XML in .NET: .NET Framework XML Classes and C# Offer Simple, Scalable Data Manipulation

Aaron Skonnard

This article assumes you're familiar with XML, C/C++, and C#

Level of Difficulty 1 2 3

Download the code for this article: Skonnard.exe (44KB)

SUMMARY Microsoft .NET introduces a new suite of XML APIs built on industry standards such as DOM, XPath, XSD, and XSLT. The .NET Framework XML classes also include innovations that offer convenience, better performance, and a more familiar programming model, tightly coupled with the new .NET data access APIs—ADO .NET. XmlWriter, XmlReader, and XmlNavigator classes and classes that derive from them, including XMLTextReader and XMLTextWriter, encapsulate a number of functionalities that previously had to be accomplished manually. A discussion of the XMLDOM-Document is also included.

The past few issues of *MSDN® Magazine* have dedicated significant space to the Microsoft® .NET architecture. The coverage has introduced the new framework, several new languages (including Visual Basic .NET and C#), the notion of Web Services, as well as several other key .NET topics.

In this article I'm going to introduce you to the new suite of .NET XML APIs, commonly referred to as the .NET Framework XML classes. I'll assume you have some knowledge of .NET fundamentals and C# (the language used for the sample code presented in this article). As a standard disclaimer, this article is based on .NET Beta 1—the final details are subject to change.

Since most aspects of .NET take advantage of XML in one way or another, a significant amount of time and energy went into improving the fundamental suite of XML APIs. The .NET Framework XML classes are built on key industry standards, such as DOM Level 2 Core, XPath 1.0, XSLT 1.0, XML Schemas (XSD), and the Simple Object Access Protocol (SOAP), but the classes also introduce several new enhancements to improve the overall programming model. In addition to the classic DOM model, the .NET Framework XML classes introduce an innovative stream-based API that offers a pull model interface (as opposed to the traditional SAX push model). Although this is probably the most significant change to the MSXML 3.0 model, several other enhancements were made to improve performance and ease-of-use.

The .NET Framework XML classes represent the natural evolution—not revolution—of MSXML 3.0 tailored to the .NET architecture. The new framework was modelled after MSXML 3.0, but also adds several improvements—including better standards compliance, extensible APIs, and a simpler programming model—all of which were achieved without sacrificing performance. MSXML 3.0 is available today and will continue to be the cornerstone for native COM applications (such as Microsoft Internet Explorer, Dynamic HTML applications, Office, Visual Basic® 6.0, and so on) that use XML and is the only viable choice for production code until .NET ships.

Developers can even continue using MSXML 3.0 in the .NET world of managed code,

using the COM Interop services provided in .NET. To use MSXML 3.0 in .NET, you must first create an assembly for the MSXML 3.0 types using the tlbimp utility that ships with the SDK. Then, you need to import the MSXML namespace and reference the generated assembly at compilation time (via/reference or /r for short). The following code fragment illustrates how to use the MSXML 3.0 DOMDocument within a C# class:

```
using MSXML;
DOMDocument30 doc = new DOMDocument30();
doc.async = false;
if (doc.load("somefile.xml"))
{
  // process document here
}
```

The main reason you may need to do this is when you have to port existing code to .NET with minimal code changes, or you have a large investment in the older XSL stylesheet language only supported by MSXML.

XML = .NET Technology Substrate

XML is truly a core technology substrate in .NET. All other parts of the .NET Framework (ASP .NET, Web Services, and so on) use XML as their native data representation format. The .NET Framework XML classes are also tightly coupled with Managed Data Access (ADO .NET). Traditionally, there have always been different programming models for working with relational versus hierarchical data. .NET breaks that tradition by offering a more deeply integrated programming model for all types of data.

The unified programming model I'm referring to is completely based on the core .NET Framework XML classes. This makes it possible to retrieve a relational ADO .NET dataset (like you would an ADO recordset) and walk it using the DOM, query or navigate it using XPath 1.0, and transform it using XSLT 1.0. And with XML Schemas, datasets can also be serialized to documents or streams, which contain the necessary metadata for future use. While XML seemed like an afterthought in ADO 2.*x*, it's at the core of ADO .NET. For more information on ADO .NET (formerly called ADO+) and its XML support, see Omri Gazitt's article "Introducing ADO+: Data Access Services for the Microsoft .NET Framework" in the November 2000 issue of *MSDN Magazine*. The remainder of this article focuses strictly on the components that make up the .NET Framework XML classes.

The .NET Framework XML classes are partitioned over several namespaces. The core types live in the System.Xml namespace. XPath/XSLT-specific types live in the System.Xml.XPath and System.Xml.Xsl namespaces, respectively (see Figure 1). All three of these namespaces are packaged in the System.Xml.dll assembly. So to start using the .NET Framework XML classes in C#, you must import the right namespaces (through the using directive):

```
using System.Xml; // now you can use these types
```

When you compile the project, you need to reference the appropriate assembly file containing the metadata for those types, in this case System.Xml.dll. A command-line compile would look like this:

```
csc /debug+ /r:System.Xml.dll helloxml.cs
```

Abstract Base Classes

At the core of the .NET Framework XML classes are two abstract classes: XmlReader and XmlWriter. XmlReader provides a fast, forward-only, read-only cursor for processing an XML document stream. XmlWriter provides an interface for producing XML document streams that conform to the W3C's XML 1.0 + Namespaces Recommendations. Applications that want to process XML documents consume XmlReader, while applications that want to produce XML documents consume XmlWriter. Both classes imply a streaming model that doesn't require an expensive in-memory cache. This makes them both attractive alternatives to the classic DOM approach.

Both XmlReader and XmlWriter are abstract base classes, which define the functionality that all derived classes must support. There are three concrete implementations of XmlReader—XmlTextReader, XmlNodeReader, and XslReader—as well as two concrete implementations of XmlWriter: XmlTextWriter and XmlNodeWriter (XmlNodeWriter is not in Beta 1, but should be coming in a future release.) XmlTextReader and XmlTextWriter support reading from and writing to a text-based stream, while XmlNodeReader and XmlNodeWriter are designed for working with in-memory DOM trees (see **Figure 2**). One of the biggest advantages of this new design is that custom readers and writers can be developed to extend the built-in functionality.

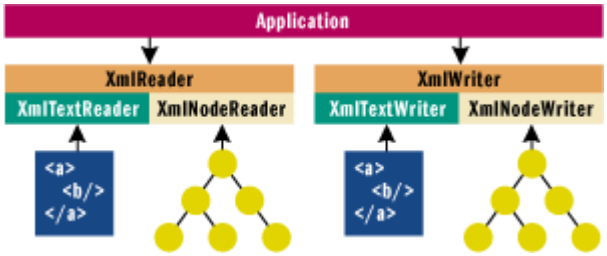

Figure 2 DOM Trees in XmlReader and XmlWriter

Before diving into the details, let's look at a quick example of how to use XmlReader and XmlWriter to process XML documents. The following example uses XmlReader to walk the document stream displaying the name of each node along the way (excluding attribute nodes):

```
public void ReadDocument(XmlReader reader)
{
  // read (pull) the next node in document order
  while (reader.Read())
  {
    // get the current node's (namespace) name
    String sNodeName = (reader.NodeType == XmlNodeType.Element ||
      reader.NodeType == XmlNodeType.EndElement) ?
      reader.NamespaceURI + "#" + reader.Name : reader.Name;
    Console.WriteLine(sNodeName);
  }
}
```

The following code illustrates how you can generate an XML document stream using the XmlWriter class:

```
public void WriteDocument(XmlWriter writer)
{
  writer.WriteStartDocument();
  writer.WriteComment("generated by SampleWriter");
  writer.WriteProcessingInstruction("hack", "on person");
  writer.WriteStartElement("p", "person", "urn:person");
  writer.WriteStartElement("name", "");
```

```
  writer.WriteString("joebob");
  writer.WriteEndElement();
  writer.WriteElementInt16("age", "", 28);
  writer.WriteEndElement();
  writer.WriteEndDocument();
}
```

This particular code fragment generates an XML document that can be serialized as follows:

```
<?xml version="1.0"?>
<!--sample person document-->
<?hack on person?>
<p:person xmlns:p="urn:person">
  <name>joebob</name>
  <age unit="year">28</age>
</p:person>
```

Let's start by digging into XmlReader, followed by XmlWriter. Then I'll look at the API support for DOM Level 2 Core, XPath, and XSLT.

XmlReader

XmlReader provides a fast, forward-only cursor for reading XML documents. It hides the complexities of working with the underlying data by serving up the document's Infoset through well-defined methods. At first glance, XmlReader feels a lot like SAX (especially ContentHandler), which is also based on streaming documents, but the two programming models are fundamentally different.

Back in 1998 when XML became a W3C Recommendation, followed later that year by DOM Level 1, developers quickly realized that the DOM didn't fit every application's needs, especially when working with large documents. As a result, SAX was developed to provide a more performance- and resource-sensitive processing model. SAX readers are responsible for parsing the character stream and for pushing the document's Infoset into consuming applications through a set of registered interfaces. The push model has several advantages over the DOM, but it comes with its own complexities, most of which revolve around state management.

XmlReader represents a compromise between the DOM and SAX APIs. It offers a streaming model (like SAX) that comes with a much simpler programming model (like DOM). XmlReader is very similar to a firehose (forward and read-only) cursor that allows the consumer to pull records one at a time and even skip records of no interest.

The XmlReader pull model has several advantages over the more common SAX-like push model (see **Figure 3**). First and foremost, the pull model is easier to use. It's just easier for developers to think about while loops than complicated state machines. Although contextual state management is still a challenge with the pull model, it's easier to deal with through consumer-driven procedural techniques that are more natural to most developers.

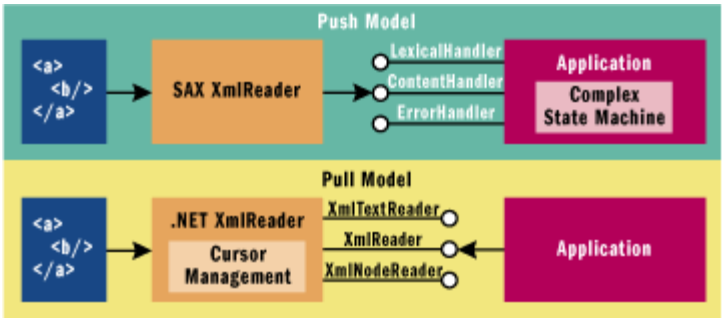

Figure 3 Push versus Pull Programming Models

The pull model can also offer improved performance through a variety of techniques. XmlReaders can take advantage of client hints to make more efficient use of character buffers (such as avoiding needless string copies). It's also possible for consumers to perform selective processing (skip over elements of no interest, don't expand entities, and so on). With a push model, everything has to be passed through the application because the reader has no way of knowing what's important.

In the end, the pull model offers a more familiar programming model along with several performance benefits. If you're still a push model aficionado, you can always layer a set of push-style interfaces on top of the XmlReader pull model, but the reverse is not true. A sample SAX2 implementation layered over XmlReader may ship with the .NET Framework SDK.

Inspecting the Current Node

XmlReader's pull model cursor defines the notion of a current node in the document stream. The current node changes by calling one of several other methods (which I'll discuss shortly). XmlReader offers several properties for inspecting the state of the current node. For example, the Name property returns the node's QName (if it has one), while LocalName and NamespaceURI return the node's namespace qualified identity. NodeType returns a type identifier similar to those used by the DOM. If the current node has a value (text nodes, attributes, and so on), you can access it through the Value property. And if the node is an element, you can check to see if it has attributes through the HasAttributes or AttributesCount properties. The following code fragment shows how to use these properties:

```
public static void InspectCurrentNode(XmlReader reader)
{
  String sQName = reader.Name;
  String sLocalName = reader.LocalName;
  String sNSURI = reader.NamespaceURI;
  XmlNodeType type = reader.NodeType;
  String sValue = reader.Value;
  boolean bAtts = reader.HasAttributes;
  int nAtts = reader.AttributeCount;
  •••
  // do something with data
}
```

Reading Nodes

XmlReader provides several methods for moving the cursor through the document stream, the most fundamental of which is Read. The Read method walks through the stream in document order one node at a time. An XmlException is thrown if the reader encounters a well-formedness error.

```
public void TraverseInDocumentOrder(XmlReader reader)
{
  try {
    while (reader.Read())
        Console.WriteLine("{0}", reader.NodeType);
}
catch(XmlException e) {
    Console.WriteLine("###error: " + e.Message);
 }
}
```

Document order is defined as the order of the structural elements in the serialized document or, in other words, the depth-first traversal of the document's logical structure (see **Figure 4**).

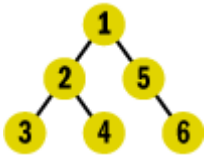

Figure 4 Doc Order

Running the code fragment shown previously against the following XML document

```
<?xml version="1.0"?>
<!--sample person document-->
<?hack on person?>
<p:person xmlns:p="urn:person">
  <name>joebob</name>
  <age unit="year">28</age>
</p:person>
```

produces the console output in Figure 5.

When an XmlReader is first initialized there is no current node, so the first call to Read moves to the first node in the document. When an XmlReader reaches the end of the document, it doesn't walk off the end, leaving it in an indeterminate state; it simply returns false when there are no more nodes to process.

XmlReader also offers several other Read methods that can provide contextual checks along the way. For example, if you want to make sure that the current node is an element with a specific name before continuing, you can use ReadStartElement as shown here:

```
try
{
  // if the current node doesn't have a local name
  // of "foo" and a namespace URI of "urn:bar", an
  // exception is thrown
  reader.ReadStartElement("foo", "urn:bar");
}
catch(XmlException e)
{
  Console.WriteLine(e.Message);
}
```

XmlReader provides several methods for reading typed text values (ReadInt16, ReadInt32,

ReadDouble, ReadString, and more). These methods provide hints to the reader about what type is expected next (allowing for parser-internal optimizations). These methods return a typed value instead of a string. For example, the following code illustrates how to quickly sum all Price elements within a document:

```
// sum all Price elements
double dTotal = 0.0;
while (reader.Read())
{
  if (reader.NodeType == XmlNodeType.Element)
  {
    if (reader.LocalName.Equals("Price"))
      dTotal += reader.ReadDouble();
  }
}
Console.WriteLine(dTotal);
```

Read doesn't encounter attribute nodes because they aren't considered part of a document's hierarchical structure. Attributes are typically considered metadata attached to structural elements. When the current node is an element, its attributes can be accessed through calls to GetAttribute by name or index. The following example walks an element's attributes by index:

```
public static void ReadAttributes(XmlReader reader)
{
  while (reader.Read())
  {
    if (reader.NodeType == XmlNodeType.Element)
      // process all of the element's attributes
      for (int i=0; i<reader.AttributeCount; i++)
        Console.WriteLine(reader.GetAttribute(i));
  }
}
```

Figure 6 helps illustrate how Read can be used to process the entire document stream. This is the canonical round-trip example that reads each node and serializes it back out to the console as XML 1.0.

Expanding Entity References

While reading a document stream, it's possible to optimize entity node expansion. When the XmlReader encounters a node of type EntityReference, it's up to the consumer to decide if expansion is necessary by explicitly calling the ResolveEntity method. If the consumer doesn't call ResolveEntity, but rather continues with a subsequent Read call, the entity's replacement content is skipped. If the consumer does call ResolveEntity, the replacement content is processed as everything else, one node at a time. The XmlReader also inserts an EndEntity node into the stream, making it possible for consumers to detect the end of the replacement content. The following code fragment illustrates these details (assuming EntityHandling has been set to ExpandCharEntities only):

```
while (reader.Read())
{
  if (reader.NodeType == XmlNodeType.Element)
    Console.WriteLine(reader.Name);
  if (reader.NodeType == XmlNodeType.EntityReference)
  {
    Console.WriteLine("*** resolve entity called ***");
```

```
    // after this call, the reader iterates over the
    // entity's content
    reader.ResolveEntity();
  }
  if (reader.NodeType == XmlNodeType.EndEntity)
    Console.WriteLine("*** end of current entity ***");
}
```

Running the previous code against the following XML document

```
<!DOCTYPE person [
  <!ENTITY fl "<first>Bob</first><last>Smith</last>">
]>
<person>
  <name>&fl;</name>
  <age>35</age>
</person>
```

produces the following console output:

```
person
name
*** resolve entity called ***
first
last
*** end of entity ***
age
```

Attribute Fidelity

Aside from the sequential read operations, XmlReader also provides several methods for moving to specific nodes within the document stream. When positioned on an element node, you can move sideways through the attached attribute nodes by calling MoveToAttribute, MoveToFirstAttribute, and MoveToNextAttribute. (You can also use the more convenient GetAttribute method mentioned earlier, but it only returns attribute values.) Conversely, when positioned on an attribute node, you can move back to the owner element node by calling MoveToElement.

If you're positioned on a non-content node (such as whitespace, comments, and processing instructions) and you want to move downstream to the next content node (element, text, and so on), you can use the handy MoveToContent method, as illustrated here:

```
public static void MoveToFirstContentNode(XmlReader reader)
{
  if (reader.Read())
  {
    Console.WriteLine(reader.Name);
    // skips all non-content nodes (pis, comments, whitespace)
    reader.MoveToContent();
    Console.WriteLine(reader.Name);
  }
}
```

To see how this works, examine the following document that contains several non-content nodes:

```
<?skip me?>
<!--skip me too-->
<?and me too...?>
```

```
<foo>
  <bar/>
</foo>
```
Processing this document with the previous code sample produces the following console output:
```
skip
foo
```

XmlTextReader

All of the examples shown so far have been coded against XmlReader, but since XmlReader is an abstract class it's helpful to introduce some of the concrete classes deriving from it. XmlTextReader is one such class that parses a text-based document stream into the tokens required of all XmlReader implementations. XmlTextReader has several different overloaded constructors that accept different types of input streams, including URI strings, Stream, and TextReader types. XmlTextReader provides several additional features that are unique to parser implementations, such as validation support and custom resolution of external entities.

XmlTextReader supports validation against DTDs, XML Data-Reduced (XDR) and XSD schemas (XSD is not currently supported in Beta 1, but is expected in a future release). Consumers can control how the reader performs validation through the Validation property. In its default state, XmlTextReader will auto-detect DTDs and schemas to process entities and default attribute values. To turn on validation, you must provide a ValidationHandler. When set to Validation.Schema, the reader automatically detects whether an XDR or XSD schema is in use (it's considered an error for a document to use both).

The ValidationHandler is an event handler for the reader to use when it encounters validation errors. The callback method is associated with the reader through the ValidationEventHandler event property. Figure 7 illustrates how to write a console application that validates the specified document against either a DTD or an XML schema.

Custom resolution of external entities makes it possible to improve the performance of your application by caching reused resources (like DTDs or Schemas) to avoid load delays. Applications that perform custom resolution must implement a class derived from XmlResolver (see the GetEntity method). Then, you can associate an instance of this new class with the XmlTextReader via its XmlResolver property. The following example shows a custom implementation of XmlResolver:

```
public class MyCustomResolver : XmlUrlResolver
{
  public override object GetEntity(string baseUri,
    string relativeUri, string role,
    Type t, out string resolvedUri)
  {
    resolvedUri = DoMyCustomResolution(baseUri, relativeUri);
    return GetResource(resolvedUri);
  }
}
```

Figure 8 shows how to use the MyCustomResolver class shown previously, along with several other XmlTextReader properties.

Other Readers

XmlTextReader is the canonical XmlReader implementation, but many other reader types are possible—including custom implementations. XmlNodeReader is an example of another implementation that reads an in-memory DOM tree as opposed to a character stream (see **Figure 2**). The result of XslTransform.Transform is another XmlReader that returns the result of the XSLT transform node-by-node. It's also possible to create your own XmlReader implementations that extend the default functionality of existing classes or that work with application-specific, non-XML data types.

To illustrate how this works, let's look at how to implement an XInclude-aware XmlReader class. XInclude defines a global attribute named href that belongs to the http://www.w3.org/1999/XML/xinclude namespace. Whenever an element containing the XInclude href attribute is encountered, it should be replaced by the Infoset of the referenced document. Figure 9 shows the beginning of such an implementation.

This implementation will merge the Infosets referenced by any XInclude elements (recursively through all included documents). For example, the following sample documents contain an XInclude relationship:

```
<!-- person-xinc.xml -->
<person xmlns:xinc="http://www.w3.org/1999/XML/xinclude">
  <fullname-ref xinc:href="fullname.xml"/>
</person>

<!-- fullname.xml -->
<name>
  <first>Joe</first>
  <last>Bob</last>
</name>
```

When person-xinc.xml is processed by this implementation of XmlReader, it produces a document stream that can be reserialized as follows:

```
<person>
  <name>
    <first>Joe</first>
    <last>Bob</last>
  </name>
</person>
```

There are many other possibilities for custom readers, especially when you need to make application-specific (binary) data look like XML to other applications.

XmlWriter

Like XmlReader, XmlWriter is an abstract class that defines the base functionality required to produce document streams conforming to the W3C's XML 1.0 + Namespaces Recommendations. As illustrated by the earlier example, XmlWriter completely shields applications from the complexities of producing XML document streams by allowing them to work with an Infoset-based API. Producing a document with XmlWriter is very similar to producing documents via SAX (see the SAX XMLWriter helper class). Currently there are two implementations of XmlWriter: XmlTextWriter and XmlNodeWriter (which is coming in a future release). These implementations are just like the reader versions, only they work in the opposite directions.

XmlWriter makes it possible to write out all standard Infoset constructs (such as elements, attributes, and processing instructions) using the corresponding Write*Xxxx* method (where *Xxxx*

is the name of the construct). If you look at the XmlWriter example shown in the second section of this article, you can see some of these methods in action (WriteStartDocument, WriteStartElement, WriteProcessingInstruction, and so on).

XmlWriter also provides methods for writing out typed elements and attributes through the WriteElement*Xxxx* and WriteAttribute*Xxxx* methods. Similarly, there are a bundle of methods for writing typed text values through the corresponding Write*Xxxx* methods. You can even write out the current node in an XmlReader (with or without attributes) by calling WriteNode. The following example illustrates how to simplify the serializer example shown in Figure 6 by using WriteNode:

```
public static void Serialize(XmlReader r, XmlWriter w)
{
  while (r.Read())
    w.WriteNode(r, true);
}
```

XmlTextWriter

XmlTextWriter is a concrete class, derived from XmlWriter, for writing out character streams. It supports many different output stream types (file URI, Stream, and TextWriter) and is very configurable. You can specify things like whether to provide namespace support, indentation options, what quote character is used for attribute values, and even what lexical representation should be used for typed values (WriteElementBool and so on), which is determined by the schema namespace in use (see DataTypeNamespace property). The following code illustrates how some of these properties can be configured:

```
public static void Main(String[] args)
{
  XmlTextWriter w = new XmlTextWriter(Console.Out);
  // write namespace names
  w.Namespaces = true;
  w.EncodeNames = true;
  // indent (pretty-print) output
  w.Formatting = Formatting.Indented;
  w.Indentation = 4;
  // use ' for attribute values
  w.QuoteChar = "'";
  WriteDocument(w);
}
```

Although XmlTextWriter is the most common XmlWriter, other XmlWriter implementations are possible. One example is XmlNodeWriter, which like XmlNodeReader writes an XML document stream into an in-memory DOM tree (see **Figure 2**).

DOM Implementation

The .NET DOM implementation (System.Xml.XmlDocument) supports all of DOM Level 1 and all of DOM Level 2 Core, but with a few minor naming changes. DOM loading is built on top of XmlReader, while DOM serialization is built on XmlWriter. This makes it possible to extend how the DOM interacts with your applications in numerous ways.

The .NET DOM API is very similar to what ships with MSXML 3.0, but with a few subtle changes. Figure 10 lays out the .NET DOM class hierarchy. The first subtle change is the

introduction of two new classes, XmlWhitespace and XmlSignificantWhitespace, for modeling whitespace in an XML document. Whitespace is considered insignificant when it appears in an element-only content model for readability purposes (the parser figures this out by inspecting the DTD/Schema). XmlLinkedNode, which derives from XmlNode, is a base class for all structural items in the document.

The XmlDocument class is equivalent to the MSXML DOMDocument coclass that models the document information item. XmlDocument provides several overloaded Load methods for building a DOM tree from an input stream. Load takes a URI for a document, a TextReader, or an XmlReader reference. The following code fragment illustrates how to load a document by URI:

```
XmlDocument doc = new XmlDocument();
doc.Load("http://staff.develop.com/aarons/xmlstuff.xml");
```

The LoadXML method expects an XML string as opposed to a stream identifier. MSXML 3.0 also offered an asynchronous loading mechanism, but this is no longer necessary in .NET since consumers have complete control over how DOM trees are built (via XmlReader or XmlWriter).

Generating DOM documents programmatically works the same as in MSXML 3.0. XmlDocument provides a factory method for each DOM node type (CreateElement, CreateAttribute, and so on). Figure 11 demonstrates how to generate the sample person document shown earlier.

Figure 11 also uses the XmlDocument Save method to serialize the DOM tree back out (with indenting!) in compliance with the XML 1.0 + Namespaces spec. There are actually several overloaded Save methods that take a URI, TextWriter, and XmlWriter references. The code in Figure 11 used an instance of XmlTextWriter for serializing to the console. There are also a few other serialization-related methods, including WriteTo and WriteContentTo, that serialize the current node (and its subtree) to the supplied XmlWriter. In addition, there are two DHTML-like properties named InnerXml and OuterXml that return a string of XML for the current node. Unlike with MSXML, the InnerXml property is settable, allowing you to build subtrees with a simple string assignment.

Navigation and XmlNavigator

XmlNode provides the basic methods for navigating the DOM tree (including ChildNodes, FirstChild, LastChild, ParentNode, NextSibling, and PreviousSibling). Document navigation can also be performed using a separate class named XmlNavigator that provides a generic navigation mechanism as well as an implementation of the XPath 1.0 Recommendation. As with XmlReader and XmlWriter, XmlNavigator is an abstract base class that defines the common functionality of all navigators. XmlNavigator supports generic navigation routines, selecting nodes, iterating over the selection, and working with the selection in more advanced ways (copying, moving, removing, and so on).

XmlNavigator is preferred over the standard DOM API; typical XmlNavigator implementations don't need to load the entire document structure into memory, whereas most DOM implementations do, making XmlNavigator a better-performing alternative. XmlNavigator also seems to provide a more intuitive interface, especially when working with XPath expressions.

XmlNavigator provides two methods for selecting node-sets, Select and SelectSingle, which

are equivalent to the selectNodes and selectSingleNode methods offered by MSXML 3.0. Both methods accept an XPath expression in the form of a string or a precompiled expression object, which are evaluated to identify a set of nodes.

```
public static void EvaluateXPath(XmlNavigator nav)
{
  // returns a node-set
  nav.Select("//Price/text()");
  while (nav.MoveToNextSelected())
  {
    // process selected nodes here
  }
}
```

If an expression will be used repeatedly, you can improve performance by compiling the expression into a cached object that can be reused.

```
Object cachedExpr = null;
cachedExpr = nav.Compile("//Invoice[@id>1000]//Price[.>10]");
// can reuse expression w/out compilation cost
nav.Select(cachedExpr);
while (nav.MoveToNextSelected())
{
  // process selected nodes here
}
```

After calling Select, use the MoveToNextSelected method to iterate over the selected nodes.

There are additional methods available for copying, moving, and removing the selected nodes. For example, the following code fragment illustrates how to copy the selected nodes between XmlNavigators:

```
public static void EvaluateXPath(XmlNavigator in, XmlNavigator out)
{
  // returns a node-set
  in.Select("//Price");
  out.CopySelected(TreePosition.FirstChild, in);
  // out navigator now contains selected nodes
}
```

And one of my favorite additions is the new Evaluate method, which evaluates XPath expressions that return XPath types other than node-set (string, number, or Boolean). For example, the following example uses an XPath expression to sum all of the Price elements in a document:

```
public static void SumPriceNodes(XmlNavigator nav) {
  // in this case, evaluate returns a number
  Console.WriteLine("sum=" + nav.Evaluate("sum(//Price)"));
}
```

Without this method, you would have to select all of the Price elements and walk through them, performing the arithmetic manually. Figure 12 shows the implementation of a convenient console utility for evaluating XPath expressions.

DocumentNavigator

DocumentNavigator is a concrete class, derived from XmlNavigator, that was designed specifically for navigating DOM trees. When you create an instance of DocumentNavigator, you simply pass it a reference to the XmlDocument that you want to navigate.

```
XmlDocument doc = new XmlDocument();
```

```
doc.Load("person.xml");
DocumentNavigator nav = new DocumentNavigator(doc);
nav.Select("/person/name");
while (nav.MoveToNextSelected())
{
  // process selection here...
}
```

As you navigate using XmlNavigator, you can access information about the current node through its exposed properties. If you want a complete DOM view of the current node, use the GetNode method to retrieve a reference to the current XmlNode object.

Thanks to this extensible design, other navigator implementations are possible as well. For starters, there is a DataDocumentNavigator class for navigating an ADO .NET dataset just like an XML document. I've even heard rumors of a registry navigator floating around. (And custom implementations of XmlNavigator get XPath support for free.)

Transformations and XslTransform

The XslTransform class manages XSLT transformations in the new framework. XslTransform lives in the System.Xml.Xsl namespace and it uses XmlNavigator during the transformation process. As with all XSLT processors, XslTransform accepts as input an XML document, an XSLT document, and some optional parameters. It can produce any type of text-based output, and yes, it supports reading the result of the transform via a custom XmlReader.

To perform a transformation using XslTransform, first create an XslTransform object and load it with the desired XSLT document (by calling Load). Then, create an XmlNavigator object and initialize it with the source XML document that you want to transform. And finally, call Transform to begin the process (see Figure 13).

Where Are We?

I've covered the core types that make up the new .NET Framework XML classes, the most important of which are XmlReader and XmlWriter, but you've really only seen the tip of the iceberg. The .NET XML story includes several other meaty topics, including XML integration into ADO .NET, XML serialization, the XSD object model, security and XML digital signatures, ASP .NET native XML integration, Web Services, and more. You'll undoubtedly be reading about all of these topics in the months to come in *MSDN Magazine*.

For related articles see:
.NET coverage in the September and October 2000 issues of MSDN Magazine
For background information see:
The Programmable Web: Web Services Provides Building Blocks for the Microsoft .NET Framework
http://msdn.microsoft.com/xml/articles/xml07242000.asp

Aaron Skonnard *is an instructor and researcher at DevelopMentor, where he develops the XML curriculum. Aaron coauthored* Essential XML *(Addison-Wesley Longman, 2000) and wrote* Essential WinInet *(Addison-Wesley Longman, 1998). Get in touch with Aaron at http://staff.develop.com/aarons.*

From the January 2001 issue of MSDN Magazine

Current Issue

Browse All MSDN Magazines