XML Programming
Using C# and .NET

Student Guide

Revision 4.0

Object Innovations Course 4125

XML Programming Using C# and .NET
Rev. 4.0

Student Guide

® is a registered trademark of Object Innovations.

Authors: Robert J. Oberg, E. J. Jones, Will Provost

Special Thanks: Gordon Hester

Object Innovations
877-558-7246
www.objectinnovations.com

Table of Contents (Overview)

Prerequisites and Pacing

- **C# programming experience is required for this course.**
- **Some experience with XML is also assumed.**
  - The student should understand basic XML grammar, and be able to read and write well-formed XML documents.
  - The concept of a **valid** XML document, as opposed to a merely well-formed one, should be understood, via some experience with either DTDs or XML Schema.
- **This course contains a great deal of material.**
  - Please give feedback to the instructor at the beginning of the course on the topics of most interest to you, and during the course about how the pacing is working for you.

Labs

- **The course relies on hands-on experience in various topics and techniques.**
- **Application code for this course is all in *XmlCs* under the top-level directory, which by default is *C:\OIC*.**
- **Where possible, starter code is provided to take work off your hands that would be largely irrelevant to the topic of the lab; thus you can be as productive as possible in the time allotted and focus on the topic at hand.**
- **The labs are installed by running the simple self-extractor:**

```
Install_XmlCs_40.exe
```

  – This can be found in the file **4125_Lab_40.zip**, which is available on Object Innovations' lab download page:

http://www.objectinnovations.com/Labs/index.html

Directory Structure for the Course

- **The course has a combination of directories under the course root directory *C:\OIC\XmlCs*.**
  - Under this root, the chapter directories **Chap01**, **Chap02**, and so on, hold code examples, including all the starting, intermediate and answer versions of all the labs.
  - The **Labs** directory holds one subdirectory for each lab in the course, named for the lab number.
  - The **Demos** directory is provided for performing in-class demonstrations led by the instructor.
  - The **CaseStudy** directory contains files pertaining to the Zenith Courseware case study.
- **Data files install to the directory *C:\OIC\Data*.**
  - In some examples, data files are contained in the source code directory. In this case, the files are configured in Visual Studio as resources that copy automatically to the output directory when the project is built.
- **The *C:\OIC\Tools* directory contains the executables of XML tools and demonstration programs that are provided with the course.**

Table of Contents (Detailed)

Chapter 1

.NET Framework XML Overview

.NET Framework XML Overview

Objectives

After completing this unit you will be able to:

- **Describe the role of parsing in XML applications.**
- **Identify the main parsing APIs in .NET, and describe the major differences between them.**
- **Describe the major .NET Framework XML classes.**
- **Describe XML serialization and its role throughout the Framework.**
- **Discuss the close relationship between XML and ADO.NET.**
- **Describe XPath and XSLT and the .NET Framework support for these XML technologies.**
- **Explain the use of Language Integrated Query (LINQ) in working with XML data sources.**

XML

- **The *eXtensible Markup Language*, or *XML*, has become a very popular choice for a wide array of software applications:**
  - Traditional web applications enhanced with XML as an HTML transformation source
  - XML as a portable format for data exchange and archiving
  - Business-to-business messaging and Web Services
  - Many more
- **It is surprising when learning the language how much can be accomplished using XML and related standards and generic, pre-built tools, without any traditional application code.**
  - **XML Stylesheet Language for Transformations**, or **XSLT**, enables moderately sophisticated document transformation.
  - Modern web browsers can present XML documents to users with the aid of XSLT, **XSL**, or **Cascading Style Sheets** (**CSS**), even including hyperlinks with the help of **XLink**.
  - Detailed document structure and content validation can be effected using **XML Schema** and a validating parser.

Parsing XML

- **At some point, however, the information in XML documents must be available to application code.**
- **At its most basic, the process by which an application reads the information in an XML document is known as *parsing* the document.**
  - Clearly, the literal meaning of the term refers to the gritty work of reading the document as a stream of characters, and interpreting that stream according to XML grammar.
  - Stated another way, the parsing task might be seen as that of abstracting the document content – often called its **information set** or **infoset** – from its lexical representation in XML proper.
  - This information set can then be read by application code, using any number of possible models.
  - Document validation can also be performed as part of parsing.
- **All these jobs are quite complex, but, thanks to the design of XML, also generic.**
- **Thus individual business applications do not have to write their own parsing code, instead leveraging prebuilt packages that offer APIs to their parsing capabilities.**

Using XML in .NET Applications

- **Microsoft is a big proponent of XML and is a huge participant in the W3C.**
- **The .NET Framework has many areas which are enabled for XML processing.**
  - Database queries can be returned in an XML format with XML Schema definitions.
  - Many configuration files in .NET projects are stored in XML format.
  - Web Services uses the XML based SOAP protocol to remotely call objects on a server.
  - The Universal Description, Discovery, and Integration (UDDI) service uses XML to request and return data to clients.
- **XML support is built into the .NET Framework**
- **The many different types of .NET applications utilize services provided in the Framework.**
  - The XML services we are going to look at include .NET classes that interact with the CLR.

The Core .NET XML Namespaces

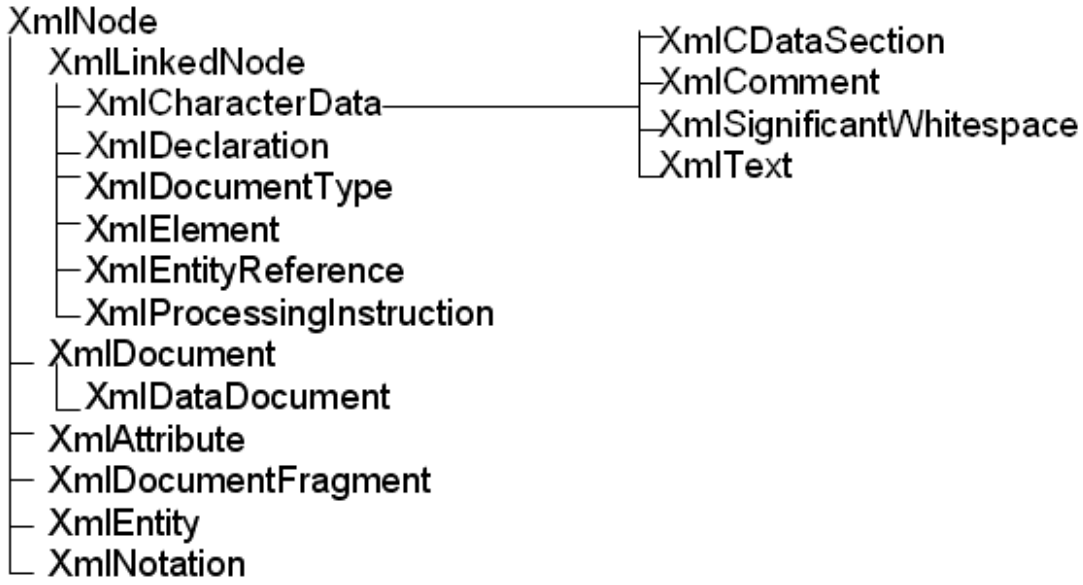
- **The *System.Xml* namespace and its subsidiary namespaces encapsulate the XML functionality in the .NET Framework.**
  - They contain the classes that parse, validate, traverse, and create XML streams.
  - The namespace classes support the following W3C XML standards.

XML 1.0 and XML namespaces	XML schemas
XPath	XSLT
DOM level 1& 2 core	SOAP (object serialization)

- **The *System.Xml* namespace contains the essential and major classes for reading and writing.**
- **The *System.Xml* namespace contains five subsidiary namespaces.**
  - **System.Xml.Schema** –XML classes that provide support for XML Schemas definition language (XSD) schemas.
  - **System.Xml.Serialization** – classes that are used to serialize objects into XML streams.
  - **System.Xml.XPath** – contains the XPath parser and evaluation engine for querying XML data streams.
  - **System.Xml.Xsl** – these classes support the Extensible Stylesheet Transformation (XSLT).
  - **System.Xml.Linq** – classes supporting LINQ to XML

The .NET XML Classes

- **The parent class for the nodes found in an XML data stream is called *XmlNode*.**
- **Depending upon the specific type of node the *XmlNode* class has six derived classes. They are:**
  - System.Xml.XmlLinkedNode
  - System.Xml.XmlAttribute
  - System.Xml.XmlDocument
  - System.Xml.XmlDocumentFragment
  - System.Xml.XmlEntity
  - System.Xml.XmlNotation
- **Each of the derived classes contains the properties and methods that are suitable for that type of node.**



.NET XML Classes and Interfaces

- **In addition to specifying class hierarchy, the classes have three interfaces in common:**
  - **ICloneable** – Xml nodes can be copied to create a new instance.
  - **IEnumerable** – Xml nodes support the **foreach** loop for C#.
  - **IXPathNavigable** – gives the ability to retrieve data from the node using XPath queries.
  - We will use all of these when we start coding.
- **Parsers in the *System.Xml* namespace are found in several classes.**
  - **XmlReader** — this class is a fast non-cached forward-only parser.
  - **XmlReaderSettings** — this class can specify features for an XmlReader object, including validating an XML input using DTDs or W3C's XML Schema definition language (XSD).
  - **XmlWriter** — this class provides the methods to assist you in writing syntactically correct XML. It can write to a file, stream, console, and other output devices. Options can be specified using an **XmlWriterSettings** object.
  - **XmlDocument** — this class implements the W3C Core Document Object Model Level 1 and Level 2. It stores the XML tree in memory and allows you traverse and modify the nodes.

Parsing Techniques

- **There are two traditional methods for parsing XML streams, and each has advantages and disadvantages.**
  - The Document Object Model (DOM).
  - The Simple API for XML (SAX). This API is **not** supported by .NET and will not be discussed further.
- **The Document Object Model has the following characteristics:**
  - The DOM is a W3C standard caching parser and is widely adopted in many programming environments.
  - Good if you need to move forward and backward in the stream and if you need to modify the node values.
  - Bad choice if you're forward scanning only and not modifying the document.
  - The DOM keeps the entire parsed tree in-memory thereby consuming computer resources.

.NET Parsing Techniques

- **.NET provides three different techniques for parsing:**
  - XML readers and writers
  - XML document editing using the DOM
  - XML document editing using **XPathNavigator**
- **XML readers provide a more effective read-only, non-cached, forward-only parser.**
  - This pull-model parser allows the application to control parser by specifying which nodes are of interest.
  - Saves processing time because only requested nodes are sent to the application.
  - **XmlWriter** supports generating a stream of XML content.
- **The *XmlDocument* class implements DOM Level 2 functionality.**
- ***XPathNavigator* provides an editable, cursor-style API for reading and editing XML documents.**
  - In .NET 1.1, this class was read-only, but write capability is now available with .NET 2.0.
  - This model is typically more useable than the DOM approach.

SimpleXML Programming Example

- **The example program *SimpleXML* in the *Chap01* folder illustrates a number of features of XML programming using .NET.**

```
using System;
using System.Xml;

namespace SimpleXML
{
   class Program
   {
      const string xmlPath =
         @"..\..\NewCarLot.xml";
      static void Main(string[] args)
      {
         WriteCars();
         ReadCars();
         ParseWithTheDom();
      }
      static void WriteCars()
      {
         ...
      }
      static void ReadCars()
      {
         ...
      }
      static void ParseWithTheDom()
      {
         ...
      }
   }
}
```

  - The constant string **xmlPath** specifies the XML file.

XmlReader Parsing Example

- **The following code example in the *ReadCars()* method shows parsing using the *XmlReader* class.**

```
XmlReader tr = XmlReader.Create(xmlPath);
Console.WriteLine("XmlReader Demo");
Console.WriteLine("==========================");
while (tr.Read())
{
   if (tr.NodeType == XmlNodeType.Element)
   {
      Console.Write("Node Name:" + tr.Name);
      Console.WriteLine("  Attribute Count:" +
         tr.AttributeCount.ToString());
   }
}
tr.Close();
```

- **The output of the code is as follows**

```
XmlReader Demo
==========================
Node Name:Dealership  Attribute Count:1
Node Name:Car  Attribute Count:0
Node Name:Make  Attribute Count:0
Node Name:Model  Attribute Count:0
Node Name:Year  Attribute Count:0
Node Name:VIN  Attribute Count:0
Node Name:Color  Attribute Count:0
Node Name:Price  Attribute Count:0
```

XmlWriter Example

- **In this example we use *XmlWriter* to create the XML file that we read in earlier code. This code is in the *WriteCars()* method.**

```
XmlWriterSettings settings =
   new XmlWriterSettings();
settings.Indent = true;
XmlWriter tw = XmlWriter.Create(xmlPath, settings);
//Opens the document
tw.WriteStartDocument();
//Write comments
tw.WriteComment("A lot of cars!");
//Write first element
tw.WriteStartElement("Dealership");
tw.WriteAttributeString("name", "Cars R Us");

tw.WriteStartElement("Car");
//Write the Make of the Car element
tw.WriteStartElement("Make");
tw.WriteString("AMC");
tw.WriteEndElement();

//Write one more element
tw.WriteStartElement("Model");
tw.WriteString("Pacer");
tw.WriteEndElement();

//... Shortened for brevity
tw.WriteStartElement("Price");
tw.WriteString("3998.99");
tw.WriteEndElement();

tw.WriteEndElement();     // end of car
tw.WriteEndElement();     // end of dealership
tw.WriteEndDocument();    // end of document
tw.Close();               // close writer
```

XmlWriter Example (Cont'd)

- **This example creates the *NewCarLot.xml* file in the *SimpleXML* directory.**

```
<?xml version="1.0"?>
<!--A lot of cars!-->
<Dealership name="Cars R Us">
  <Car>
    <Make>AMC</Make>
    <Model>Pacer</Model>
    <Year>1977</Year>
    <VIN>CZ7821</VIN>
    <Color>Blue</Color>
    <Price>3998.99</Price>
  </Car>
</Dealership>
```

.NET DOM Parser Features

- **The *XmlReader* and *XmlWriter* classes aren't resource demanding but they lack the ability to move around or modify the XML stream.**
- **The *XmlDocument* class is resource intensive because it stores the parsed XML internally offering you the advantages to navigate, modify, or create the data.**
- **The downside is the resources used on the machine will be proportional to the size of the entire XML stream you've read or are creating.**
- **The DOM is a language-independent W3C specification; Microsoft's *XmlDocument* class implementation has many of the same property and methods (with some extensions).**
- **The *XmlNode* class, like the DOM Node interface, specifies the basic functionality for the different types of nodes in an XML stream.**

XmlDocument Example

- **The following code, found in the method *ParseWithTheDOM()*, reads the XML file created in the previous example.**
  - This code uses the DOM parser with the **XmlNode** class.

```
XmlDocument doc =  new XmlDocument();
doc.Load(xmlPath);

XmlNode root = doc.DocumentElement;
XmlNodeList list = root.SelectNodes("//*");
foreach ( XmlNode elem in list )
{
   Console.WriteLine( elem.Name);
}
```

- **The output is:**

```
Dealership
Car
Make
Model
Year
VIN
Color
Price
```

Other XML Features in .NET

- **XML serialization converts the state of an object into an XML byte stream suitable for persisting or transporting.**
  - .NET supports XML serialization in the namespace **System.Xml.XmlSerialization**.
- **The foundation of XML serialization is XML Schema, which is a W3C Recommendation.**
  - XML Schema is a complete type system.
  - .NET supports reading and writing XML Schema in the namespace **System.Xml.Schema**.
- **ADO.NET is tightly coupled to XML.**
  - You can exchange both data and schema information between XML and DataSets.
  - Support is provided in the **System.Data** namespace.
  - Classes such as **DataSet** have explicit methods for working with XML data.
- **XPath provides a mechanism to query for content in an XML document.**
  - .NET support is provided in **System.Xml.XPath**.
- **XSLT enables transformation of XML into text, HTML or other XML.**
  - .NET support is provided in **System.Xml.Xsl**.

LINQ to XML

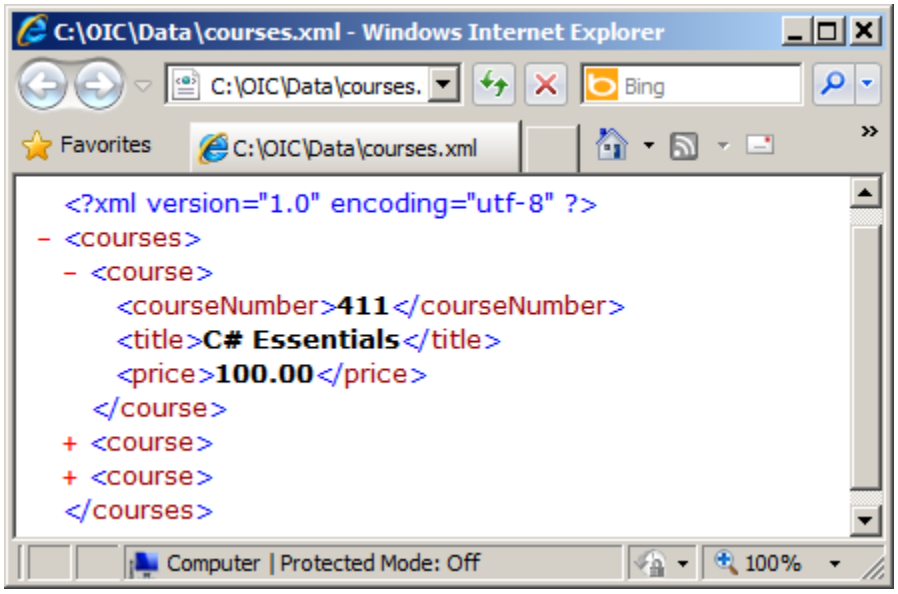
- **Language-Integrated Query (LINQ) provides an intuitive syntax for querying a variety of data sources using C# and Visual Basic.**
- **The query syntax is part of the programming language, giving the advantages of strong typing and tool support such as IntelliSense in Visual Studio.**
- **LINQ provides a consistent API that can be used with many different kinds of data, including .NET collections, SQL Server databases and XML documents.**
- **LINQ to XML is a programming model for manipulating XML documents using .NET languages.**
- **It is similar in goals to the Document Object Model (DOM) but lighter weight and easier to work with.**
  - With respect to query capability, the programming model is consistent with the model for other LINQ data sources.
- **The namespace is *System.Xml.Linq*.**
  - Important classes include **XDocument**, **XElement** and **XAttribute**.

XML and the Web

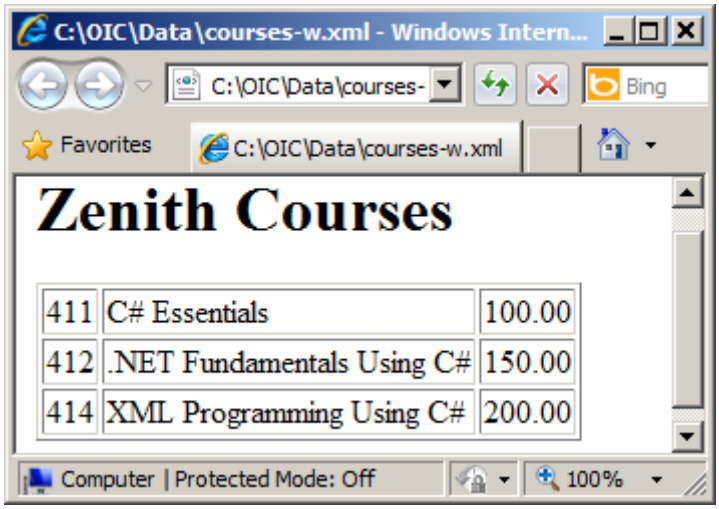
- **A major motivation for the development of XML was to support Web applications.**
- **Both XML and HTML are based on SGML, which was quite complex.**
  - HTML is quite simple, but is only concerned with *presentation*. Also, although similar to XML, it does not conform to precise XML syntax.
  - XML is also simple and is concerned with the information content of a document without regard to presentation.
  - XHTML is a markup language understood by modern browsers that is HTML with precise XML syntax.
- **A robust way to manage complex information in a Web site is through XML, with transformation to HTML as needed for presentation.**
  - XSLT is an XML technology that supports transformation of XML to HTML and other formats through a *stylesheet*.
  - There are many other ways to integrate XML with a Web site, including its use in data management, configuration, and Web services.

Internet Explorer and XML

- **The basic Microsoft Web tool is the Internet Explorer Web browser.**
  - It will display well-formed XML in a collapsible tree view.



  - Through a style sheet it can format XML.



  - The example files are **courses.xml** and **courses-w.xml** in the **Data** folder.

Summary

- **XML parsing is the cornerstone of .NET Framework application development. Many higher-level application capabilities can make use of XML enabled features:**
  - Cached/Non-cached, push model, syntax/validating type parsers available
  - XML object serialization
  - XML messaging, for instance using SOAP
- **Also, there are a number of XML-related specifications that define their own "languages," or really their own XML vocabularies – examples are XSLT and XML Schema.**
  - Each of these allows some information to be defined in an XML document: an XSLT style sheet or transform, an XML Schema.
  - Because these each leverage basic XML, the style sheets and schema can themselves be parsed and manipulated, just like any other XML document.
- **XML is tightly integrated with ADO.NET.**
- **Language-Integrated Query (LINQ) provides an intuitive syntax for querying a variety of data sources, including XML documents, from a programming language.**

Chapter 6

Manipulating XML Information with the DOM

Manipulating XML Information with the DOM

Objectives

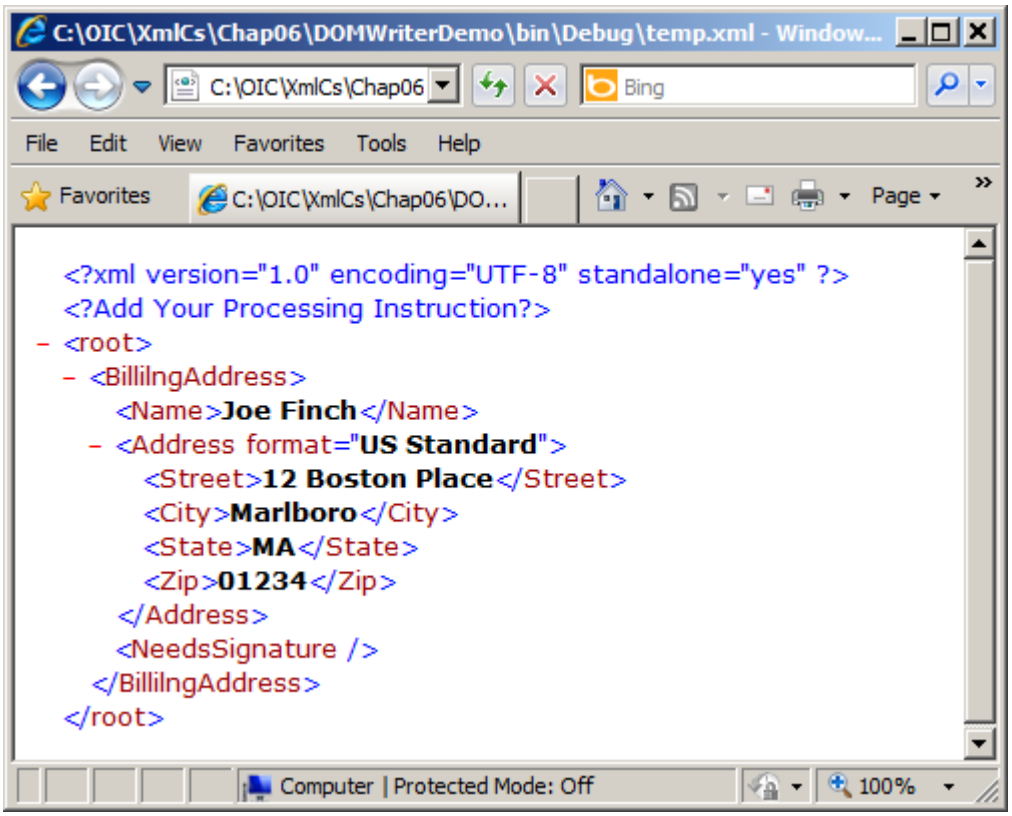
After completing this unit you will be able to:

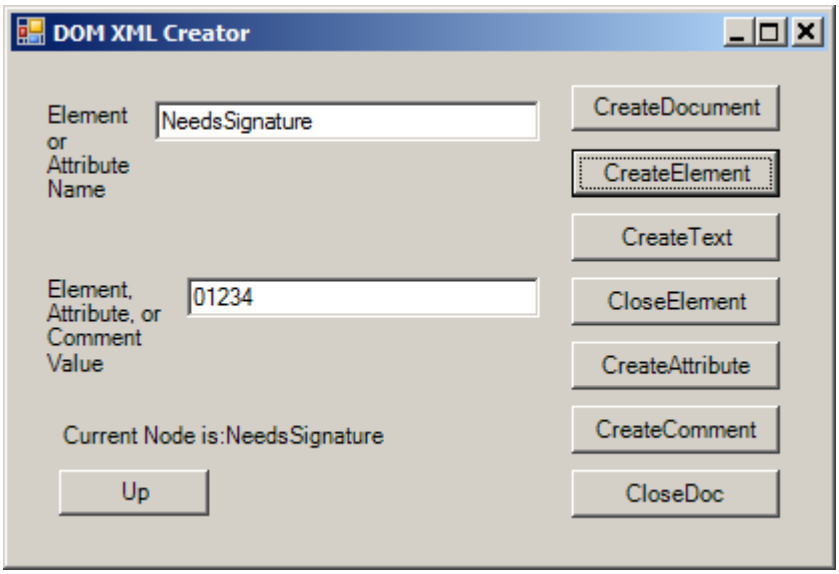
- **Build entirely new documents using the DOM, and populate them with desired information to create a new XML document.**
- **Add, remove, and replace nodes as children of other nodes in a DOM tree.**
- **Clone nodes and subtrees for processing or document modification.**
- **Change element and attribute values.**

Modifying Documents

- **In the previous chapter we focused exclusively on using the *XmlDocument* as a parser.**
- **The *XmlDocument* is actually a read/write class.**
  - Nodes of all types have both accessors and mutators, and can be modified, added and removed as children of other nodes.
  - In some cases content can be modified; some node types have certain immutable properties that can only be "changed" by removing the original node with a partially-modified copy.
  - To create a new XML document, simply create an instance of **XmlDocument** and start adding element, attribute or other node(s). The DOM tree will be maintained in memory and can be written to a file using the **save** method.
- **In this chapter we'll learn how to use the DOM classes to modify existing XML documents, and to create new ones.**

Build A DOM Tree – Demo



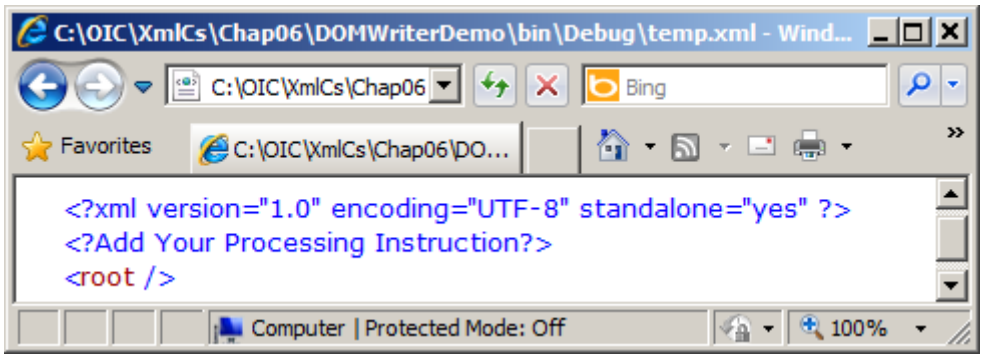


Build A DOM Tree – Demo

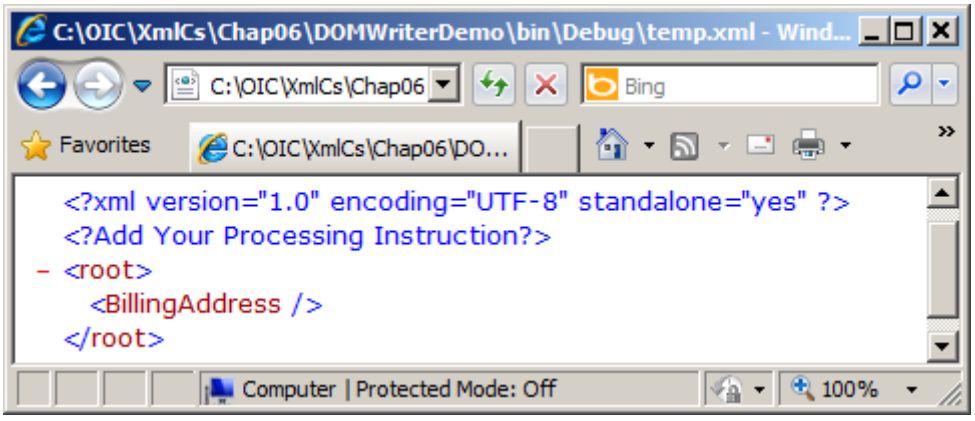
- **In this demo you will create and navigate a DOM tree. The source code is in the *DOMWriterDemo* folder in the chapter directory.**

1. Locate the executable file **DOMWriterDemo.exe** in the folder **\OIC\Tools**. Run this program, which will create a file **temp.xml**, which you can view with Internet Explorer.

2. Once the program starts, press the **CreateDocument** button and you should see the following XML in Internet Explorer. Refresh after each operation. Current node is "root."

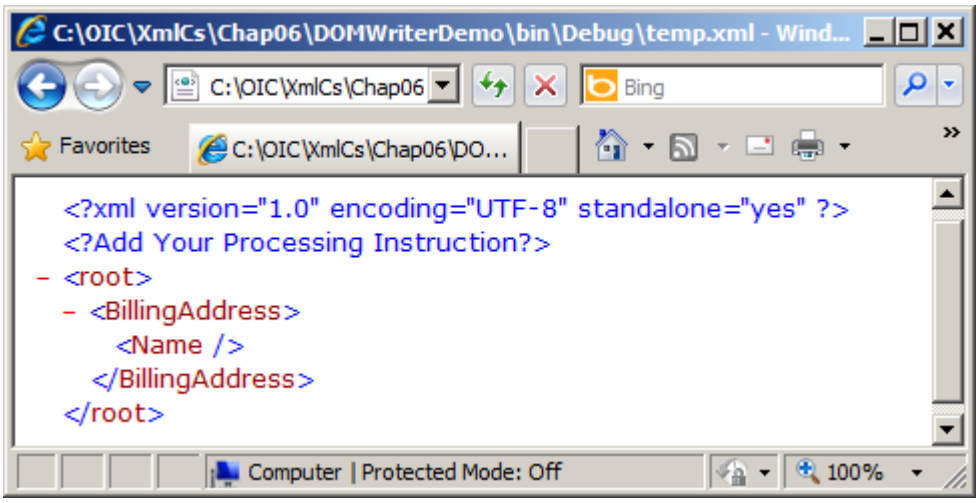


3. Now type in "BillingAddress" in the **Element or Attribute Name** text box and then press the **CreateElement** button. Notice your Current Node is now "BillingAddress."



Build A DOM Tree – Demo

4. Now type in “Name” in the **Element or Attribute Name** text box and then press the **CreateElement** button. Your Current Node is now “Name.”



5. Next we create a **Text** node. Type in your name into the **Element, Attribute, or Comment Value** text box and then press the **CreateText** button. The current node is still “Name” so press the **CloseElement** button, and the current node moves up the tree to “BillingAddress.”

6. Next add the “Address” node and the “format” attribute. In adding an attribute you should specify both the name and value.

7. Add the rest of the nodes to the tree using the buttons. The Current Node will always indicate which element you are going to add a child or attribute to. Closing the element always moves you up one level. You can also press the **Up** button to move up one level.

8. Experiment as you like.

Modifying Documents

- **The *XmlDocument* class is key to this capability, since it has all the factory methods for various node types (the important ones are listed as follows) :**

```
XmlElement CreateElement(string name);

XmlAttribute CreateAttribute(string name);

XmlText CreateTextNode(string text);

XmlComment CreateComment(string data);

void Save(destination);
    // string, Stream, TextWriter, XmlWriter
```

The XmlNode Class – Modifications

- **Here is yet another slice of the larger *XmlNode* class, this one including the main mutators:**

```
XmlNode InsertAfter(XmlNode newChild,
   XmlNode refChild);

XmlNode InsertBefore(XmlNode newChild,
   XmlNode refChild);

XmlNode ReplaceChild(XmlNode newChild,
   XmlNode oldChild);

XmlNode RemoveChild(XmlNode oldChild);

XmlNode AppendChild(XmlNode newChild);

XmlNode CloneNode(bool deep);

XmlDocument OwnerDocument {get;}

string Value {get; set;}

string OuterXml {get;}

string InnerXml {get; set;}
```

  - Much of this interface is concerned with managing the child list: insert, append, remove, and replace operations.
  - Note also that from any node one can get the owning document. This is important when creating new content.
  - The **Xml** properties will give you a string that represents the XML content of the node and its subtree.

Legal and Illegal Modifications

- **Obviously, not all combinations and orderings of nodes as parents and children are legal: an attempt to add an element to a comment, for instance, must fail.**
- **The *XmlException* class encapsulates exceptional conditions in DOM programming, many of which have to do with making changes to existing nodes.**
  - But be aware that some exceptions may be fit into another exception class. The code below throws an **InvalidOperationException**.

```
try
{
   XmlDocument doc = new XmlDocument();
   XmlNode commentNode, elementNode;
   commentNode = doc.CreateComment("my comment");
   elementNode = doc.CreateElement("myElement");
   doc.AppendChild(commentNode);
   doc.FirstChild.AppendChild(elementNode);
}
catch (XmlException e)
{
   Console.WriteLine(e.GetType() + " " +
                     e.Message);
}
catch (Exception e)
{
   Console.WriteLine(e.GetType() + " " +
                     e.Message);
}
```

  - See **IllegalDOM** in the chapter directory.

Managing Children

- **Use of the *XmlNode* class to add or remove child elements is simple enough.**
- **Additions can be managed using either A*ppendChild()*, *InsertAfter()*, or *InsertBefore()*.**
  - The choice between them is really a question of convenience in a particular algorithm.
  - Each will assure uniqueness in the child list by first removing the node if it is already in the list somewhere.
- **To simply remove a child, call *RemoveChild()*.**
- **The *ReplaceChild()* method has the effect of an *InsertBefore()* combined with a *RemoveChild()*.**
  - There is a subtle difference having to do with error recovery.
  - **ReplaceChild()** is typically implemented to assure atomicity and consistency of the operation: if the new child node is rejected for any reason, the entire replacement will fail.
  - Sometimes this is the desired behavior, and sometimes not. Choose your approach to node replacement carefully in case of unexpected failure: should the existing node be removed regardless, or should it stay if the new node is unacceptable?

Cloning

- **The *XmlNode* class also provides the *CloneNode()* method.**
  - The DOM recommendation calls it a “generic copy constructor,” imprecisely echoing C++ terminology.
- **The method returns a new node of the same type and content as the source on which the method was called.**
- **The lone parameter, *deep*, affects the resulting node by directing the clone operation to make either a *shallow* or *deep copy*.**
  - If this parameter is true, the node and all its descendant nodes will be cloned into a new subtree.
  - If it is false, only the target node will be cloned; it will have no children.
  - Shallow-copied element clones will have the attributes of the source element, but none of the true child nodes.
- **Especially when used to make deep copies, the *CloneNode()* method can save quite a lot of code!**

Modifying Elements

- **There are several possible changes to an element.**
  - The tag name is immutable; to change it you must replace the element with a new one.
  - The character content of an element is captured in a separate text node as a child of the element. Thus, changing this means setting a new value on the child element.
  - The **XmlCharacterData** class includes a number of mutators that allow the text to be modified:

```
string Data {get; set;}

void AppendData(string strData);

void InsertData(int offset, string strData);

void DeleteData(int offset, int count);

void ReplaceData(int offset, int count,
   string strData);
```

  - Alternatively, a new text node can be emplaced or replaced.

Splitting Text and Normalizing

- **The *XmlText* class has the method *SplitText()*, which breaks the node in two at a given offset.**
  - The target of the call retains the text information up to the offset.
  - A new **XmlText** node is created which holds the information after the offset, and this node is returned by the method.
  - The new node is automatically inserted as a child of the target node's parent, right after the target node in the child list.
- **Separately, the *XmlNode* class offers the *Normalize()* method.**
  - This affects the entire subtree.
  - It rearranges text nodes as necessary to assure that there are no empty text nodes and no consecutive text nodes.
  - Most parsers will create a document in normal form, but after modifications, especially using **SplitText()**, it is possible that normalization will be necessary to support further processing by certain algorithms.
  - Once normal form is broken, text-processing algorithms must take care to accumulate text information from adjacent text nodes. If an applications need to gather contiguous character data to get the whole picture of an element's content it should call this method.

Modifying Attributes

- **Many of the *XmlElement* class mutators concern management of the element's attributes.**

```
void SetAttribute(string name, string value);

void RemoveAttribute(string name);

XmlAttribute SetAttributeNode(
   XmlAttribute newAttr);

XmlAttribute RemoveAttributeNode(
   XmlAttribute oldAttr);
```

  - **SetAttribute()** will assure that the desired attribute has the given value. If such an attribute already exists, the value is overwritten, and if not it is created.
  - **RemoveAttribute()** acts pretty much as advertised.
  - Note that one can choose to work directly with attribute values or to derive, create, manipulate, and use **XmlAttribute**-type nodes to capture the appropriate information.
  - **XmlAttribute** objects can be unwieldy by comparison, but have some advantages as separate node objects, such as being collectable and cloneable.

Lab 6

Shipping Information for Zenith Courseware

In this lab you will continue the **PrintShipDOM** program from the previous chapter to create an XML file that specifies shipping and handling charges for each destination. Very simple algorithms are used for determining shipping and handling costs. You are supplied a file **Ship.cs** that encapsulates these algorithms.

Detailed instructions are contained in the Lab 6 write-up in the Lab Manual.

Suggested time: 60 minutes

Summary

- **We've seen the DOM from both sides, now.**
  - The DOM offers quite a lot as a parsing technology.
  - Now we've learned how to use it to modify existing documents, and even to create new documents from scratch.
- **With the DOM API, application can be written to read and write XML documents of any complexity.**
- **As of DOM Level 2, and looking only at the Core recommendation, we can see a few shortcomings.**
- **There is no support yet for XML Schema in the DOM specification.**
  - In many ways DOM parsing is independent of type information, by design.
  - However, it would be very helpful to capture metadata for application use, and certainly for the parser to validate against an associated schema before returning the DOM tree.
  - Microsoft's DOM implementation does provide extensions that support XML Schema and DTD.

Chapter 10

Introduction to XSLT

Introduction to XSLT

Objectives

After completing this unit you will be able to:

- **Describe the origins of XSLT.**
- **Distinguish XSLT as a rule-based language from procedural languages used in application programming.**
- **Apply an XSLT transform to an XML source document to produce a transformed document.**
- **Use classes in the *System.Xml.Xsl* namespace to perform XSLT transforms programmatically.**

The Strange Ancestry of XSLT

- **Enamored of XML as they are, the W3C has over the last few years set out to adopt or replace almost every relevant web technology for use with XML.**
- **For styling purposes, the *eXtensible Stylesheet Language*, or *XSL*, was conceived.**
- **The XSL framework included its own transformation language, *XSL Transformations*, or *XSLT*.**
- **The more general applicability of XSLT quickly became clear, and it was identified as a distinct language and specification.**
- **XSL proper is now focused on styling and formatting – it is also known as *XSLFO*, or *XSL with Formatting Objects*.**
- **Oddly, the descendant technology, XSLT, is an integral part of the XSL process, which calls for a transformation from the source document to a tree of formatting objects.**

Input and Output

- **Transformation is fundamentally a process of taking some set of inputs and producing some set of outputs.**
  - In XSLT, the input set is defined by a single source XML document.
  - The output is a stream of characters generated by the transformation.
- **When one turns to XSLT for the first time, one often has a specific problem in mind, which usually involves extracting some source document content and reshaping it: filtering, sorting, reformatting, etc.**
- **To do whatever you want with XSLT requires a thorough knowledge of the subject, which is beyond the scope of this chapter.**
- **We'll introduce the subject in this chapter, by examining the transformation process, the rules by which templates are matched to source content, and the basic means of generating output.**
  - Most of our output will be static—that is, written into the transform, rather than extracted from the source document.
  - This will enable us to develop a good sense of control over the process and the look of the output, with few distractions.

Rule-Based Transformation

- **XSLT is a *rule-based* language.**
  - If XSLT were a **procedural** language, then a transformation would be described as a process, with steps in a certain order: get this element, read this value, write that attribute, etc.
  - As a rule-based language, XSLT instead defines a transformation as a series of rules, each of which dictates what output to produce based on certain types of input, if found.
  - The primary means of expressing a rule in XSLT is the **template**.
  - To apply a transformation is to look for elements in the source document that match the templates, and then to apply the output directives in that template based on the matching element and its content.

Stylesheets and Transforms

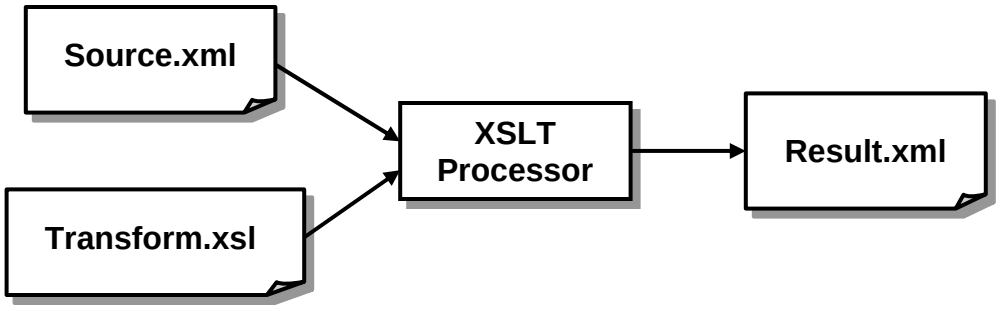
- **Thanks to the XSL legacy, XSLT uses elements defined in the "XSL/Transform" namespace.**
- **To differentiate between styling and true transformation tasks, an element has been added to the namespace.**
  - A **stylesheet** is a tree with **xsl:stylesheet** at its root; a **transform** is a tree with **xsl:transform** at its root.
  - Most often this element is also the XML document element.
  - This distinction is informal and is applied "to taste"; formally, the two elements are identical, and either will be accepted by a transformation engine (or web browser).
- **In either case, this topmost element must:**
  - Define the version of XSLT in use – we're covering version 1.0 here.
  - Import the XSL namespace.

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

- **From here, there is a subset of all XSLT elements whose members are defined as *top-level* elements.**
  - Only instances of these types can appear as children of the **stylesheet** or **transform** element.
  - Such elements can occur nowhere else.

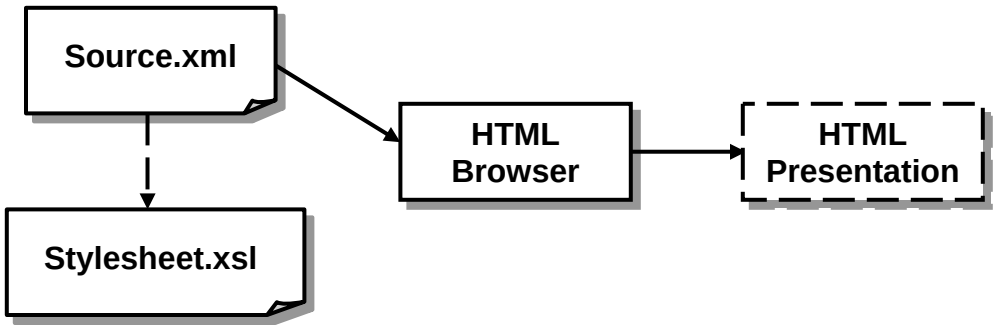
Applying a Transform to a Document

- **Generally, XSLT fosters a decoupling of XML source document from the transformed output, and also from the transform document as an *.xsl* file.**
  - Commonly, a transformation engine is applied to the source and the transform, and generates the output text or document.
  - This is often done using a command-line tool, as part of a script for instance.
  - The process can also be brought into the realm of application programming and be hosted by a software component that coordinates one or more transformations.



Referencing a Stylesheet

- **It is possible for an XML document to directly reference an external .xsl file, as a stylesheet.**
  - This is most useful when XSLT is being used to generate HTML for presentation.
  - The browser would not otherwise know that a transformation (stylesheet) were to be applied.



- **A transform can also be embedded in an XML document, so the *xsl:transform* element is no longer the root element, but a child of some other element.**
  - This is, again, most useful for presentation, and takes the progression to its extreme by packing the XML content and presentation logic all in one document.
  - The document still makes a stylesheet reference, but to a part of its own tree, rather than to an external file.

Templates

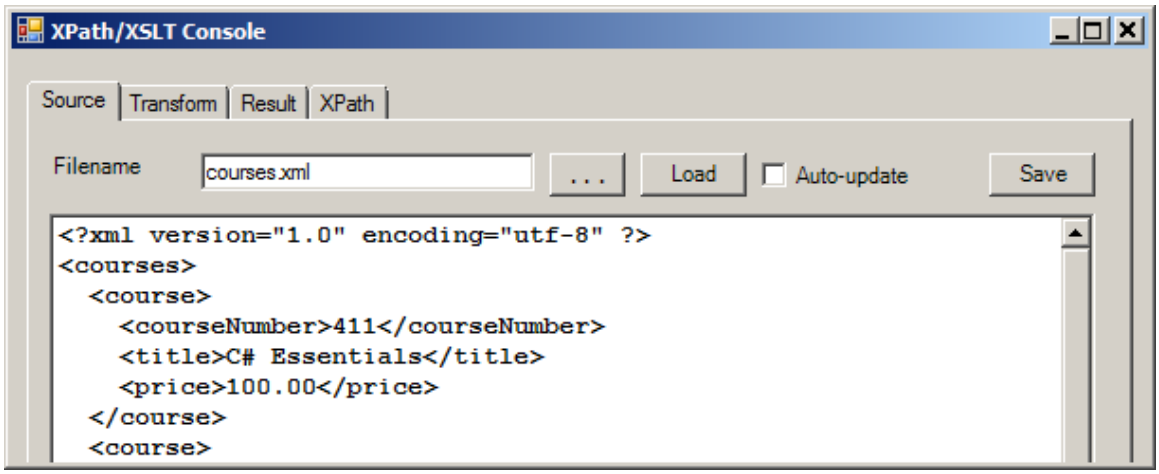
- **A template is defined using the top-level element *xsl:template*.**
- **To function as a rule, a template must define two things:**
  - What elements to use as source material—this is defined in the **match** attribute
  - What to produce based on these elements—this is the template content itself, some combination of XSLT elements, other child elements, and text
- **The *match* attribute has an XPath expression as its value.**
  - Remember that XSLT is not a procedural language, so it would be wrong to say that this XPath expression is then applied to the source document. This would imply that each template was applied in sequence, and that each got its opportunity to find relevant source elements and to produce output from them.
  - In fact, there is an algorithm in play, but it is iterating first over the document tree, and then testing each template's select expression against a single element at a time.
  - The difference may seem academic, but the result is not only a different ordering of output, but in some cases different content entirely.

XSLT Tools and Setup

- **For our work with XSLT and XPath we will continue to use the XSLT/XPath console.**
- **This is a simple tabbed GUI that can**
  - Load, edit and save source XML documents
  - Load, edit and save XSLT transform/stylesheet documents
  - Apply the XSLT transformations to the source, and show the results as raw text, HTML, or indented XML
  - Save the results to a file
- **It can also evaluate XPath expressions—we used this feature in the previous chapter.**
- **The tool is installed in *C:\OIC\Tools*.**
  - Your instructor may have chosen a different location.
  - The name of the executable is **xtcon.exe**.

Using the XSLT Console

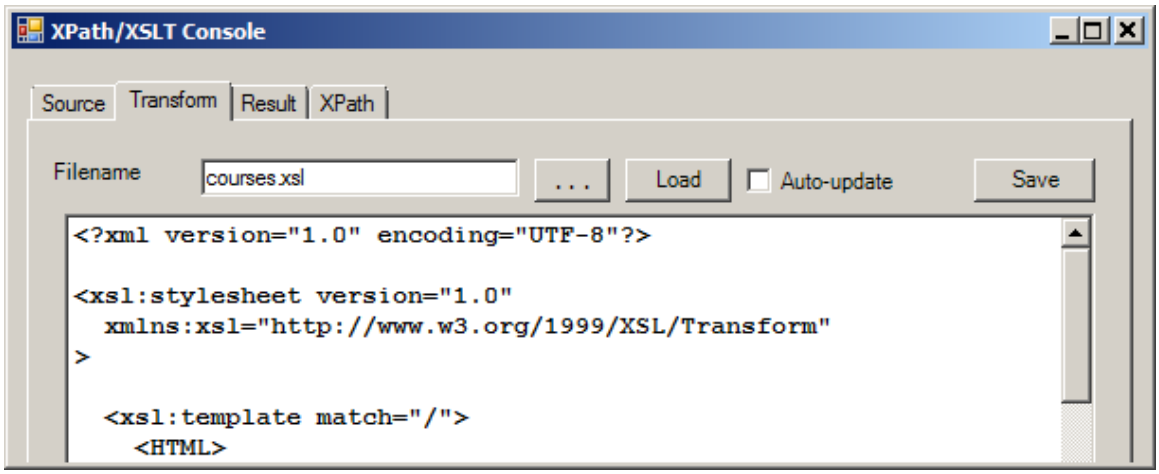
- **Start the application by double-clicking on *xtcon.exe*.**



- **The first three tabs are organized according to the usual XSLT workflow: Source, Transform, Result.**
- **In the Source tab, you can type the name or relative path of an XML file to load, or you may use the ... button to navigate with a File Open dialog box.**
  - Click the **Load** button to load the document into view.
- **You can edit the document in place, if you like.**
  - Save changes by clicking **Save**.
  - You don't need to save in order to test your changes.
- **Or, you can edit in a more full-featured text editor, and the console will pick up changes to the document.**
  - Check the **Auto-update** box.
  - Be careful not to leave this checked when editing in place!

Transform Tab

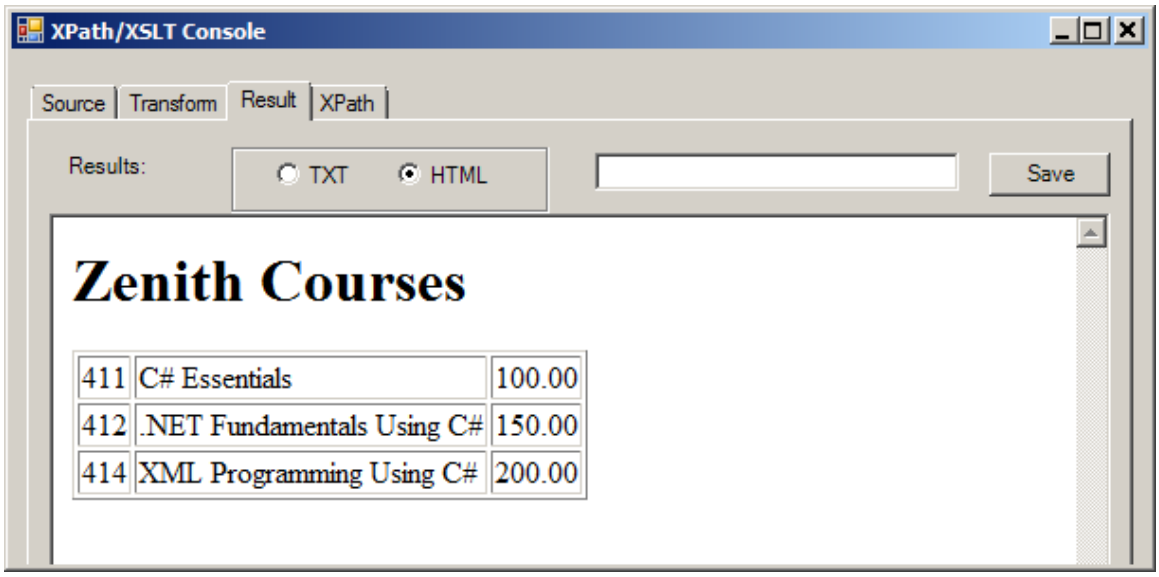
- **The Transform tab allows you to manage an XSLT transform or stylesheet.**



- **It works the same way the Source tab works.**
  - Click **Load** to load the document.
  - Edit in place, save changes, and/or auto-update to pick up changes made in an external editor.

Result Tab

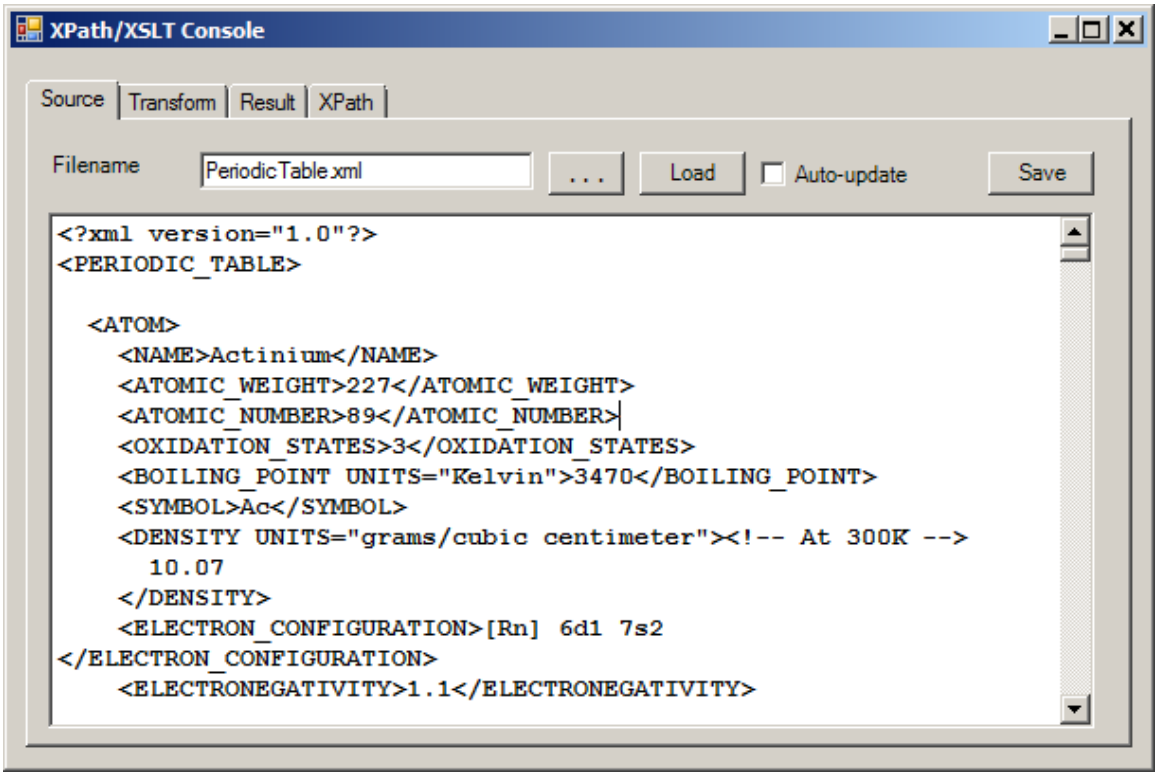
- **The Result tab shows the results of the transform.**
- **Clicking the Result tab triggers the transformation, so it is performed just in time to view it.**



- **You may use the radio buttons to view the result in either text or HTML.**
  - The HTML view shows the result as it would be seen in a browser.
- **You can save the results to a file in order to make them available to other tools, such as HTML browsers, XML parsers, or additional XSLT transformations.**

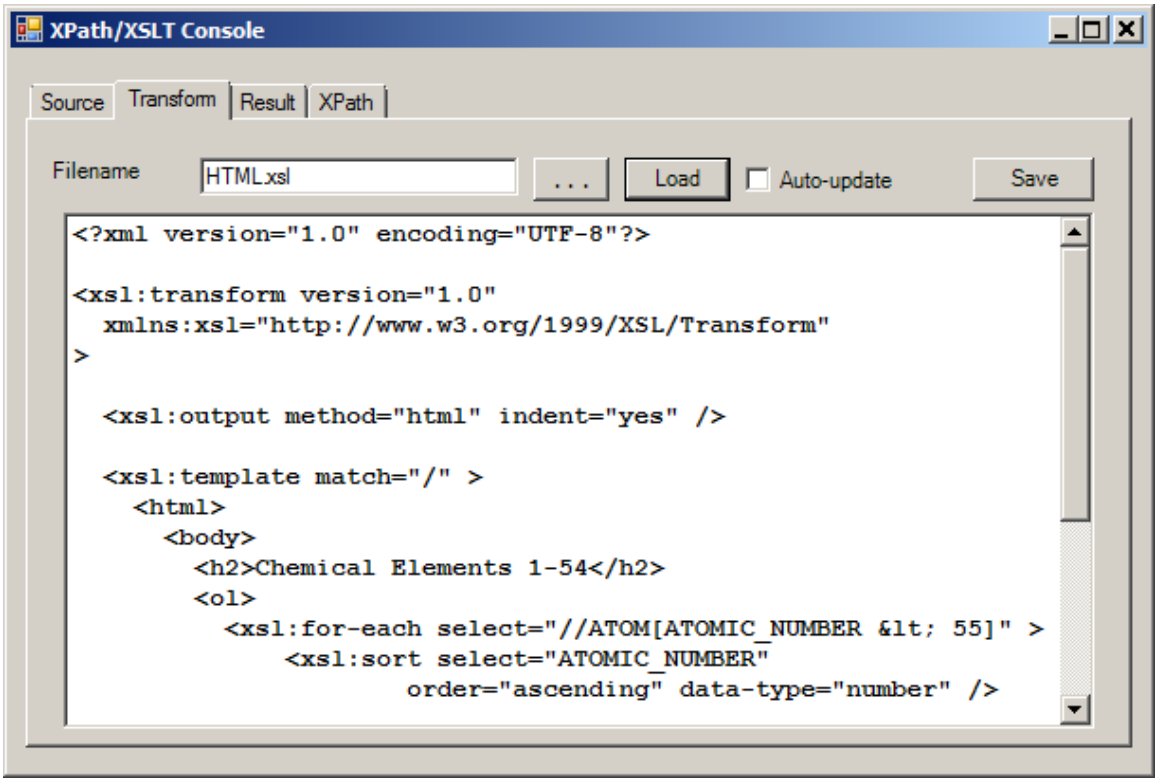
Transform Examples

- **In the *Chemistry* folder in the *Data* directory, there are a number of different transforms that operate on the document *PeriodicTable.xml*.**
  - This document expresses a great deal of information about the periodic table of chemical elements:
  - Load this document as the source. It will take a while.



HTML Transform

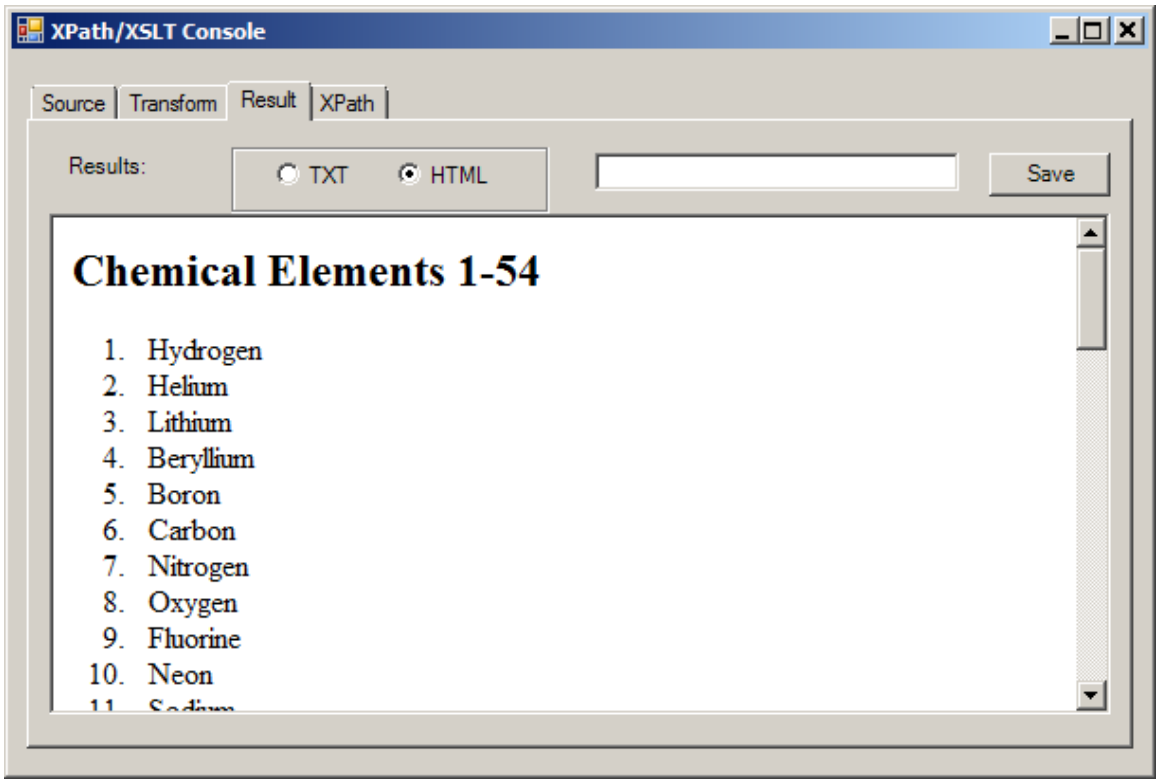
- **First we'll look at a transform that produces a bulleted list of elements.**
  - In the Transform tab, load **HTML.xsl**:



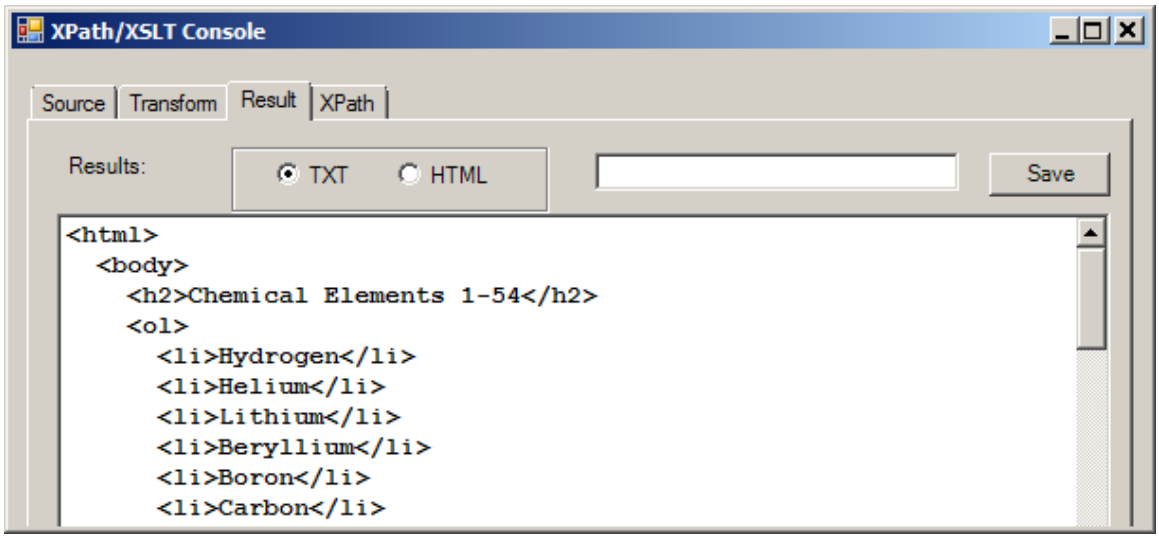
  - It runs a loop over the first 54 elements in the table.
  - It sorts by atomic number, not in document order as the text transform did.
  - For each one, it produces an item in a numbered list.

HTML Transform (Cont'd)

- **Run this transform and view it in the HTML:**



  - You can see the raw text by clicking the TXT radio button:



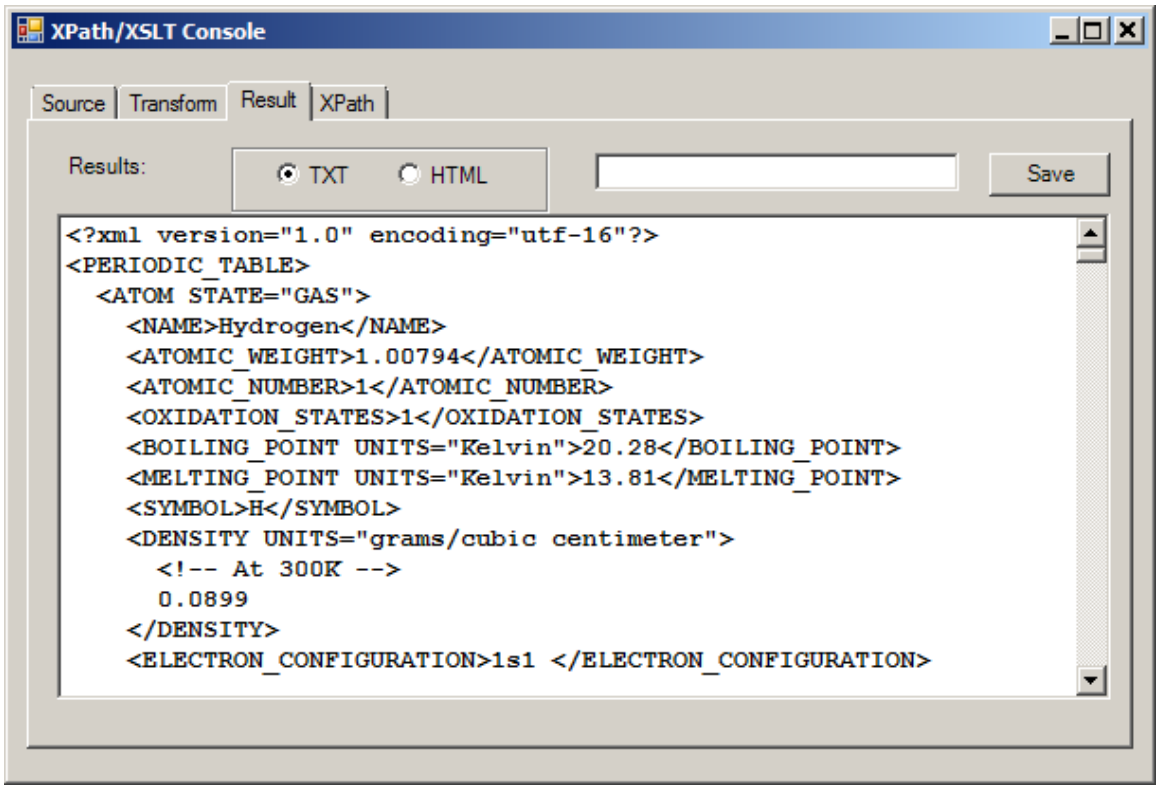
XML Transform

- **The *XML.xsl* transform filters the source document to the first 54 elements, and sorts them as the HTML one does.**
  - However this one creates a deep copy of each element, producing a smaller, but similar, XML document:

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

  <xsl:output method="xml" indent="yes" />

  <xsl:template match="/" >
<PERIODIC_TABLE>
    <xsl:apply-templates
select="//ATOM[ATOMIC_NUMBER &lt; 55]" >
      <xsl:sort select="ATOMIC_NUMBER"
order="ascending" data-type="number" />
    </xsl:apply-templates>
</PERIODIC_TABLE>
  </xsl:template>

  <xsl:template match="ATOM" >
  <xsl:copy-of select="." />
  </xsl:template>

</xsl:transform>
```

XML Transform (Cont'd)

- **Run this transform and view the result as text:**

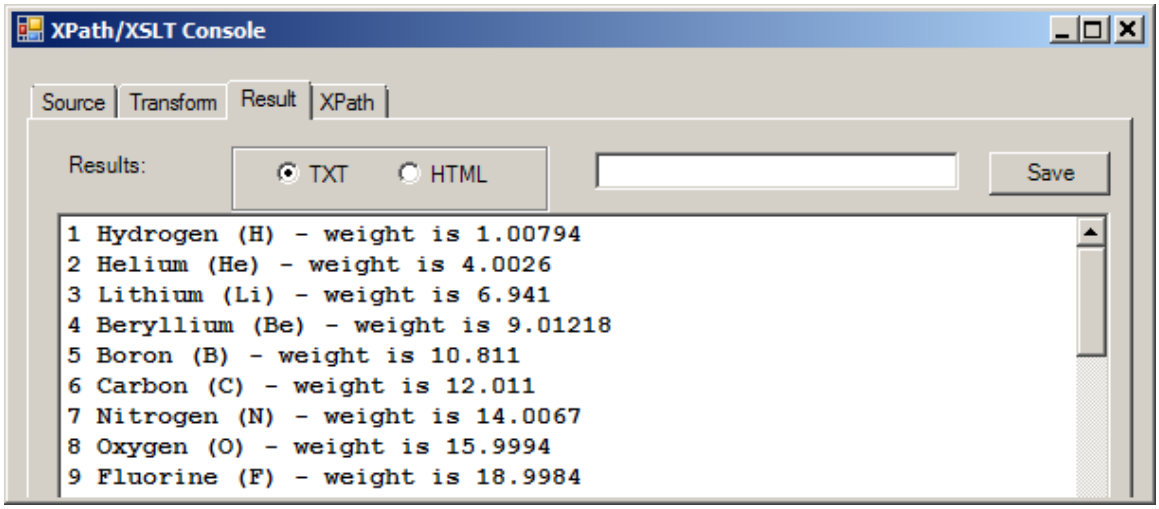


XSLT and XPath

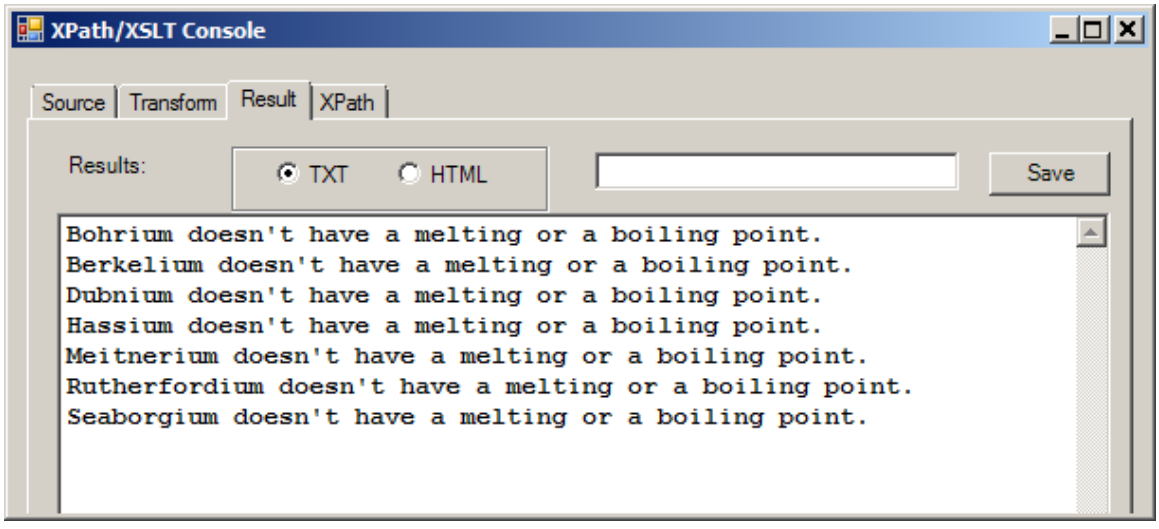
- **An in-depth study of XSLT relies heavily on XPath.**
  - XSLT relies heavily on XPath to **address** XML content in the source document.
  - XPath expressions are evaluated to decide what XSLT templates match to what source nodes.
  - Then, to produce certain values to the output stream, an XSLT template uses still more XPath.
  - So it is impossible to do anything interesting with XSLT without getting a firm grasp of XPath.
- **So you need what we learned in the previous chapter!**

Some More Examples

- **Some additional examples of transformations are provided in the *Chemistry* folder.**
  - Summary.xsl



  - MeltBoil.xsl



Style Sheets in the Browser

- **You can invoke an XSLT transformation automatically in the Web browser by means of a processing instruction in the XML file.**
  - For an example, see **Data\Courses\courses-w.xml**.

```
<?xml version="1.0" encoding="utf-8" ?>
<?xml-stylesheet type="text/xsl" href="courses.xsl"
?>
<courses>
  <course>
    <courseNumber>411</courseNumber>
    <title>C# Essentials</title>
    <price>100.00</price>
  </course>
  <course>
    <courseNumber>412</courseNumber>
    <title>.NET Fundamentals Using C#</title>
    <price>150.00</price>
  </course>
  <course>
    <courseNumber>414</courseNumber>
    <title>XML Programming Using C#</title>
    <price>200.00</price>
  </course>
</courses>
```

A Style Sheet for Browser Display

- **Here is the style sheet.**
  - See **Data\Courses\courses.xsl**.
  - Note the use of the <**xsl:stylesheet**> tag, which is a synonym for <**xsl:transform**>. In this context, one would normally use the style sheet nomenclature.

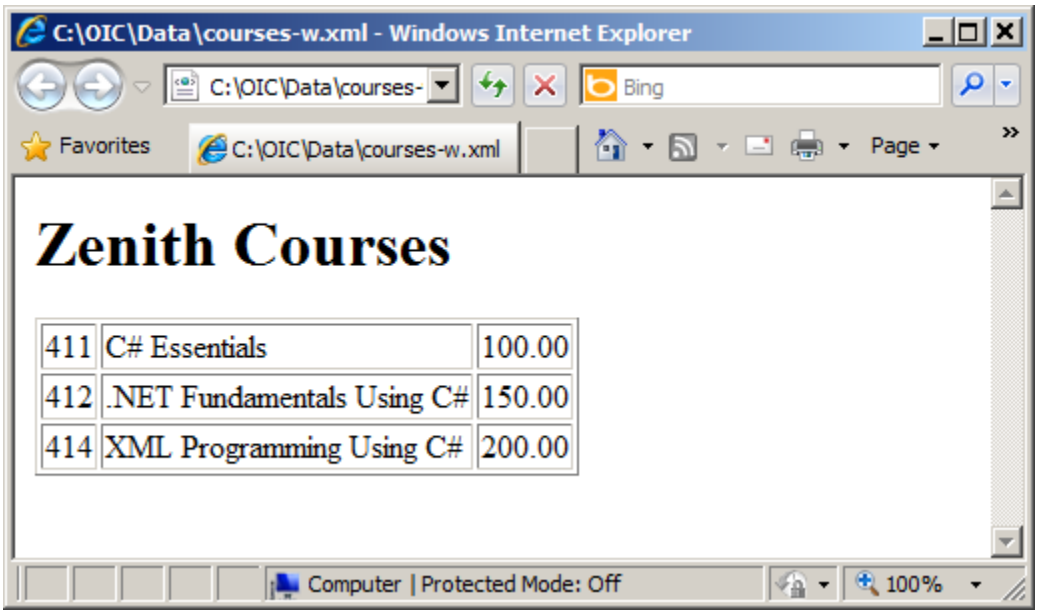
```
<?xml version="1.0" encoding="UTF-8"?>

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

  <xsl:template match="/">
    <HTML>
      <BODY>
       <H1>Zenith Courses</H1>
      <TABLE border="1">
          <xsl:apply-templates
            select="courses/course">
          </xsl:apply-templates>
      </TABLE>
      </BODY>
    </HTML>
  </xsl:template>
  ...
```

A Style Sheet for Browser Display

```
...
<xsl:template match="course">
   <TR>
   <xsl:apply-templates select="courseNumber" />
   <xsl:apply-templates select="title" />
   <xsl:apply-templates select="price" />
   </TR>
</xsl:template>

<xsl:template match="courseNumber">
 <TD >
      <xsl:value-of select="." />
 </TD>
</xsl:template>

<xsl:template match="title">
 <TD >
    <xsl:value-of select="."/>
 </TD>
</xsl:template>

<xsl:template match="price">
 <TD >
      <xsl:value-of select="."/>
 </TD>
</xsl:template>
</xsl:stylesheet>
```

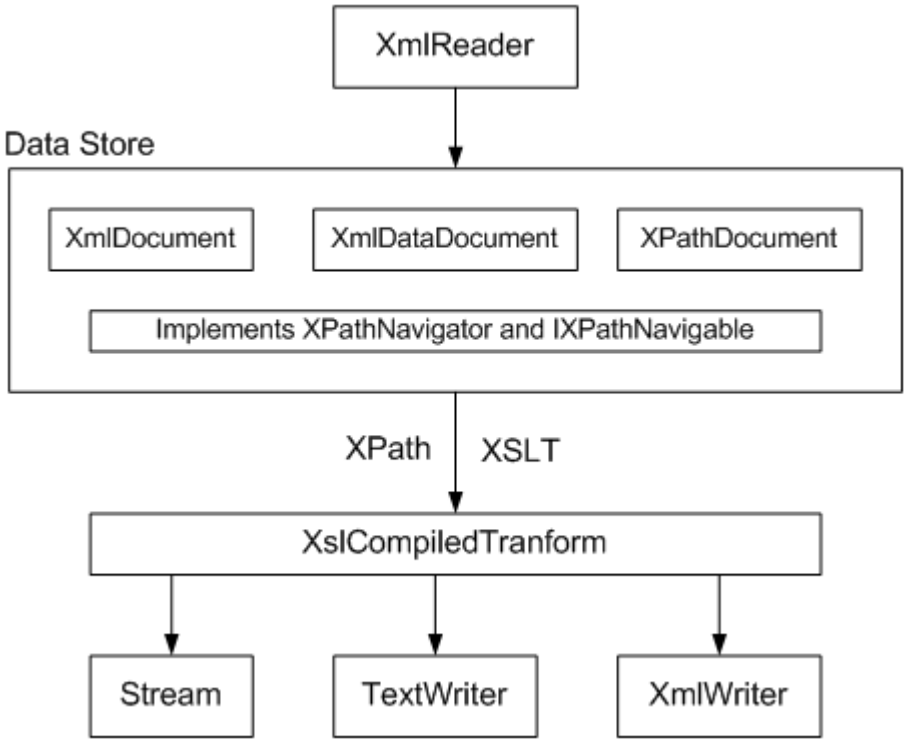
Browser Display

- **To see the result of applying the style sheet in the browser, double click on the XML file *courses-w.xml*.**



XSLT in the .NET Framework

- **The .NET Framework implements an XSLT processor in the *XslCompiledTransform* class of the *System.Xml.Xsl* namespace.**
- **The overall transformation architecture in .NET is shown in the diagram.**



- **XPath is used to select portions of an XML document during the transformation.**
- **The output of the transformation is written to a Stream, a TextWriter, or an XmlWriter.**

New XSLT Processor

- **.NET 2.0 introduced a new XSLT processor.**
- **The class *XslCompiledTransform* is new, replacing *XslTransform*, which is now obsolete.**
- **The *Transform()* method of the new class is much faster than the corresponding method of the old class.**

Sample Program

- **The program *XTran* uses a stylesheet to transform an XML document.**
  - File names are entered at the command line.

```
string source, sheet;
if (args.Length != 2)
{
   Console.WriteLine("Requires two arguments:");
   Console.WriteLine("   XSL styleheet");
   Console.WriteLine("   XML document");
   return;
}
sheet = args[0];
source = args[1];
```

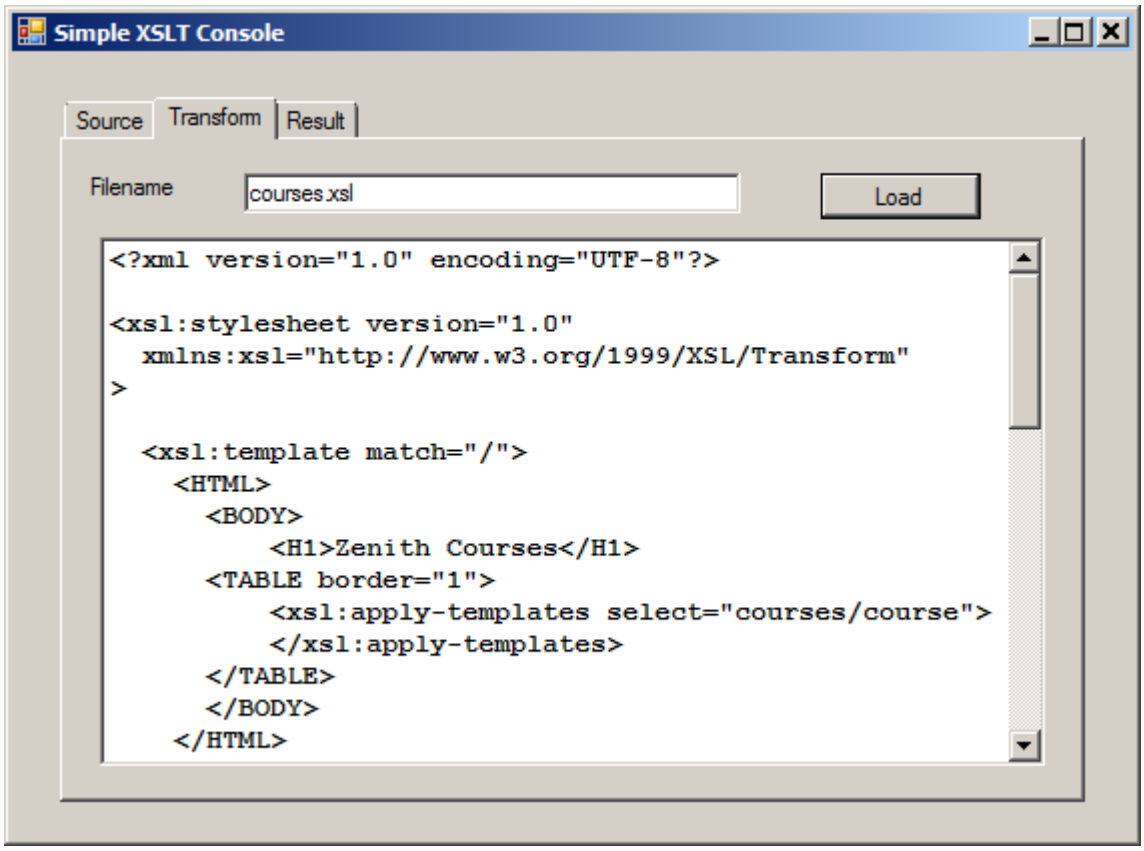
  - The transformation is then done, with the output written to the file **output.txt**.

```
try
{
   XPathDocument doc = new XPathDocument(source);
   XslCompiledTransform xt =
      new XslCompiledTransform();
   xt.Load(sheet);
   StreamWriter wr =
      new StreamWriter("output.txt");
   xt.Transform(doc, null, wr);
}
catch (Exception ex)
{
   Console.WriteLine(ex.Message);
}
```

Lab 10

A Simplified XSLT Console

In this lab you will use .NET Framework XML classes to implement a simplified version of the XSLT console tool. Like the full-blown tool we've used in this chapter, your program will have a tabbed user interface. It will simply allow you to load XML and XSL files from the current directory and perform the transform. You are provided with a starting UI.



Detailed instructions are contained in the Lab 10 write-up in the Lab Manual.

Suggested time: 30 minutes

Summary

- **XSLT was originally designed to support XSL.**
- **It was never intended to be a general-purpose transformations language, but it has become the de-facto standard nonetheless.**
- **And it is an excellent solution!**
- **It is however an unusual language, and particularly tricky for programmers of structured and object-oriented languages to learn.**
  - It is first and foremost based on matching rules.
  - Although it has procedural aspects, it is not a programming language, and it is a mistake to approach it as such.
- **The .NET Framework provides an XSLT processor in the *XslCompiledTransform* class in the *System.Xml.Xsl* namespace.**
  - The new XSLT processor in .NET 2.0 is much faster than the one in the previous .NET Framework.