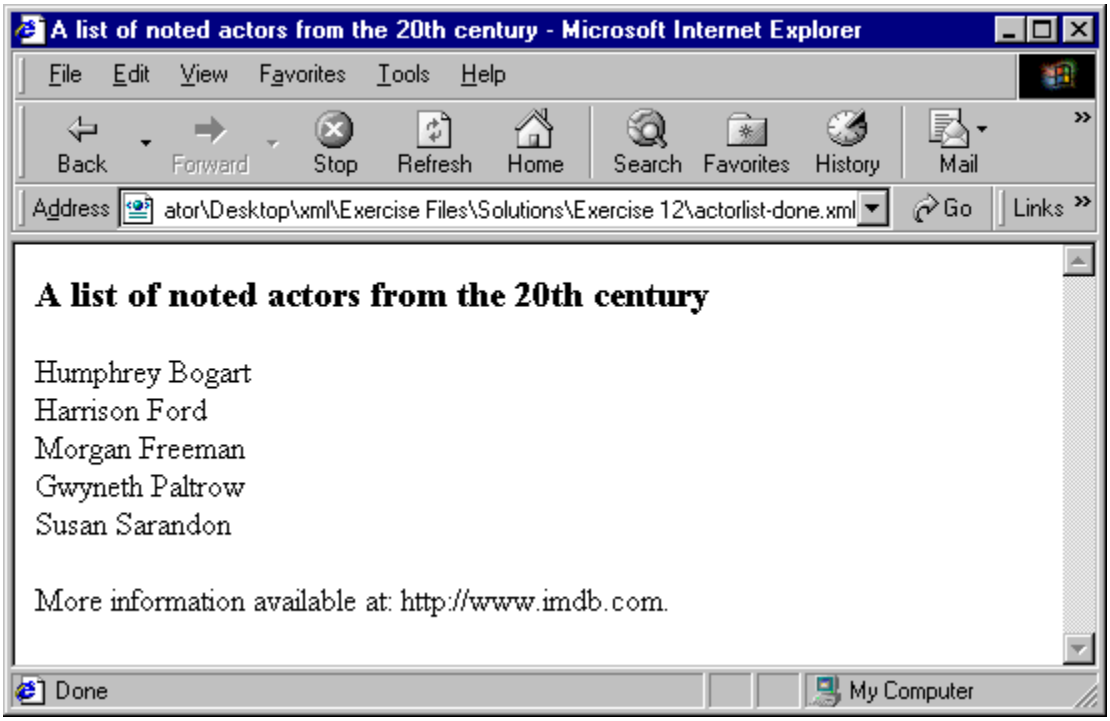
## Modifying xsl:sort with order and data-type attributes

The default order for sorts is ascending, and, by default, all data is treated as text. However, you can override both of these. If you were to wish to sort by a numeric criteria in descending order, you would write an **xsl:sort** tag like the following:

```
<xsl:sort select="course/number" data-type="number" order="descending" />
```

## Exercise 13: Adding a Sort Order to your XSL

In this exercise, you will be improving your XSL stylesheet to order your list of actors according to last name. When you are done, your display should look like:




To complete this exercise, please do the following:

1. Re-open **actorxsl.xsl** in Notepad.
2. Add an **xsl:sort** element to your stylesheet. You want to sort your list of names by **lastname**.
3. Save your file.
4. Open **actorlist.xml** in Internet Explorer.
5. If you get an error message, go back into Notepad, edit your file, and reload.

### If you are done early...

- Sort your list of names by the date of their first movie (for these purposes, just measure the date of the first movie listed for each actor, not the first chronologically).
- Sort in descending, rather than ascending, order.

## A Possible Solution to Exercise 13

As contained in **actorxsl-ex13-done.xsl**:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
         <head>
            <title><xsl:value-of select="actors/description" /></title>
         </head>
         <body>
         <h3><xsl:value-of select="actors/description" /></h3>
         <xsl:apply-templates select="actors/actor/name">
            <xsl:sort select="lastname" />
         </xsl:apply-templates>
         <p>
            More information available at:
            <xsl:value-of select="actors/url" />.
         </p>
         </body>
      </html>
   </xsl:template>
   <xsl:template match="name">
      <div><xsl:value-of select="." /></div>
   </xsl:template>
</xsl:stylesheet>
```

## Generating Hyperlinks with XSL

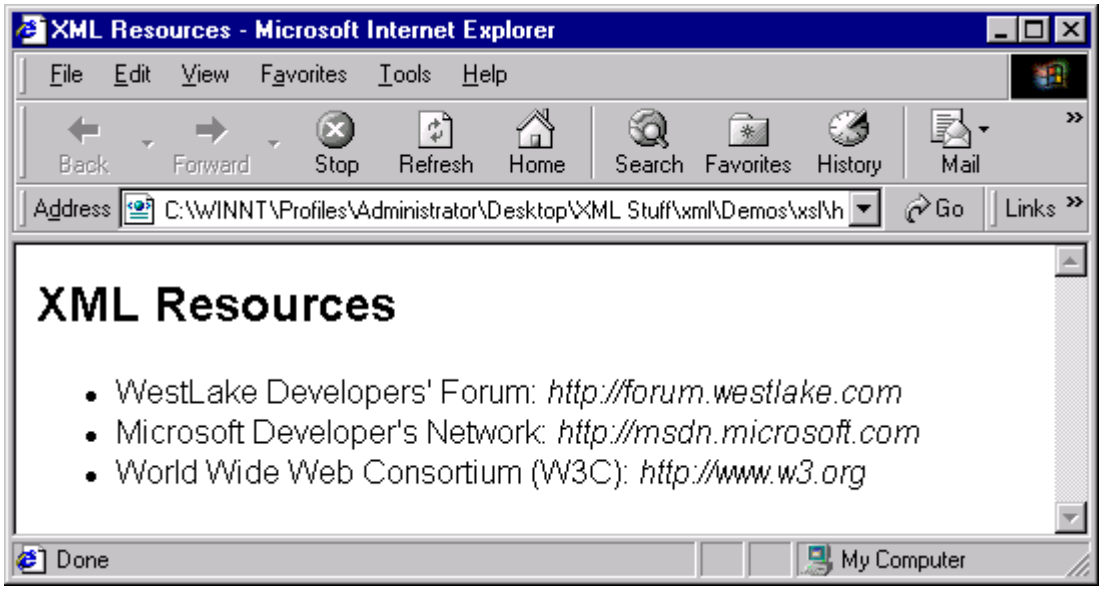
If you have tried to generate a hyperlink using XSL with syntax like:

```
<a href="<xsl:value-of select='url' />">Link Text</a>
```

you will have found that you cannot nest tags within tags in XSL. However, you need to be able to generate dynamic content for the value of attributes if you want to be able to generate hyperlinks with your XSL stylesheets. To get a feel for how you generate hyperlinks, we will be using an XML file that contains a list of resources. The XML code for this file is below:

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="resource_list.xsl" ?>
<resources>
   <resource>
      <name>WestLake Developers' Forum</name>
      <url>http://forum.westlake.com</url>
   </resource>
   <resource>
      <name>Microsoft Developer's Network</name>
      <url>http://msdn.microsoft.com</url>
   </resource>
   <resource>
      <name>World Wide Web Consortium (W3C)</name>
      <url>http://www.w3.org</url>
   </resource>
</resources>
```

A standard stylesheet, designed to output the content but not (yet) generate hyperlinks might look like (**demos > xsl > hyperlinks > resource_list.xml**):

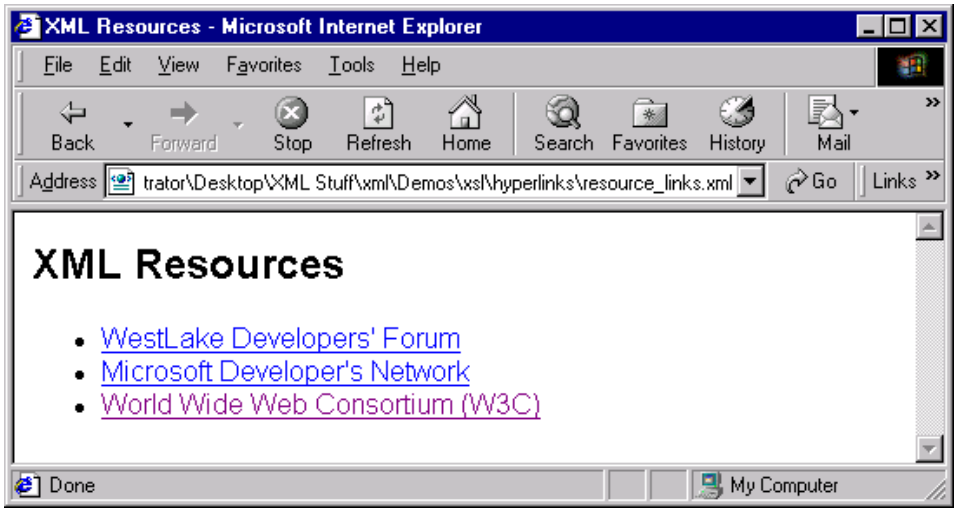


The XSL code for this stylesheet should be quite logical (**demos > xsl > hyperlinks > resource_list.xsl**):

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
         <head>
            <title>XML Resources</title>
         </head>
         <body>
            <div style="font-size:12pt;font-family:Arial">
            <h2>XML Resources</h2>
            <ul>
               <xsl:apply-templates select="//resource" />
            </ul>
            </div>
         </body>
      </html>
   </xsl:template>

   <xsl:template match="resource">
      <li>
         <xsl:value-of select="name" />:
         <i>
            <xsl:value-of select="url" />
         </i>
      </li>
   </xsl:template>
</xsl:stylesheet>
```

Note that the resource template displays a list item for each resource, and inside the list item displays the name of the resource, followed by a colon, followed by the URL, in italics.

Contrast the previous example with the next (**demos > xsl > hyperlinks > resource_links.xml**):



The output code now produces hyperlinks (which, if you're suspicious, actually do work; try them). The new **resource** template is below (**demos > xsl > hyperlinks > resource_links.xsl**):

```
<xsl:template match="resource">
   <li>
      <a>
         <xsl:attribute name="href">
            <xsl:value-of select="url" />
         </xsl:attribute>
         <xsl:value-of select="name" />
      </a>
   </li>
</xsl:template>
```

You will notice a new XSL tag: **xsl:attribute**. Its role is to declare attributes for its parent tag. In this case, the parent tag is the HTML <a> tag. The structure of the **xsl:attribute** tag is that it takes a name attribute to specify the **name** portion of the **name**="value" structure of the attribute. The value portion is specified by whatever is between the <xsl:attribute> and the </xsl:attribute> tags, in our case the text of the **resource/url** element.

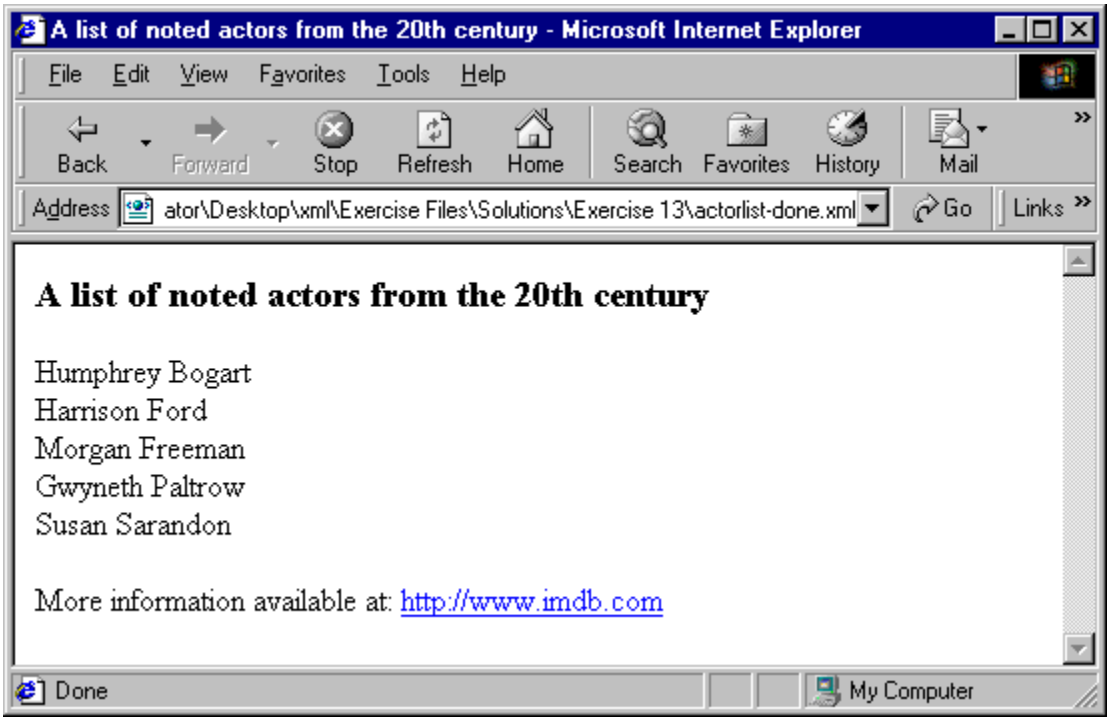
Finally, in order to specify the visible text of the URL, we print out the **resource/name** element, which will be contained between the <a> and the </a> tags, forming the link.

## xsl:element

There is also an XSL tag **xsl:element** tag that is used to dynamically generate new elements in the output code. It is only rarely used in generating HTML output (as you will almost always know the HTML tags you're producing) but is commonly used when outputting new XML with your XSL stylesheet. **xsl:element** and applications to produce new XML with an XSL stylesheet are covered in WestLake's *Advanced XML and XSL* class.

## Exercise 14: Generating Hyperlinks with XSL

In this exercise, you will be improving your XSL stylesheet so that the Internet Movie Database URL shows up as a hyperlink, rather than as plain text. When you're done, your page should look like:




To complete this exercise, please do the following:

1. Re-open **actorxsl.xsl** in Notepad.
2. Modify your XSL stylesheet so that the URL listed in **actorlist.xml** is displayed as a hyperlink. Note that you will be using the same value for both the **href** attribute and the displayed text.
3. Save **actorxsl.xsl**.
4. Open **actorlist.xml** in Internet Explorer. If you get an error message, go back into Notepad, edit your file, and reload.

### If you are done early...

- Hyperlink each actor's name with a **mailto:** link so that clicking on it generates an email in the form ***firstname*@westlake.com**. The syntax of a mailto link is **<a href="mailto:email@address.ext">Link Text</a>**.

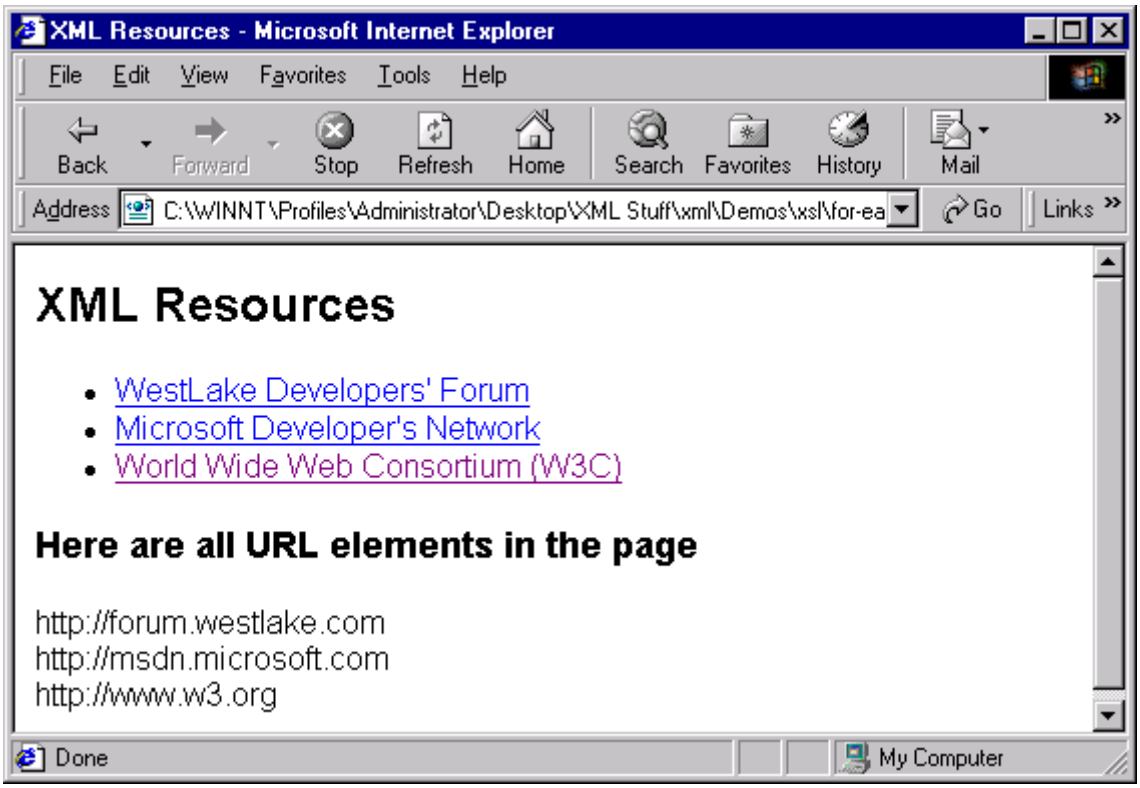
## A Possible Solution to Exercise 14

As contained in **actorxsl-ex14-done.xsl**:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
         <head>
            <title><xsl:value-of select="actors/description" /></title>
         </head>
         <body>
         <h3><xsl:value-of select="actors/description" /></h3>
         <xsl:apply-templates select="actors/actor/name">
            <xsl:sort select="lastname" />
         </xsl:apply-templates>
         <p>More information available at:
            <a>
               <xsl:attribute name="href">
                    <xsl:value-of select="actors/url" />
               </xsl:attribute>
               <xsl:value-of select="actors/url" />
            </a>
         </p>
         </body>
      </html>
   </xsl:template>
   <xsl:template match="name">
      <div><xsl:value-of select="." /></div>
   </xsl:template>
</xsl:stylesheet>
```

## Loops with XSL

With **xsl:apply-templates**, we have the basic functionality of a loop. However, there are times when you would prefer to keep the loop in-line rather than having to call an external template. As such, it is largely a matter of style as to when you choose to use **xsl:apply-templates**, and when you choose to use **xsl:for-each**. To get a sense of how **xsl:for-each** works, take a look at the following example: (**demos > xsl > for-each > resource_foreach.xsl**):




The code for this example is:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match="/">
<html>
<head>
   <title>XML Resources</title>
</head>
<body>
   <div style="font-size:12pt;font-family:Arial">
   <h2>XML Resources</h2>
   <ul>
      <xsl:apply-templates select="//resource" />
   </ul>
   <h3>Here are all URL elements in the page</h3>
```

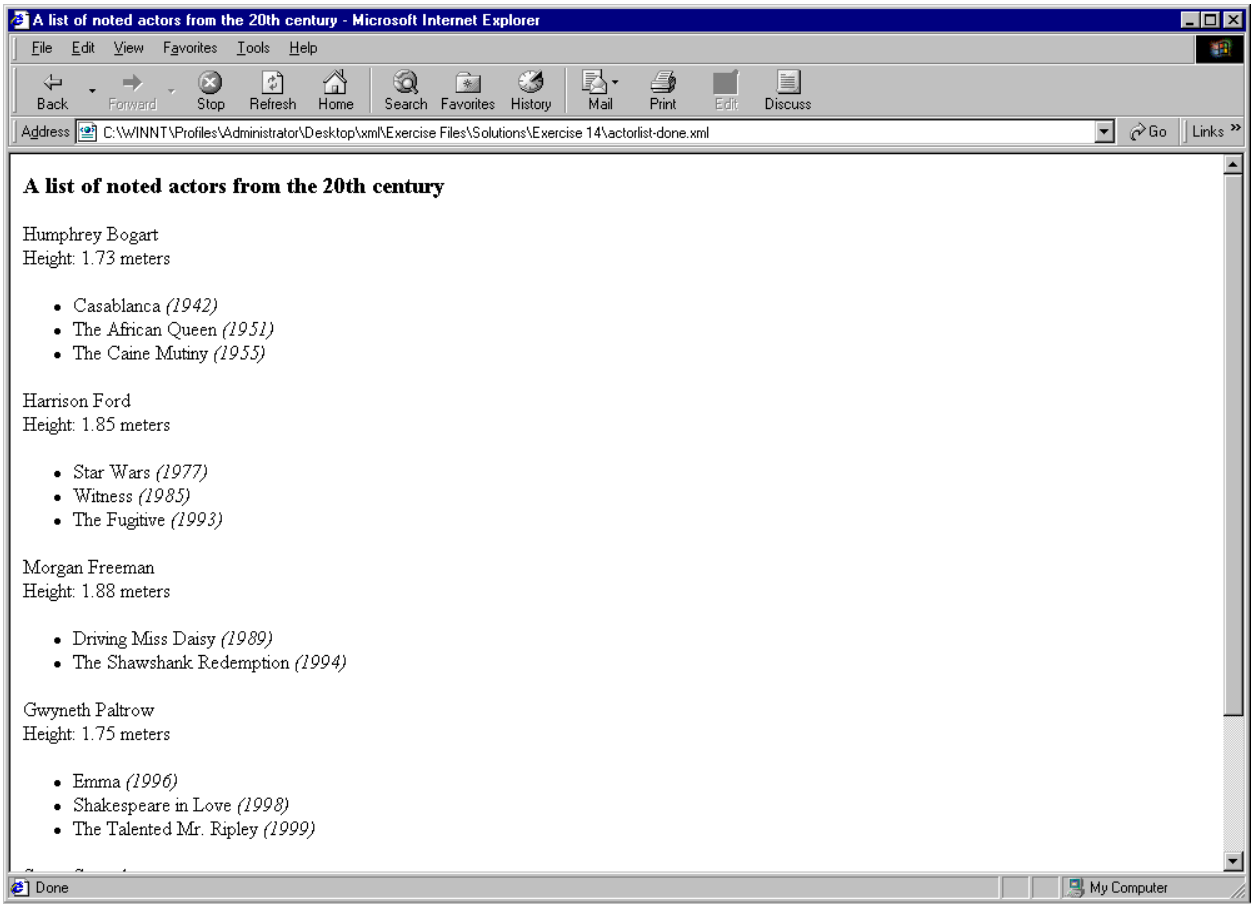
```
   <xsl:for-each select="//url">
      <div><xsl:value-of select="." /></div>
   </xsl:for-each>
   </div>
</body>
</html>
</xsl:template>

<xsl:template match="resource">
   <li>
      <a>
         <xsl:attribute name="href">
            <xsl:value-of select="url" />
         </xsl:attribute>
         <xsl:value-of select="name" />
      </a>
   </li>
</xsl:template>

</xsl:stylesheet>
```

## The xsl:for-each element

**xsl:for-each** takes a required **select** attribute (exactly like xsl:apply-templates), whose value must contain some expression selecting a node or list of nodes. It can also accept as a child element **xsl:sort**, should you wish to order your results.

## Exercise 15: Adding an xsl:for-each Loop to Your Stylesheet

In this exercise, you will be using an xsl:for-each loops to list each actor's movies in a bulleted list beneath their name. You will also be displaying the actor's height (which we will use in an upcoming exercise). When you are done, your display should look like:




To complete this exercise, please do the following:

1. Re-open **actorxsl.xsl** in Notepad.
2. Modify your XSL stylesheet so your name template displays the actor's height, below his or her name.
3. Below the actor's name, add an unordered list (<ul>…</ul>).
4. Inside the unordered list, add an **xsl:for-each** loop. This loop should display for each film:
    - a list item (<li>) tag
    - the film's title

- the film's date (inside parentheses).

5. The loop should sort the films by their date.
6. Save **actorxsl.xsl**.
7. Open **actorlist.xml** in Internet Explorer.
8. If you get an error message, go back into Notepad, edit your file, and reload.

## If you are done early...

- Add another loop that displays an asterisk next to the actor's name for each film of his/hers that has been either nominated for an oscar or has won an oscar.
- Modify your stylesheet so that it uses an **xsl:for-each** loop instead of a separate template (and the **xsl:apply-templates** tag that calls it) to display the rows of your table.

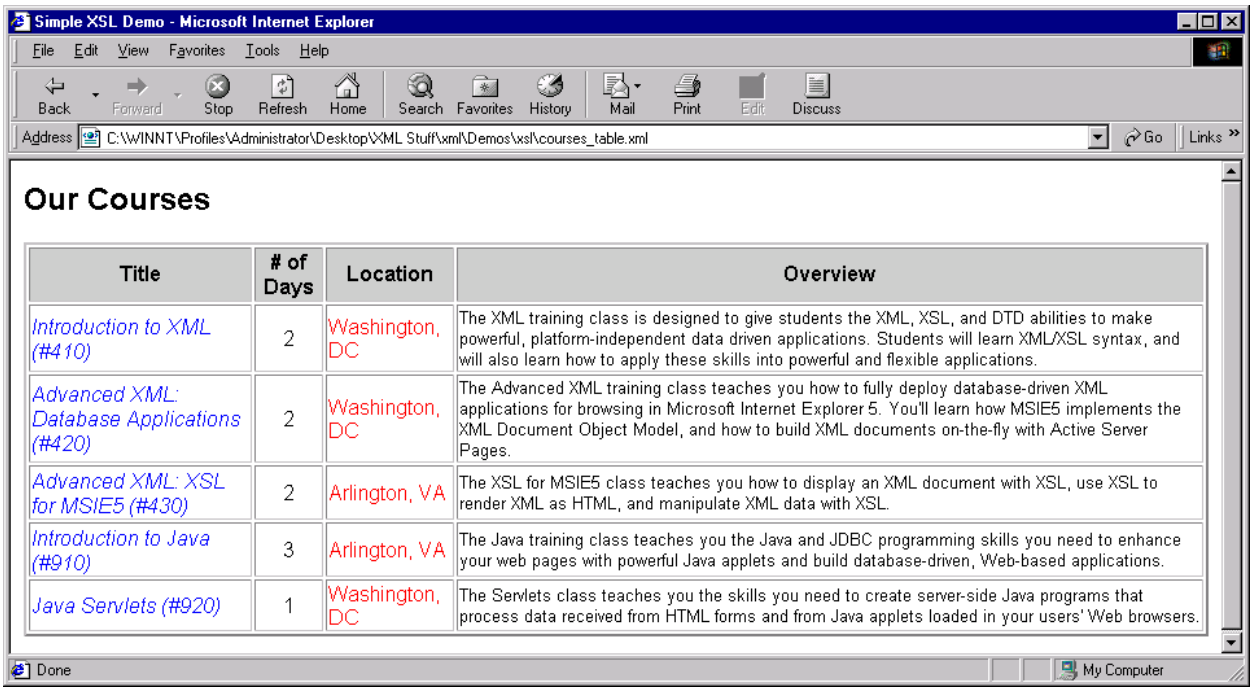
## A Possible Solution to Exercise 15

As contained in **actorxsl-ex15-done.xsl**:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
         <head>
            <title><xsl:value-of select="actors/description" /></title>
         </head>
         <body>
         <h3><xsl:value-of select="actors/description" /></h3>
         <xsl:apply-templates select="actors/actor/name">
            <xsl:sort select="lastname" />
         </xsl:apply-templates>
         <p>More information available at:
            <a>
               <xsl:attribute name="href">
                    <xsl:value-of select="actors/url" />
               </xsl:attribute>
               <xsl:value-of select="actors/url" />
            </a>
         </p>
      </body>
      </html>
   </xsl:template>
   <xsl:template match="name">
      <div>
         <xsl:value-of select="." /><br />
         Height: <xsl:value-of select="../height" />
         <xsl:text> </xsl:text>
         <xsl:value-of select="../height/@units" />
         <ul>
            <xsl:for-each select="../films/film">
               <xsl:sort select="date" order="ascending" />
               <li>
                    <xsl:value-of select="title" />
                    <xsl:text> </xsl:text>
                    <i>(<xsl:value-of select="date" />)</i>
               </li>
            </xsl:for-each>
         </ul>
      </div>
   </xsl:template>
</xsl:stylesheet>
```

# Displaying Complex Structures with XSL

XSL is not restricted to simple HTML structures. Take a look at the following demo (**xml > demos > xsl > courses_table.xml**:




The table you see above is the product of the same XML datasheet from the previous example, linked to the following XSL stylesheet (**demos > xsl > courses_table.xsl**):

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match="/">
   <html>
   <head>
      <title>Simple XSL Demo</title>
   </head>
   <body>
      <div style="font-size:12pt;font-family:Arial">
      <h2>Our Courses</h2>
      <table border="2">
         <tr bgcolor="#cccccc">
            <th>Title</th>
            <th># of Days</th>
            <th>Location</th>
            <th>Overview</th>
         </tr>
         <xsl:apply-templates select="courses/course">
            <xsl:sort select="number" data-type="number" />
         </xsl:apply-templates>
```

```
      </table>
      </div>
   </body>
   </html>
</xsl:template>

<xsl:template match="course">
   <tr>
   <td style="font-style:italic;color:blue">
      <xsl:value-of select="title" />
      (#<xsl:value-of select="number" />)
   </td>
   <td align="center">
      <xsl:value-of select="course-length" />
   </td>
   <td style="color:red">
      <xsl:value-of select="location/city" />,
      <xsl:value-of select="location/state" />
   </td>
   <td style="font-size:10pt">
      <xsl:value-of select="description" />
   </td>
   </tr>
</xsl:template>

</xsl:stylesheet>
```

What you see is essentially the same as the previous example. You have a root node match, that allows the coder to specify beginning and ending HTML, HEAD, and BODY tags. Inside, there is an iterative HTML structure (in this case a table) that will display one row for each course in the courses datasheet.

## The xsl:value-of select attribute

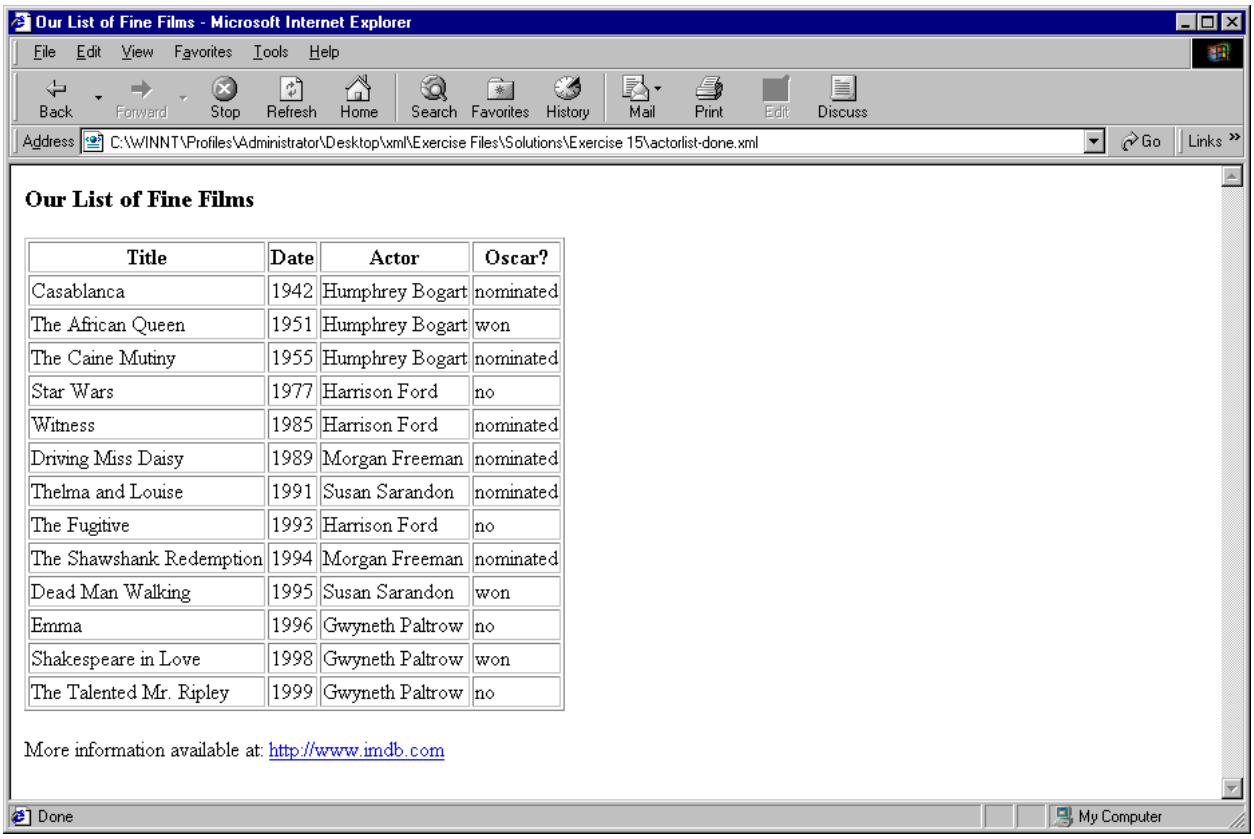
When you match a node with the **xsl:apply-templates** tag, you are selecting an entire element. That element includes all its sub-elements. So, you have available not only the data content, but also the hierarchical structure of the XML data. If you wish to further distinguish the portions of a datasheet for display, you do so with the **select** attribute of the **xsl:value-of** tag. In the above example, one data cell contains the content:

```
<td>
   <span style="color:red">
      <xsl:value-of select="location/city" />,
      <xsl:value-of select="location/state" />
   </span>
</td>
```

As you are selecting the **course** element in the **xsl:template** tag, you specify your select *with respect to that element*. In this case, we have specified the **city** and **state** sub-elements of the **location** element (which is itself a sub-element of **course)**.

## Exercise 16: Building an HTML Table with XSL

In this exercise, you will be creating an XSL stylesheet that will list all the films in the datasheet in an HTML table. When you have completed the exercise, you should see:




To complete this exercise:

1. Re-open **actorlist.xml** in Notepad
2. Modify the xml-stylesheet tag so that it points to the file moviexsl.xsl
3. Open **moviexsl.xsl** in your **Exercises-Main** folder. It has already been started for you.
4. Create the XSL stylesheet to create the display above. Some features:
    - You will probably want 2 templates: a root template, and a template that matches the **film** element.
    - The root template contains a heading with the text "Our List of Fine Films". You will need to add a table and the table's first row, as well as **xsl:apply-templates** tag that selects all **film** elements.

- The **film** template should produce a table row with 4 cells. The cells should display the title, the date, the actor, and whether or not the film won or was nominated for an oscar.
- The films should be ordered by **date**.

5. Save your XSL file (**moviexsl.xsl**).
6. Open **actorlist.xml** in Internet Explorer.
7. If you get an error message, go back into Notepad, edit your file, and reload.

## If you are done early…

- Add another row of headers at the bottom of the table.
- Sort your table first by oscar status, second by actor, and third by date.
- Add some CSS formatting to the individual cells of your table.

## A Possible Solution to Exercise 16

As contained in **moviexsl-ex16-done.xsl**:

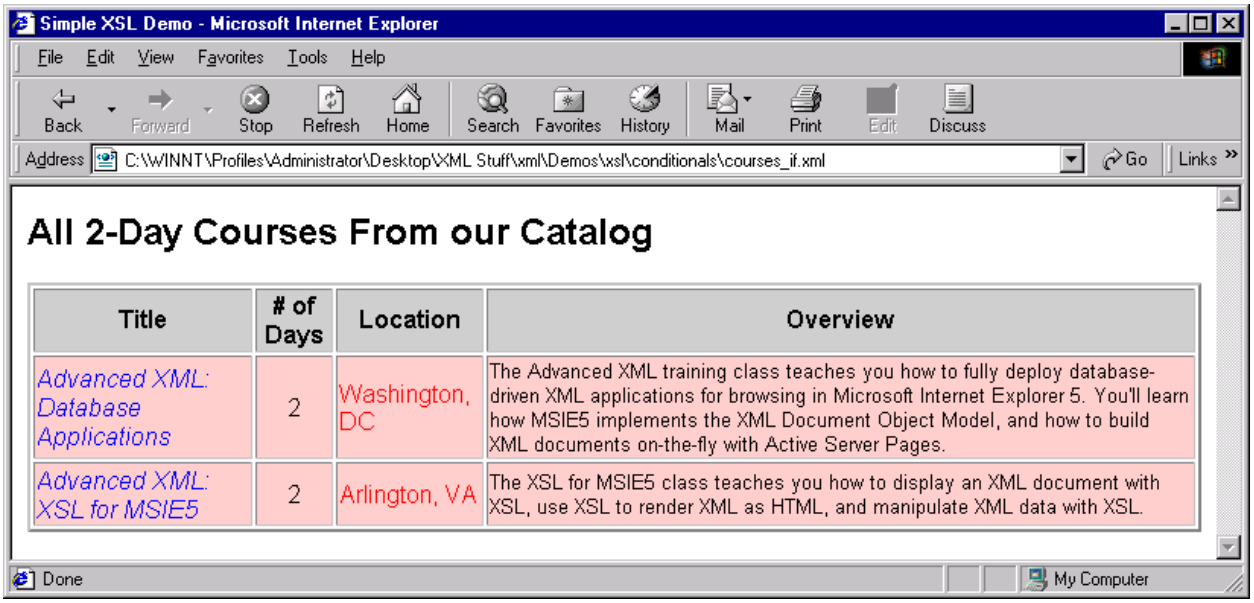
```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
         <head>
            <title>Our List of Fine Films</title>
         </head>
         <body>
         <h3>Our List of Fine Films</h3>
         <table border="1">
         <tr>
            <th>Title</th>
            <th>Date</th>
            <th>Actor</th>
            <th>Oscar?</th>
         </tr>
         <xsl:apply-templates select="//film">
            <xsl:sort select="date" order="ascending" />
         </xsl:apply-templates>
         </table>
         <p>More information available at:
            <a>
               <xsl:attribute name="href">
                    <xsl:value-of select="actors/url" />
               </xsl:attribute>
               <xsl:value-of select="actors/url" />
            </a>
         </p>
         </body>
      </html>
   </xsl:template>
   <xsl:template match="film">
      <tr>
         <td><xsl:value-of select="title" /></td>
         <td><xsl:value-of select="date" /></td>
         <td><xsl:value-of select="../../name" /></td>
         <td><xsl:value-of select="@oscar" /></td>
      </tr>
   </xsl:template>
</xsl:stylesheet>
```

# Conditional Logic in XSL

In XSL, as in any scripting language, we need some way to test whether some condition is satisfied. We have a rough approximation with the XPath expressions, but we would often like a more explicit if…else construction. The tags we will be using are the **xsl:if**, **xsl:choose**, **xsl:when**, and **xsl:otherwise** tags. To get a feel for their syntax, we will look at a couple of demos.

## xsl:if For Conditional Output

The first demo uses the **xsl:if** conditional (**demos > xsl > conditionals > courses_if.xml**):




The XSL stylesheet that produces the above output is below (**xml > demos > xsl > courses_if.xsl**). Pay particular attention to the portions in **bold**:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match="/">
   <html>
   <head>
      <title>Simple XSL Demo</title>
   </head>
   <body>
      <div style="font-size:12pt;font-family:Arial">
      <h2>All 2-Day Courses From our Catalog</h2>
      <table border="2">
         <tr bgcolor="#cccccc">
```

```
            <th>Title</th>
            <th># of Days</th>
            <th>Location</th>
            <th>Overview</th>
         </tr>
         <xsl:apply-templates select="courses/course">
            <xsl:sort select="title" />
         </xsl:apply-templates>
      </table>
      </div>
   </body>
   </html>
</xsl:template>

<xsl:template match="course">
   <xsl:if test="course-length='2'">
      <tr style="background-color:#ffcccc">
         <td style="font-style:italic;color:blue">
            <xsl:value-of select="title" />
         </td>
         <td align="center">
            <xsl:value-of select="course-length" />
         </td>
         <td style="color:red">
            <xsl:value-of select="location/city" />,
            <xsl:value-of select="location/state" />
         </td>
         <td style="font-size:10pt">
            <xsl:value-of select="description" />
         </td>
      </tr>
   </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

As you can see above, the entire output of the **xsl:template match="course"** block is wrapped inside an **xsl:if** tag. This tag specifies that the output is only to be executed if the condition specified by the **test** attribute tests true. We'll take a deeper look at the test attribute a little later. For now, it is important to recognize that all test attributes contain expressions that are evaluated with respect to the active node. So, we are checking the condition, from the current course node, that the **course-length** value equal '**2**'.

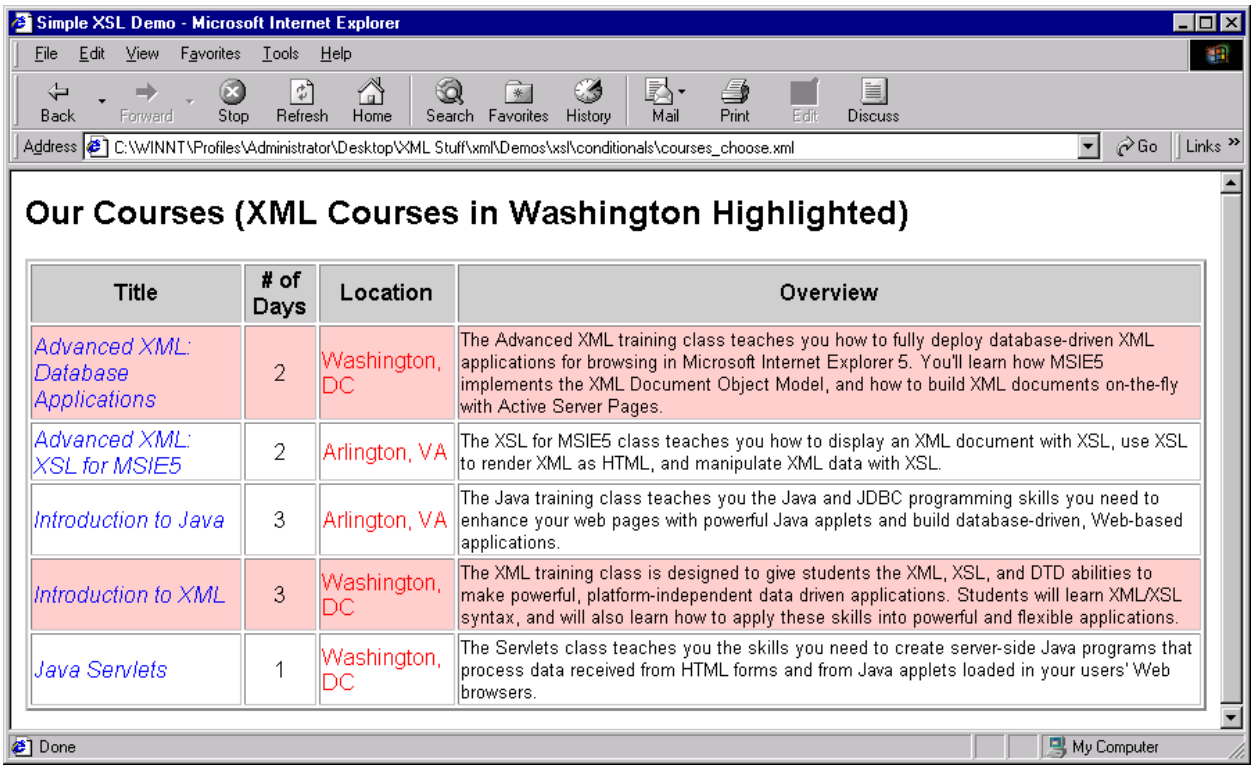
If you have programmed in a procedural language such as JavaScript, C, VB, or Perl, you might well wonder what the syntax is for an "else" condition to the above **xsl:if** condition. However, in XSL, there is no "else" condition. Instead, for any condition that has two or more possible outcomes, you use a construction similar to the case construct in many languages: a combination of the tags **xsl:choose**, **xsl:when**, and **xsl:otherwise**.

If you are wondering why the designers of XSL chose not to implement an xsl:else tag, you aren't alone. The reason is that there is a benefit in performance to have the program flow for

all tags be unambiguous. So, when the processor reaches an xsl:if tag, it doesn't have to wait until that tag ends to know whether or not there are additional conditions it has to check.

## Multi-Option Branching with xsl:choose, xsl:when, and xsl:otherwise

To get a feel for the syntax involved in multi-option branching with XSL, take a look at the following example (**demos > xsl > conditionals > courses_choose.xml**):




You will note that the certain courses are highlighted with a pink background, while others are not. How was this done? Take a look at the code for the XSL stylesheet (**demos > xsl > conditionals > courses_choose.xsl**), particularly the sections in **bold**:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match="/">
   <html>
   <head>
      <title>Simple XSL Demo</title>
   </head>
   <body>
      <div style="font-size:12pt;font-family:Arial">
      <h2>Our Courses (XML Courses in Washington Highlighted)</h2>
      <table border="2">
```

```
         <tr bgcolor="#cccccc">
            <th>Title</th>
            <th># of Days</th>
            <th>Location</th>
            <th>Overview</th>
         </tr>
         <xsl:apply-templates select="courses/course">
            <xsl:sort select="title" />
         </xsl:apply-templates>
      </table>
      </div>
   </body>
   </html>
</xsl:template>

<xsl:template match="course">
   <xsl:choose>
      <xsl:when test="@subject='XML' and location/city='Washington'">
         <tr style="background-color:#ffcccc">
            <td style="font-style:italic;color:blue">
               <xsl:value-of select="title" />
            </td>
            <td align="center">
               <xsl:value-of select="course-length" />
            </td>
            <td style="color:red">
               <xsl:value-of select="location/city" />,
               <xsl:value-of select="location/state" />
            </td>
            <td style="font-size:10pt">
               <xsl:value-of select="description" />
            </td>
         </tr>
      </xsl:when>
      <xsl:otherwise>
         <tr>
            <td style="font-style:italic;color:blue">
               <xsl:value-of select="title" />
            </td>
            <td align="center">
               <xsl:value-of select="course-length" />
            </td>
            <td style="color:red">
               <xsl:value-of select="location/city" />,
               <xsl:value-of select="location/state" />
            </td>
            <td style="font-size:10pt">
               <xsl:value-of select="description" />
            </td>
         </tr>
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>
</xsl:stylesheet>
```

As you can see from the above example, we are surrounding our output in our xsl:template block with an **xsl:choose** block. Inside that **xsl:choose** block, we have two options, **xsl:when**, and **xsl:otherwise**.

## The xsl:choose, xsl:when, and xsl:otherwise tags

To identify a block inside which you have multiple possibilities, you wrap the entire condition within **xsl:choose** tags. **xsl:choose** takes no attributes. You may have one or more **xsl:when** tags inside the **xsl:choose** block. If none of them are satisfied, an optional **xsl:otherwise** block may be specified. The simplified syntax looks like:

```
<xsl:choose>
   <xsl:when test="some condition">
      code_to_execute
   </xsl:when>
   <xsl:when test="some other condition">
      other_code_to_execute
   </xsl:when>
   <xsl:otherwise>
      code_to_execute_if_neither_above_condition_tested_true
   </xsl:otherwise>
</xsl:choose>
```

Whenever one condition tests true, the XSL parser executes the appropriate code for that test block, and exits out of the **xsl:choose**.

| NOTE  | **Why All the Redundant Code?**<br><br>You may have noticed in the above example that you repeated a great deal of code between your xsl:when and your xsl:otherwise tags. The reason for this is that all XSL documents have to be nested correctly, as they must be valid XML. If you didn't surround the entire <tr> element with your condition, you would provoke an XML error before your script were processed by XSL! |
|---|---|

## Conditional Operators in XSL

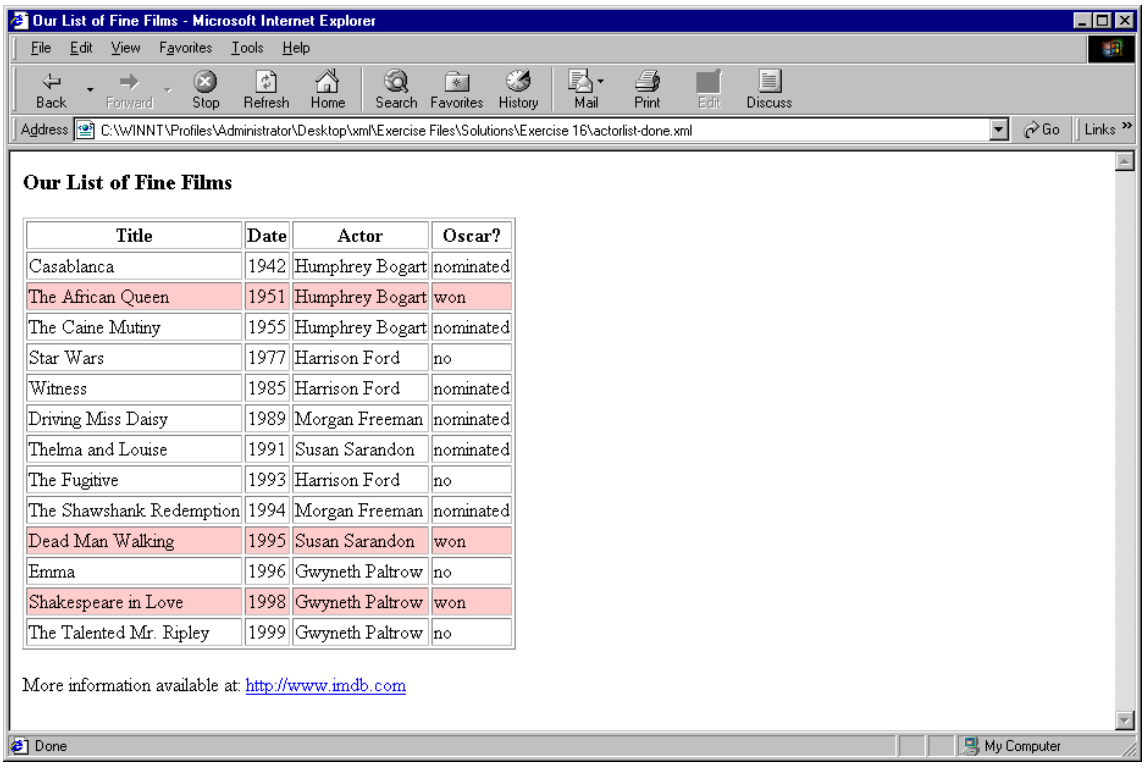
For conditional operators in XSL, you use standard comparison operators. **However**, as you cannot use either < or <= in XSL, as the less-than sign is always interpreted as an opening tag element, you must use the escaped equivalent. A complete table of options is below:

| Condition | Recommended Operator |
|---|---|
| greater-than | `>` |
| less-than | `<` |
| greater-than-or-equal | `>=` |
| less-than-or-equal | `<=` |
| equal to | `=` |
| not equal to | `!=` |
| and | `and` |
| or | `or` |

In addition, in XSL, there is extensive ability to perform more complex operations on the data, or to work with aggregate, or evaluative functions. We will begin our investigation of these functions and expressions in the next section.

# Exercise 17: Using XSL Conditionals to Identify Oscar-Winners

In this exercise, you will be improving your movie list to note the films that won Oscars. When you have completed the exercise, your display should look as follows:




To complete this exercise:

1. Re-open **moviexsl.xsl** in Notepad.
2. Modify your film template so that if the film's **oscar** attribute is 'won' it displays a background color of pink (#ffcccc) for that row.
3. Save your file.
4. Open **actorlist.xml** in Internet Explorer.
5. If you get an error message, go back into Notepad, edit your file, and reload.

## If you are done early…

- Modify the display so that in addition to marking the Oscar-winning films, Oscar-nominated films are highlighted with a lighter-pink background (#ffeeee).

## A Possible Solution to Exercise 17

As contained in **moviexsl-ex17-done.xsl**:

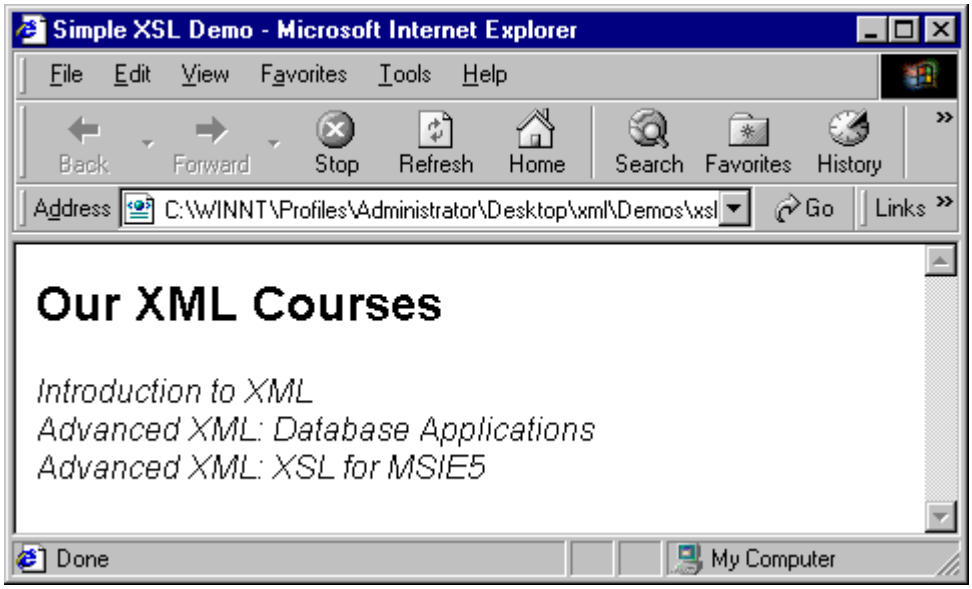
```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:template match="/">
      <html>
         <head>
            <title>Our List of Fine Films</title>
         </head>
         <body>
         <h3>Our List of Fine Films</h3>
         <table border="1">
         <tr>
            <th>Title</th>
            <th>Date</th>
            <th>Actor</th>
            <th>Oscar?</th>
         </tr>
         <xsl:apply-templates select="//film">
            <xsl:sort select="date" order="ascending" />
         </xsl:apply-templates>
         </table>
         <p>More information available at:
            <a>
               <xsl:attribute name="href">
                     <xsl:value-of select="actors/url" />
               </xsl:attribute>
               <xsl:value-of select="actors/url" />
            </a>
         </p>
         </body>
      </html>
   </xsl:template>
   <xsl:template match="film">
      <tr>
         <xsl:attribute name="style">
            <xsl:if test="@oscar='won'">background-color:#ffcccc</xsl:if>
         </xsl:attribute>
         <td><xsl:value-of select="title" /></td>
         <td><xsl:value-of select="date" /></td>
         <td><xsl:value-of select="../../name" /></td>
         <td><xsl:value-of select="@oscar" /></td>
      </tr>
   </xsl:template>
</xsl:stylesheet>
```

# XPath Expressions and XSL Functions

We looked briefly at XPath earlier in the course. It is worth spending some additional time examining the filtering capabilities of XPath.

## XPath Expressions and Filters

Very often, you want to apply a template to some nodes of a given name but not others, depending on their contents. You do this by filtering a node identification with an XPath expression similar to one you would use as the value of a **test** attribute in a conditional expression. To get a sense of how this works, take a look at **demos > xsl > courses-expression.xml**:




What do you notice about the display? It now only displays courses with the subject attribute of XML. You could do this by adding an xsl:if test around the contents of your **course** template, but it is more efficient to filter the node list before the sub-template is called. The code for this file (**courses-expression.xsl**) is below:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
      <head>
         <title>Simple XSL Demo</title>
      </head>
      <body>
         <div style="font-size:12pt;font-family:Arial">
         <h2>Our XML Courses</h2>
```

```
            <xsl:apply-templates select="courses/course[@subject='XML']" />
            </div>
         </body>
         </html>
      </xsl:template>

      <xsl:template match="course">
         <div style="font-style:italic"><xsl:value-of select="title" /></div>
      </xsl:template>

   </xsl:stylesheet>
```

You may find it useful to think of the expression inside the square brackets as the equivalent of an SQL "**WHERE**" clause. So, to translate the above expression: *all **course** children of **courses** where the **subject** attribute equals 'XML".*

## Operators in XSL Expressions

| Operator | Description |
|---|---|
| = | equals |
| != | not equal to |
| < | less than |
| > | greater than |
| <= | less than or equal to |
| >= | greater than or equal to |
| and | joins two conditions. i.e. [@subject='XML' and course-length='2'] |
| or | tests for either of two conditions. i.e. [location/state='DC' or location/state='NY'] |
| not(*expr*) | tests for the opposite of the expression |

Note that you cannot use the less-than symbol in XSL, as it is a reserved character in XML. So, you use **<** (its character equivalent).

## The not() function

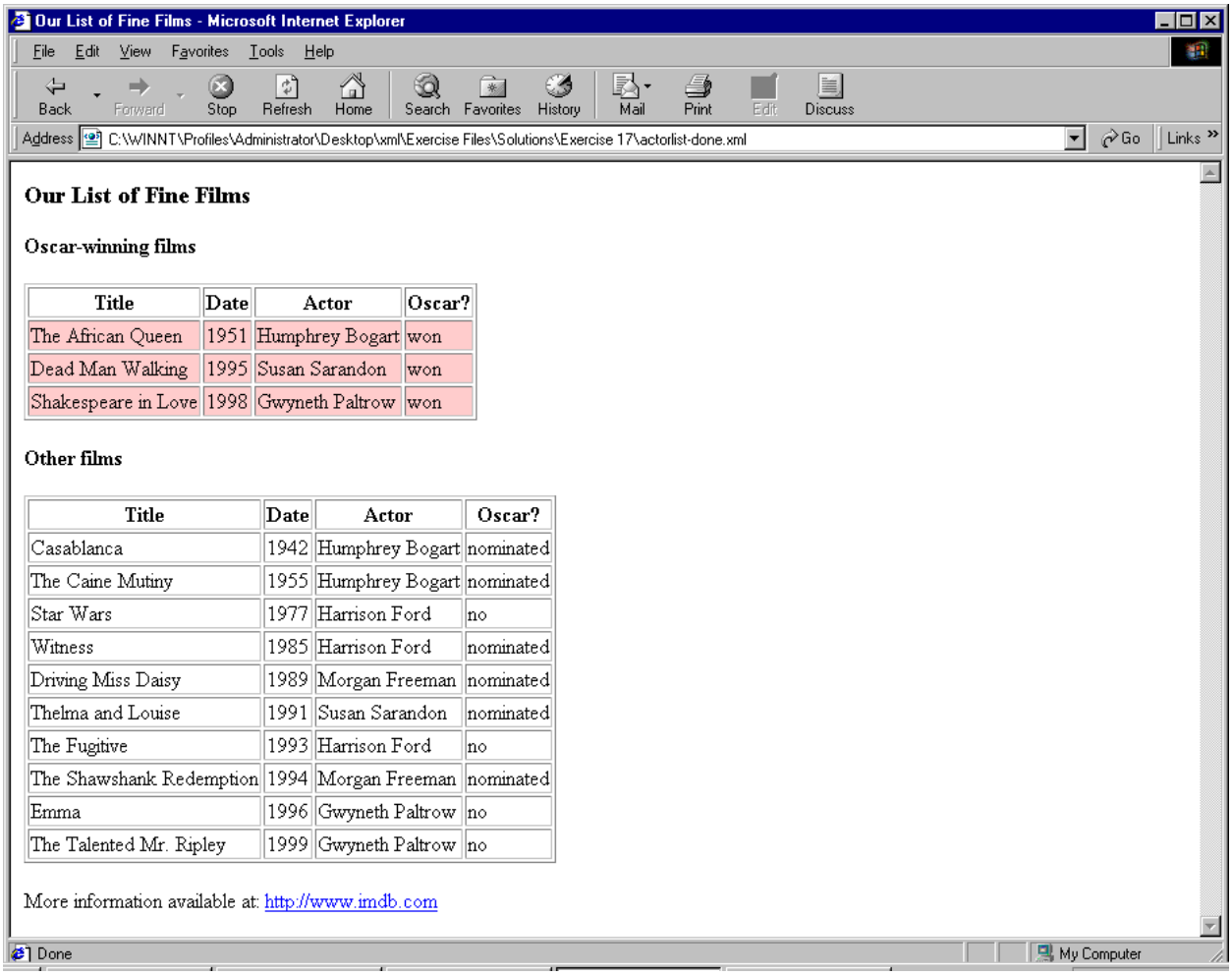
If you wish to test for the logical converse of a given expression, you can use the **not()** function:

```
select="courses/course[not(contains(., 'DTD'))]"
```

This would test for any course which does not contain the text 'DTD'. If you're curious, the **contains()** operator is one of the first instances we've seen of string manipulation and testing functions. Both operators and functions will be covered more thoroughly in the next section.

# Exercise 18: Using XPath Filtering Expressions

In this exercise, you will be splitting your xsl:apply-templates expression into 2 separate expressions. Each will contain a filter. The first expression will select only Oscar-winning films, while the second will select all other films. When you are done, your page will look like:




For this exercise, complete the following:

1. Open **actorlist.xml** in Notepad. Change the stylesheet link so that it points at **moviexsl-separated.xsl**.

2. Save **actorlist.xml**.

3. Re-open **moviexsl.xsl** in Notepad. Save it as **moviexsl-separated.xsl**.

4. Modify your root template to contain an additional table tag, with an **xsl:apply-templates** expression inside the table.

5. The first expression should select all films where the **oscar** attribute is equal to 'won'.

6. The second expression should select all other films (note that there are a couple of different ways to solve this).

7. Add some heading text to identify the difference between the two tables.

8. Save your file (again, as **moviexsl-separated.xsl**).

9. Open **actorlist.xml** in Internet Explorer.

10. If you get an error message, go back into Notepad, edit your file, and reload.

## If you are done early...

- Make a third table and separate the films that were nominated for an Oscar from those that were not.

- Separate your tables so that they have a section for "modern" films (since 1990) and another for "classic" films (before 1990).

## A Possible Solution to Exercise 18

As contained in **moviexsl-ex18-done.xsl**:

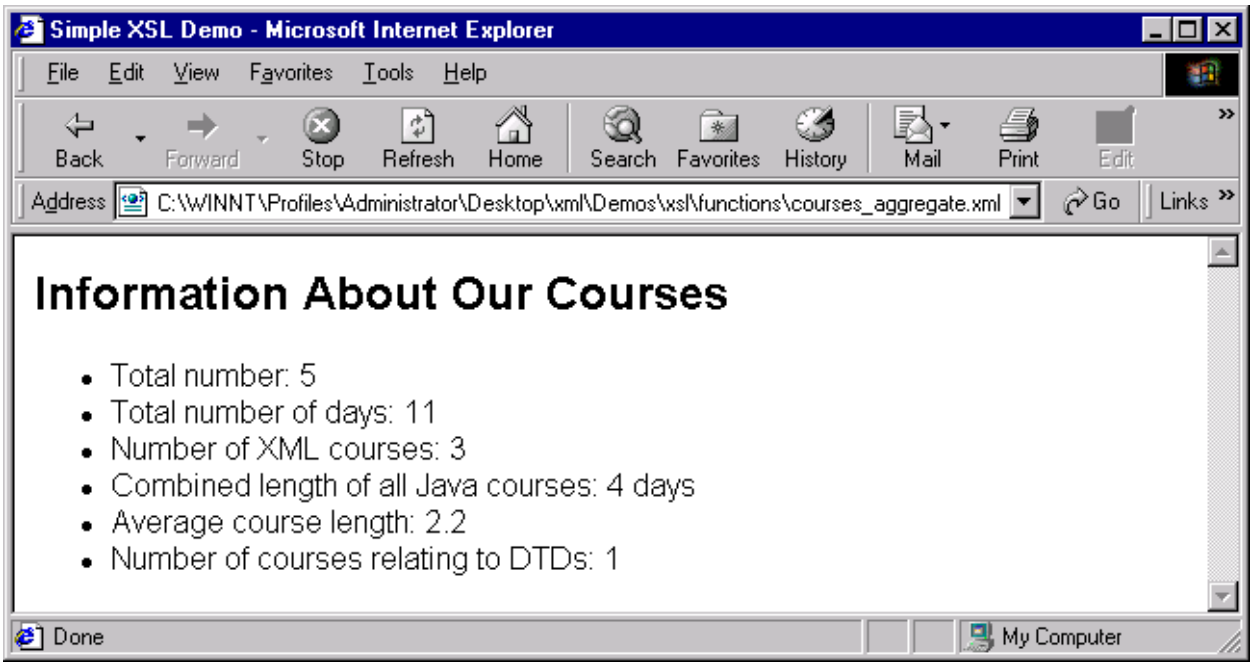
```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
         <head>
            <title>Our List of Fine Films</title>
         </head>
         <body>
         <h3>Our List of Fine Films</h3>
         <h4>Oscar-winning films</h4>
         <table border="1">
         <tr>
            <th>Title</th>
            <th>Date</th>
            <th>Actor</th>
            <th>Oscar?</th>
         </tr>
         <xsl:apply-templates select="//film[@oscar='won']">
            <xsl:sort select="date" order="ascending" />
         </xsl:apply-templates>
         </table>
         <h4>Other films</h4>
         <table border="1">
         <tr>
            <th>Title</th>
            <th>Date</th>
            <th>Actor</th>
            <th>Oscar?</th>
         </tr>
         <xsl:apply-templates select="//film[@oscar!='won']">
            <xsl:sort select="date" order="ascending" />
         </xsl:apply-templates>
         </table>
         <p>More information available at:
            <a>
               <xsl:attribute name="href">
                    <xsl:value-of select="actors/url" />
               </xsl:attribute>
               <xsl:value-of select="actors/url" />
            </a>
         </p>
         </body>
      </html>
   </xsl:template>
   <xsl:template match="film">
      <tr>
         <xsl:attribute name="style">
```

```
            <xsl:if test="@oscar='won'">background-color:#ffcccc</xsl:if>
          </xsl:attribute>
          <td><xsl:value-of select="title" /></td>
          <td><xsl:value-of select="date" /></td>
          <td><xsl:value-of select="../../name" /></td>
          <td><xsl:value-of select="@oscar" /></td>
       </tr>
    </xsl:template>
</xsl:stylesheet>
```

## Aggregate Functions

In addition to its simple comparative operators, XSL makes available a wide array of functions, which can be applied to any set of nodes that you select. Some of the most useful of these are the two basic aggregate functions, **sum()** and **count()**.

To get a sense of how this works, take a look at the following example page (**demos > xsl > functions > courses_aggregate.xml**):




The XSL code used to produce this report is relatively straightforward (**demos > xsl > functions > courses_aggregate.xsl**):

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match="/">
<html>
   <head>
      <title>Simple XSL Demo</title>
   </head>
   <body>
      <div style="font-size:12pt;font-family:Arial">
      <h2>Information About Our Courses</h2>
      <ul>
         <li>Total number:
            <xsl:value-of select="count(//course)" /></li>
         <li>Total number of days:
            <xsl:value-of select="sum(//course-length)" /></li>
         <li>Number of XML courses:
```

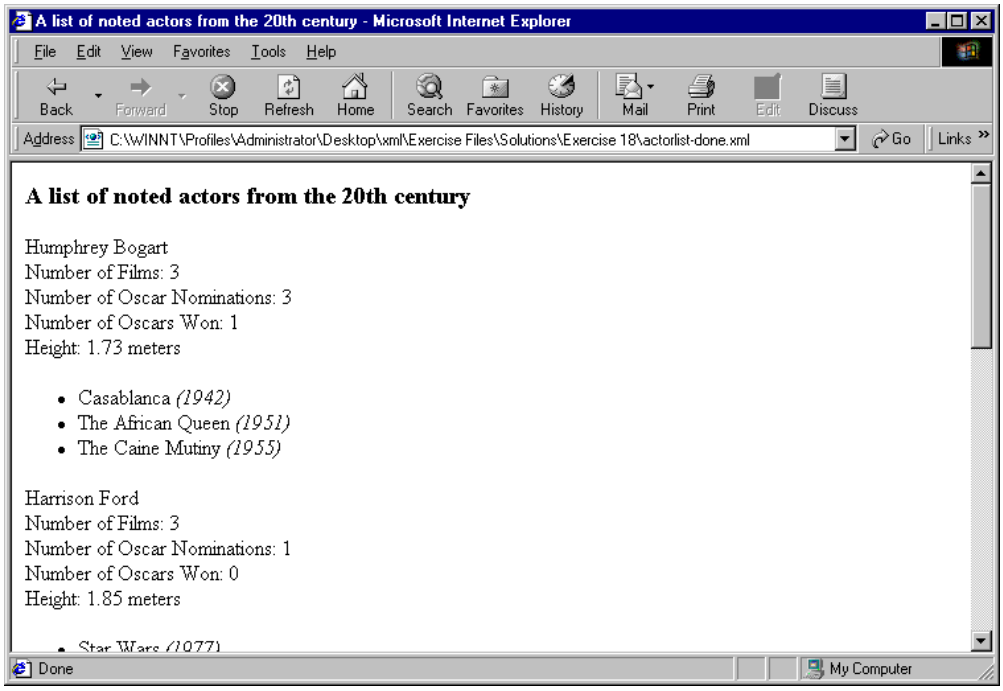
```
            <xsl:value-of select="count(//course[@subject='XML'])" /></li>
         <li>Combined length of all Java courses:
            <xsl:value-of select="sum(//course-length[../@subject='Java'])" /> days</li>
         <li>Average course length:
            <xsl:value-of select="sum(//course-length) div count(//course)" /></li>
         <li>Number of courses relating to DTDs:
            <xsl:value-of select="count(//course[contains(., 'DTD')])" /></li>
      </ul>
      </div>
   </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

As you can see from the above example, **sum()** and **count()** take XPath expressions as their arguments, and return the appropriate values, either summing the numeric values or counting the number or nodes in the node list.

One final note: the **div** operator (in the next to last of the 6 list items in the demo) is the XSL operator for division. While you might expect the forward-slash (/) to control division, it has a reserved XPath meaning.

# Exercise 19: Adding Aggregate Functions to your Stylesheet

In this exercise, you'll be adding a few aggregate functions to your stylesheet. You will count the number of films, nominations, and Oscars for each actor, and display them. When you have completed the exercise, your page should look like:




To complete this exercise, please do the following:

1. Re-open **actorxsl.xsl** in Notepad.
2. Inside your actor template, add 3 lines in the format above. The first should count the number of films in the list. The second should count the number of films that were nominated (remember to count the ones that won as nominations). The third should count the number of Oscar wins.
3. Save **actorxsl.xsl**.
4. Open **actorlist.xml** in Internet Explorer.
5. If you get an error message, go back into Notepad, edit your file, and reload.

## If you are done early...

- Complete the following sentence at the bottom of your root template: "If you were to lay all 5 actors end-to-end, they would be ____ long".

## A Possible Solution to Exercise 19

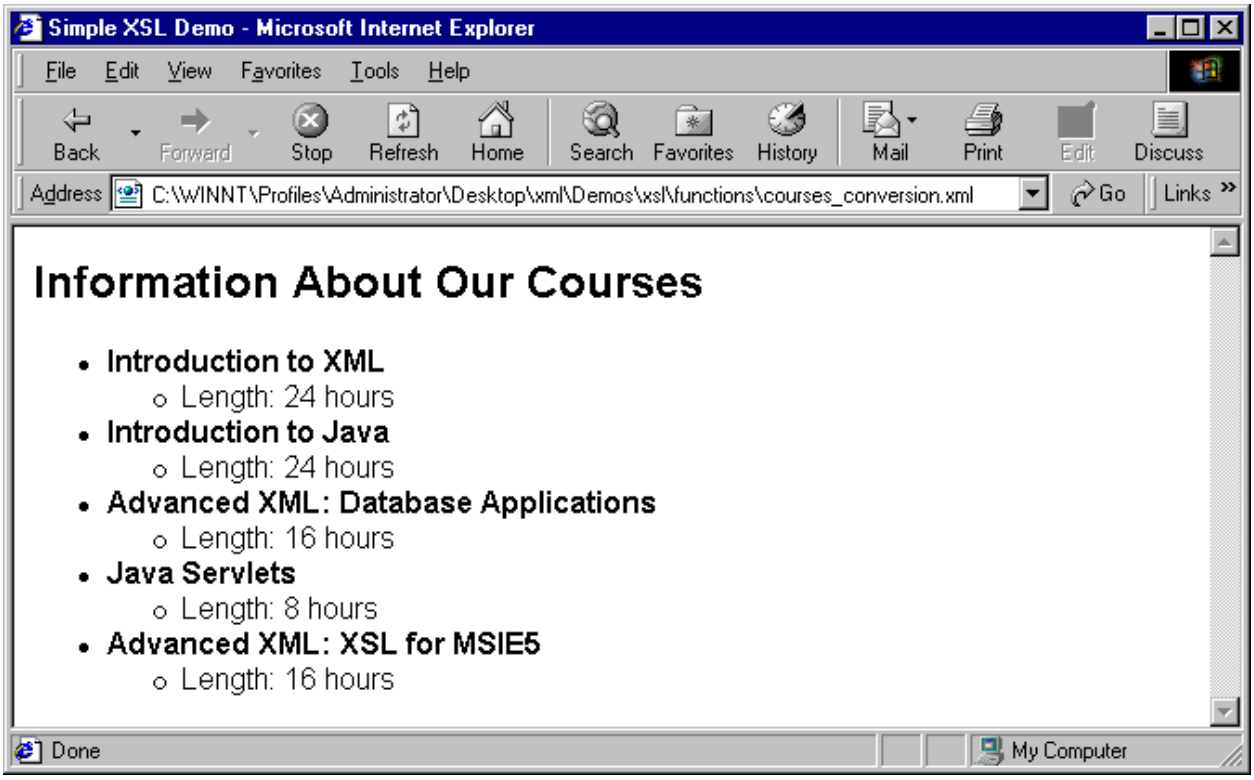
As contained in **actorxsl-ex19-done.xsl**:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
         <head>
            <title><xsl:value-of select="actors/description" /></title>
         </head>
         <body>
         <h3><xsl:value-of select="actors/description" /></h3>
         <xsl:apply-templates select="actors/actor/name">
            <xsl:sort select="lastname" />
         </xsl:apply-templates>
         <p>More information available at:
            <a>
               <xsl:attribute name="href">
                   <xsl:value-of select="actors/url" />
               </xsl:attribute>
               <xsl:value-of select="actors/url" />
            </a>
         </p>
      </body>
      </html>
   </xsl:template>
   <xsl:template match="name">
      <div>
         <xsl:value-of select="." /><br />
         Number of Films: <xsl:value-of select="count(../films/film)" /><br/>
         Number of Oscar Nominations: <xsl:value-of
select="count(../films/film[@oscar='nominated' or @oscar='won'])" /><br/>
         Number of Oscars Won: <xsl:value-of
select="count(../films/film[@oscar='won'])" /><br/>
         Height: <xsl:value-of select="../height" />
         <xsl:text> </xsl:text>
         <xsl:value-of select="../height/@units" />
         <ul>
            <xsl:for-each select="../films/film">
               <xsl:sort select="date" order="ascending" />
               <li>
                   <xsl:value-of select="title" />
                   <xsl:text> </xsl:text>
                   <i>(<xsl:value-of select="date" />)</i>
               </li>
            </xsl:for-each>
         </ul>
      </div>
   </xsl:template>
</xsl:stylesheet>
```

## Data Conversion, Calculations, and Variables

A common use of XSL is to transform information from one format into another. When this information is numeric, these transformations often involve multiplying, dividing, adding, or subtracting. When the transformations are strings, they can involve excerpting substrings, concatenating, or transforming. XSL allows all these sorts of operations.

To get a sense of how basic data transformation works, take a look at the following example (**demos > xsl > functions > courses_conversion.xml**):




You should note that there is a manipulation going on in the display that you see. The course-length has been changed into hours, at the conversion factor of 8 hours per day of class. The XSL code for this example is below (**demos > xsl > functions > courses_conversion.xsl**):

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match="/">
<html>
   <head>
      <title>Simple XSL Demo</title>
   </head>
   <body>
      <div style="font-size:12pt;font-family:Arial">
```

```
      <h2>Information About Our Courses</h2>
      <ul>
      <xsl:for-each select="courses/course">
         <li><b><xsl:value-of select="title" /></b><ul>
            <li>Length: <xsl:value-of select="course-length * 8" /> hours</li>
         </ul></li>
      </xsl:for-each>
      </ul>
      </div>
   </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

As you can see, we are performing a standard mathematical calculation on the value of the **course-length** element. The standard mathematical operators in XSL are:

| Operator | Description |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| div | division (note that the standard division operator, the / forward-slash, has a reserved meaning in XPath) |
| mod | modulus (the remainder) |

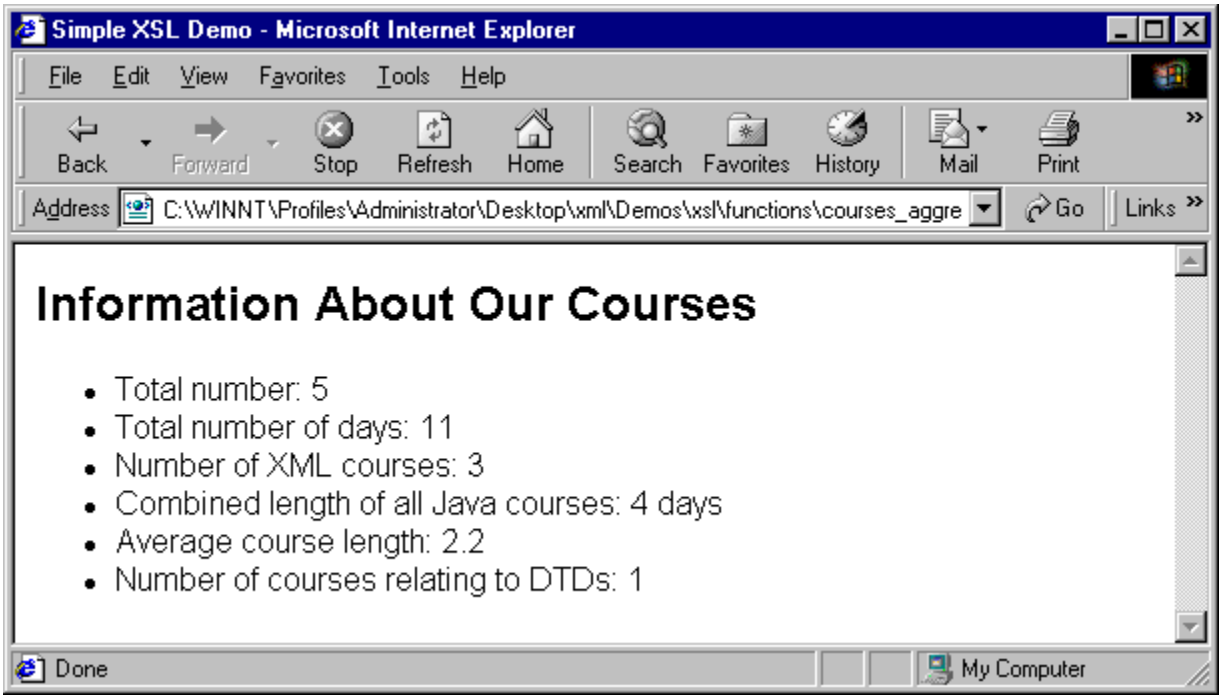
## Variables in XSL and the xsl:variable Tag

In the course of your coding, you may want to declare a value once and use it (once or many times) later in the document. The XSL tag that is used for this is **xsl:variable**. The tag is a bit deceptively named, as it actually can't be varied: once the value of the variable is set, it cannot be changed until the variable goes out of scope. Though this may seem restrictive, there are a number of situations where it is useful to store constants for easy access later.

- **Ease of use**: variables can be declared once, at the top of your page (or even in a separate document which is included into other templates as a library). This means that should you need to access a variable at different points, it is easily accessed. Similarly, if you need to set the value of a variable with an external process (such as a JavaScript command) the value is held once, rather than distributed throughout the page.

- **Conservation of processing:** Declaring a node expression as a variable at the beginning of a template may save your system processing resources, as the expression is evaluated just once, when the variable is declared. Thereafter, the results of the expression can be used without incurring additional processing.

To get a sense of how this might be valuable, take a look at the file **demos > xsl > functions > courses_aggregate_var.xml**:




This is the same display as before. However, the code is significantly more efficient (**courses_aggregate_var.xsl**):

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:variable name="course_length_list" select="//course-length" />
<xsl:variable name="course_list" select="//course" />

<xsl:template match="/">
<html>
   <head>
      <title>Simple XSL Demo</title>
   </head>
   <body>
      <div style="font-size:12pt;font-family:Arial">
      <h2>Information About Our Courses</h2>
      <ul>
         <li>Total number:
            <xsl:value-of select="count($course_list)" /></li>
         <li>Total number of days:
            <xsl:value-of select="sum($course_length_list)" /></li>
         <li>Number of XML courses:
            <xsl:value-of select="count($course_list[@subject='XML'])" /></li>
         <li>Combined length of all Java courses:
            <xsl:value-of select="sum($course_length_list[../@subject='Java'])"
/> days</li>
         <li>Average course length:
            <xsl:value-of select="sum($course_length_list) div
count($course_list)" /></li>
         <li>Number of courses relating to DTDs:
```

```
            <xsl:value-of select="count($course_list[contains(., 'DTD')])"
/></li>
      </ul>
      </div>
   </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

In the above example, we store two node lists as the variables **course_list** and **course_length_list**. These node lists are evaluated once, when the variable is declared. After that, they are held in processor memory, and can be referenced without forcing the processor to go back to the XML document. 2 trips through the XML document rather than 7 is a major improvement in efficiency. For example, the **course_list** variable is declared once:

```
<xsl:variable name="course_list" select="//course" />
```

and then referenced four times, once for the count of courses:

```
<li>Total number: <xsl:value-of select="count($course_list)" /></li>
```

again for the count of XML courses:

```
<li>Number of XML courses:
   <xsl:value-of select="count($course_list[@subject='XML'])" /></li>
```

again for the average course length:

```
<li>Average course length:
   <xsl:value-of select="sum($course_length_list) div count($course_list)" /></li>
```

and finally for the count of courses that contain the text 'DTD':

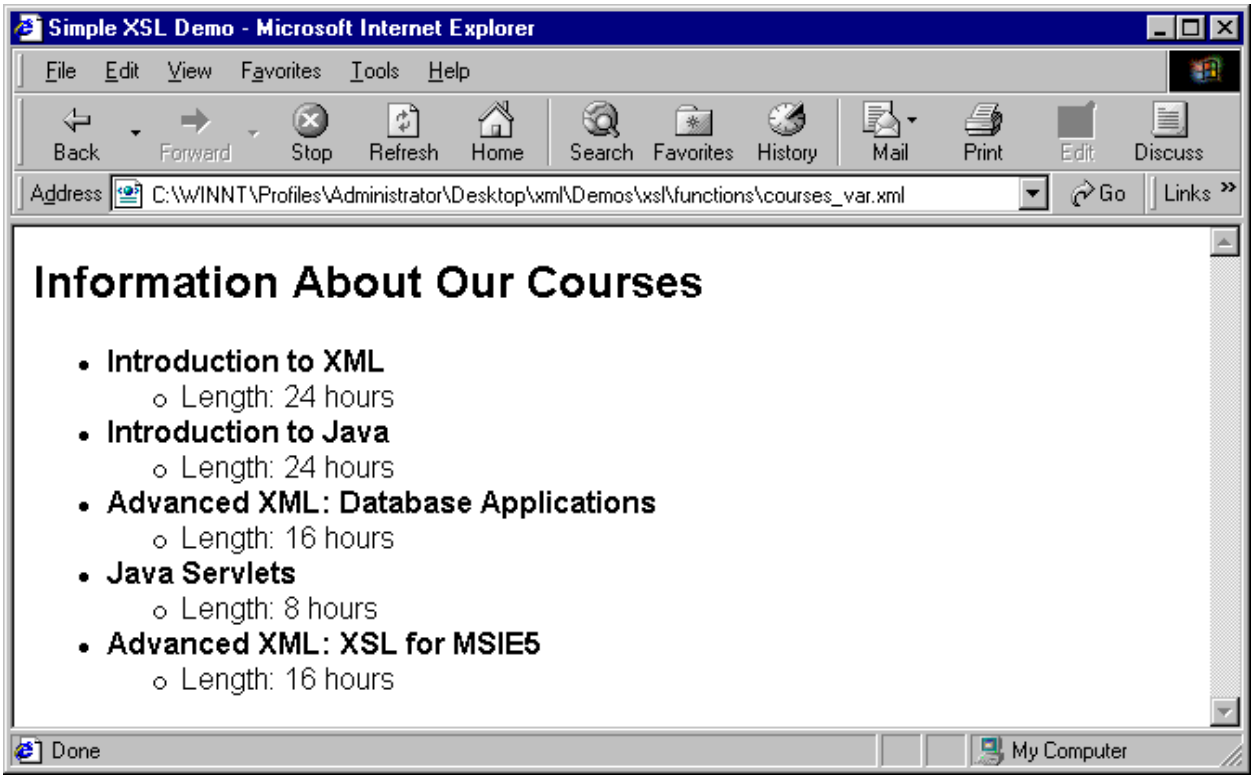
```
<li>Number of courses relating to DTDs:
   <xsl:value-of select="count($course_list[contains(., 'DTD')])" /></li>
```

## Variable Scope

All variables in XSL are strictly scoped. Therefore, a variable that you set in one template is unavailable inside other templates, and, should you wish to, you are welcome to use variables with the same name inside different templates for different purposes.

To set a **global variable**, the variable must be declared before the first **xsl:template** element, as a direct child of the **xsl:stylesheet** tag.

To get another sense of a variable in action, take a look at the following example (**demos > xsl > functions > courses_var.xml**):




The code is below. Pay attention to the differences in **bold**:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:variable name="hrsperday" select="8" />
<xsl:template match="/">
<html>
<head>
   <title>Simple XSL Demo</title>
</head>
<body>
   <div style="font-size:12pt;font-family:Arial">
   <h2>Information About Our Courses</h2>
   <ul>
   <xsl:for-each select="//course">
      <li><b><xsl:value-of select="title" /></b><ul>
         <li>Length: <xsl:value-of select="course-length * $hrsperday" /> hours</li>
      </ul></li>
   </xsl:for-each>
   </ul>
   </div>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

As you can see, once a variable is declared, it can be referenced with the notation *$varname*. The advantages of this are chiefly in clarity and maintainability. It is usually easier and more sustainable to declare constants at the beginning of scripts, rather than leaving hard-coded values scattered throughout. In addition, it makes dynamic transformations of XSL data easier if the data is only held in one place.

## xsl:variable declaration and types

You can declare a value for the **xsl:variable** tag in one of two ways: as you saw above, using the **select** attribute:

```
<xsl:variable name="hrsperday" select="8" />
```

or by writing the expression between the **<xsl:variable>** and **</xsl:variable>** tags:

```
<xsl:variable name="hrsperday">8</xsl:varible>
```

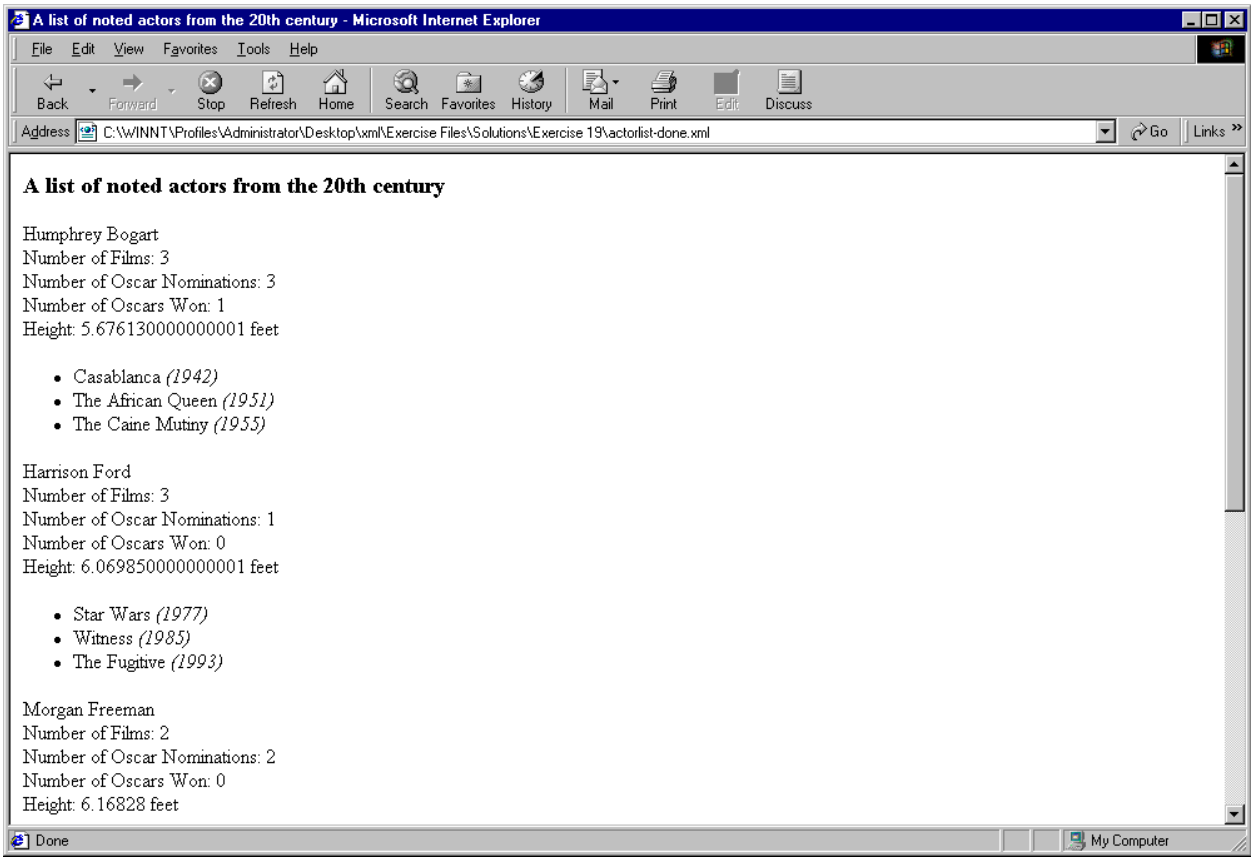
There is no difference in the behavior of the variable depending on how you declare it.

Finally, XSL variables can contain the following types of data:

- Constants
- Single Nodes
- Node Lists

## Exercise 20: Translating Meters to Feet Using XSL

In this exercise, you will be converting the height of actors from meters to feet. We will leave the values in decimal format for now; in the next exercise you will get to calculate round numbers of feet and inches. When you are done, your page should look like:




To complete this exercise, please do the following:

1. Re-open **actorxsl.xsl** in Notepad.

2. Declare 2 variables:

    - At the top of your page, declare a variable named "**meterstofeet**". The value of the variable should be **3.281** (the number of feet in a meter).

    - Inside the **name** template, declare a variable named "**filmlist**". The value of this variable should be the list of all films corresponding to the current actor.

3. In your stylesheet, multiply the values of the height element by the **meterstofeet** variable. If you completed the challenge to the previous exercise (summing the heights of the actors) be sure to convert that value as well.

4. In your **name** template reference the variable **filmlist** (rather than the XML data directly) for your aggregate functions and for your for-each loop.

5. Save **actorxsl.xsl**.

6. Open **actorlist.xml** in Internet Explorer.

7. If you get an error message, go back into Notepad, edit your file, and reload.

## If you are done early...

- Add another listing that displays the actors' heights in centimeters (100 to a meter) or fathoms (0.547 to a meter).

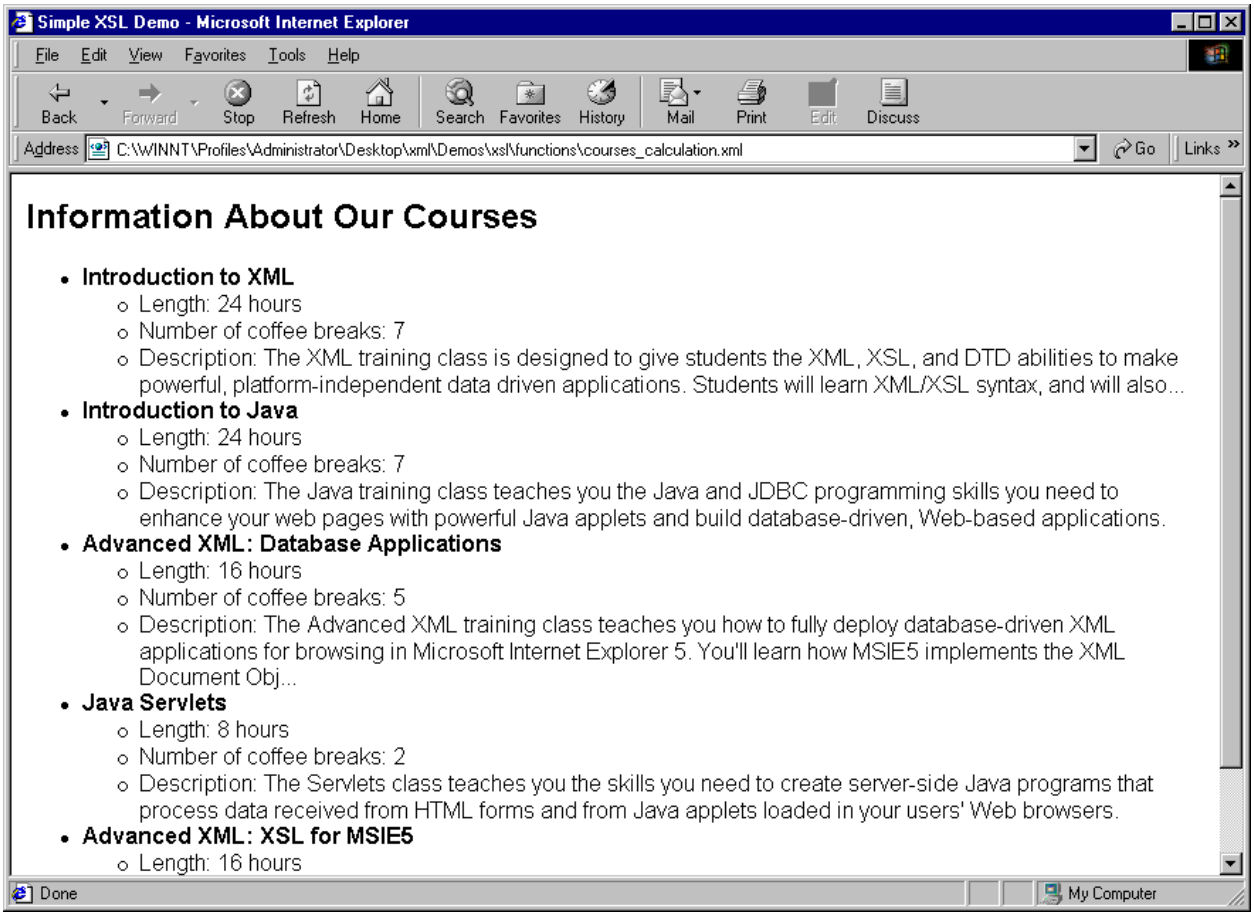
## A Possible Solution to Exercise 20

As contained in **actorxsl-ex20-done.xsl**:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:variable name="meterstofeet">3.281</xsl:variable>
   <xsl:template match="/">
      <html>
         <head>
            <title><xsl:value-of select="actors/description" /></title>
         </head>
         <body>
         <h3><xsl:value-of select="actors/description" /></h3>
         <xsl:apply-templates select="actors/actor/name">
            <xsl:sort select="lastname" />
         </xsl:apply-templates>
         <p>More information available at:
            <a>
               <xsl:attribute name="href">
                   <xsl:value-of select="actors/url" />
               </xsl:attribute>
               <xsl:value-of select="actors/url" />
            </a>
         </p>
      </body>
      </html>
   </xsl:template>
   <xsl:template match="name">
      <xsl:variable name="filmlist" select="../films/film" />
      <div>
         <xsl:value-of select="." /><br />
         Number of Films: <xsl:value-of select="count($filmlist)" /><br/>
         Number of Oscar Nominations: <xsl:value-of
select="count($filmlist[@oscar='nominated' or @oscar='won'])" /><br/>
         Number of Oscars Won: <xsl:value-of
select="count($filmlist[@oscar='won'])" /><br/>
         Height: <xsl:value-of select="../height * $meterstofeet" /> feet
         <ul>
            <xsl:for-each select="$filmlist">
               <xsl:sort select="date" order="ascending" />
               <li>
                   <xsl:value-of select="title" />
                   <xsl:text> </xsl:text>
                   <i>(<xsl:value-of select="date" />)</i>
               </li>
            </xsl:for-each>
         </ul>
      </div>
   </xsl:template>
</xsl:stylesheet>
```

## Calculations and Number Formatting Functions

Let's take a look at the next extension of our example (**demos > xsl > functions > courses_calculation.xml**):




The code for the above example is below (courses_calculation.xsl):

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:variable name="hrsperday" select="8" />
<xsl:variable name="coffeebreaksperhour" select=".3" />
<xsl:variable name="maxlength" select="200" />
<xsl:template match="/">
<html>
   <head>
      <title>Simple XSL Demo</title>
   </head>
   <body>
      <div style="font-size:12pt;font-family:Arial">
      <h2>Information About Our Courses</h2>
```

```
      <ul>
      <xsl:for-each select="courses/course">
         <li><b><xsl:value-of select="title" /></b><ul>
            <li>Length: <xsl:value-of select="course-length * $hrsperday"
/> hours</li>
            <li>Number of coffee breaks: <xsl:value-of
select="round(course-length * $hrsperday * $coffeebreaksperhour)" /></li>
            <li>Description: <xsl:value-of select="substring(description,
1, $maxlength)" />
            <xsl:if test="string-length(description) >
$maxlength">...</xsl:if></li>
         </ul></li>
      </xsl:for-each>
      </ul>
      </div>
   </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

As you can see, there are a few additional calculations, including a usage of the **round()** function. A more complete list of the standard mathematical operators in XSL are:

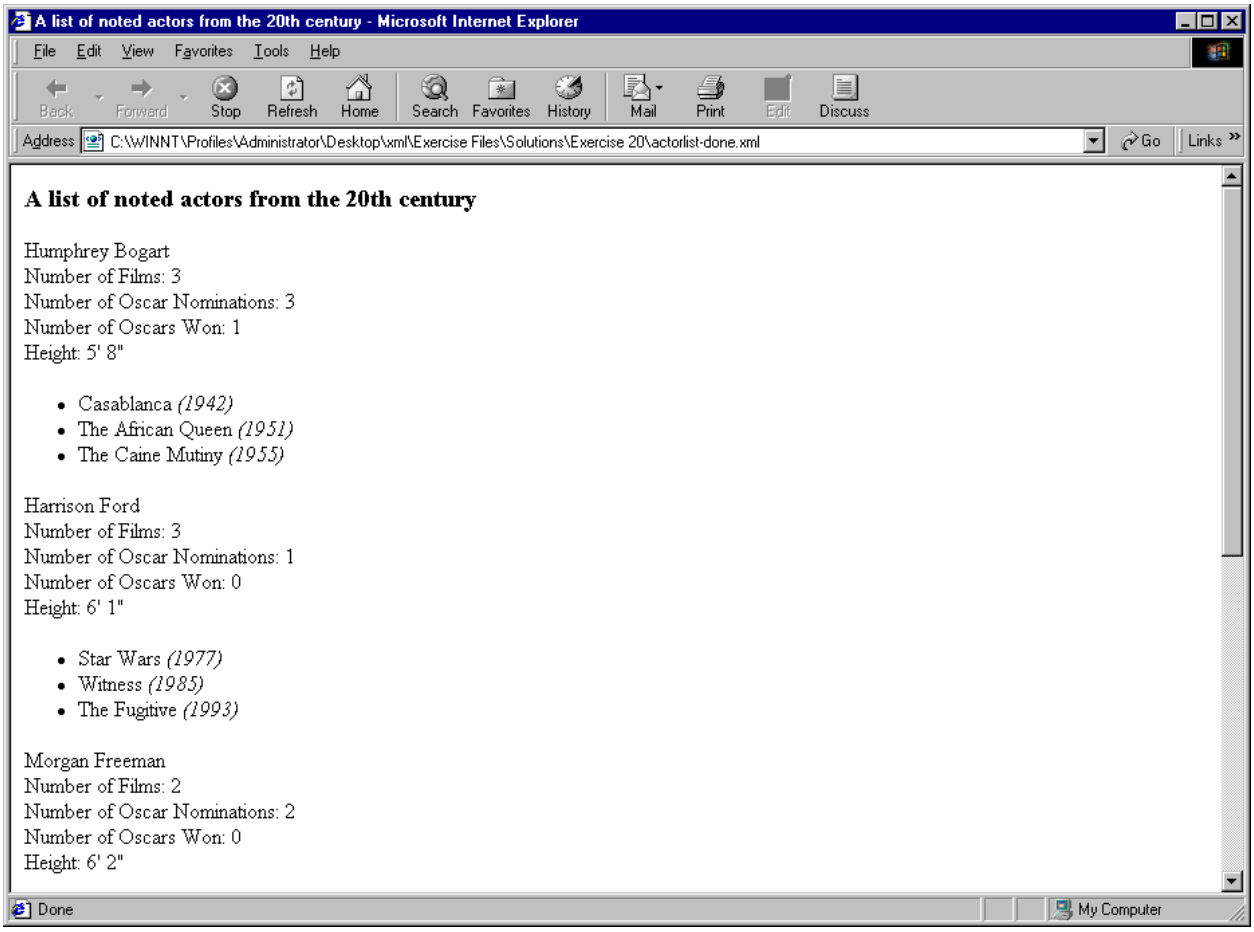
| Operator | Description |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| div | division (note that the standard division operator, the / forward-slash, has a reserved meaning in XPath) |
| mod | modulus (the remainder) |
| round() | rounded to the nearest integer |
| floor() | the nearest integer equal to or less than the value |
| ceiling() | the nearest integer equal to or greater than the value |

Also in the above example, we saw the **contains()** function. Some other common string-manipulation and testing functions are:

| Function | Usage |
|---|---|
| concat() | concatenates a list of comma-delineated values |
| contains() | returns Boolean value (true/false), testing if the node contains the specified value: *contains(node, substring)* |
| starts-with() | like *contains()*, returns a Boolean value, testing if the node begins with the specified value: *starts-with(node, substring)* |
| string-length() | returns a numeric measurement of the number of characters in a given string |
| substring() | returns a substring from a given string, beginning position, and length, in the form: *substring(string, begin, length)* |
| translate() | returns a string, with characters translated as specified from orig to new: *translate(string, orig, new)* |

# Exercise 21: Using XSL Calculations to Produce Feet and Inches

In this exercise, you will be using some mathematical calculations to produce round numbers of feet and inches. Don't worry; we'll give you the mathematical operations you'll need to perform. When you are done, your page should look like (note the nicely rounded heights):




For this exercise, complete the following:

1. Re-open **actorxsl.xsl** in Notepad.

2. Inside your **name** template, declare 3 variables:

    - A variable named "**decimalfeet**". The value of the variable should be the value that you calculated in the previous exercise (the height in meters multiplied by **3.281**).

    - A variable named "**numfeet**". The value of the variable should be the number of feet, passed through the **floor()** function to get a round number of feet.

- A variable named "**numinches**". The value of the variable should be the difference between **decimalfeet** and **numfeet**, multiplied by **12** (to get inches) and then rounded to the nearest whole number with the **round()** function.

3. Where you were displaying the decimal number of feet, now display the number of feet and the number of inches.

4. Save **actorxsl.xsl**.

5. Open **actorlist.xml** in Internet Explorer.

6. If you get an error message, go back into Notepad, edit your file, and reload.

## If you are done early...

- Display, next to the actor's name, the percentage of his or her films that won Oscars.

- Can you figure out a way, given what you know to this point, of rounding the number you calculate above to 2 decimal places? Hint: you'll have to multiply, then round, then divide.

## A Possible Solution to Exercise 21

As contained in **actorxsl-ex21-done.xsl**:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:variable name="meterstofeet">3.281</xsl:variable>
   <xsl:template match="/">
      <html>
         <head>
            <title><xsl:value-of select="actors/description" /></title>
         </head>
         <body>
         <h3><xsl:value-of select="actors/description" /></h3>
         <xsl:apply-templates select="actors/actor/name">
            <xsl:sort select="lastname" />
         </xsl:apply-templates>
         <p>More information available at:
            <a>
               <xsl:attribute name="href">
                    <xsl:value-of select="actors/url" />
               </xsl:attribute>
               <xsl:value-of select="actors/url" />
            </a>
         </p>
      </body>
      </html>
   </xsl:template>
   <xsl:template match="name">
      <xsl:variable name="filmlist" select="../films/film" />
      <div>
         <xsl:value-of select="." /><br />
         Number of Films: <xsl:value-of select="count($filmlist)" /><br/>
         Number of Oscar Nominations: <xsl:value-of
select="count($filmlist[@oscar='nominated' or @oscar='won'])" /><br/>
         Number of Oscars Won: <xsl:value-of
select="count($filmlist[@oscar='won'])" /><br/>
         <xsl:variable name="decimalfeet" select="../height * $meterstofeet" />
         <xsl:variable name="numfeet" select="floor($decimalfeet)" />
         <xsl:variable name="numinches" select="round(12 * ($decimalfeet - $numfeet))" />
         Height: <xsl:value-of select="$numfeet" />' <xsl:value-of
select="$numinches" />"
         <ul>
            <xsl:for-each select="../films/film">
               <xsl:sort select="date" order="ascending" />
               <li>
                    <xsl:value-of select="title" />
                    <xsl:text> </xsl:text>
                    <i>(<xsl:value-of select="date" />)</i>
               </li>
            </xsl:for-each>
         </ul>
      </div>
   </xsl:template>
</xsl:stylesheet>
```

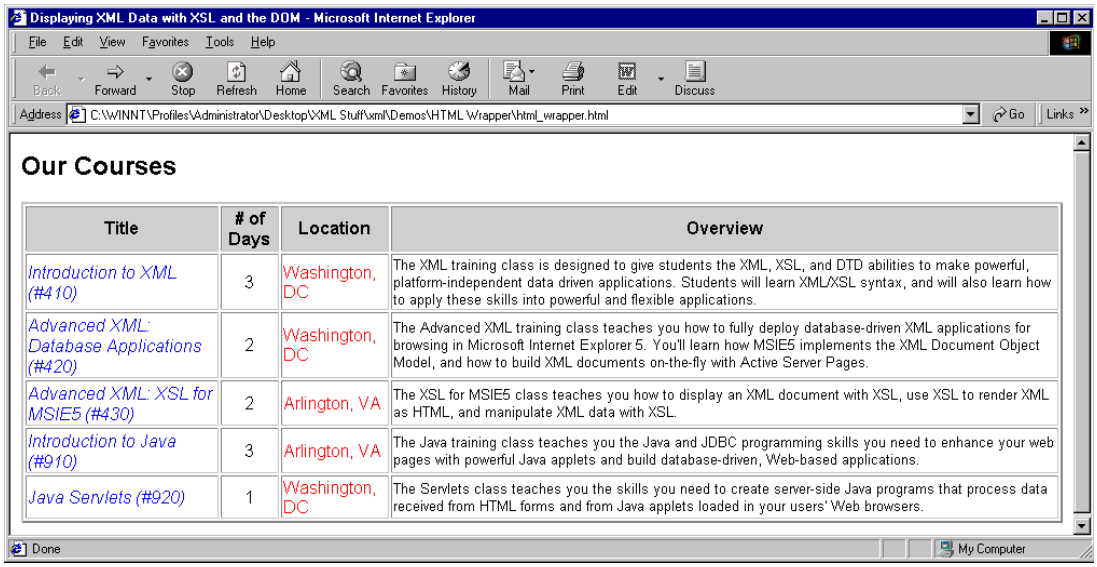
# Building the HTML Front End to XML Data

In most XML applications, whether running on the client or the server, the XML document does not directly reference an XSL stylesheet. The main reason to avoid hard-coding this relationship is that doing so prevents one XML document from having multiple stylesheets. Why would you want multiple stylesheets? One could transform your XML data into HTML for posting on your Intranet, while another could transform it into text for sending out as an email. A third could produce a PDF. If your data changes, you only have to update the XML document once, and the stylesheets can produce new data in all three formats.

So, to allow for flexibility in the application of stylesheets, many applications make use of a "**wrapper page**". This page usually does not contain its own content. Instead, its job is to play traffic cop: to load, as appropriate, the XML document, XSL and/or CSS stylesheets, and script libraries. For client-side XML applications, it is usually HTML that organizes and references the different components. For server-side XML processing, a middleware such as ASP, JSP, PHP, Coldfusion or Perl will usually perform the same tasks. The principal tasks of a wrapper page, whether client-side or server-side, are:

- Load the XML datasheet into memory
- Load the XSL stylesheet into memory
- Evaluate the XML data with respect to the XSL stylesheet
- Place the output on the page

As a quick review, take a look at our demo, reorganized to use an HTML wrapper to pull together our different components (**xml > demos > HTML Wrapper > ie > ie_wrapper.html**):




You should not notice any change. However, as the XML, XSL, and DTD documents are being accessed from an HTML page, it will be easy to integrate CSS and JavaScript information into our application. Take a look at the code for the above HTML page:

```
<html>
<head>
   <title>Displaying XML Data with XSL and the DOM</title>
   <xml id="courselist" src="../courses.xml"></xml>
   <xml id="coursedisplay" src="../courses.xsl"></xml>
</head>

<body onLoad="dataTarget.innerHTML=courselist.transformNode(coursedisplay)">

<div id="dataTarget"></div>

</body>
</html>
```

There are four critical pieces in our HTML wrapper. The first two are relatively self-explanatory. Both the XML document and the XSL stylesheet are loaded into the HTML page as data islands with the <XML> tag. Each is given an **id**. However, if we stopped there, we would not have any visible display. We have not told our HTML page how to associate these data islands with each other, or how they should integrate with the HTML page content.

Also, note the DIV inside the body of the document, named **dataTarget**, but without any initial content. This DIV will eventually contain the formatted HTML output from the XSL stylesheet.

## Data Islands and the HTML <XML> tag

Data islands are produced whenever XML data is loaded into an HTML document. Microsoft has programmed Internet Explorer to allow you to do this very easily with its **<xml>** tag. This

tag can either contain XML data directly, or, much more commonly, can load XML data from an external data source. To process XML data from a data island, the island must be given an **id** attribute:

```
<xml id="dataIsland" src="dataislanddemo.xml"></xml>
```

Once loaded, a data island can be accessed either through JavaScript and the DOM, or through the data binding controls inherent to Internet Explorer 5+.

## Using the transformNode() method to Apply XSL to an XML Document

Any XML-aware technology will have a command for integrating XML and XSL objects. Internet Explorer uses the DOM and JavaScript to do this. Here, we see a built-in method for interrelating XML and XSL documents. When both are loaded as data islands, the syntax is:

```
divReference.innerHTML=XMLIslandID.transformNode(XSLIslandID)
```

As you can see, we are creating HTML output with our XSL stylesheet. So, we need a location for that HTML. Using the **innerHTML** property (not currently approved by the W3C HTML specifications, but understood in IE5+ as well as Netscape 6+), we can reference the text content of a particular page element. In the example above, this is the **dataTarget** div.

**transformNode()** is an IE-specific method of any XML datasheet referenced by a data island, and takes as an argument an XSL stylesheet object (referenced by data island id).

## Linking to CSS from a Wrapper Page

When you build an HTML wrapper page, you create a head element that is independent of your XSL output. However, when you perform your XSL transormation, the full results will appear inside a  in the body of your HTML page. In the XSL sheets we've written thus far, you'll find that there are redundant tags produced by your XSL (such as the <html>, <head>, and <body> tags). Although Internet Explorer will just ignore the tags, do not pull a Blanche DuBois and rely on the kindness of strangers: Netscape/Mozilla will give you bad results if you leave extraneous head elements. Therefore, eliminate any of these redundant tags.

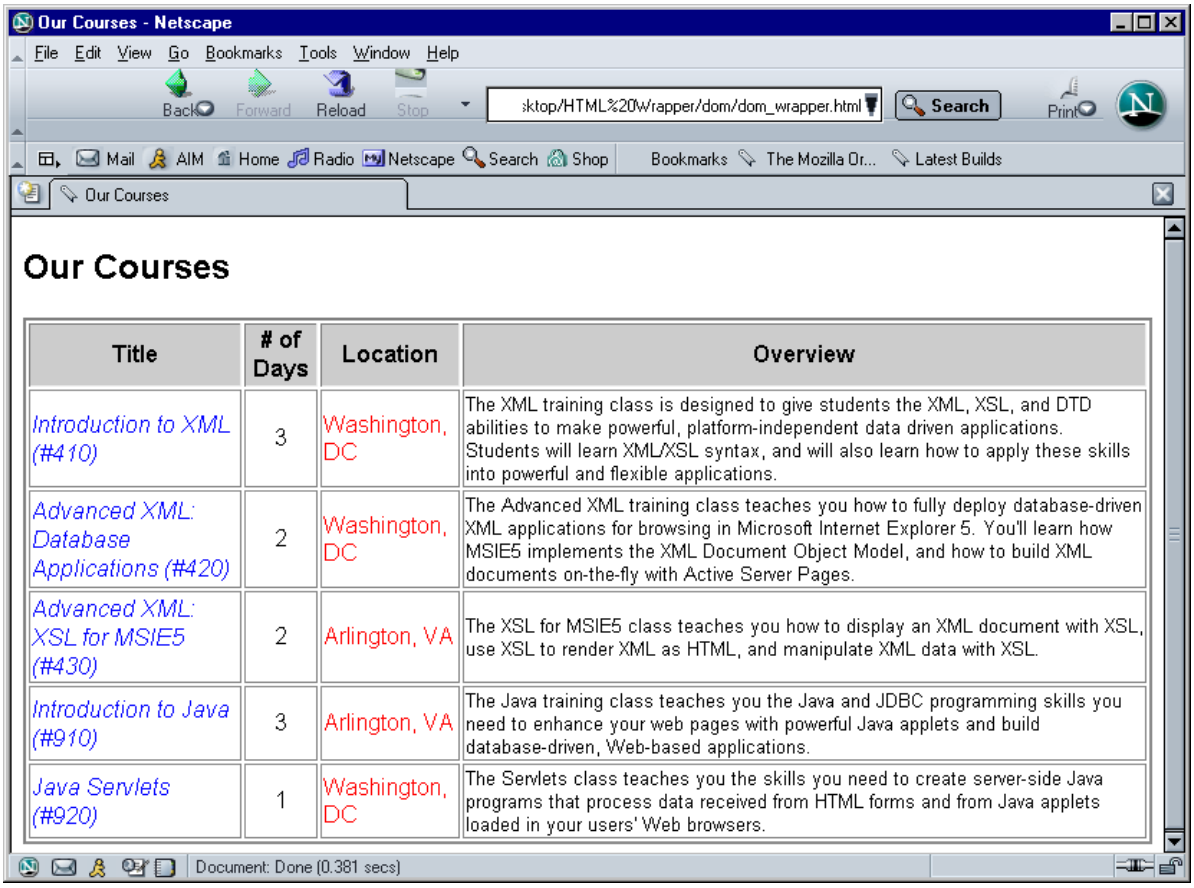
Furthermore, according to HTML specifications, the HTML <link> tag should work only inside the <head> tag (although some browsers will be forgiving about this). So, you'll need to remove any <link> tags from your XSL-generated HTML, and put them in the head of your wrapper page instead.

## Wrapper Pages in Netscape/Mozilla

You may have noticed that the xml-related parts of our wrapper page are IE-specific: the xml data islands, as well as the transformNode() method cannot be used with other browsers. This was acceptable when IE was the only browser that could handle client-side XSL

transformations. However, with browsers such as Mozilla and Netscape 6.2+, IE is no longer the only player in town. That's the good news.

The bad news is that the APIs for other browsers are a bit more cumbersome. Take a look at how we would construct the same wrapper page in Netscape 7 (**xml > demos > HTML Wrapper > dom > dom_wrapper.html**):




## Our Courses

| Title | # of Days | Location | Overview |
|---|---|---|---|
| *Introduction to XML (#410)* | 3 | Washington, DC | The XML training class is designed to give students the XML, XSL, and DTD abilities to make powerful, platform-independent data driven applications. Students will learn XML/XSL syntax, and will also learn how to apply these skills into powerful and flexible applications. |
| *Advanced XML: Database Applications (#420)* | 2 | Washington, DC | The Advanced XML training class teaches you how to fully deploy database-driven XML applications for browsing in Microsoft Internet Explorer 5. You'll learn how MSIE5 implements the XML Document Object Model, and how to build XML documents on-the-fly with Active Server Pages. |
| *Advanced XML: XSL for MSIE5 (#430)* | 2 | Arlington, VA | The XSL for MSIE5 class teaches you how to display an XML document with XSL, use XSL to render XML as HTML, and manipulate XML data with XSL. |
| *Introduction to Java (#910)* | 3 | Arlington, VA | The Java training class teaches you the Java and JDBC programming skills you need to enhance your web pages with powerful Java applets and build database-driven, Web-based applications. |
| *Java Servlets (#920)* | 1 | Washington, DC | The Servlets class teaches you the skills you need to create server-side Java programs that process data received from HTML forms and from Java applets loaded in your users' Web browsers. |

Here is the code:

```
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
    <title>Our Courses</title>
    <script type="text/javascript">

     xsltProc = new XSLTProcessor();

     sourceDoc = document.implementation.createDocument("", "", null);
     styleDoc = document.implementation.createDocument("", "", null);
     resultDoc = document.implementation.createDocument("", "", null);

     sourceDoc.onload = loadStyle;
     sourceDoc.load("../courses.xml");
```

```
        function loadStyle() {
           styleDoc.onload = transform;
           styleDoc.load("../courses.xsl");
        }

        function transform() {
           xsltProc.transformDocument(sourceDoc, styleDoc, resultDoc, null);
           myResult = new XMLSerializer().serializeToString(resultDoc);
           document.getElementById("dataTarget").innerHTML = myResult;
        }

      </script>
    </head>
    <body>
        <div id="dataTarget"></div>
     </body>
  </html>
```

First, we instantiate the parser that will do our transformations and assign it to a variable:

```
xsltProc = new XSLTProcessor();
```

Then, we create three blank documents. The first will eventually contain our XML data. The second will contain our XSL stylesheet. And the third will contain the result of the XSL transformation. To do this, we use the document object's implementation property, which in turn, has a createDocument() method. The createDocument method takes three arguments: a namespace, the name of the root node in the resulting document, and document type. The first two are only relevant if we are producing fragments that would otherwise be invalid xml and if we want to use a namespace for the resulting document. The third argument has yet to be implemented by the browser, so we use the default: null.

```
sourceDoc = document.implementation.createDocument("", "", null);
styleDoc = document.implementation.createDocument("", "", null);
resultDoc = document.implementation.createDocument("", "", null);
```

Now that these document objects have been created, we load XML files into the first two. We also want to make sure that our script does not proceed until these documents are fully loaded: if the script began trying to evaluate the xml with respect to the xsl before both documents were fully loaded, we would get an error. So we add event handlers to these objects and then load data into them.

```
sourceDoc.onload = loadStyle;
sourceDoc.load("../courses.xml");
```

In other words, do not do the "loadStyle" function until sourceDoc has finished loading courses.xml from the parent directory.

```
function loadStyle() {
    styleDoc.onload = transform;
    styleDoc.load("../courses.xsl");
}
```

loadStyle then loads the courses.xsl stylesheet and instructs the script to go on to the transform() function when the stylesheet has been fully loaded.

Finally we come to the `transform()` function. This function evaluates the xml with respect to the xsl and assigns the result to the still-empty `resultDoc` object. It then instantiates an "XMLSerializer" which converts an XML document from an object to a string. We assign this string to the `myResult` variable, and then place the string in the empty `dataTarget` div:

```
function transform() {
   xsltProc.transformDocument(sourceDoc, styleDoc, resultDoc, null);
   myResult = new XMLSerializer().serializeToString(resultDoc);
   document.getElementById("dataTarget").innerHTML = myResult;
}
```

## Cross-Browser Wrapper Pages

So you're dying to know: is it possible to make a wrapper that will work in both browsers? Yes! Let's take a look at **xml > demos > HTML Wrapper > cross-browser > cross_wrapper.html**:

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
       <title>Cross-browser Courses Table</title>

    </head>
    <body>
       <div id="dataTarget"></div>

    </body>
    <script type="text/javascript">

   var ie, dom, sourceDoc, styleDoc;
   ie=false;
   dom=false;
   navigator.appName == "Microsoft Internet Explorer"?ie=true:dom=true;

   if (dom) {
      xsltProc = new XSLTProcessor();
      sourceDoc = document.implementation.createDocument("", "", null);
      styleDoc = document.implementation.createDocument("", "", null);
      resultDoc = document.implementation.createDocument("", "", null);
      sourceDoc.onload = loadStyle;
      sourceDoc.load("../courses.xml");
   } else {
      sourceDoc = new ActiveXObject("MSXML2.DOMDocument.3.0");
      sourceDoc.async = false;
      sourceDoc.load("../courses.xml");
      styleDoc = new ActiveXObject("MSXML2.DOMDocument.3.0");
      styleDoc.async = false;
      styleDoc.load("../courses.xsl");
      transform();
   }
```

```
    function loadStyle() {
       styleDoc.onload = transform;
       styleDoc.load("../courses.xsl");
    }

    function transform() {
       if (dom){
         xsltProc.transformDocument(sourceDoc, styleDoc, resultDoc, null);
         myResult = new XMLSerializer().serializeToString(resultDoc);
       } else {
         myResult = sourceDoc.transformNode(styleDoc);
       }
         document.getElementById("dataTarget").innerHTML = myResult;
    }

     </script>
</html>
```

There's not too much that's new here. First we declare some variables we'll be using. Then we do a browser check to determine if the user is using IE or Netscape/Mozilla. Note that this script will only work in IE or Netscape 6.2+ or Mozilla. When other browsers add xsl functionality, this code can be updated. Note also that the Mozilla API is not considered stable. Changes may be implemented so you will want to test your code periodically in the latest version of the browser.

Next, our main "if" condition determines how to load the xml and xsl files. If the browser is Netscape or Mozilla, we instantiate our XSLTProcessor, instantiate blank document objects and start the loading process. If the browser is IE, we instantiate "ActiveX" objects. This is a type of Microsoft object that allows us to do various kinds of data processing. In this case, our ActiveX Object is the Microsoft XSL processor.

```
sourceDoc = new ActiveXObject("MSXML2.DOMDocument.3.0");
```

This object will eventually be used to load our XML document. This is the same thing that "sourceDoc" does in our Mozilla/Netscape code, so we give it the same name. Pretty clever, eh? Now, we tell the script not to proceed until it has finished loading this document:

```
sourceDoc.async = false;
```

This "turns off" asynchronous loading. That is, the script should not move to the next line of code until it is finished loading `sourceDoc`. We already came across with this problem with the Mozilla code and saw its somewhat less elegant solution of adding event handlers. Now we're ready to load the XML.

```
sourceDoc.load("../courses.xml");
```

Then we repeat this for the stylesheet.

```
 styleDoc = new ActiveXObject("MSXML2.DOMDocument.3.0");
 styleDoc.async = false;
```

```
        styleDoc.load("../courses.xsl");
```

Since we can't add event handlers for IE ActiveX Objects, we simply call the transform function after everything is loaded. Notice, however, that our script comes after the body tags. The reason this is necessary is that we are referencing a div inside the body. With asynchronous loading, the div tag will not have been loaded when we call the transform function. If we put the script after the body tag, however, the div will have been loaded and our page will work as planned.

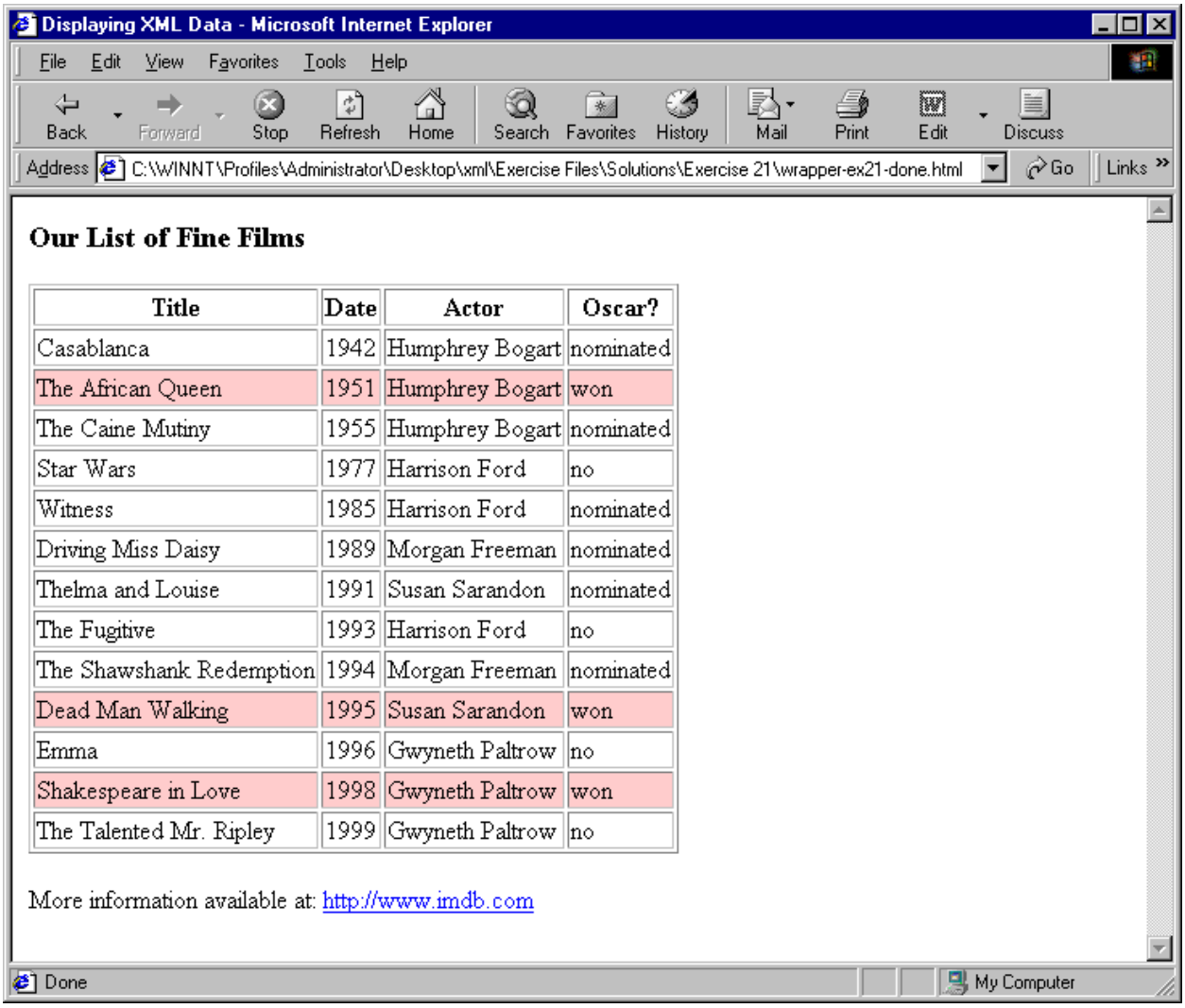
The transform() function has been updated so that it performs the XSL transformation appropriately for the two browsers and saves the result to a variable called myResult. Once the transformation is complete, the code for putting that content into our target  is the same for both browsers:

```
function transform() {
    if (dom){
      xsltProc.transformDocument(sourceDoc, styleDoc, resultDoc, null);
      myResult = new XMLSerializer().serializeToString(resultDoc);
    } else {
      myResult = sourceDoc.transformNode(styleDoc);
    }
      document.getElementById("dataTarget").innerHTML = myResult;
}
```

Voilà! To see an even more flexible version of this code, feel free to check out **xml > demos > HTML Wrapper > cross-browser > cross_wrapper-modular.html**.

## Exercise 22: Creating an HTML Wrapper for your Movie List Application

In this exercise, you will be building an HTML front-end to the XML and XSL files you have been working on over the previous exercises. When you have finished, your display should look just like it did at the end of the previous exercise:




To complete this exercise:

1. Re-open **actorlist.xml** in a text editing program.
2. Remove the link to your XSL stylesheet.
3. Save **actorlist.xml**.

4. Open a new document in Notepad.

5. Build an HTML wrapper for your application. This wrapper should:

    - Load both your XML document and your XSL stylesheet (choose **moviexsl.xsl**) as Data Islands inside your HEAD tag.

    - Contain an empty DIV named "dataTarget" inside the BODY tag.

    - Have an **onLoad** event handler that inserts HTML content into **dataTarget** by executing the **transformNode()** method of the XML datasheet. You will have to pass the contents of the XSL stylesheet as the argument of the transformNode() method.

6. Save your file as **wrapper.html**.

7. Open **wrapper.html** in Internet Explorer.

8. If you get an error message, go back into Notepad, edit your file, and reload.

## If you are done early…

- Some of the content in your XSL stylesheet (such as the <HTML> tags) is now redundant. Eliminate it to clean up your code.

- Add some other HTML content (maybe a page heading?) to your HTML page. Then try adding the same content to the XSL stylesheet.

- Make a Mozilla/Netscape wrapper.

- If you're feeling adventurous, make an IE/Netscape-compatible wrapper.

- If you're feeling really crazy, look at the modular file in your demos folder (**xml > Demos > HTML Wrapper > cross-browser > cross_wrapper-modular.html**). See if you can figure it out. Try making your own version. You might even be able to make a version where people could choose how they want things displayed from a drop-down list of stylesheets (**xml > Demos > HTML Wrapper > cross-browser > cross_wrapper-modular-userdefined.html**).

- Try the XSL challenges from the previous exercises.

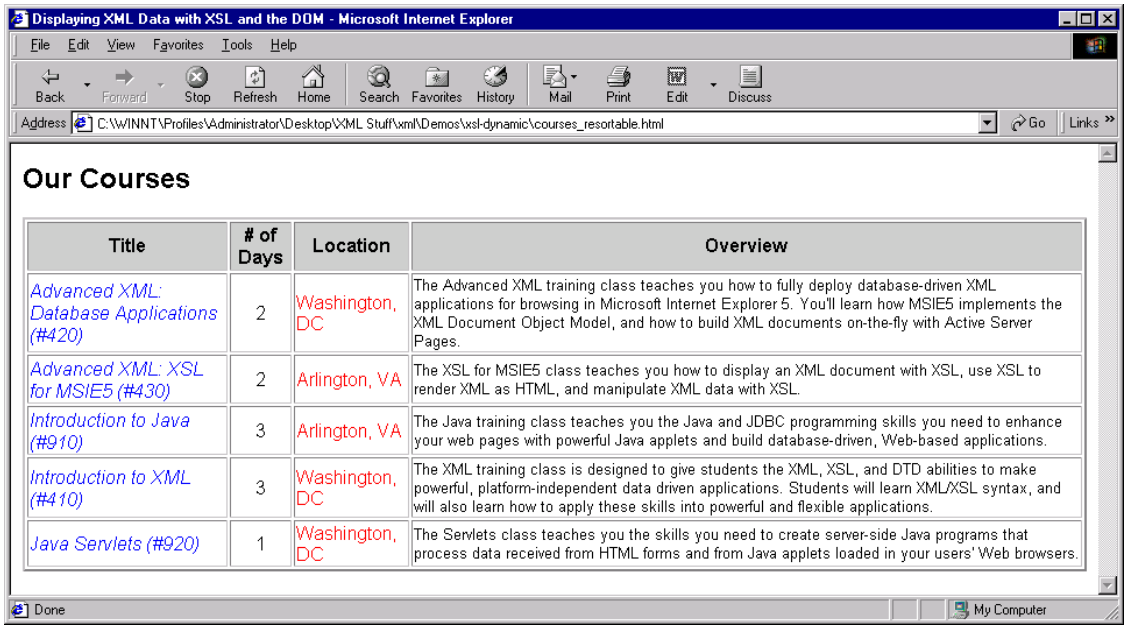
## A Possible Solution to Exercise 22

As contained in **wrapper-ex22-done.html**:

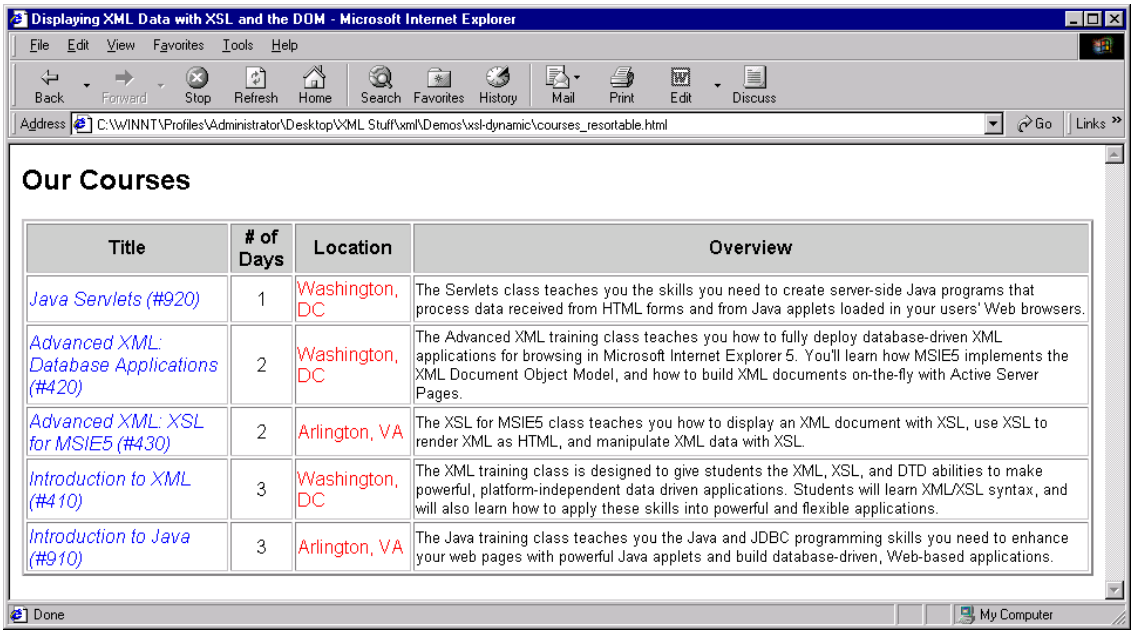
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
   <title>Displaying XML Data</title>

<xml id="allactors" src="actorlist-done.xml"></xml>
<xml id="actorxsl" src="moviexsl-done.xsl"></xml>

</head>

<body onLoad="dataTarget.innerHTML=allactors.transformNode(actorxsl)">

<div id="dataTarget"></div>

</body>
</html>
```

# Dynamic XSL Changes

In the previous section, you used JavaScript to transform an XML datasheet with an XSL stylesheet. It is a relatively small step from what we saw to allow dynamic changes in the XSL stylesheet based on user actions. Take a look at the following demo (**xml > demos > xsl-dynamic > ie > courses_resortable-ie.html**):




At first examination, there appears to be nothing new. The courses are sorted alphabetically by title. However, try clicking on the number of days:



What you've seen is a 4-step process:

- The measurement of a user event, and the passing of details about that event to a JavaScript function.
- The JavaScript function then modifies the XSL stylesheet slightly.
- The JavaScript function re-evaluates the XML with respect to the (now modified) XSL document.
- The JavaScript function then places the new HTML into the page, replacing the old HTML.

## Using XSL Updating to Re-Sort XML Data

First, take a look at the HTML wrapper for the application. It will organize and refer to the different application components (**xml > demos > xsl_dynamic > ie > courses_resortable-ie.html**):

```
<html>
<head>
   <title>Displaying XML Data with XSL and the DOM</title>
<xml id="courselist" src="../courses.xml"></xml>
<xml id="coursedisplay" src="../courses_resortable.xsl"></xml>
<script language="JavaScript" src="js-sortfuncs.js"></script>
</head>

<body onLoad="dataTarget.innerHTML=courselist.transformNode(coursedisplay)">

<div id="dataTarget"></div>

</body>
</html>
```

The one additional component is the link to the JavaScript library **js-sortfuncs-ie.js**. We will look at that in a moment, but first, take a look at the code for the XSL stylesheet (**xml > demos > xsl_dynamic > courses_resortable.xsl**), particularly the sections in **bold**:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match="/">
<html>
<head>
   <title>Simple XSL Demo</title>
</head>
<body>
   <div style="font-size:12pt;font-family:Arial">
   <h2>Our Courses</h2>
```

```
   <table border="2">
   <tr bgcolor="#cccccc">
      <th onclick="reSort('title')">Title</th>
      <th onclick="reSort('course-length')"># of Days</th>
      <th onclick="reSort('location/city')">Location</th>
      <th onclick="reSort('description')">Overview</th>
   </tr>
   <xsl:apply-templates select="courses/course">
      <xsl:sort select="title" />
   </xsl:apply-templates>
   </table>
   </div>
</body>
</html>
</xsl:template>

<xsl:template match="course">
   <tr>
   <td style="font-style:italic;color:blue">
      <xsl:value-of select="title" />
      (#<xsl:value-of select="number" />)
   </td>
   <td align="center">
      <xsl:value-of select="course-length" />
   </td>
   <td style="color:red">
      <xsl:value-of select="location/city" />,
      <xsl:value-of select="location/state" />
   </td>
   <td style="font-size:10pt">
      <xsl:value-of select="description" />
   </td>
   </tr>
</xsl:template>

</xsl:stylesheet>
```

With the above code, we're taking advantage of Internet Explorer's built-in ability to reformat and redisplay content dynamically. We're linking the **onclick** event handler to the table header tags. In each case, we call a function named **reSort()**, and pass it the name of a particular node. Also note that the default initial sort is by **title**.

The code for **xml > demos > xsl_dynamic > ie > js-sortfuncs-ie.js**:

```
function reSort(sortValue)
{
   sortField = coursedisplay.selectSingleNode("//xsl:sort/@select");
   sortField.value=sortValue;
   dataTarget.innerHTML=courselist.transformNode(coursedisplay);
}
```

**js-sortfuncs-ie.js** contains one function named **reSort()**. The function expects a field name, which it will call **sortValue**. The next step is to locate the **select** attribute of the **xsl:sort** tag in the XSL stylesheet:

```
sortField = coursedisplay.selectSingleNode("//xsl:sort/@select");
```

This is accomplished with the Mirosoft-specific **selectSingleNode()** method of the `coursedisplay` XML Document object. **selectSingleNode()** takes a pattern to match, and returns the corresponding node object. In this case, we are looking for the **select** attribute of the **xsl:sort** node.

We then set the **nodeValue** of the **select** attribute to the field name we have passed from the event handler:

```
sortField.nodeValue=sortValue;
```

Finally, we re-build the HTML of the page, based on the newly-modified XSL stylesheet:

```
dataTarget.innerHTML=courselist.transformNode(coursedisplay);
```

## The XMLDocument Object

Above, we're using the **selectSingleNode()** method, which will return the first matching node object that it finds, beginning from a particular starting point (above, the entire document). The 4 major **XMLDocument** methods are:

| Method | Description |
|---|---|
| getElementsByTagName (*tagname*) | Returns an array of elements matching the specified *tagname* |
| selectNodes(*pattern*) | Applies a pattern to a particular node object, and returns a **node list object** containing all nodes that match the pattern. |
| selectSingleNode(*pattern*) | Applies a pattern to a particular node object, and returns the first matching node object. |
| transformNode(*stylesheet*) | Interprets a node object (and any sub-elements) based on a particular stylesheet document. Returns the output of that stylesheet, typically HTML code. |

**An important note:** of the 4 above methods, only the first is in the W3C XML DOM specifications. The last 3 are all MSXML extensions, and have been presented to the W3C for adoption. However, as they are so useful for dynamic processing of XML/XSL data, we use them in this section of the course.

## Dynamically Sorting in Descending Order

As you may remember, in order to sort in descending order, the xsl:sort tag receives a second attribute (in addition to select, which specifies the field to sort by). This attribute is named **order**. The syntax is as follows:

```
<xsl:sort select="fieldname" order="descending" />
```

So, in order dynamically reverse the order you're sorting by, you'll need to select a second node, and adjust its value as well. A sample JavaScript function might look like:

```
function reSort(sortValue, sortOrder)
{
    var sortField, orderField;
    sortField = coursedisplay.selectSingleNode("//xsl:sort/@select");
    orderField = coursedisplay.selectSingleNode("//xsl:sort/@order");
    sortField.value = sortValue;
    orderField.value = sortOrder;
    dataTarget.innerHTML=courselist.transformNode(coursedisplay);
}
```

Note that you now need to pass in a second parameter from the event handler: either **ascending** or **descending**, depending on the field.

Another option would be to have the order toggle back and forth between ascending and descending. This would allow the user to click on a column and have it sort in ascending order, then click on it again and have it resort in descending order. A sample script might look like this (this can be found in your folder as **xml > demos > xsl-dynamic > ie > js-sortfuncs-ie-toggle.js**:

```
function reSort(sortValue)
{
    sortField = coursedisplay.selectSingleNode("//xsl:sort/@select");
    sortField.nodeValue = sortValue;

    orderField = coursedisplay.selectSingleNode("//xsl:sort/@order");
    if (orderField.nodeValue == "descending"){
        orderField.nodeValue = "ascending"
    } else {
        orderField.nodeValue = "descending";
    }

    dataTarget.innerHTML=courselist.transformNode(coursedisplay);
}
```

## Using the W3C DOM and Netscape/Mozilla

Again, the **selectSingleNode()** method will only work in a Microsoft browser. Let's examine how to use the World Wide Web Consortium standards to do the same thing in Netscape/Mozilla (**xml > demos > xsl-dynamic > dom > js-sortfuncs-dom.js**):

```
function reSort(sortValue){
    sortList = styleDoc.getElementsByTagName("sort");
    sortList[0].setAttribute("select",sortValue);
```

```
        xsltProc.transformDocument(sourceDoc, styleDoc, resultDoc, null);
        myResult = new XMLSerializer().serializeToString(resultDoc);
        document.getElementById("dataTarget").innerHTML = myResult;
    }
```

First we get a list of tags named "sort". The matching tags are automatically stored as an array. Mozilla/Netscape ignores the xsl prefix when getting elements by tag name. Since we know that we are using the first sort tag, we refer to it by its index number – 0 (all JavaScript arrays start from zero). We then use the DOM setAttribute() method. This sets the "select" attribute to the value of sortValue. With our XSL sheet now adjusted, we retransform the document, turn it into a string, and write it back to our container .

## Putting it all together

So how would we write a function that could work in both browsers? Funny you should ask, because we just happen to have one possible answer at **xml > demos > xsl-dynamic > cross-browser > courses-resortable-cross.html**. The wrapper is the same as what we saw before. But the function is slightly different. It can be found in the file: **js-sortfuncs-cross.js**.

```
function reSort(sortValue){
       dom ? sort="sort":sort="xsl:sort";
       sortList = styleDoc.getElementsByTagName(sort);
       sortList[0].setAttribute("select",sortValue);
       sortOrder = sortList[0].getAttribute("order");
       if (sortOrder == "ascending") {
          sortList[0].setAttribute("order", "descending");
       } else {
          sortList[0].setAttribute("order", "ascending");
       }

       transform();
}
```

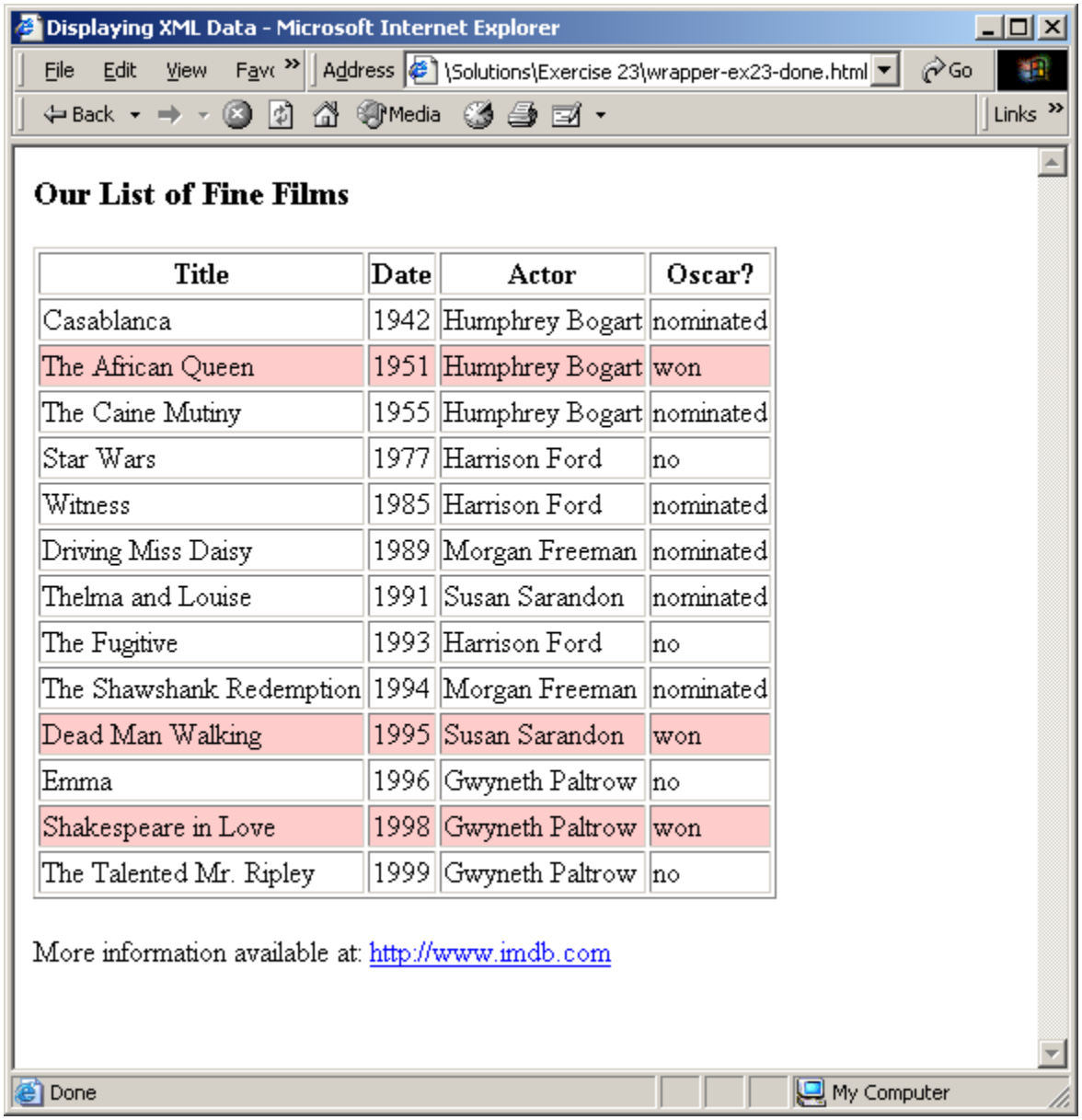
First, we check to see if we're using a dom-compliant browser like Mozilla (the "dom" variable came from our wrapper page's initial browser check). Depending on whether this is true, we set the value of a variable called "sort". IE's implementation of getElementsByTagName() looks for the name of the node along with any prefix, whereas Mozilla's looks for it without the prefix. The only other piece of this script that is new is:

```
sortOrder = sortList[0].getAttribute("order");
```

This assigns to the variable sortOrder the value of the order attribute associated with the first sort tag on the page. The transform() function is in the head of our wrapper file. It uses the browser check to perform the XSL transformation as appropriate for the browser being used to view this page. And that's it!

## Exercise 23: Allowing Users to Re-Sort their Display

In this exercise, you will be upgrading the XSL stylesheet for your application, and building a JavaScript sorting function. When you have completed the exercise, your display should look like it did before, except that users can now click on a table heading to resort their display. The below display comes after a user has clicked on the **Title** heading:




To complete this exercise:

1. Re-open **wrapper.html** in Notepad.

2. Add a <script> tag that references the file **jsLibrary.js.**

3. Save **wrapper.html**.

4. Re-open **moviexsl.xsl** in Notepad.

5. Modify the stylesheet so that the four column headings each have an **onclick** event handler that calls the JavaScript function **reSort()**, and passes it the appropriate XSL element name for sorting.

6. Save **moviexsl.xsl**.

7. Open a new document in Notepad.

8. This document will be your JavaScript library. You will be building one function, the **reSort()** function. It should:

    - Select the node containing the **select** attribute of the **xsl:sort** node.
    - Replace the existing value with the value passed from the event handler
    - Re-parse the XML datasheet with respect to the new XSL information.
    - Place the resulting content in the DIV in the HTML page.

9. Save your file as **jsLibrary.js**.

10. Open **wrapper.html** in Internet Explorer.

11. If you get an error message, go back into Notepad, edit your file, and reload.

## If you are done early…

- See if you can get this to work in Netscape/Mozilla. If that works, try making it cross-browser compatible.
- Rebuild your sort so that the **Oscar** column sorts in descending, rather than ascending, order. We have a solution to this challenge as **wrapper-ex23-challenge1-done.html**.
- Allow your user to sort by reverse order by clicking again on the column heading after having chosen it once already. We have a solution to this challenge as **wrapper-ex23-challenge2-done.html**.
- Allow a secondary sort order, no matter what the user-chosen sort order is. So, for example, this might be **lastname**, and when a user sorts by preferred, you will execute the sort so that it sorts by first preferred, and then **lastname**.

## A Possible Solution to Exercise 23

As contained in **actors-ex23-done.html**:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
   <title>Displaying XML Data</title>

<xml id="allactors" src="actorlist-done.xml"></xml>
<xml id="actorxsl" src="moviexsl-ex23-done.xsl"></xml>

<script language="javascript" src="jsLibrary-ex23-done.js"></script>

</head>

<body onLoad="dataTarget.innerHTML=allactors.transformNode(actorxsl)">

<div id="dataTarget"></div>

</body>
</html>
```

And as contained in **jsLibrary-ex23-done.js**:

```
function reSort(sortValue) {
   var sortField = actorxsl.selectSingleNode("//xsl:sort/@select");
   sortField.value=sortValue;
   dataTarget.innerHTML=allactors.transformNode(actorxsl);
}
```

And as contained in **moviexsl-ex23-done.xsl**:

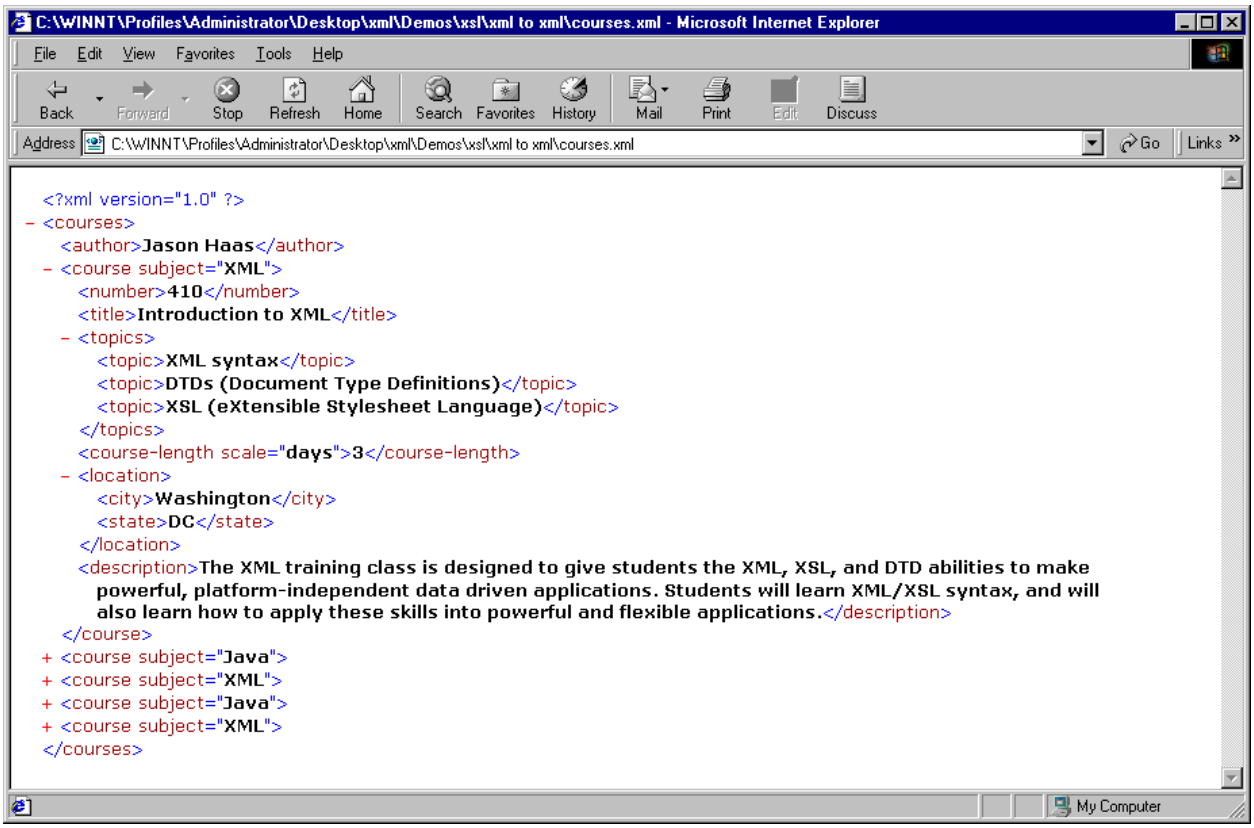
```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
      <html>
         <head>
            <title>Our List of Fine Films</title>
         </head>
         <body>
         <h3>Our List of Fine Films</h3>
         <table border="1">
         <tr>
            <th onclick="reSort('title')">Title</th>
            <th onclick="reSort('date')">Date</th>
            <th onclick="reSort('../../name/lastname')">Actor</th>
            <th onclick="reSort('@oscar')">Oscar?</th>
         </tr>
         <xsl:apply-templates select="//film">
```

```
            <xsl:sort select="date" order="ascending" />
         </xsl:apply-templates>
         </table>
         <p>More information available at:
            <a>
               <xsl:attribute name="href">
                    <xsl:value-of select="actors/url" />
               </xsl:attribute>
               <xsl:value-of select="actors/url" />
            </a>
         </p>
         </body>
      </html>
   </xsl:template>
   <xsl:template match="film">
      <tr>
         <xsl:attribute name="style">
            <xsl:if test="@oscar='won'">background-color:#ffcccc</xsl:if>
         </xsl:attribute>
         <td><xsl:value-of select="title" /></td>
         <td><xsl:value-of select="date" /></td>
         <td><xsl:value-of select="../../name" /></td>
         <td><xsl:value-of select="@oscar" /></td>
      </tr>
   </xsl:template>
</xsl:stylesheet>
```

# Using XSL to Produce New XML

A common use of XSL is to transform XML into different XML. For example, you may receive information that is in one XML format, and may already have applications that expect the information in another format. XSL can allow systems to be structured however makes the most sense for their internal operations, and can provide the bridge between different XML document types when necessary.

To get a sense of how this might work, we will transform the datasheet we've been using for demonstrations throughout the course. In case you've forgotten, the structure of this data is as follows (**demos > xsl > xml to xml > courses.xml**):



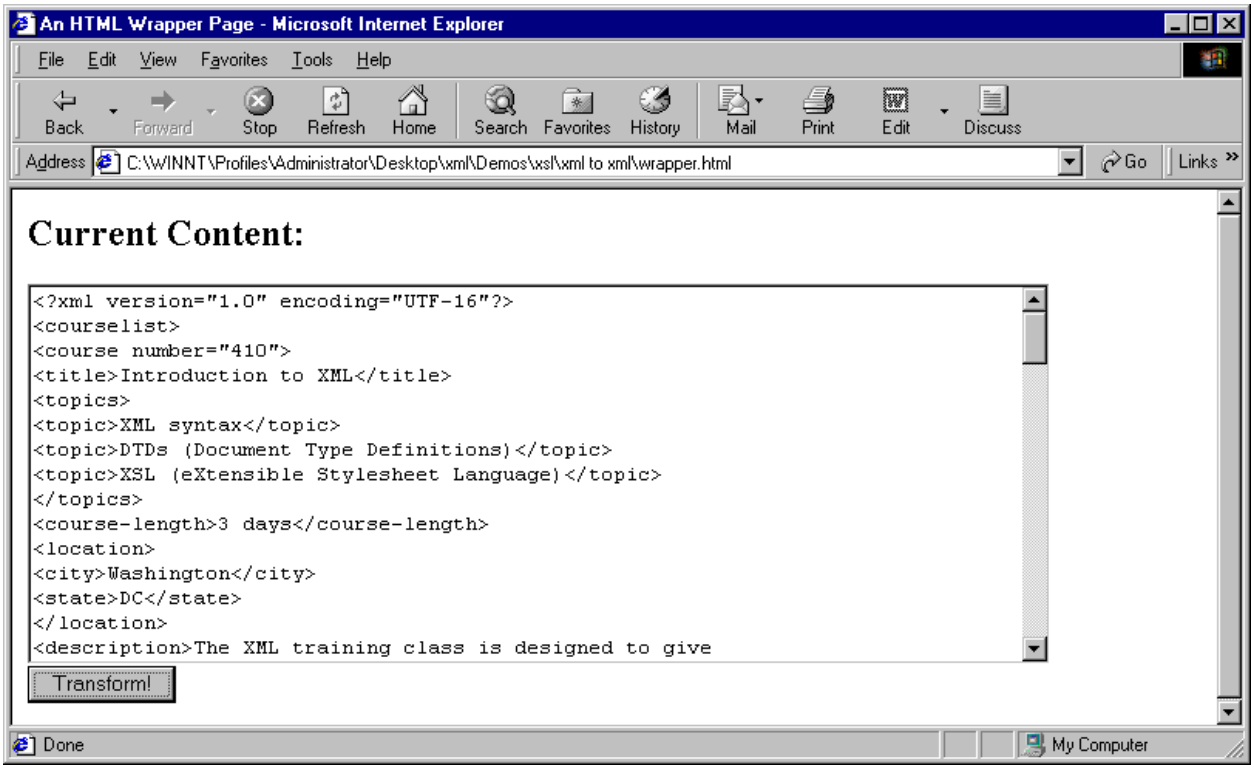

As you can see, we have a **courses** element, which contains an **author** element and a series of **course** elements. Each course has additional information, such as the **number**, **title**, **topics**, **course-length**, **location**, and **description**.

Our application is going to rebuild this XML document to suit a different set of demands. These demands are:

- There should be no author element, only a series of course elements.
- The course number should be an attribute of course, instead of a sub-element.

- We will discard the course subject.
- Course-length should be a concatenated value in the form "4 days" rather than a numeric value with a units attribute.
- Otherwise, the structure will remain the same.

To see this application in action, take a look at the demo **demos > xml to xml > wrapper.html**. After loading the XML data into the textbox and clicking "Transform" we see:




You will note that we have made the changes we specified: the **author** element and the **subject** attribute have been removed, the course **number** has been turned into an attribute, and the **course-length** is now a concatenated string.

The XSL stylesheet that produces this result is **demos > xsl > xml to xml > transform.xsl**:

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

<xsl:template match="/">
   <courselist>
      <xsl:apply-templates select="//course" />
   </courselist>
</xsl:template>
```

```
<xsl:template match="course">
   <course>
      <xsl:attribute name="number">
         <xsl:value-of select="number" />
      </xsl:attribute>
      <title><xsl:value-of select="title" /></title>
      <topics>
         <xsl:apply-templates select="topics/topic" />
      </topics>
      <course-length><xsl:value-of select="concat(course-length, ' ',
course-length/@scale)" /></course-length>
      <location>
         <city><xsl:value-of select="location/city" /></city>
         <state><xsl:value-of select="location/state" /></state>
      </location>
      <description><xsl:value-of select="description" /></description>
   </course>
</xsl:template>

<xsl:template match="topic">
   <topic><xsl:value-of select="." /></topic>
</xsl:template>

</xsl:stylesheet>
```

There is really not too much new to this stylesheet. You have a standard construction of templates, except that instead of HTML tags, you are writing XML tags.

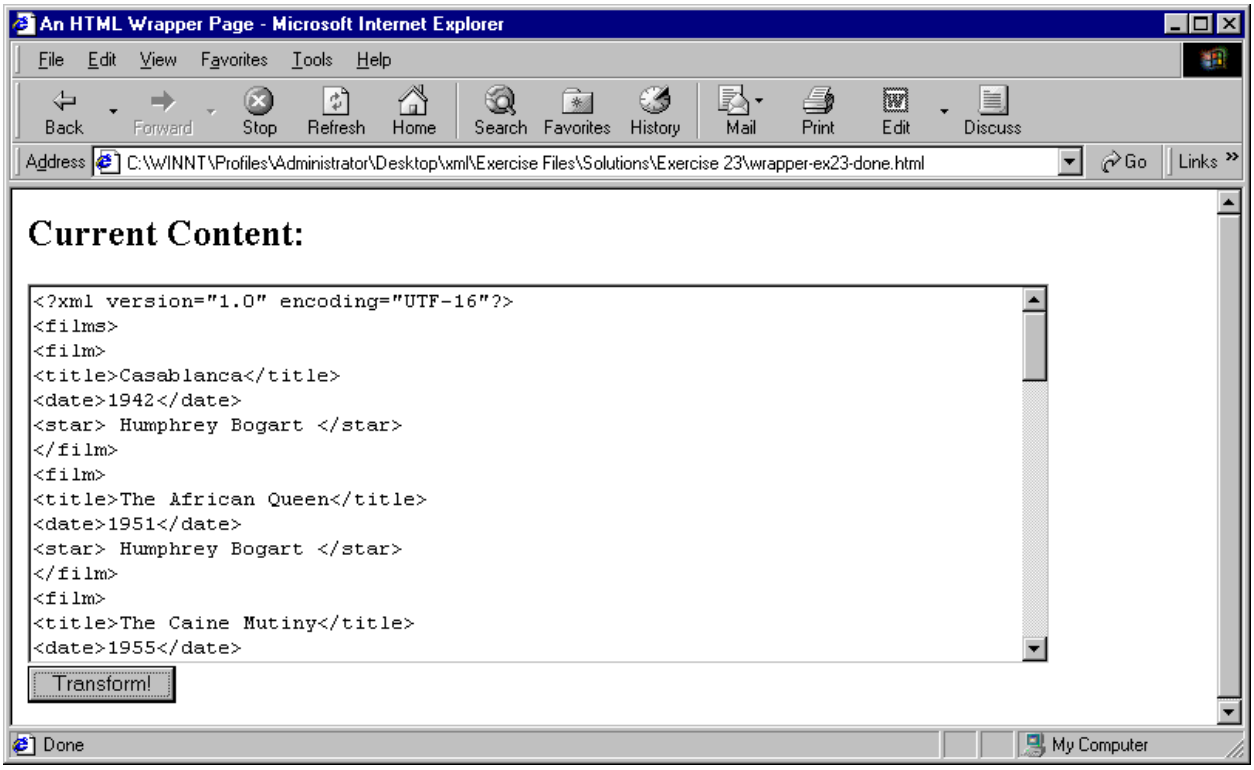
## xsl:output method="xml"

There are a few attributes of the **xsl:output** tag that you will want to know about if you are producing XML. We use the **indent** attribute, which is somewhat misnamed (at least in the MSXML implementation; the exact representation of indentation varies from parser to parser). What it does is puts each element in the output stream on a line by itself. Without that, you will get one long string of XML code, which is difficult to read. Note that adding the whitespace to produce nicely-formatted XML will add file size to a resulting output file, and the extra text nodes may cause problems in some applications. Other useful attributes are listed below:

| Attribute | Explanation |
|---|---|
| version | For XML, allows you to specify the version (default is version 1.0, and at the moment, you shouldn't use anything else) |
| encoding | Specifies the encoding of the output (e.g. *utf-8*). |
| doctype-system | Specifies a path to a system DTD that is to be included in the output file. |
| omit-xml-declaration | If you are producing a document fragment, setting this attribute to "yes" will prevent the <?xml?> tag and its attributes from being created. |

# Exercise 24: Producing a datasheet organized by film rather than by actor

In this exercise, you will be completing an XSL stylesheet that will transform the data stored in **actorlist.xml** into a different format: one organized by film rather than by actor. You will be completing a template; other parts of the application are done for you. When you are done, you will visit a wrapper page. Clicking "Transform" will produce:




For this exercise, complete the following:

1. Open **transform-ex24-temp.xsl** in from your **Exercise 24** folder in Notepad. You should see the following code:

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

<!--

Add the required templates to produce new XML with the top-level
element "films" and child elements "film".

Each "film" element should have the children "title", "date", and "star"
```

```
-->

</xsl:stylesheet>
```

2. Complete your code so that it produces the appropriate XML output:

    - The root template should produce the top-level tag **<films>**, and apply-templates on all **film** elements.
    - Each **film** element should contain 3 children: **title**, **date**, and **star**.
    - Add the appropriate content into each element.

3. Save **transform-ex24-temp.xsl**, and open **wrapper-ex24-temp.html** in Internet Explorer.

4. If you encounter errors, edit, save, and re-test.

## If you are done early...

- Add the **oscar** attribute to the film tag.
- You may notice that the names in the <star> node have extra spaces before and after them. This comes from the spaces in the actual xml file. Try to get rid of these spaces. There are a number of ways you could do this: use the xsl **concat** function; use separate **xsl:value-of** elements for firstname and lastname; use the xsl **normalize-space** function.
- Change the **star** element so that it contains separate **firstname** and **lastname** elements.

## A Possible Solution to Exercise 24

As contained in **transform-ex24-done.xsl**:

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

<xsl:template match="/">
   <films>
      <xsl:apply-templates select="//film" />
   </films>
</xsl:template>

<xsl:template match="film">
   <film>
      <title><xsl:value-of select="title" /></title>
      <date><xsl:value-of select="date" /></date>
      <star><xsl:value-of select="../../name" /></star>
   </film>
</xsl:template>

</xsl:stylesheet>
```

# Conclusions: Why XML?

To this point, you have seen many different pieces of different XML applications. It is almost certainly worth taking a step back to think about the sorts of applications that usually can benefit from XML. These are:

## When you need to output the same data in multiple formats

For example, consider the case of a press release. Normally, press releases have to be output in (at least) formatted text for a fax or mailing, HTML for posting on a Web site, and plain text for sending out as an email. There may be other needs (WML for the wireless web) as well. If you write the release document in XML, you can apply different XSL stylesheets to that document and have the end results automatically generated. What's more, if you standardize your press releases on a specific set of tags (usually defined in a DTD or Schema) your stylesheets will be reusable without modification.

## When you need to exchange data between otherwise incompatible systems, applications, or organizations

For example, consider the case of a manufacturer. They routinely need to subcontract out to produce various parts for inclusion into their products. In the pre-XML world, they had to send out a document specifying what they needed in a bid to produce these products, and might easily receive the information back in various formats: paper documents, faxes, emails, attachments, and spreadsheets, at least. A person would have to then enter this information into some application so that the different bids could be compared. The added layer of data entry adds time, expense, and error to the process.

With XML the same manufacturer can send out their requirements in the form of a DTD or Schema, and expect an XML document in return. The first step of the process, when a bid is submitted, is to validate the data against its DTD/Schema. If it validates, the manufacturer can run a pre-made program to pull the information out of the XML file and enter it into a database. If the bid does not match, the submitter can be notified.

## When you need to store structured data, but a database is not appropriate or necessary

For many applications, particularly applications where data is being generated in multiple locations with questionable connectivity, it can be difficult or undesirable to have team members connecting directly to the main database for a project. XML allows the opportunity to store structured information in a format that can be easily imported into a database, or evaluated without one.

## When you want multiple different views of information, presented dynamically

For client-side applications (currently mostly Intranet applications, although with the increasing penetration of XML-capable browsers, Internet applications are not far off) it is possible to get extremely flexible and responsive displays of information, without having to make additional demands on the server or database. For example, in a standard database driven application, if you wish to see a display of information sorted differently (say, by last name rather than zip code) you have to follow the following process:

1. Browser makes a new HTTP request to the server
2. Server processes the request with a middleware application
3. Middleware application builds a new SQL query for the database
4. Middleware application creates a connection to the database, and passes it the SQL query
5. Database receives the query and retrieves the recordset
6. Database returns the recordset to the middleware application
7. Middleware application generates the HTML from the recordset, and sends it via HTTP to the browser
8. Browser displays the information

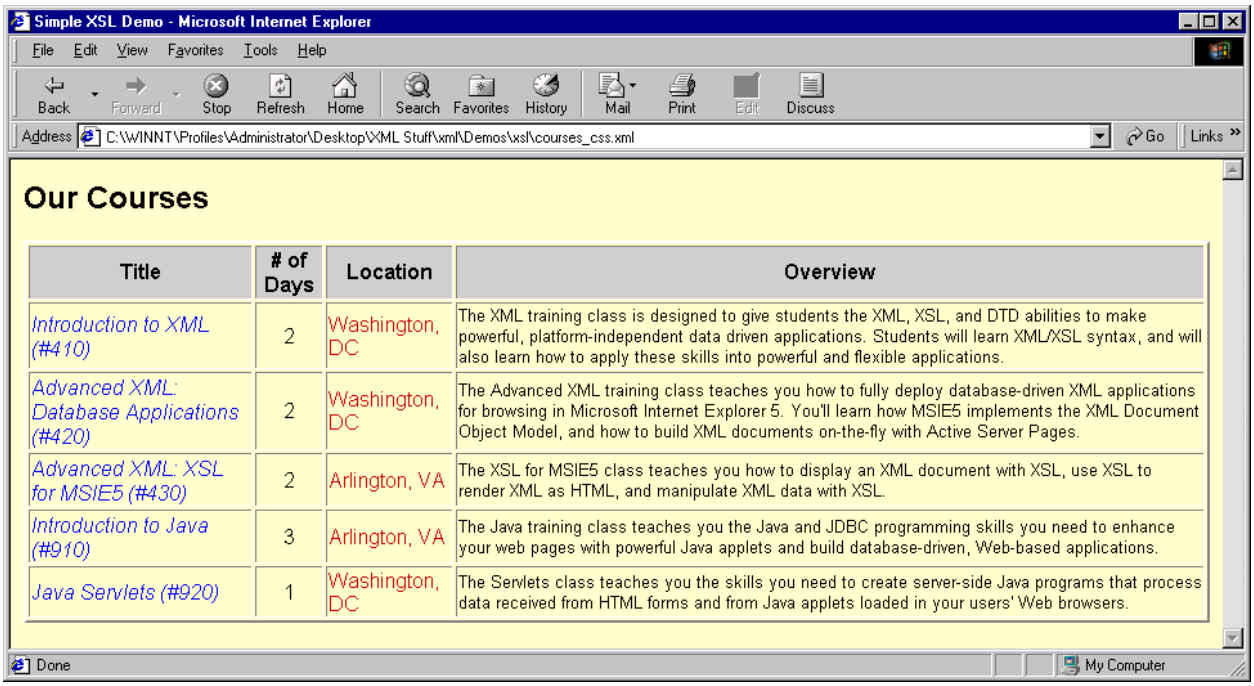
Whew! That's a lot of steps for one application, and requires your server and your database to do extra work for reformatting. However, in a client-side XML application, the process is simplified to:

1. Browser uses JavaScript to modify XSL stylesheet
2. JavaScript reapplies (now modified) XSL stylesheet to XML data
3. JavaScript places the generated HTML content into the page

As you can see, not only is the process a much shorter one, but it also is entirely processed by the browser's processor. By taking advantage of efficiency savings like these, you can make applications that are extraordinarily responsive.

# Appendix A: Incorporating CSS with XSL

Since your XSL generates HTML code, you can use it to build an HTML <LINK> tag in the head of your generated HTML document. Then, you'll be able to use any CSS classes that you have declared in your XSL-generated HTML code. To get a sense of how it works, take a look at the file **demos > xsl > courses_css.xml**:




You will notice, in addition to the previous formatting, there is a background color for the page. In addition, the styles are now applied through CSS classes, rather than as in-line CSS style rules. The code for the XSL is below (**demos > xsl > courses_css.xsl**):

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match="/">
   <html>
   <head>
      <title>Simple XSL Demo</title>
      <link href="courses.css" rel="stylesheet" />
   </head>
   <body>
      <div style="font-size:12pt;font-family:Arial">
      <h2>Our Courses</h2>
      <table border="2">
         <tr bgcolor="#cccccc">
            <th>Title</th>
            <th># of Days</th>
            <th>Location</th>
```

```
            <th>Overview</th>
          </tr>
          <xsl:apply-templates select="//course">
            <xsl:sort select="number" data-type="number" />
          </xsl:apply-templates>
        </table>
        </div>
    </body>
    </html>
</xsl:template>

<xsl:template match="course">
    <tr>
    <td class="title">
        <xsl:value-of select="title" />
        (#<xsl:value-of select="number" />)
    </td>
    <td align="center">
        <xsl:value-of select="course-length" />
    </td>
    <td class="location">
        <xsl:value-of select="location/city" />,
        <xsl:value-of select="location/state" />
    </td>
    <td class="description">
        <xsl:value-of select="description" />
    </td>
    </tr>
</xsl:template>

</xsl:stylesheet>
```

Note that the <LINK> tag is a naturally empty HTML element, and you will need to identify it as such with the trailing forward slash. The CSS declarations are now in the form of CSS **classes.**

## Classes in CSS

The contents of the CSS file are:

```
body            {background-color: #ffffcc}

.title          {color: #0000ff;
                 font-style: italic}

.location       {color: #cc0000}

.description    {font-size: 10pt}
```

The first declaration specifies a background color for the body of the page, while the next three declarations create **classes**, or names that represent certain style characteristics. Although we've chosen intuitive names for the classes, they do not need to match XML element names.

# Appendix B: The XML DOM

The XML specifications allow for direct access of XML objects with JavaScript (or other scripting languages). XML applications can make use of the JavaScript understanding of the XML DOM to create, modify, or delete nodes, or even to parse and display an entire XML object hierarchy, even when the individual element names or the internal structure are not known. A final important use of the DOM is for text searching.

For client-side applications, all references begin with the **XMLDocument** property of an XML data island. From there, you can use the following built-in methods to identify the elements you wish to select:

| **Method** | **Description** |
|---|---|
| createAttribute(*name*) | Creates a new attribute with the specified *name* |
| createElement(*name*) | Creates a new element with the specified *name* |
| createTextNode(*value*) | Creates a new text() node, with the specified *value* |
| getElementsByTagName (*tagname*) | Returns an array of elements matching the specified *tagname* |
| transformNode(*stylesheet*) | Interprets a node object (and any sub-elements) based on a particular *stylesheet* document. Returns the output of that stylesheet |
| selectSingleNode(*pattern*) | Applies a *pattern* to a particular node object, and returns the first matching node object |
| selectNodes(*pattern*) | Applies a *pattern* to a particular node object, and returns a node list object containing all nodes that match the pattern |

Once you have a **node** element selected, that node has the following properties:

| **Property** | **Description** |
|---|---|
| nodeName | Returns the name of the current element |
| nodeType | Returns the node type as a number |
| nodeValue | Returns the value of the element as text |
| firstChild | Returns a reference to the first child element of the current element |
| lastChild | Returns a reference to the last child element of the current element |
| parentNode | Returns a reference to the parent of the current element |
| childNodes | Returns a reference to a node list containing all child elements of the current element. The number of child elements can be obtained with **childNodes.length** |
| previousSibling | Returns a reference to the previous sibling element of the current element |

| | |
|---|---|
| nextSibling | Returns a reference to the next sibling element of the current element |
| ownerDocument | Returns a reference to the top-level node containing the current element |

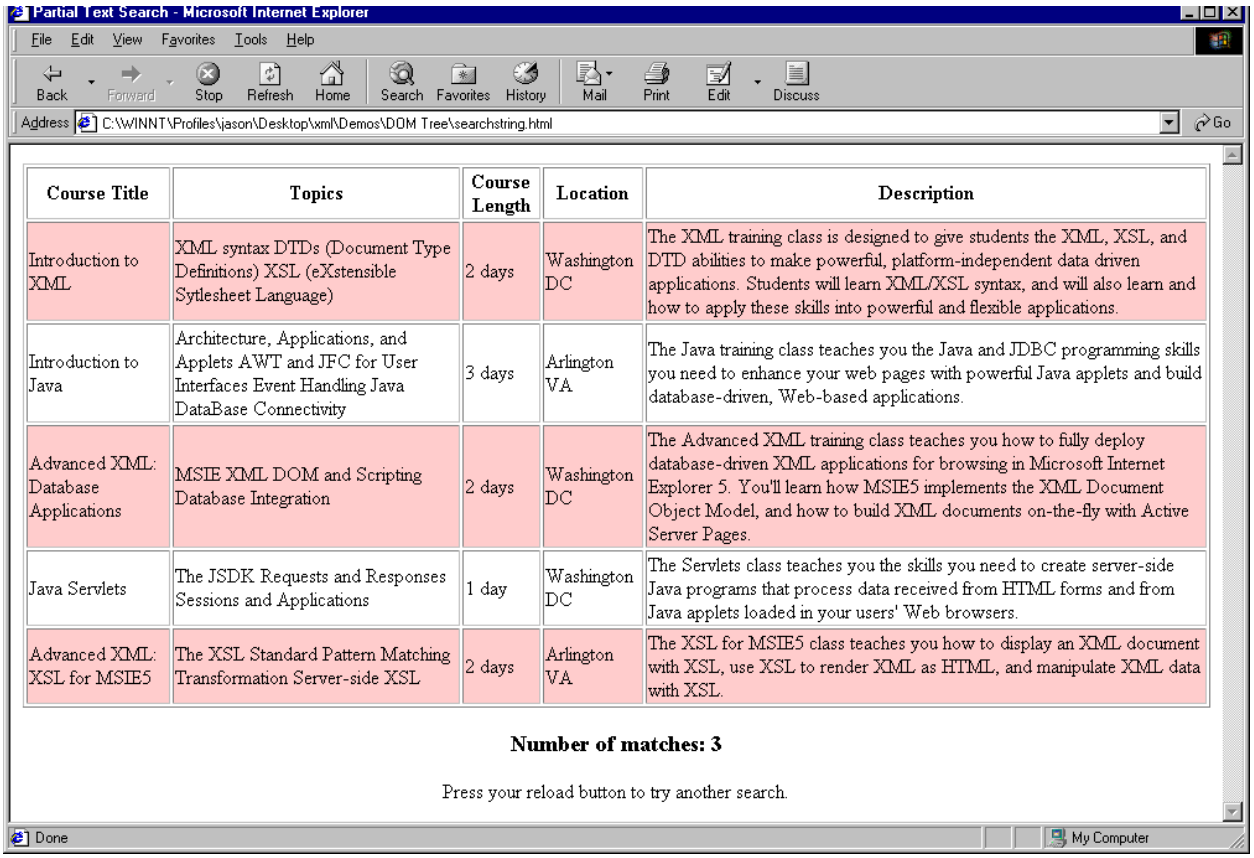
In addition, there are a few useful properties that are Internet Explorer-specific:

| Property | Description |
|---|---|
| nodeTypeString | Returns a string value of the node type |
| text | Returns the text value of the current element, as well as any child elements. |
| xml | Returns the XML of the current element, including any child elements. |

## Partial Searches with JavaScript

One of the most important uses of JavaScript in XML is to parse through text to find certain matching words or phrases. In our example (**xml > demos > DOM Tree > searchstring.html**), we will be using JavaScript's pattern-matching capabilities to search for text matches in XML data. The initial screen queries the user for a value to search for from the courses list:


Entering a value (in our case, "XML"), and clicking the button produces the following result:




As you can see, the table display shows several pieces of information for each course: the Title, the topics, the course length, the location, and the description. Also, elements that contained the search text are highlighted in pink.

The code for this page is below. Note the sections in **bold**:

```
<html>
<head>
   <title>Partial Text Search</title>
<xml id="courselist" src="courses.xml"></xml>

<script language="javascript">
var theXML, myNodeList, textToSearch;
var strNode = "";
var matches=0;

function showMatches() {
   theXML = courselist.XMLDocument;
   textToSearch = document.forms[0].searchtext.value;

   myNodeList = theXML.getElementsByTagName('course');
   strNode += "<table border='1'><tr><th>Course Title</th><th>Topics</th>";
   strNode += "<th>Course Length</th><th>Location</th><th>Description</th></tr>";
   for (var count=0; count<myNodeList.length; ++count) {
      theItem = myNodeList[count];
```

```
      if (theItem.text.match(textToSearch)) {
         //Build a row with a pink background for matching rows
         strNode += "<tr bgcolor=#ffcccc>";
         ++matches;
      } else {
         //Build a row with a white background for non-matching rows
         strNode += "<tr>";
      }
      strNode += "<td>" + theItem.childNodes[0].text + "</td>"
      strNode += "<td>" + theItem.childNodes[1].text + "</td>";
      strNode += "<td>" + theItem.childNodes[2].text + "</td>";
      strNode += "<td>" + theItem.childNodes[3].text + "</td>";
      strNode += "<td>" + theItem.childNodes[4].text + "</td></tr>";
   }
   strNode += "</table>";
   strNode += "<h3>Number of matches: " + matches + "</h3>";
   strNode += "Press your reload button to try another search.";
   dataTarget.innerHTML = strNode;
}

</script>
</head>
<body>
<div align="center" id="dataTarget">
<form>
   <h2>Course Information Search</h2>
   Please enter a value to search for in our course list<br>
   (partial queries OK): <input name="searchtext"><br><br>
   <input type="Button"
       value="Search the XML Document"
       onClick="showMatches()">
</form>
</div>

</body></html>
```

We will look at three pieces of the above document.

## Matching Node Names with the getElementsByTagName() method

In the above example, we are retrieving a list of all nodes matching a particular name using the **getElementsByTagName()** method:

```
   theXML = courselist.XMLDocument;
   myNodeList = theXML.getElementsByTagName('course');
```

The current document has been stored as **theXML**, and **myNodeList** stores the list of nodes (in JavaScript terms, a 5-element array containing all the **course** nodes).

### Testing for a match with the match() string method

In order to find our match, we need to test the text of our course element against the text value entered into our form, using the JavaScript match() string method:

```
textToSearch = document.forms[0].searchtext.value;
strNode += "<table border='1'><tr><th>Course Title</th><th>Topics</th>";
strNode += "<th>Course Length</th><th>Location</th><th>Description</th></tr>";
for (var count=0; count<myNodeList.length; ++count) {
   theItem = myNodeList[count];
   if (theItem.text.match(textToSearch)) {
      //Build a row with a pink background for matching rows
      strNode += "<tr bgcolor=#ffcccc>";
      ++matches;
   } else {
      //Build a row with a white background for non-matching rows
      strNode += "<tr>";
   }
```

First, we save as **textToSearch** the value from the form. Then, if the **match()** method of the XML text returns true when passed the text in **textToSearch**, we build a table row tag with a background color of pink. Otherwise, the default background color (white) will be used.

### Building each row of HTML output

Next, we need to build the HTML output for our page. It is structured so that each row contains the same 5 elements (title, topics, course length, location, and description):

```
strNode += "<td>" + theItem.childNodes[0].text + "</td>"
strNode += "<td>" + theItem.childNodes[1].text + "</td>";
strNode += "<td>" + theItem.childNodes[2].text + "</td>";
strNode += "<td>" + theItem.childNodes[3].text + "</td>";
strNode += "<td>" + theItem.childNodes[4].text + "</td></tr>";
```

Note that the HTML content is being stored in the variable **strNode**. Finally, we need to place the content into the HTML page:

```
dataTarget.innerHTML = strNode;
```

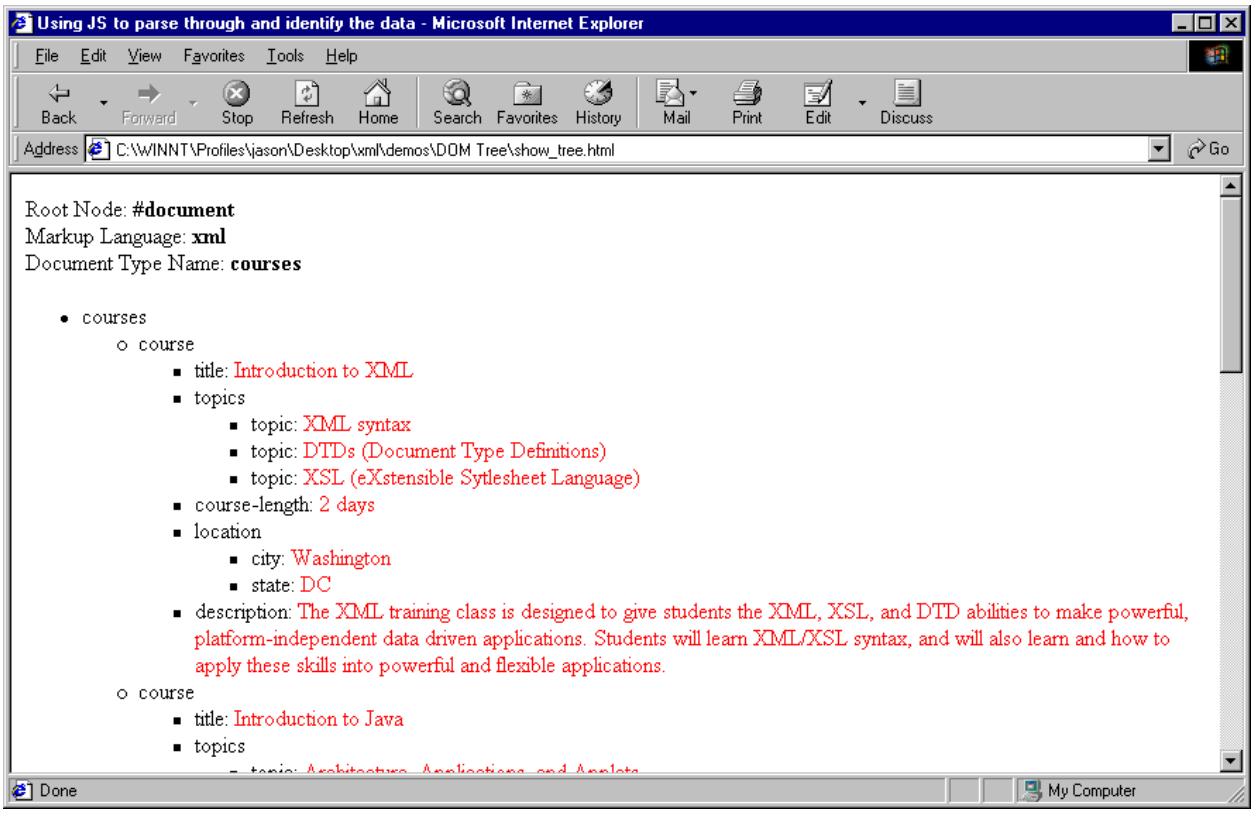
We are using the one DIV in the body of the page (which formerly contained the form and its contents). By substituting in the **innerHTML** property of the DIV, we can display the results. To retrieve the initial form, all we need to do is reload the page.

## Accessing the XML DOM Tree with JavaScript

Although you may already be comfortable with using the XML Document object model (DOM) it is useful to be able to move comfortably through it

## Building a Tree Display of your XML Content

Take a look at **xml > demos > DOM Tree > show_tree.html**:



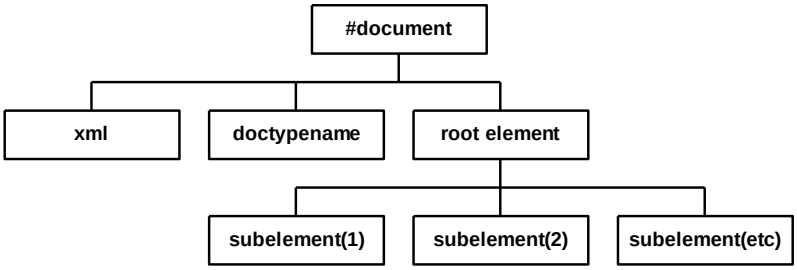

Using JavaScript, we've dynamically built a tool that will parse through any XML datasheet and display the hierarchy (in black) and any text values (in red). The code used to create this is below:

```
<html>
<head>
   <title>Using JS to parse through and identify the data</title>
<xml id="courselist" src="courses.xml"></xml>
<script language="javascript">

   var strNode = "";

   function startTreeWalk() {
      var topNode = document.all.courselist.XMLDocument;
      strNode += "Root Node: <b>" + topNode.nodeName + "</b><br>";
      languageNode = topNode.firstChild;
      strNode += "Markup Language: <b>" + languageNode.nodeName + "</b><br>";
      docTypeNode = languageNode.nextSibling;
      strNode += "Document Type Name: <b>" + docTypeNode.nodeName + "</b><br><br>";
      contentNode = docTypeNode.nextSibling;
      showChildNodes(contentNode);
   }
```

```
   function showChildNodes(thisNode) {
      var intNode = 0;
      var numChildren = 0;

      if (thisNode.nodeName.indexOf("#text") == -1) {
         //This is a node with sub-nodes
         strNode += "<ul><li>" + thisNode.nodeName + "";
      } else {
         //This is text content
         strNode += ": <span style='color:red'>"+thisNode.nodeValue+"</span>";
      }
      numChildren = thisNode.childNodes.length;
      if (numChildren > 0) {
         for(intNode = 0; intNode<numChildren; intNode++) {
            showChildNodes(thisNode.childNodes[intNode]);
         }
      strNode += "</ul>";
      }
      dataTarget.innerHTML = strNode;
   }

</script>
</head>
<body onLoad="startTreeWalk()">

<div id="dataTarget"></div>

</body>
</html>
```

The key to the above page is that, for JavaScript, all XML documents have the following hierarchy:

**Hierarchy of an XML document in JavaScript**




So, in our initial **startTreeWalk()** function, we work our way from the document root to the root XML element, taking account of the nodes along the way, and beginning to add to our **strNode** variable, which will contain the HTML content of our page:

```
function startTreeWalk() {
   var topNode = document.all.courselist.XMLDocument;
```

```
    strNode += "Root Node: <b>" + topNode.nodeName + "</b><br>";
    languageNode = topNode.firstChild;
    strNode += "Markup Language: <b>" + languageNode.nodeName + "</b><br>";
    docTypeNode = languageNode.nextSibling;
    strNode += "Document Type Name: <b>" + docTypeNode.nodeName + "</b><br><br>";
    contentNode = docTypeNode.nextSibling;
    showChildNodes(contentNode);
}
```

The last line of the function calls **showChildNodes()**, which will walk recursively through the XML hierarchy of any page:

```
function showChildNodes(thisNode) {
   var intNode = 0;
   var numChildren = 0;

   if (thisNode.nodeName.indexOf("#text") == -1) {
      //This is a node with sub-nodes
      strNode += "<ul><li>" + thisNode.nodeName + "";
   } else {
      //This is text content
      strNode += ": <span style='color:red'>"+thisNode.nodeValue+"</span>";
   }
   numChildren = thisNode.childNodes.length;
   if (numChildren > 0) {
      for(intNode = 0; intNode<numChildren; intNode++) {
         showChildNodes(thisNode.childNodes[intNode]);
      }
   strNode += "</ul>";
   }
   dataTarget.innerHTML = strNode;
}
```

There's a lot going on in that little function. The fundamental logic behind it is that any XML content will either be a tag, or it will be text content. Our initial if… condition texts for this. If it is a tag, we will indent with the **<ul>** tag, and print out a list item with the **name** of the node. If it is text content, we will print out the **value** of the node in red.

So, all tags receive bulleted lists, and those that have text values receive their values next to them in red. Finally, a for… loop provides the engine for the recursive search:

```
for(intNode = 0; intNode<numChildren; intNode++) {
   showChildNodes(thisNode.childNodes[intNode]);
}
```

For each child of the current element, we call the function again. That means that the page builds itself from the inside out, finishing each individual branch of the tree before moving on to the next one.

# Appendix C: Glossary

- **ADO:** ActiveX Data Object, a Microsoft capability for sorting through, organizing, and displaying recordset data. Present client-side in Internet-Explorer 5, as well as in server-side Microsoft technologies such as ASP.
- **Attribute:** A name and value pair contained in an element's opening tag.
- **Class:** A CSS specification that incorporates a group of declarations not tied to any particular tag. For example, you could make a class "loud" that is color:red and font-family:impact. That class could then be applied to any bit of HTML text, whether enclosed in HTML headings, paragraphs, or other tags.
- **CSS:** Cascading Style Sheets, a technology that extends HTML formatting capabilities, and allows designers access to HTML defaults. Syntax is in the form of rule:declaration. Introduced in Internet Explorer 3 and Netscape 4, and adopted completely in Internet Explorer 4 and 5. Can be applied to format XML datasheets.
- **Data Binding:** The process of linking an XML Data Island (see below) to an HTML structure through ActiveX controls. IE5-Specific. Can be single-record or tabular, which will iterate through an entire recordset.
- **Data Island:** XML data loaded into an HTML Web page through the HTML <XML> tag. This data can be displayed directly in IE5 through data binding and ADO, or accessed through JavaScript control of the DOM.
- **Datasheet:** An XML document that contains the marked-up data content. Saved with an .xml extension**.**
- **DOM:** The Document Object Model. The internal organization of objects that describes the logical environment of a language. For example, the JavaScript DOM is organized in a hierarchy with the window as the top-level object, and various sub-objects below that.
- **DTD:** Document Type Definition. A set of codes, included at the beginning of XML datasheets, that specifies the valid structure and relationship of elements and attributes. Can be either internal or external.
- **Dynamic HTML:** Not one language, but instead a collection of interactive effects enabled in 4th generation and later browsers by a combination of JavaScript, CSS, and 4th generation HTML. Characterized by a Web page's ability to monitor and respond to a wide array of user actions.
- **Element:** A complete object in XML, as defined by the opening and closing tags, any attributes, and any content, including sub-elements and text. A synonym is **node**.

- **Namespace:** A subset of XML tags, defined by their prefix and a unique URL. Namespaces are designed to allow extensibility and the ability to define tags that retain fixed meaning while allowing maximum flexibility for designers. Commonly used namespaces include xsl (for XSL tags), xsd (for XML Schema Documents) and xsi (for XML Schema Instance).

- **Node:** An XML element, including any child elements and their attributes.

- **Regular Expression**: A pattern-matching language developed for Unix and popularized in Perl and JavaScript that allows concise specification of very precise patterns. Implemented in XML Schemas using the xsd:pattern element.

- **SGML:** Standard Generalized Markup Language. The predecessor of both HTML and XML, which provided for tags-based description of data. XML is designed to be a valid sub-set of SGML.

- **Schema**: A set of declarations that define the structure and contents of an XML document. Similar to DTDs but more powerful, Schemas were developed by the W3C.

- **Tabular Data Binding:** The process of binding XML data to an HTML table through ADO controls. This allows you to match an iterative HTML feature (a table) to your XML datasheet.

- **Tag:** An individual string of text that determines the opening or closing component of an element.

- **Well-formed:** A markup language document is well-formed if it follows the appropriate rules of syntax for that language. With respect to XML, a well-formed document must be properly nested, must contain closing tags for all tags with content, must end all "empty tags" with a forward slash, and must surround attribute values with either single or double quotes.

- **World Wide Web Consortium (W3C):** The independent organization that proposes and ratifies standards for Internet languages such as HTML, CSS, and XML. They can be visited at *http://www.w3.org*.

- **Wrapper:** The general term for an document that serves to organize and refer to other pieces of an XML application, such as XML datasheets, XSL or CSS stylesheets, and JavaScript libraries. For client-side applications, the wrapper is typically an HTML file, but wrappers are equally important server-side, where middleware applications such as ASP or JSP serve the same function.

- **XML:** Extensible Markup Language. A child language of SGML, where writers design their own document structures through the use of user-defined tags.

- **XML Working Group**: The group at the W3C which evaluates, proposes, and publishes changes to the standards of any of the XML family of languages.

- **XPath**: A protocol for specifying nodes within the hierarchy of an XML document. The syntax is similar to that of hyperlinks. Nodes and attributes can be selected by relationships to other nodes or by location, and can have filters applied to them. XPath has been incorporated into XSLT.

- **XSL**: Extensible Stylesheet Language. XSL is really two languages, XSLT (for transformations) and XSLFO (for formatting). Of the two, XSLT is by far the more widely used, and XSL and XSLT are often used interchangeably. Both are subsets of XML.

- **XSLFO**: XSL Formatting Objects. A stylesheet language designed to allow very precise formatting and display for XML data, similar in many ways to CSS. XSLFO is not supported by browsers, and is most commonly used in custom applications to format and display output in non-English languages.

- **XSLT:** XSL Tranformation. A language designed by the W3C to translate XML data into other formats, including HTML, WML, text, and even new XML. One of the most fluid of the XML family of languages.

# Appendix D: Cascading Style Sheets (CSS)

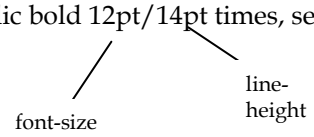
| Declaration | Property | Example | Notes |
|---|---|---|---|
| font-family | specific font name, font family name | {font-family: times}<br>{font-family: serif} | |
| font-size | by size unit*, by percentage, by size name** | {font-size: 12pt}<br>{font-size: 150%}<br>{font-size: small} | |
| font-style | Specific font styling | {font-style: italic}<br>{font-style: normal}<br>{font-style: oblique} | |
| font-weight | normal, bold, bolder, lighter, 100, 200, 300, 400, …, 800, 900 | {font-weight: bold}<br>{font-weight: 700} | |
| text-transform | uppercase, lowercase, capitalize | {text-transform: lowercase}<br>{text-transform: capitalize} | {text-transform: capitalize} Capitalizes the first letter of each word, but does not convert other capital letters to lowercase letters. |
| text-decoration | none, underline, line-through | {text-decoration: none} | A {text-decoration: none} will remove the underline from your links. |
| color | color name, RGB triplet | {color: red}<br>{color: #ff0000}<br>{color: rgb(255,0,0)} | All of these formats return the color red. |
| line-height | by number, by size unit, by percentage | {line-height: 2}<br>{line-height: 5px}<br>{line-height: 150%} | {line-height: 2} returns double-spacing. {line-height: 3} returns triple spacing, etc. |
| text-align | left, right, center, justify | {text-align: justify} | only works on elements that define a block of data, like <P>, <H1>-<H6>, <UL>, <OL>, <DIV> and <BLOCKQUOTE>. |

* Size units can be defined in pixels (px), point (pt), inches (in) centimeters (cm), millimeters (mm), and pica (pc).

** Size names can be declared as xx-small, x-small, small, medium, large, x-large, xx-large, larger and smaller.

| Declaration | Property | Example | Notes |
|---|---|---|---|
| `margin-top`<br>`margin-bottom`<br>`margin-right`<br>`margin-left` | `by size unit,`<br>`by percentage` | {margin-top: 25px}<br>{margin-bottom: 1in}<br>{margin-right: 10%}<br>{margin-left: 5cm} | These can be used to change the whitespace around an object or piece of text, effectively moving it up, down or to the right or left. |
| `border-width` | `by size unit` | {border-width: 2px} | |
| `border-style` | `double, groove, ridge, outset, inset` | {border-style: double} | |
| `border-color` | `color name, RGB triplet` | {border-color: red}<br>{border-color: #ff0000}<br>{border-color: #f00}<br>{border-color: rgb(255,0,0)} | |
| `background-color` | `color name, RGB triplet` | {background-color: red}<br>{background-color: #ff0000} | Handled differently by different browsers. |
| `background-image` | `url(image_name)` | {background-image: url(imp.gif)} | |
| `background-repeat` | `no-repeat, repeat-x, repeat-y` | {background-repeat: no-repeat} | |
| `position` | `absolute, relative, static` | {position: absolute} | `{position: relative}` positions elements with respect to where HTML would normally place them |
| `top` | `by size unit, by percentage` | {position: absolute; top: 10%} | |
| `left` | `by size unit, by percentage` | {position: relative; left: 10px} | |
| `height` | `by size unit, by percentage` | {height: 20%} | |
| `width` | `by size unit, by percentage` | {width: 300px} | |
| `display` | `block, in-line (default), none` | {display: block} | `block` treats elements as thought they were inside DIV tags, each starting on a line of its own. |
| `z-index` | `by number` | {z-index: 20} | `z-index` controls the stack order of elements. Higher z-index values appear on top. |

# Shortcut Declarations

| Declaration | Property | Example | Notes |
|---|---|---|---|
| `font` | `font-style,`<br>`font-variant,`<br>`font-weight,`<br>`font-size,`<br>`line-height,`<br>`font-family` | {font: italic bold 12pt/14pt times, serif}<br>  <br> | The `font` declaration can be used to declare many different declaration properties at once. |
| `background` | `background-color,`<br>`background-image,`<br>`background-repeat` | {background: url(background.gif) #CCFFCC repeat-y} | The `background` declaration can be used to declare many different declaration properties at once. |

# Appendix E: Special Characters

XML includes support within **PCDATA** sections for characters such as © and ñ that do not appear on American keyboards. These characters may be inserted by entering their numeric **code**. The 5 items listed first may also be specified by their **entity name**.

| Description | Char | Code | Entity name |
|---|---|---|---|
| carriage return | |  --> | |
| standard space | |   --> | |
| quotation mark | " | " --> " | " --> " |
| quotation mark | ' | ' --> " | ' --> " |
| ampersand | & | & --> & | & --> & |
| less-than sign | < | < --> < | < --> < |
| greater-than sign | > | > --> > | > --> > |
| non-breaking space | |   --> | |
| inverted exclamation mark | ¡ | ¡ --> ¡ | |
| cent sign | ¢ | ¢ --> ¢ | |
| pound sign | £ | £ --> £ | |
| currency sign | ¤ | ¤ --> ¤ | |
| yen sign | ¥ | ¥ --> ¥ | |
| broken vertical bar | ¦ | ¦ --> ¦ | |
| section sign | § | § --> § | |
| spacing diaresis | ¨ | ¨ --> ¨ | |
| copyright sign | © | © --> © | |
| feminine ordinal indicator | ª | ª --> ª | |
| angle quotation mark, left | « | « --> « | |
| negation sign | ¬ | ¬ --> ¬ | |
| soft hyphen | - |  --> - | |
| circled R registered sign | ® | ® --> ® | |
| spacing macron | ¯ | ¯ --> ¯ | |
| degree sign | ° | ° --> ° | |
| plus-or-minus sign | ± | ± --> ± | |
| superscript 2 | ² | ² --> ² | |
| superscript 3 | ³ | ³ --> ³ | |
| spacing acute | ´ | ´ --> ´ | |
| micro sign | µ | µ --> µ | |
| paragraph sign | ¶ | ¶ --> ¶ | |
| middle dot | · | · --> · | |
| spacing cedilla | ¸ | ¸ --> ¸ | |
| superscript 1 | ¹ | ¹ --> ¹ | |
| masculine ordinal indicator | º | º --> º | |
| angle quotation mark, right | » | » --> » | |
| fraction 1/4 | ¼ | ¼ --> ¼ | |
| fraction 1/2 | ½ | ½ --> ½ | |
| fraction 3/4 | ¾ | ¾ --> ¾ | |
| inverted question mark | ¿ | ¿ --> ¿ | |
| capital A, grave accent | À | À --> À | |
| capital A, acute accent | Á | Á --> Á | |
| capital A, circumflex accent | Â | Â --> Â | |
| capital A, tilde | Ã | Ã --> Ã | |
| capital A, dieresis or umlaut mark | Ä | Ä --> Ä | |
| capital A, ring | Å | Å --> Å | |
| capital AE diphthong (ligature) | Æ | Æ --> Æ | |
| capital C, cedilla | Ç | Ç --> Ç | |

| Description | Char | Code | Entity name |
|---|---|---|---|
| capital E, grave accent | È | È --> È | |
| capital E, acute accent | É | É --> É | |
| capital E, circumflex accent | Ê | Ê --> Ê | |
| capital E, dieresis or umlaut mark | Ë | Ë --> Ë | |
| capital I, grave accent | Ì | Ì --> Ì | |
| capital I, acute accent | Í | Í --> Í | |
| capital I, circumflex accent | Î | Î --> Î | |
| capital I, dieresis or umlaut mark | Ï | Ï --> Ï | |
| capital Eth, Icelandic | Ð | Ð --> Ð | |
| capital N, tilde | Ñ | Ñ --> Ñ | |
| capital O, grave accent | Ò | Ò --> Ò | |
| capital O, acute accent | Ó | Ó --> Ó | |
| capital O, circumflex accent | Ô | Ô --> Ô | |
| capital O, tilde | Õ | Õ --> Õ | |
| capital O, dieresis or umlaut mark | Ö | Ö --> Ö | |
| multiplication sign | × | × --> × | |
| capital O, slash | Ø | Ø --> Ø | |
| capital U, grave accent | Ù | Ù --> Ù | |
| capital U, acute accent | Ú | Ú --> Ú | |
| capital U, circumflex accent | Û | Û --> Û | |
| capital U, dieresis or umlaut mark | Ü | Ü --> Ü | |
| capital Y, acute accent | Ý | Ý --> Ý | |
| capital THORN, Icelandic | Þ | Þ --> Þ | |
| small sharp s, German (sz ligature) | ß | ß --> ß | |
| small a, grave accent | à | à --> à | |
| small a, acute accent | á | á --> á | |
| small a, circumflex accent | â | â --> â | |
| small a, tilde | ã | ã --> ã | |
| small a, dieresis or umlaut mark | ä | ä --> ä | |
| small a, ring | å | å --> å | |
| small ae diphthong (ligature) | æ | æ --> æ | |
| small c, cedilla | ç | ç --> ç | |
| small e, grave accent | è | è --> è | |
| small e, acute accent | é | é --> é | |
| small e, circumflex accent | ê | ê --> ê | |
| small e, dieresis or umlaut mark | ë | ë --> ë | |
| small i, grave accent | ì | ì --> ì | |
| small i, acute accent | í | í --> í | |
| small i, circumflex accent | î | î --> î | |
| small i, dieresis or umlaut mark | ï | ï --> ï | |
| small eth, Icelandic | ð | ð --> ð | |
| small n, tilde | ñ | ñ --> ñ | |
| small o, grave accent | ò | ò --> ò | |
| small o, acute accent | ó | ó --> ó | |
| small o, circumflex accent | ô | ô --> ô | |
| small o, tilde | õ | õ --> õ | |
| small o, dieresis or umlaut mark | ö | ö --> ö | |
| division sign | ÷ | ÷ --> ÷ | |
| small o, slash | ø | ø --> ø | |
| small u, grave accent | ù | ù --> ù | |
| small u, acute accent | ú | ú --> ú | |
| small u, circumflex accent | û | û --> û | |
| small u, dieresis or umlaut mark | ü | ü --> ü | |
| small y, acute accent | ý | ý --> ý | |
| small thorn, Icelandic | þ | þ --> þ | |

# Appendix F: Recommended Resources

## Books

Castro, Elizabeth, XML for the World Wide Web. Berkeley: Peachpit, 2001.

Kay, Michael. XSLT Programmer's Reference. Chicago: Wrox, 2000.

Marchal, Benoît. XML By Example. Indianapolis: Que, 2000.

Homer, Alex. XML IE5. Chicago: Wrox, 1999.

## Newsgroups

- **comp.text.xml**
- **microsoft.public.xml**

## Web Sites

- **World Wide Web Consortium XML Pages** (*http://www.w3.org/XML/*): The main resource for XML, including current and proposed standards, sample documents, and news articles.
- **Brown University Scholarly Technology Group** (*http://www.stg.brown.edu/service/xmlvalid/*): An excellent public resource for validating XML documents against their DTDs.
- **XML.com** (*http://www.xml.com/*): An excellent resource for code samples, developers forums, news releases, and the like.
- **MSDN** (*http://msdn.microsoft.com*): Hundreds of pages of documentation on the Microsoft XML standards, code samples, and more, including an excellent search interface. A tremendous resource for writing XML for the MSXML parser.
- **Perfect XML (http://www.perfectxml.com):** Good references to xml software and other resources.
- **ZVON (http:**//www.zvon.org): Good xslt and schema references and tutorials.

20040816v3.1.0_mrosenshield