

XML and the .NET Framework

The .NET (pronounced “dot net”) Framework is the foundation of Microsoft’s next generation of development tools. Given the increasing importance of XML in all areas of programming and data processing, the inclusion of powerful, integrated XML support in .NET is not surprising. This chapter provides a brief overview of .NET and then details the XML tools that are provided by the Framework and the Visual Studio .NET development environment. Code samples in this chapter are written in the C# language. This is a new language introduced with Visual Studio .NET.

.NET Overview

Microsoft’s .NET initiative has its origins in the increasing importance of the Web in almost all areas of application development. Previous development tools, exemplified by Visual Studio version 6.0, were designed for the needs of a decade ago, when the ruling paradigm was applications that were stand-alone or were distributed over a local area network (LAN). As the need for Web-related capabilities grew, ad hoc solutions were crafted as enhancements to existing tools. Because the Web capabilities were not built into the development tools from the beginning, however, there were inevitable problems with deployment, maintenance, and efficiency.

Things are different with .NET. The .NET Framework provides a comprehensive set of classes that are designed for just about any programming task you can imagine. From the very beginning, the Framework was designed to integrate Web-related programming functionality. The Framework can be used by any of Microsoft’s three programming languages: Visual Basic, C++, and C# (pronounced “C sharp”). The new releases of Visual Basic and C++ will be familiar to anyone who has used earlier versions, although there are

numerous changes to accommodate the .NET architecture. C# is new language that is similar to Java in many respects, although there are significant differences between the two. Some observers consider C# to be a Java replacement made necessary because legal problems have forced Microsoft to stop supporting Java (or Visual J++, as Microsoft's version of Java was called).

For the XML developer, .NET was designed to support XML from the ground up. There are no add-ons required, such as the MSXML Parser or the SOAP Toolkit. Everything you need is provided by the Framework. Please remember that as of this writing, the .NET Framework is a beta product. It is believed that the XML support is fairly stable, but it is possible that there will be some changes before the final product is released (which may happen by the time you read this).

The System.XML Assembly

XML support in .NET is provided by the classes in the System.XML namespace, or assembly. An assembly is a collection of related classes. In the case of System.XML, the classes are related by having to do with XML processing. The primary classes are as follows:

- `Xm1TextReader`: Provides forward-only, fast, noncached access to XML data
- `Xm1ValidatingReader`: Used in conjunction with the `Xm1TextReader` class to provide the capability for DTD, XDR, and XSD schema validation
- `Xm1Document`: Implements both level 1 and level 2 of the W3C Document Object Model specification (<http://www.w3.org/TR/DOM-Level-1/> and <http://www.w3.org/TR/DOM-Level-2/>)
- `Xm1TextWriter`: Permits generation of XML documents that conform to the W3C XML 1.0 specification
- `Xm1Navigator`: Supports evaluation of XPath expressions

Note that the Simple API for XML (SAX) is not supported in .NET. Similar functionality is provided by the `Xm1TextReader` class, although there are significant differences, which are detailed later in the chapter.

The Xm1TextReader Class

The `Xm1TextReader` class is designed for fast, resource nonintensive access to the contents of an XML file. Unlike the `Xm1Document` class, the `Xm1TextReader` class does not create a node tree of the entire document in memory. Rather, it

processes the XML as a forward-only stream. The entire XML document is never available at the same time (as is the case with the `XmlDocument` class)—your code can extract individual items from the XML file as they stream by.

In some ways the `XmlTextReader` class is similar to the SAX model covered in Chapter 11, and in fact, the .NET programmer would tend to use the `XmlTextReader` class for the same types of processing that SAX would be used for. There is a major difference between the two models, however. SAX uses a *push* model in which the XML processor uses events to inform the host program that node data is available, and the program can use the data or not as its needs dictate. The data is pushed from the XML processor to the host, and it can be accepted or ignored. As an analogy, think of a Chinese dim sum restaurant where the available food is brought around on carts and you select what you want.

In contrast, the `XmlTextReader` class uses a *pull* model. The host program requests that the XML processor read a node, and then requests data from that node as needed. The host program pulls the data from the processor as it is needed. Pull processing is analogous to a traditional restaurant where you request items from a menu. A pull model has numerous advantages over a push model for XML processing. Perhaps most important is that a push model can easily be built on top of a pull model, while the reverse is not true.

The `XmlTextReader` class operates by stepping through the nodes of an XML document, one at a time, under the control of the host program. At any given time, there is a current node. For the current node, the host program can determine the type of the node, its attributes (if any), its data, and so on. Once the needed information about the current node has been obtained, the program will step to the next node. In this manner the entire XML file can be processed.

The `XmlTextReader` class has a large number of public properties and methods. The ones you will need most often are explained in Table 13.1 and Table 13.2.

Table 13.1 Commonly Needed Properties of the `XmlTextReader` Class

Property	Description
<code>AttributeCount</code>	Returns the number of attributes of the current node
<code>Depth</code>	Returns the depth (nesting level) of the current node
<code>EOF</code>	Returns True if the XML reader is at the end of the file

(continued)

Table 13.1 (cont.)

Property	Description
HasAttributes	Returns True if the current node has attributes
HasValue	Returns True if the current node can have a value
IsEmptyElement	Returns True if the current node is an empty element (for example, <ElementName/>)
Item	Returns the value of an attribute
LocalName	Returns the name of the current node without any namespace prefix
Name	Returns the name of the current node with any namespace prefix
NodeType	Returns the type of the current node as an <code>Xm1NodeType</code> (see Table 13.3)
Value	Returns the value of the current node

Table 13.2 Commonly Needed Methods of the `Xm1TextReader` Class

Method	Description
<code>Close()</code>	Closes the XML file and reinitializes the reader.
<code>GetAttribute(Att)</code>	Gets the value of an attribute. <i>Att</i> is a number specifying the position of the attribute, with the first attribute being 0, or a string specifying the name of the attribute.
<code>IsStartElement()</code>	Returns True if the current node is a start element or an empty element.
<code>MoveToAttribute(Att)</code>	Moves to a specific attribute. <i>Att</i> is a number specifying the position of the attribute, with the first attribute being 0, or a string specifying the name of the attribute.

Table 13.2 (*cont.*)

Method	Description
MoveToElement()	Moves to the element that contains the current attribute.
MoveToFirstAttribute()	Moves to the first attribute.
MoveToNextAttribute()	Moves to the next attribute.
Read()	Reads the next node from the XML file. Returns True on success or False if there are no more nodes to read.

Table 13.3 XmlNodeType Values Returned by the NodeType Property

Value	Meaning
Attribute	An attribute
CDATA	A CDATA section
Comment	A comment
Document	The document node (root element)
DocumentType	A DOCTYPE element
Element	An element (opening tag)
EndElement	The end of an element (closing tag)
EntityReference	An entity reference
ProcessingInstruction	An XML processing instruction
Text	The text content of an element
XmlDeclaration	The XML declaration element

The basic steps required to use the `XmlTextReader` class are as follows:

1. Create an instance of the class, passing the name of the XML file to process as an argument to the class constructor.
2. Create a loop that executes the `Read()` method repeatedly until it returns `False`, which means that the end of the file has been reached.
3. In the loop, determine the type of the current node.
4. Based on the node type, either ignore the node or retrieve node data as needed.

Listing 13.1 presents an example of using the `XmlTextReader` class. It is an ASP Web page, with the script components written using the C# language. The script opens an XML file and processes it using the `XmlTextReader` class. The root element in the file, its child element, and their attributes and data are formatted as HTML and written to the output for display in the browser.

The script defines a class called `DisplayXmlFile` that does all of the work. This class contains one public method, `ReadDoc()`, that is passed the name of the XML file to be processed and returns the HTML to be displayed in the document. It also contains two private methods: `ProcessXml()`, which performs the actual processing of the XML file, and `Spaces()`, which is a utility function to provide indentation to format the output.

The script also contains a procedure named `Page_Load()`. This is an event procedure that is called automatically when the browser first loads the page. Code in this procedure creates an instance of the `DisplayXml` class, and then calls its `ReadDoc()` method, passing the name of the XML file to be processed. The HTML returned by this method is displayed by assigning it to the `InnerHTML` property of a `<div>` element in the page.

Figure 13.1 shows the output when this script is used to process `Listing1506.xml`, an XML file that is presented in Chapter 15. Though not visible in the figure, the data (attribute and element values) are displayed in blue while the remainder of the document is in black.

Listing 13.1 Using the `XmlTextReader` Class to Read Data from an XML File

```
<%@ Import Namespace="System.Xml" %>

<script language="C#" runat=server>

    public class DisplayXmlFile
        // This is the class that reads and processes the XML file.
```

```
{
    StringBuilder result = new StringBuilder();

    public string ReadDoc(String XmlFileName) {
        XmlTextReader xmlReader = null;
    try {
        // Create an instance of the XMLTextReader.
        xmlReader = new XmlTextReader(XmlFileName);
        // Process the XML file.
        ProcessXml(xmlReader);
    }
    catch (Exception e){
        result.Append("The following error occurred: " +
            e.ToString());
    }
    finally
    {
        if (xmlReader != null)
            xmlReader.Close();
    }
    return result.ToString();
}

private void ProcessXml(XmlTextReader xmlReader) {
    // This method reads the XML file and generates the output HTML.
    while (xmlReader.Read()) {
        // Process a start of element node.
        if (xmlReader.NodeType == XmlNodeType.Element) {
            result.Append(spaces(xmlReader.Depth*3));
            result.Append("<" + xmlReader.Name);
            if (xmlReader.AttributeCount > 0) {
                while (xmlReader.MoveToNextAttribute()) {
                    result.Append("&nbsp" + xmlReader.LocalName +
                        "=<font color=#0000ff>" + xmlReader.Value +
                        "</font>&nbsp");
                }
            }
            result.Append("><br>");
            // Process an end of element node.
        } else if (xmlReader.NodeType == XmlNodeType.EndElement) {
            result.Append(spaces(xmlReader.Depth*3));
            result.Append("</" + xmlReader.Name + "><br>");
            // Process a text node.
        } else if (xmlReader.NodeType == XmlNodeType.Text) {
```

```
        if (xmlReader.Value.Length != 0) {
            result.Append(spaces(xmlReader.Depth*3));
            result.Append("<font color=#0000ff>" + xmlReader.Value +
                "<br></font>");
        }
    }
}
}

private string spaces(int n) {

// Returns the specified number of non-breaking
// spaces (&nbsp;).

    string s = "";
    for (int i=0; i < n; i++) {
        s += "&nbsp;";
    }
    return s;
}
} //End DisplayXmlFile Class

private void Page_Load(Object sender, EventArgs e){

    // Create a class instance.
    DisplayXmlFile DisplayXmlFileDemo = new DisplayXmlFile();

    // Add the HTML generated by the class to the HTML document.
    show.InnerHtml =
DisplayXmlFileDemo.ReadDoc(Server.MapPath("list1506.xml"));
}

</script>
<html>
<head>
</head>
<body>
    <font size="4">Using the XmlTextReader Class</font><p>
    <div id="show" runat="server"/>
</body>
</html>
```

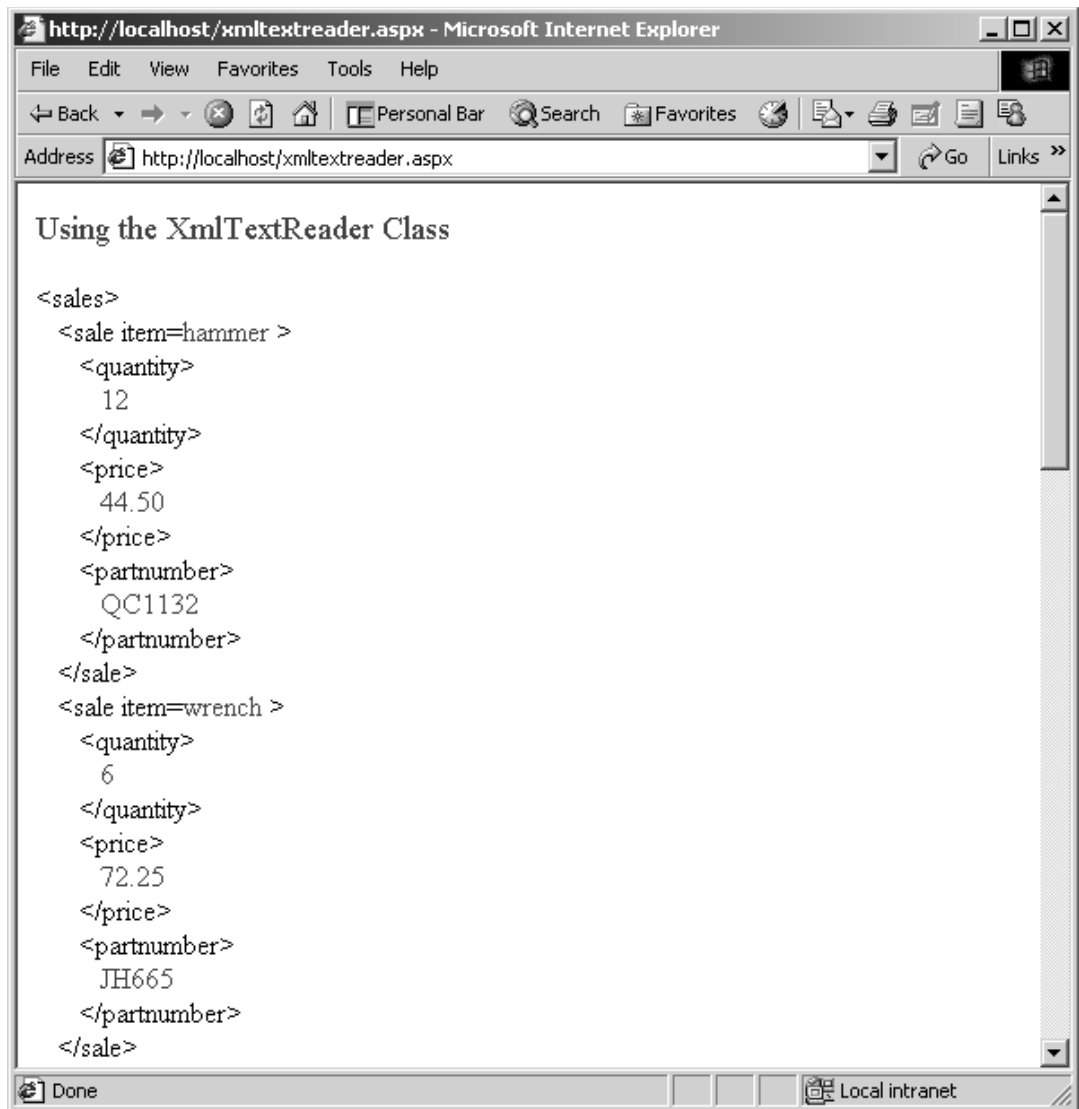



Figure 13.1 An XML file displayed in Internet Explorer by the script in Listing 13.1

The XmlValidatingReader Class

The `XmlValidatingReader` class is used to validate an XML file against a DTD or a schema (either XDR or XSD). This class is used in conjunction with the `XmlTextReader` class and provides the same access to the document contents.

Table 13.4 ValidationType Constants for the ValidationType Property

Constant	Description
Auto	Validates using information contained in the XML document (a DTD defined in a DOCTYPE element, a “schemalocation” attribute, or an inline schema). If no validation information is found, it acts as a nonvalidating parser.
DTD	Validate against a DTD.
None	Does not validate.
Schema	Validate against an XSD Schema.
XDR	Validate against an XDR Schema.

The properties and methods of these two classes are essentially identical. The differences between them lie in two properties of the `XmlValidatingReader` class that are related to validation.

The `ValidationType` property specifies the type of validation to be performed. The possible settings for this property are the `ValidationType` constants described in Table 13.4.

The `ValidationEventHandler` property is used to inform the XML reader of the event procedure you have created to handle validation errors. This event procedure takes the following form:

```
public void ValidationCallback(object sender, ValidationEventArgs args)
// Code to handle error goes here.
}
```

The name of the procedure can be anything you like. When a validation error occurs, the procedure is called by the XML reader with information about the error contained in the `args` argument. Use `args.ErrorCode` and `args.Message` to obtain a numerical code and text description of the error, respectively. You can also use the `LineNumber` and `LinePosition` properties of the `XmlTextReader` class to get information about the location of the error in the XML file.

To inform the `XmlValidatingReader` object of your event handler, use the following syntax (assuming that “`vrdr`” is an instance of the class):

```
vrdr.ValidationEventHandler +=  
    new ValidationEventHandler( NameOfEventHandler );
```

Note that you do not have to specify a handler for validation errors. If you do not, the reader will throw an exception when a validation error occurs. The advantage of using a validation error handler is that multiple validation errors can be detected and reported during a single pass over the XML file.

The general procedure for using the `XmlValidatingReader` class to validate an XML document is as follows. This assumes that the XML file contains the DTD/schema to be used for validation, either inline or as a reference.

1. Create an instance of the `XmlTextReader` class and load the XML file into it.
2. Create an instance of the `XmlValidatingReader` class and pass it a reference to the `XmlTextReader` class created in step 1.
3. Create an event handler procedure to handle validation errors. Code in this procedure can display messages to the user, set flags, or perform other actions as required by the program.
4. Set the `XmlValidatingReader` object's `ValidationType` and `ValidationEventHandler` properties.
5. Call the `XmlValidatingReader` object's `Read()` method repeatedly until the end of the XML file is reached.

The program in Listing 13.2 shows an example of how to do these tasks. This is a console, or command-line, application written in C# (a console application runs in a “DOS box”). Passed the name of an XML file as a command-line argument, the program validates the file against the DTD or schema information contained or referenced in the file. If the validation is successful, a message to that effect is displayed. If there is a validation error, or an exception is thrown, the relevant information is displayed to the user.

Listing 13.2 C# Program to Demonstrate the `XmlValidatingReader` Class

```
using System;  
using System.IO;  
using System.Xml;  
using System.Xml.Schema;  
  
public class ValidateXML  
{
```

```
private XmlTextReader rdr = null;
private XmlValidatingReader vrdr = null;
private Boolean succeeded = true;

public ValidateXML(string filename)
// This method performs the validation.
{
    try
    {
        //Create an XmlTextReader.
        rdr = new XmlTextReader( filename );
        // Create an XmlValidatingReader.
        vrdr = new XmlValidatingReader( rdr );
        // Set validation type to DTD.
        vrdr.ValidationType=ValidationType.DTD;
        // Set the validation callback method.
        vrdr.ValidationEventHandler +=
            new ValidationEventHandler( ValidationCallBack );
        // Read the XML document.
        while (vrdr.Read()) {}
        // Display success or failure message.
        if (succeeded)
            Console.WriteLine("Validation succeeded.");
        else
            Console.WriteLine("Validation failed.");
    }
    catch (Exception e)
    {
        Console.WriteLine( "Xml Exception: " + e.ToString() );
    }
    finally
    {
        if ( rdr != null )
            rdr.Close();
        if ( vrdr != null )
            vrdr.Close();
    }
}

public static void Main(string[] args)
{
    // Execution starts here.
    // The class reference.
    ValidateXML validate;
    // Ensure that 1 command line argument (the XML file name) was passed.
```

```
        if (args.Length != 1)
            Console.WriteLine("Usage: validate xmlfilename");
        else
            validate = new ValidateXML(args[0]);
    }

    public void ValidationCallBack( object sender, ValidationEventArgs args
    )
    {
        // This callback method is called when a validation error occurs.
        succeeded = false;
        // Display error information to the user.
        Console.Write( "\r\n\tValidation error: " + args.Message );
        if ( rdr.LineNumber > 0 )
            Console.WriteLine( "Line: " + rdr.LineNumber + " Position: " +
                rdr.LinePosition );
    }
}
```

The XmlTextWriter Class

The `XmlTextWriter` class provides the ability to write properly formed XML to a file or other stream. The XML created conforms to the W3C XML specification version 1.0, and also to the Namespaces in XML specification. Using this class is straightforward:

1. Create an instance of the class, passing the name of the file to be used for output and the type of encoding to use. Pass a null reference for the encoding argument to use UTF-8 encoding.
2. Set object properties as needed to control the formatting of the output.
3. Call object methods to write elements and attributes to the file.
4. Close the file.

The properties and methods of the `XmlTextWriter` class that you will use most often are described in Table 13.5 and Table 13.6.

The program in Listing 13.3 demonstrates how to use the `XmlTextWriter` class. This is a C# console application that creates a new XML file named `XmlOutput.xml`. Some data is written to the file, and then it is closed. Finally, the file is read back using the `XmlTextReader` class and then written to the console. Reading the file with `XmlTextReader` is often a good idea to verify that the file is well-formed. The results of running the program are shown in Figure 13.2.

Table 13.5 Commonly Needed Properties of the `XmlTextWriter` Class

Property	Description
Formatting	Specifies the formatting of the output. Possible settings are <code>Formatting.Indented</code> to indent child elements with respect to their parents. Set to <code>Formatting.None</code> for no indentation (the default).
Indentation	Specifies how many characters to indent by for each level in the element hierarchy when <code>Formatting</code> is set to <code>Indented</code> . The default is 2.
IndentChar	Specifies the character to use for indenting when <code>Formatting</code> is set to <code>Indented</code> . The default is the space character. Must be a valid white space character.
Namespaces	Set to <code>True</code> to enable namespace support, and set to <code>False</code> for no namespace support. The default is <code>True</code> .
QuoteChar	Specifies the character to use for quoting attribute values. Must be either the double quote (<code>&#34;</code>) or the single quote (<code>&#36;</code>). The default is the double quote.

Table 13.6 Commonly Needed Methods of the `XmlTextWriter` Class

Method	Description
<code>Close()</code>	Closes the output stream or file.
<code>Flush()</code>	Flushes the writer buffer to the output file or stream.
<code>WriteAttributeString</code> <i>(localName, value)</i>	Writes an attribute with the specified local name and value. Use the other forms of the method to include as namespace URI and a prefix.
<code>WriteAttributeString</code> <i>(localName, ns, value)</i>	
<code>WriteAttributeString</code> <i>(prefix, localName, ns, value)</i>	
<code>WriteCData(text)</code>	Writes a CDATA section containing <i>text</i> .
<code>WriteComment(text)</code>	Writes an XML comment containing <i>text</i> .

(continued)

Table 13.6 (cont.)

Method	Description
<code>WriteDocType(name, pubid, sysid, subset)</code>	Writes a DOCTYPE element. <i>Name</i> is a required argument specifying the name of the DOCTYPE. The other arguments are optional and are for writing PUBLIC “ <i>pubid</i> ,” SYSTEM “ <i>sysid</i> ,” and [<i>subset</i>] to the DOCTYPE element.
<code>WriteElementString(localName, value)</code> <code>WriteElementString(localName, ns, value)</code>	Writes an element with the specified local name and value. Use the second form of the method to include a namespace URI.
<code>WriteEndAttribute()</code>	Completes an attribute started with <code>WriteStartAttribute()</code> .
<code>WriteEndDocument()</code>	Closes any open elements or attributes.
<code>WriteEndElement()</code>	Writes the closing tag for an element started with <code>WriteStartElement()</code> . If the element is empty it will be closed with a short end tag: <code><element /></code> .
<code>WriteFullEndElement()</code>	Writes the closing tag for an element started with <code>WriteStartElement()</code> . If the element is empty it will be closed with a separate end tag: <code><element></element></code> .
<code>WriteProcessingInstruction(name, text)</code>	Writes a processing instruction with the specified name and text.
<code>WriteRaw(text)</code>	Writes raw text to the output.
<code>WriteStartAttribute(localName, ns)</code> <code>WriteStartAttribute(prefix, localName, ns)</code>	Writes the start of an attribute with the specified local name and namespace URI. Use the second form of the method to add a prefix to the local name.
<code>WriteStartDocument()</code> <code>WriteStartDocument(standalone)</code>	Writes the XML declaration with the version “1.0.” Use the second form of the method to include “ <code>standalone=yes</code> ” or “ <code>standalone=no</code> ” in the declaration.

(continued)

Table 13.6 (cont.)

Method	Description
<code>WriteStartElement(<i>localName</i>)</code>	Writes a start element with the specified local name. Use the other forms of the method to include a namespace URI and a prefix in the element.
<code>WriteStartElement(<i>localName</i>, <i>ns</i>)</code>	
<code>WriteStartElement(<i>prefix</i>, <i>localName</i>, <i>ns</i>)</code>	
<code>WriteWhitespace(<i>string</i>)</code>	Writes white space to the output. If <i>string</i> contains non-white space characters, an exception occurs.

Listing 13.3 Console Application Demonstrating the `XmlTextWriter` Class

```
using System;
using System.IO;
using System.Xml;

class XmlWriter
{
    private const string m_FileName = "XmlOutput.xml";

    static void Main()
    {
        XmlTextWriter w = null;
        XmlTextReader rdr = null;

        try
        {
            w = new XmlTextWriter(m_FileName, null);
            w.Formatting = Formatting.Indented;
            w.Indentation = 4;

            //Start the document.
            w.WriteStartDocument();

            //Write the root element.
            w.WriteStartElement( "contacts" );
```



```
// Start a "person" element.
w.WriteStartElement( "person" );

//Write a "category" attribute.
w.WriteAttributeString("category", "personal");

//Write a "name" element.
w.WriteElementString("name", "John Adams");

//Write a "phone" element.
w.WriteElementString("phone", "555-555-1212");

//Write an "email" element.
    w.WriteElementString("email", "john.adams@nowhere.net");

//Close the "person" element.
w.WriteEndElement();

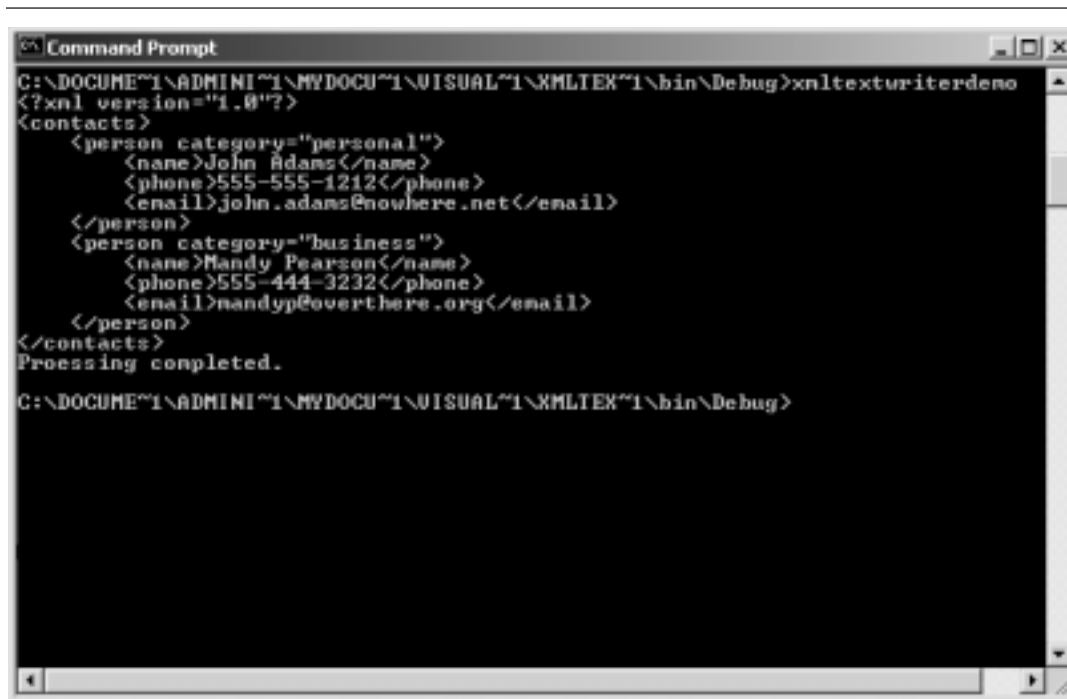
//Write another "person" element.
    w.WriteStartElement( "person" );
    w.WriteAttributeString("category", "business");
    w.WriteElementString("name", "Mandy Pearson");
    w.WriteElementString("phone", "555-444-3232");
    w.WriteElementString("email", "mandyp@overthere.org");
    w.WriteEndElement();

// Close the root element.
w.WriteEndElement();

//Flush and close.
w.Flush();
w.Close();

//Read the file back in and display it.
rdr = new XmlTextReader( m_FileName );
XmlDocument xmlDoc = new XmlDocument();
// Preserve white space for readability
xmlDoc.PreserveWhitespace = true;
xmlDoc.Load( rdr );
// Write the content to the console
Console.Write( xmlDoc.InnerXml );
}
catch (Exception e)
{
```

```
        Console.WriteLine( "Exception: ", e.ToString() );
    }
    finally
    {
        Console.WriteLine();
        Console.WriteLine( "Processing completed." );
        if ( rdr != null )
            rdr.Close();
        if ( w != null )
            w.Close();
    }
}
}
```



```
Command Prompt
C:\DOCUMENTS\ADMINISTRATOR\MYDOCUMENTS\VISUAL\XMLTEXT\bin\Debug>xmltexturiterdemo
<?xml version="1.0"?>
<contacts>
  <person category="personal">
    <name>John Adams</name>
    <phone>555-555-1212</phone>
    <email>john.adams@nowhere.net</email>
  </person>
  <person category="business">
    <name>Mandy Pearson</name>
    <phone>555-444-3232</phone>
    <email>nandyp@overthere.org</email>
  </person>
</contacts>
Processing completed.
C:\DOCUMENTS\ADMINISTRATOR\MYDOCUMENTS\VISUAL\XMLTEXT\bin\Debug>
```

Figure 13.2 Running the C# console application in Listing 13.3

The XmlDocument Class

The `XmlDocument` class provides support for the Document Object Model (DOM) levels 1 and 2, as defined by W3C. This class represents the entire XML document as an in-memory node tree, and it permits both navigation and editing of the document. The DOM implemented by the `XmlDocument` class is essentially identical to the DOM implemented by the MSXML Parser, as was covered in detail in Chapter 10. The properties and methods are the same, and rather than duplicating that information here I suggest that you turn to that chapter.

When do you use the `XmlDocument` class in preference to the `XmlTextReader` class? The criteria are similar to those for deciding between using the MSXML DOM and the Simple API for XML.

Use `XmlTextReader` when

- Memory and processing resources are a consideration, particularly for large documents.
- You are looking for specific pieces of information in the document. For example, in a library catalog, use `XmlTextReader` when you need to locate all works by a specific author.
- You do not need to modify the document structure.
- You want to only partially parse the document before handing it off to another application.

Use `XmlDocument` when

- You need random access to all of document's contents.
- You need to modify the document structure.
- You need complex XPath filtering.
- You need to perform XSLT transformations.

There are various ways to use the `XmlDocument` class. You can use it alone, applying the class methods and properties to “walk the tree” and make changes. You can also use the `XmlDocument` class in conjunction with the `XPathNavigator` class, which provides more sophisticated navigational and editing capabilities as well as XPath support. The following sections look at both. The first section presents a C# demonstration of using the `XmlDocument` class to modify the contents of an XML file. The second section explains how to use the `XPathNavigator` class.

Using the XmlDocument Class to Modify an XML Document

The first demonstration of the `XmlDocument` class shows how to use it to modify the contents of an XML document. In this case the task is to add a new `<person>` element to the XML file shown in Listing 13.4 and save the modified file under the name `OutputFile.xml`. The program, shown in Listing 13.5, is a C# console application, and the code is fully commented so you can figure out how it works.

Listing 13.4 InputFile.xml Is the File to Be Modified

```
<?xml version="1.0"?>
<contacts>
  <person category="personal">
    <name>John Adams</name>
    <phone>555-555-1212</phone>
    <email>john.adams@nowhere.net</email>
  </person>
  <person category="business">
    <name>Mandy Pearson</name>
    <phone>555-444-3232</phone>
    <email>mandyp@overthere.org</email>
  </person>
  <person category="family">
    <name>Jack Sprat</name>
    <phone>000-111-2222</phone>
    <email>jack001@earth.net</email>
  </person>
</contacts>
```

Listing 13.5 C# Program to Modify the Contents of InputFile.xml

```
using System;
using System.IO;
using System.Xml;

class Class1
{
  private const string m_InFileName = "InputFile.xml";
  private const string m_OutFileName = "OutputFile.xml";

  static void Main()
  {
```

```
bool ok = true;
XmlDocument xmlDoc = new XmlDocument();

try
{
    //Load the input file.
    xmlDoc.Load( m_InFileName );
    //Create a new "person" element.
    XmlElement elPerson = xmlDoc.CreateElement( "person" );
    //Add the "category" attribute.
    elPerson.SetAttribute( "category", "family" );
    //Create "name," "phone," and "email" elements.
    XmlElement elName = xmlDoc.CreateElement( "name",
        "Ann Winslow" );
    XmlElement elPhone = xmlDoc.CreateElement( "phone",
        "000-000-0000" );
    XmlElement elEmail = xmlDoc.CreateElement( "email",
        "anne123@there.net" );
    //Add them as children of the "person" element.
    elPerson.AppendChild( elName );
    elPerson.AppendChild( elPhone );
    elPerson.AppendChild( elEmail );
    //Get a reference to the document's root element.
    XmlElement elRoot = xmlDoc.DocumentElement;
    //Add the "person" element as a child of the root.
    elRoot.AppendChild( elPerson );
    //Save the document.
    xmlDoc.Save( m_OutFileName );
}
catch ( Exception e )
{
    ok = false;
    Console.WriteLine( "Exception: " + e.Message );
}
finally
{
    if (ok)
        Console.WriteLine( "Element added successfully." );
    else
        Console.WriteLine( "An error occurred." );
}
}
```

Using XPathNavigator with XmlDocument

The `XPathNavigator` class is designed specifically to facilitate navigating through XML that is contained in an `XmlDocument` object. It provides a cursor model, meaning that the navigator almost always has a position within the document's node tree. Many of the actions you can take with the navigator are performed relative to the current position, such as "move to the next node." When an action is performed successfully, the cursor is left pointing at the location where the action occurred. When an action fails, the cursor remains at its original position. You can always use the `MoveToRoot()` method to move the cursor to the document's root node.

Much of the power of the `XPathNavigator` class comes from its support for XPath expressions. You can select all of the nodes that match an XPath expression, and then conveniently work with them. However, many of the uses of this class do not in fact involve XPath expressions and hence its name is a bit misleading. These are the steps required to work with the `XPathNavigator` class if you are not going to use XPath expressions:

1. Create an instance of the `XmlDocument` class.
2. Load the XML document into the `XmlDocument` object.
3. Call the `XmlDocument` object's `CreateNavigator()` method to create an instance of the `XPathNavigator` class and return a reference to it.
4. Use the `XPathNavigator` object's properties and methods to move around the document and access its content.

The following code fragment shows how the preceding steps would be done in C#:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load( "original.xml" );
XPathNavigator nav = xmlDoc.CreateNavigator();
// Work with navigator here.
```

If you want to use XPath expressions, you call the `XPathNavigator` object's `Select()` method, which returns a type `XPathNodeIterator` that contains the nodes matching the XPath expression. This is explained later in this chapter.

When the `XPathNavigator` is first created, it is by default positioned on the document's root node. Even so, many programmers call the `MoveToRoot()` method to ensure that they know where they are starting. Then a call to `MoveToFirstChild()` moves to the first element in the file, typically the `<?xml version="1.0"?>` node. At this point, a typical approach is to call `MoveToNext()`

repeatedly until you reach the document's root element (the <contacts> element in Listing 13.4). Then you can use the various methods to move around the document as needed. You'll see this in the first demonstration program later in this chapter.

Note that there is some potential confusion regarding the use of the term "root" because the root node as seen by `XPathNavigator` is not the same as the document's root element. The root node encompasses the entire XML document, and the root element is a child node of this root node.

The `XPathNavigator` class has a large number of properties and methods, and many of them are infrequently needed. Rather than presenting all of them here, I have limited coverage to those properties and methods that you most often need. (You can refer to the .NET online documentation for information on the others.) Table 13.7 lists these properties and methods of the `XPathNavigator` class. Following the tables, I present two sample programs that use the `XPathNavigator` class.

Demonstrating XPathNavigator

This first demonstration shows how to use the `XPathNavigator` class to "walk" the tree of an XML document. The demonstration makes use of the XML data

Table 13.7 Commonly Used Properties and Methods of the `XPathNavigator` Class

Property/Method	Description
<code>GetAttribute (name, ns)</code>	Returns the value of the attribute with the specified name and namespace URI. Returns null if a matching attribute is not found.
<code>HasAttributes</code>	Returns True if the current node has attributes. Returns False if the current node has no attributes or is not an element node.
<code>HasChildren</code>	Returns True if the current node has child nodes.
<code>IsEmptyElement</code>	Returns True if the current node is an empty element (such as <element/>).
<code>LocalName</code>	Gets the name of the current node without its namespace prefix.

(continued)

Table 13.7 (cont.)

Property/Method	Description
Matches (<i>XPathExpr</i>)	Returns True if the current node matches the specified XPath expression. The argument can be a string or a type <code>XPathExpression</code> .
MoveTo()	Moves to the first sibling of the current node. Returns True if there is a first sibling node or False if not or if the current node is an attribute node.
MoveToAttribute (<i>name, ns</i>)	Moves to the attribute with the matching local name and namespace URI. Returns True if a matching attribute is found or False if not.
MoveToFirstChild()	Moves to the first child of the current node. Returns True on success or False if there is no child node.
MoveToID(<i>id</i>)	Moves to the node that has a type ID attribute with the specified value. Returns True on success or False if there is no matching node.
MoveToNext()	Moves to the next sibling of the current node. Returns True on success or False if there are no more siblings or if the current node is an attribute node.
MoveToNextAttribute()	Moves to the next attribute node. Returns True on success or False if there are no more attribute nodes or if the current node is not an attribute node.
MoveToParent()	Moves to the current node's parent. Returns True on success or False if the current node has no parent (is the root node).
MoveToPrevious()	Moves to the previous sibling node. Returns True on success or False if there is no previous sibling or if the current node is an attribute node.
MoveToRoot()	Moves to the root node. This method is always successful and has no return value.
Name	Returns the name of the current node with namespace prefix (if any).

(continued)

Table 13.7 (cont.)

Property/Method	Description
NodeType	Returns an XPathNodeType value identifying the type of the current node. See Table 13.8 for possible values.
Select(<i>match</i>)	Selects a node set that matches the specified XPath expression and returns a type XPathNodeIterator. The argument can be a string or a type XPathExpression.
Value	Returns the text value of the current node—for example, the value of an attribute node or the text in an element node.

Table 13.8 Members of the XPathNodeType Enumeration Returned by the XPathNavigator Class's NodeType Property

Constant	Description
All	All node types
Attribute	Attribute node
Comment	Comment node
Element	Element node
Namespace	A namespace node (for example, xmlns="xxx")
ProcessingInstruction	A processing instruction (not including the XML declaration)
Root	Root node
SignificantWhitespace	A node that contains white space and has xml:space set to "preserve"
Text	A text node (the text content of an element or attribute)
Whitespace	A node that contains only white space characters

file presented later in the book in Listing 18.5. This file contains a database of books and is structured as shown in this fragment:

```
<books>
<book category="reference">
  <title>The Cambridge Biographical Encyclopedia</title>
  <author>David Crystal</author>
  <publisher>Cambridge University Press</publisher>
</book>
...
</books>
```

The objective of the demonstration is to let the user select a category of books, and then display a list of all matching books. It is created as a Web application. The user selects the category on an HTML page, as shown in Listing 13.6. This page presents a list of categories from which the user selected. The request is sent to the ASP.NET application in Listing 13.7. The code in this page uses an `XPathNavigator` object to move through the XML file. Specifically, the code locates each `<book>` element and checks its “category” attribute. If the value of this attribute matches the category requested by the user, the program walks through the `<book>` element’s children (the title, author, and publisher elements), extracts their data, and outputs it in the form of an HTML table. The results of a search are shown in Figure 13.3.

Listing 13.6 The HTML Page That Lets the User Select a Book Category

```
<html>
<head>
<title>Book search</title>
</head>
<body>
<h2>Find books by category.</h2>
<hr/>
<form method="GET" action="list1307.aspx">
<p>Select your category, then press Submit.</p>
<p>Category:
<select name="category" size="1">
  <option value="biography">Biography</option>
  <option value="fiction">Fiction</option>
```

```
<option value="reference">Reference</option>
<select>
</p>
<p><input type="submit" value="Submit"/>
</p><hr/>
</form>
</body>
</html>
```

Listing 13.7 ASP.NET Script That Uses the XPathNavigator Class to Access XML Data

```
<%@ Import Namespace="System.Xml" %>
<%@ Import Namespace="System.Xml.XPath" %>
<script language="C#" runat="Server">
void Page_Load(object sender, EventArgs e) {
    try {
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.Load(Server.MapPath("list1805.xml"));
        XPathNavigator nav = xmlDoc.CreateNavigator();
        //Get the query string submitted by the client.
        NameValueCollection coll = Request.QueryString;
        string category = coll.Get( "category" );
        //Move to the document's root element.
        nav.MoveToRoot();
        //Move to the first child.
        nav.MoveToFirstChild();
        while (nav.LocalName != "books")
            nav.MoveNext();
        //At this point we are positioned at the root element.
        //Move to the first child (the first <book> element).
        nav.MoveToFirstChild();
        //Start writing the HTML to the output.
        Response.Write( "<html><body>" );
        Response.Write("<h2>Books in the '" + category +
            "' category:</h2><hr/>");
        //Write out the table headings.
        Response.Write(" <table cellpadding='4'>" );
        Response.Write( "<thead><tr>" );
        Response.Write( "<th>Title</th><th>Author</th>" );
        Response.Write( "<th>Publisher</th></tr></thead>" );
        Response.Write( "<tbody>" );
```

```
bool more = true;
while (more)
{
    //Is this book in the selected category?
    if (nav.GetAttribute( "category", "" ) == category)
    {
        //Move to the first child (<title>) and write its data.
        nav.MoveToFirstChild();
        Response.Write( "<tr><td>" + nav.Value + "</td>");
        //Move to next (<author>).
        nav.MoveNext();
        Response.Write( "<td>" + nav.Value + "</td>");
        //Move to next (<publisher>).
        nav.MoveNext();
        Response.Write( "<td>" + nav.Value + "</td></tr>");
        //Move back to the parent <book> node.
        nav.MoveToParent();
    }
    //Move to the next <book> node, if any.
    more = nav.MoveNext();
}
//Finish the table.
Response.Write( "</tbody></table><hr/></body></html>");
}
catch(Exception ex) {
    Response.Write(ex.ToString());
}
}
</script>
```

Using the Select() Method and the XPathNodeIterator Class

The `XPathNavigator` class has the `Select()` method, which permits you to select a node set that matches an XPath expression. The method returns an object of type `XPathNodeIterator` that contains the matching nodes. If there are no matching nodes, the `XPathNodeIterator` object's `Count` property will be 0; otherwise, this property returns the number of nodes. For example, this code assumes that the variable `selectExpr` contains the XPath expression that you want to use:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load( "InputFile.xml" );
```

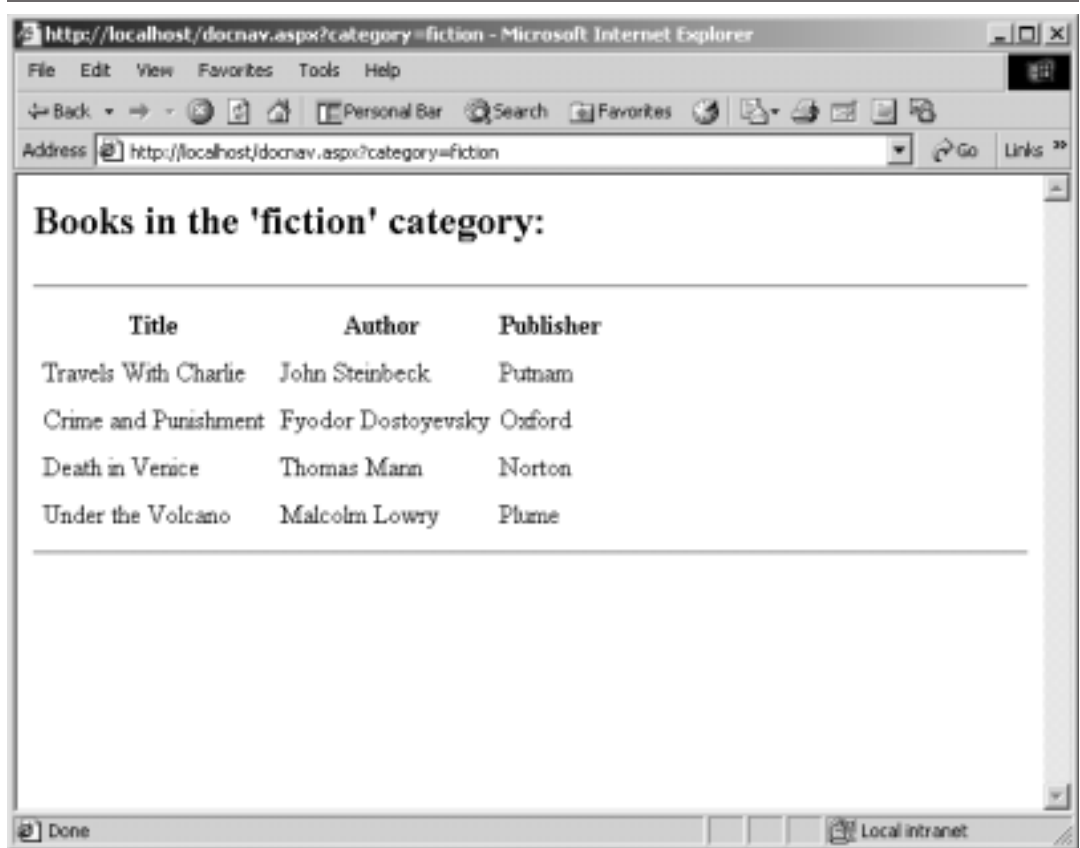


Figure 13.3 The results of a book query displayed by Listing 13.7

```
XPathNavigator nav = xmlDoc.CreateNavigator();
XPathNodeIterator xpi = nav.Select( selectExpr );
if ( xpi.Count != 0 )
{
    // At least one matching node was found.
}
else
{
    // No matching nodes were found.
}
```

When to Use XPathNodeIterator

There's not much that really requires the use of the XPathNodeIterator class, but it does make certain tasks more efficient. You can always locate the node(s) that you want by using the XPathNavigator class's methods to move around the document tree and examine nodes as you go. However, the ability to quickly select a subset of nodes based on an XPath expression can make this sort of brute force technique unnecessary.

Table 13.9 describes members of the XPathNodeIterator class. You will note that the Current property returns a reference to an XPathNavigator object that is positioned on the current node. However, you cannot use this XPathNavigator object to move away from the current node (unless you first clone it)—you can use it only to get information about the current node.

To demonstrate using the Select() method and the XPathNodeIterator class, I turn again to the XML file Inputfile.xml from Listing 13.4. The goal of this application is to list the names of all the people in the XML database. In other words, the application needs to go through the XML file, select all <name> nodes, and display their values. This could be done using the “brute force” method of going through all the nodes in the document, but the code is a lot simpler if you use the Select() method. This is a console application that opens the file and displays the names on the screen. Listing 13.8 presents the source code.

Table 13.9 Members of the XPathNodeIterator Class

Member	Description
Count	Returns the index of the last selected node or 0 if there are no nodes.
Current	Returns a type XPathNavigator positioned on the current node.
CurrentPosition	The 1-based index of the current node.
MoveNext()	Moves to the next selected node. Returns True on success or False if there are no more selected nodes.

Listing 13.8: Using XPathNavigator and XPathNodeIterator to Access XML Data

```
using System;
using System.IO;
using System.Xml;
using System.Xml.XPath;

namespace XPathNavDemo
{
    class SearchXML
    {
        static void Main(string[] args)
        {
            SearchXML ex = new SearchXML();
        }

        public SearchXML()
        {
            try
            {
                XmlDocument xmlDoc = new XmlDocument();
                xmlDoc.Load( "InputFile.xml" );
                XPathNavigator nav = xmlDoc.CreateNavigator();
                // Select all the <name> nodes.
                string select = "descendant::person/name";
                XPathNodeIterator xpi = nav.Select(select);
                if ( xpi.Count != 0 )
                {
                    // At least one <name> node was found.
                    // Move through them and display the values.
                    Console.WriteLine("The following people are in this file:");
                    while (xpi.MoveNext())
                        Console.WriteLine(xpi.Current.Value);
                }
                else
                    Console.WriteLine("No <name> elements found.");
            }
            catch ( System.Exception ex )
            {
                Console.WriteLine("Exception: " + ex.Message );
            }
            finally
            {

```

```
        Console.ReadLine();  
    }  
}  
}
```

Summary

The .NET Framework provides a rich and comprehensive set of classes that support almost every imaginable aspect of desktop and Internet-enabled computing. Coupled with the Visual Studio .NET programming environment, .NET represents Microsoft's response to the new challenges facing developers as the Internet assumes greater importance in all aspects of computing. .NET includes excellent support for XML development, with several classes for reading, writing, modifying, and navigating XML documents.