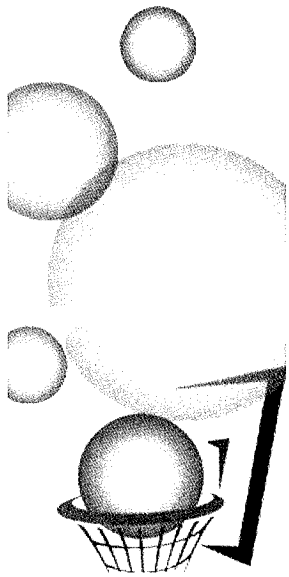
Get to know VB .NET's XML objects for easy parsing

by Matthew MacDonald

Application: Microsoft Visual Basic .NET

Download: http://download.elementkjournals.com/ivb/200306.zip



As you know, XML offers a simple idea: a universal, text-based format for organizing structured or semi-structured data. Using XML makes it easy to send data from an application on one platform to an application running on another (most modern programming languages include some type of XML parser). XML also makes it easy for different organizations to share data and integrate business processes. XML even simplifies life when creating in-house applications that need to be standardized so they can be enhanced and modified long after the original coders have left the company. With everything it has going for it, it's no wonder that Microsoft has integrated XML so effectively into .NET. In this article, we'll briefly discuss the essentials of XML.

XML meets .NET

We think you'll find XML an integral and easy tool in .NET. To help you become acquainted with using this language, we'll introduce you to some of the classes found in the `System.Xml` namespace that read and write XML in .NET. Finally, we'll create an example .NET project that will use the most common of these objects, `XMLDocument`, to read an XML document and output its values to the Debug window.

An encapsulated intro to XML

XML is an all-purpose format for encoding data according to a few simple rules. By using XML, you store data in a standardized way, rather than using a proprietary format. The actual location of this data—in memory, in a file, in a network stream, or in a database—becomes irrelevant.

XML provides a universal way to identify data using tags. These tags use the same sort of format found in an HTML file, but while HTML tags indicate formatting, XML tags indicate content. (Because an XML file is just about data, there's no standardized way to display it in a browser.) You can see in **Listing A** an example of an order list in an XML format.

As you can see, every product item is enclosed in an `<Item>` tag, and every piece of information has its own tag with an appropriate name. Tags are nested several layers deep to show relationships. Essentially, XML provides the basic tag syntax, and you (the programmer) define the tags you want to use.

Listing A: *A basic XML document*

```
<?xml version="1.0"?>
<Order id="2003-04-12-4996">
   <Client>
      <Name>Fred Murphy</Name>
   </Client>
   <Item id="1">
      <Name>Calculator</Name>
      <Price>24.99</Price>
   </Item>
   <Item id="2">
      <Name>Laser Printer</Name>
      <Price>400.75</Price>
   </Item>
</Order>
```

Some tips of the tag

When creating your own XML document, keep in mind a few important rules:

- XML automatically collapses whitespace, so you can use tabs and hard returns to format the elements in the document. To add a real space, you'll need to use the   entity equivalent, as in HTML.
- XML only accepts valid characters. Special characters, like greater than and less than symbols (> <) and the ampersand (&), aren't allowed. Instead, you'll have to use the entity equivalents (like < and > for angle brackets, and & for the ampersand). These equivalents are the same as in HTML coding, and will be automatically converted back to the original characters when you read them with the appropriate .NET classes.
- Every start tag must have an end tag, or you must use the special "empty tag" format, which includes a forward slash at the end. For example, you can

use <Name>Content</Name> or <Name />, but you can't use <Name> on its own.

- All tags must be nested in a root tag. In the product list, the root tag is <Order>. As soon as the root tag is closed, the document is finished, and you can't add any more content.
- Every element must be fully enclosed. In other words, when you start a subordinate tag, you must close it before you can close the parent. So, <Product><ID></ID></Product> is valid, but <Product><ID></Product></ID> isn't.

Well-formed and valid

If you meet these requirements, an XML parser can parse and display the XML as a basic tree. This means that your document is well-formed, but it doesn't mean that it's valid. For example, you may have elements in the wrong order (for example, <ID><Product></Product></ID>), or you may have the wrong type of data in a given field (for example, <ID>Chair</ID><Name>2</Name>). These elements would be considered well-formed, but they wouldn't be valid representations of your data structure. There are ways to impose these additional rules on XML documents through DTDs or Schemas, but that's really outside the scope of this article.

Case matters

When you create tags in an XML document, remember that *case matters*. As a case-sensitive language, XML treats tag names with different cases as entirely different elements. So, <Item> wouldn't be the same element as <item>. The same goes for attribute names inside XML elements. An HREF attribute wouldn't be viewed the same as an href attribute.

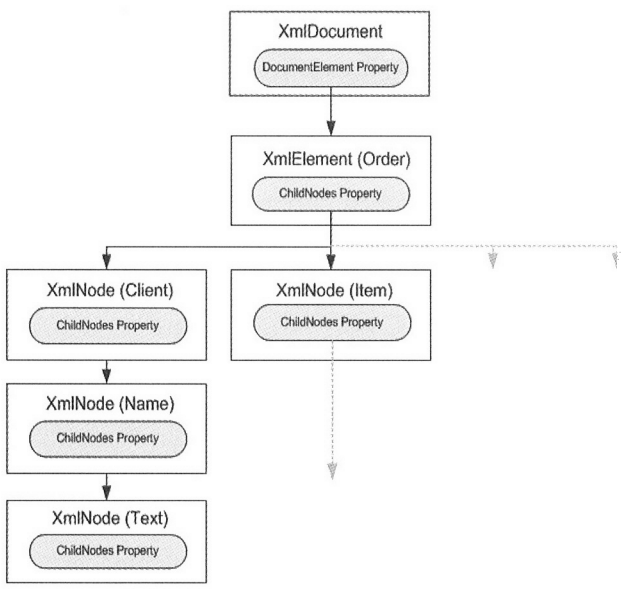


Figure A: *The XmlDocument object creates tree hierarchy objects from an XML file.*

XML in .NET

When you use an XML document in VB .NET, you can take advantage of the available XML parsers to make your life easier. With these tools, your code won't need to worry about detecting where a tag starts and stops, collapsing whitespace, or identifying attributes. Instead, you can just read the file into some helpful XML data objects that make navigating through the entire document much easier.

.NET provides a rich set of classes for XML manipulation in the System.Xml namespace, including:

- XmlTextReader, which allows you to read XML data from a forward-only, read-only stream.
- XmlTextWriter, which allows you to write XML data to a file in a forward-only, write-only stream.
- XmlDocument, which allows you to load a tree representing the entire XML document into memory, edit it, and then save it back to disk at a later point.
- XmlDataDocument, which works like XmlDocument, but also allows you to access XML data through the same ADO.NET objects you use to manipulate relational data drawn from a database.

You can perform equivalent operations with any of these classes. In the remainder of this article, we'll hone in on just one object, the XmlDocument, which is often the most flexible option when dealing with XML data that isn't excessively large.

Processing XML with XmlDocument

The XmlDocument class includes two key methods. Load() retrieves the XML data and creates the in-memory XML tree. Save() writes the current contents of the XML tree back to a file.

Once VB .NET loads the XML data into memory, your code can access the elements in the tree-like structure. VB.NET represents each element in the document hierarchy as a node (like those in the TreeView control). As in the text file, the topmost element of the XML document object is the root node. The root node (represented by an XmlNode object) can contain any number of additional nodes. These nodes can, in turn, contain even more nodes, and so on. To drill down from one layer to the next, you examine the collection of nodes provided in the XmlNode.ChildNodes property.

For example, consider the XML order list we provided earlier. **Figure A** shows a diagram of how

this list maps to some of the `XmlNode` objects in an `XmlDocument`.

Creating an example app

Before we continue, let's create a sample .NET project that we can work with. As a first step, launch Visual Studio .NET and select New Project from the Start page. In the New Project dialog box, select the Visual Basic Projects folder (if it isn't already selected), and then select the Windows Application item. Next, in the Name field, enter *XMLNET* as the project's name. Use the Location field to store the application wherever you wish. Click OK to create the new project.

At this point, we'll need an XML document to manipulate. To create this file, open your favorite text editor and enter the XML we provided earlier in **Listing A**. Save the file as *orders.xml* in the same bin directory folder where you stored your VB .NET project.

An introductory approach

To get used to working with the `XmlDocument`, let's create a recursive procedure that loops through all the nodes in a given XML document and displays them in the Debug window. To do so, right-click on our sample application's default form and select View Code from the resulting shortcut menu. When the IDE displays the code window, first add a reference to the `System.XML` namespace above the form's class declaration, as in:

```
Imports System.XML
```

Next, enter the code from **Listing B** anywhere within the form's class block. This procedure will loop through the document's nodes and output the results to the Debug window.

Now, all we need to do is execute this procedure after the form loads. To accommodate this task, from the Method dropdown menu, select the form's `New()` method. As you can see, in addition to the pre-generated code, the IDE indicates where to place code that will run when it initializes the form. We'll call the `DisplayNode()` procedure in this section. With this in mind, modify the `New()` method so that it appears as shown in **Listing C**.

To view the code in action, select Build | Build Solution from the main menu (or press [Ctrl][Shift]B) to build the EXE file. Then, press [F5] to run the project. When you do, VB .NET creates a new version of the form, triggering the form's `New()` method. After initializing the form, the application creates a new `XmlDocument` object, and then loads the XML file we created earlier.

Listing B: *Recursively accessing XML nodes*

```
Private Sub DisplayNode(ByVal node As XmlNode)
   ' Display the node type.
   Debug.WriteLine(node.NodeType.ToString() & _
      ": <" & node.Name & ">")

   ' Display the node content, if applicable.
   If node.Value <> String.Empty Then
      Debug.WriteLine("Value: " & node.Value)
   End If

   ' Display all nested nodes.
   Dim Child As XmlNode
   For Each Child In node.ChildNodes
      DisplayNode(Child)
   Next
End Sub
```

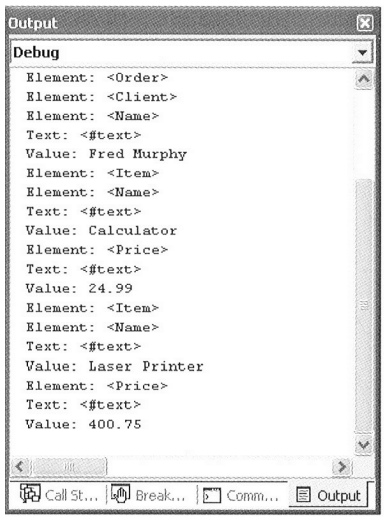
Listing C: *The form's new New() method*

```
Public Sub New()
   MyBase.New()

   'This call is required by the Windows
   'Form Designer.
   InitializeComponent()

   'Add any initialization after the
   'InitializeComponent() call

   ' Load the document.
   Dim Doc As New XmlDocument()
   Doc.Load("order.xml")

   ' Start the node walk at the root node
   (depth = 0).
   DisplayNode(Doc.DocumentElement)
End Sub
```

Next, it calls the `DisplayNode()` procedure and passes in the document's root node.

From here, the `DisplayNode()` procedure takes over. As its first order of business, the code writes out the node's type. Then, it places the node's name within opening and closing brackets. Next, it determines if the node contains a value. If it does, it also writes this value to the Debug window. Otherwise, it continues to the next section of code.

The loop we provided serves two purposes. First, it lets the application handle any and all child nodes associated with the current node. In addition, it triggers a recursive call to `DisplayNode()`, which in turn outputs nested child element values to the Debug window. As a result of this simple



recursive code, the procedure will process every node in the document, as seen in the output shown in **Figure B**.

VB .NET makes it easy

In this article, we looked at how to read XML data into VB .NET and obtain some basic information from a document of this type. In a future article, we'll consider an interesting trick: a way to automatically map custom .NET objects directly to XML nodes. ✳

Figure B: *Each XMLNode object exposes all the properties necessary to determine its type, value, and whether it contains child nodes.*

Dig deep into the operating system with Windows Management Instrumentation

by Jason Fisher

Application: Microsoft Visual Basic 5.0/6.0
Operating System: Microsoft Windows

Download: http://download.elementkjournals.com/ivb/200306.zip

Although it's been around for several years now, Windows Management Instrumentation (WMI) is still one of the best-kept secrets in the development community. Just as with XML, it probably hasn't fully caught on yet because developers are either confused by it, overwhelmed by it, or just not quite sure what to do with it. Admittedly, the WMI object model isn't as straightforward as that of many other, more familiar COM components—it feels a little more C++-centric—but with a little bit of background and some helpful samples, you'll see that it really isn't terribly difficult to use from Visual Basic. And, what's so exciting about WMI is that, through it, you can tap into features of the operating system that you never thought you could access so easily. A few examples of these features include processes, threads, services, hardware, and software settings.

Our agenda

In this article, we'll give you a crash course in starting to use WMI. As we alluded above, the topic is a vast one, easily filling a 1,000-page book, so we'll have to keep to a tightly focused plan. We'll start with just enough background in WMI to get you up to speed, but not so much as to put you to sleep. Then, we'll show you how to connect to the WMI namespace and begin looking around. Finally, for the real meat and potatoes of the article, we'll build the Visual Basic application shown in **Figure A**, with which we'll prove just how powerful and flexible WMI can be. Even this is a tall order for a single article, so we'd better jump right in.

What *is* WMI?

Without getting too buried in the theory that underlies it, WMI is basically the Microsoft implementation of an emerging standard called Web-Based Enterprise Management (WBEM). The idea is to derive a logical model of hardware and software components, irrespective of platform or vendor, and then each individual vendor provides a platform-specific implementation of this logical model.

The object model itself is constructed to be maximally flexible. As a result, the model relies heavily on inheritance and polymorphism—although you don't have to worry too much about this yourself—and upon a system of dynamic properties and methods.