12

Using XML in Visual Basic 2005

In this chapter, we'll look at how you can generate and manipulate Extensible Markup Language (XML) using Visual Basic 2005. However, using XML in Visual Basic is a vast area to cover (more than possibly could be covered in this chapter). The .NET Framework exposes five XML-specific namespaces that contain over a hundred different classes. In addition, there are dozens of other classes that support and implement XML-related technologies, such as ADO.NET, SQL Server, and BizTalk. Consequently, we'll concentrate on the general concepts and the most important classes.

Visual Basic relies on the classes exposed in the following XML-related namespaces to transform, manipulate, and stream XML documents:

- `System.Xml` provides core support for a variety of XML standards (including DTD, namespace, DOM, XDR, XPath, XSLT, and SOAP).
- `System.Xml.Serialization` provides the objects used to transform objects to and from XML documents or streams using serialization.
- `System.Xml.Schema` provides a set of objects that allow schemas to be loaded, created, and streamed. This support is achieved using a suite of objects that support the in-memory manipulation of the entities that compose an XML schema.
- `System.Xml.XPath` provides a parser and evaluation engine for the XML Path Language (XPath).
- `System.Xml.Xsl` provides the objects necessary when working with Extensible Stylesheet Language (XSL) and XSL Transformations (XSLT).

The XML-related technologies utilized by Visual Basic include other technologies that generate XML documents and allow XML documents to be managed as a data source:

- **ADO** — The legacy COM objects provided by ADO have the ability to generate XML documents in stream or file form. ADO can also retrieve a previously persisted XML document and manipulate it. (Although ADO will not be used in this chapter, ADO and other legacy COM APIs can be accessed seamlessly from Visual Basic.)
- **ADO.NET** — This uses XML as its underlying data representation: the in-memory data representation of the ADO.NET `DataSet` object is XML; the results of data queries are represented as XML documents; XML can be imported into a `DataSet` and exported from a `DataSet`. (ADO.NET is covered in Chapter 11.)
- **SQL Server 2000** — XML-specific features were added to SQL Server 2000 (`FOR XML` queries to retrieve XML documents and `OPENXML` to represent an XML document as a rowset). Visual Basic can use ADO.NET to access SQL Server's XML-specific features (the documents generated and consumed by SQL Server can then be manipulated programmatically). Recently, Microsoft also released SQLXML, which provides an SQL Server 2000 database with some excellent XML capabilities, such as the ability to query a database using XQuery, get back XML result sets from a database, work with data just as if it was XML, take huge XML files and have SQLXML convert them to relational data, and much more. SQLXML allows you to perform these functions and more via a set of managed .NET classes. You can download SQLXML for free from the Microsoft SQLXML Web site at `http://msdn.microsoft.com/sqlxml`.
- **SQL Server 2005** — SQL Server has now been modified with XML in mind. SQL Server 2005 can natively understand XML because it is now built into the underlying foundation of the database. The ability to query and understand XML documents is a valuable addition to this database server. SQL Server 2005 also comes in a lightweight (and free) version called SQL Server Express Edition.

In this chapter, we'll make sense of this range of technologies by introducing some basic XML concepts and demonstrating how Visual Basic, in conjunction with the .NET Framework, can make use of XML. Specifically, you will:

- Learn the rationale behind XML.
- Look at the namespaces within the .NET Framework class library that deal with XML and XML-related technologies.
- Take a closer look at some of the classes contained within these namespaces.
- Gain an overview of some of the other Microsoft technologies that utilize XML, particularly SQL Server and ADO.NET.

At the end of this chapter, you will be able to generate, manipulate, and transform XML using Visual Basic.

An Introduction to XML

XML is a tagged markup language similar to HTML. In fact, XML and HTML are distant cousins and have their roots in the Standard Generalized Markup Language (SGML). This means that XML leverages one of the most useful features of HTML — readability. However, XML differs from HTML in that XML represents data, while HTML is a mechanism for displaying data. The tags in XML describe the data, for example:

```
<?xml version="1.0" encoding="utf-8"?>
<Movies>
  <FilmOrder name="Grease" filmId="1" quantity="21"></FilmOrder>
  <FilmOrder name="Lawrence of Arabia" filmId="2" quantity="10"></FilmOrder>
  <FilmOrder name="Star Wars" filmId="3" quantity="12"></FilmOrder>
  <FilmOrder name="Shrek" filmId="4" quantity="14"></FilmOrder>
</Movies>
```

This XML document is used to represent a store order for a collection of movies. The standard used to represent an order of films would be useful to movie rental firms, collectors, and others. This information can be shared using XML because:

- The data tags in XML are self-describing.
- XML is an open standard and supported on most platforms today.

XML supports the parsing of data by applications not familiar with the contents of the XML document. XML documents can also be associated with a description (a schema) that informs an application as to the structure of the data within the XML document.

At this stage, XML looks simple — it's just a human-readable way to exchange data in a universally accepted way. The essential points that you should understand about XML are

- XML data can be stored in a plain text file.
- A document is said to be well formed if it adheres to the XML standard.
- Tags are used to specify the contents of a document, for example, `<FilmOrder>`.
- XML elements (also called nodes) can be thought of as the objects within a document.
- Elements are the basic building blocks of the document. Each element contains a start tag and end tag. A tag can be both a start and an end tag, for example, `<FilmOrder/>`. Such a tag is said to be empty.
- Data can be contained in the element (the element content) or within attributes contained in the element.
- XML is hierarchical. One document can contain multiple elements, which can themselves contain child elements, and so on. However, an XML document can only have one root element.

This last point means that the XML document hierarchy can be thought of as a tree containing nodes:

- The example document has a root node, `<Movies>`.
- The branches of the root node are elements of type `<FilmOrder>`.
- The leaves of the XML element, `<FilmOrder>`, are its attributes: `name`, `quantity`, and `filmId`.

Of course, we're interested in the practical use of XML by Visual Basic. A practical manipulation of the example XML is to display for the staff of the movie supplier firm a particular movie order in some application so that this supplier could fill the order, and then save the information to a database. In this chapter, you'll look at how you can perform such tasks using the functionality provided by the .NET Framework class library.

XML Serialization

The simplest way to demonstrate Visual Basic's support for XML is not with a complicated technology, such as SQL Server or ADO.NET. Instead, we will demonstrate a practical use of XML by serializing a class.

The serialization of an object means that it is written out to a stream, such as a file or a socket (this is also known as dehydrating an object). The reverse process can also be performed: An object can be deserialized (or rehydrated) by reading it from a stream.

> *The type of serialization you are discussing in this chapter is XML serialization, where XML is used to represent a class in serialized form.*

To help you understand XML serialization, let's examine a class named `FilmOrder` (which can be found in the code download from `www.wrox.com`). This class is implemented in Visual Basic and is used by the company for processing an order for movies. This class could be instantiated on a firm's PDA, laptop, or even mobile phone (so long as the .NET Framework was installed).

An instance of `FilmOrder` corresponding to each order could be serialized to XML and sent over a socket using the PDA's cellular modem. (If the person making the order had a PDA which did not have a cellular modem, the instance of `FilmOrder` could be serialized to a file.) The order could then be processed when the PDA was dropped into a docking cradle and synced. What we are talking about here is data in a propriety form, an instance of `FilmOrder` being converted into a generic form — XML — that can be universally understood.

The `System.Xml.Serialization` namespace contains classes and interfaces that support the serialization of objects to XML and the deserialization of objects from XML. Objects are serialized to documents or streams using the `XmlSerializer` class. Let's look at how you can use `XmlSerializer`. First, you need to define an object that implements a default constructor, such as `FilmOrder`:

```
Public Class FilmOrder

  ' These are Public because we have yet to implement
  ' properties to provide program access.

  Public name As String
  Public filmId As Integer
  Public quantity As Integer

  Public Sub New()
  End Sub

  Public Sub New(ByVal name As String, _
                 ByVal filmId As Integer, _
```

```
                    ByVal quantity As Integer)
      Me.name = name
      Me.filmId = filmId
      Me.quantity = quantity
    End Sub
  End Class
```

This class should be created in a console application. From there, let's move onto the module. Within the module's `Sub Main`, create an instance of `XmlSerializer`, specifying the object to serialize and its type in the constructor:

```
  Dim serialize As XmlSerializer = _
    New XmlSerializer(GetType(FilmOrder))
```

Create an instance of the same type as was passed as parameter to the constructor of `XmlSerializer`:

```
  Dim MyFilmOrder As FilmOrder = _
    New FilmOrder("Grease", 101, 10)
```

Call the `Serialize` method of the `XmlSerializer` instance, and specify the stream to which the serialized object is written (parameter one, `Console.Out`) and the object to be serialized (parameter two, `prescription`):

```
  serialize.Serialize(Console.Out, MyFilmOrder)
  Console. WriteLine()
```

To make reference to the `XmlSerializer` object, you are going to have to make reference to the `System.Xml.Serialization` namespace:

```
  Imports System.Xml
  Imports System.Xml.Serialization
```

Running the module, the following output is generated by the preceding code:

```
<?xml version="1.0" encoding="IBM437"?>
<FilmOrder xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <name>Grease</name>
  <filmId>101</filmId>
  <quantity>10</quantity>
</FilmOrder>
```

This output demonstrates the default way that the `Serialize` method serializes an object:

- Each object serialized is represented as an element with the same name as the class, in this case `FilmOrder`.
- The individual data members of the class serialized are contained in elements named for each data member, in this case `name`, `filmId`, and `quantity`.

Also generated are

- The specific version of XML generated, in this case `1.0`
- The encoding used, in this case `IBM437`
- The schemas used to describe the serialized object, in this case `www.w3.org/2001/XMLSchema-instance` and `www.w3.org/2001/XMLSchema`

A schema can be associated with an XML document and describe the data it contains (name, type, scale, precision, length, and so on). Either the actual schema or a reference to where the schema resides can be contained in the XML document. In either case, an XML schema is a standard representation that can be used by all applications that consume XML. This means that applications can use the supplied schema to validate the contents of an XML document generated by the `Serialize` method of `XmlSerializer`.

The code snippet that demonstrated the `Serialize` method of `XmlSerializer` displayed the XML generated to `Console.Out`. Clearly, we do not expect an application to use `Console.Out` when it would like to access a `FilmOrder` object in XML form. The basic idea shown was how serialization can be performed in just two lines of code (one call to a constructor and one call to method). The entire section of code responsible for serializing the instance of `FilmOrder` is

```
Try
    Dim serialize As XmlSerializer = _
                New XmlSerializer(GetType(FilmOrder))
    Dim MyMovieOrder As FilmOrder = _
            New FilmOrder("Grease", 101, 10)
    serialize.Serialize(Console.Out, MyMovieOrder)
    Console.Out.WriteLine()
    Console.Readline()
Catch ex As Exception
    Console.Error.WriteLine(ex.ToString())
End Try
```

The `Serialize` method's first parameter is overridden so that it can serialize XML to a file (the file name is given as type `String`), a `Stream`, a `TextWriter`, or an `XmlWriter`. When serializing to `Stream`, `TextWriter`, or `XmlWriter`, adding a third parameter to the `Serialize` method is permissible. This third parameter is of type `XmlSerializerNamespaces` and is used to specify a list of namespaces that qualify the names in the XML-generated document. The permissible overrides of the `Serialize` method are:

```
Public Sub Serialize(Stream, Object)
Public Sub Serialize(TextWriter, Object)
Public Sub Serialize(XmlWriter, Object)
Public Sub Serialize(Stream, Object, XmlSerializerNamespaces)
Public Sub Serialize(TextWriter, Object, XmlSerializerNamespaces)
Public Sub Serialize(XmlWriter, Object, XmlSerializerNamespaces)
```

An object is reconstituted using the `Deserialize` method of `XmlSerializer`. This method is overridden and can deserialize XML presented as a `Stream`, a `TextReader`, or an `XmlReader`. The overloads for `Deserialize` are:

```
Public Function Deserialize(Stream) As Object
Public Function Deserialize(TextReader) As Object
Public Function Deserialize(XmlReader) As Object
```

Before demonstrating the `Deserialize` method, we will introduce a new class, `WXClientMultiPrescription`. This class contains an array of prescriptions (an array of `WXClientPrescription` objects). `WXClientMultiPrescription` is defined as follows:

```
Public Class FilmOrder_Multiple

    Public multiFilmOrders() As FilmOrder

    Public Sub New()
    End Sub

    Public Sub New(ByVal multiFilmOrders() As FilmOrder)
        Me.multiFilmOrders = multiFilmOrders
    End Sub
End Class
```

The `FilmOrder_Multiple` class contains a fairly complicated object, an array of `FilmOrder` objects. The underlying serialization and deserialization of this class is more complicated than that of a single instance of a class that contains several simple types. However, the programming effort involved on your part is just as simple as before. This is one of the great ways in which the .NET Framework makes it easy for you to work with XML data, no matter how it is formed.

To work through an example of the deserialization process, let's start by first creating a sample order stored as an XML file called `Filmorama.xml`.

```
<?xml version="1.0" encoding="utf-8" ?>
<FilmOrder_Multiple xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <multiFilmOrders>
         <FilmOrder>
            <name>Grease</name>
            <filmId>101</filmId>
            <quantity>10</quantity>
         </FilmOrder>
         <FilmOrder>
            <name>Lawrence of Arabia</name>
            <filmId>102</filmId>
            <quantity>10</quantity>
         </FilmOrder>
         <FilmOrder>
            <name>Star Wars</name>
            <filmId>103</filmId>
            <quantity>10</quantity>
         </FilmOrder>
   </multiFilmOrders>
</FilmOrder_Multiple>
```

Once the XML file is in place, the next step is to change your console application so it will take this XML file and deserialize its contents.

From there, it is important to make sure that your console application has made the proper namespace references:

```
Imports System.Xml
Imports System.Xml.Serialization
Imports System.IO
```

Then, the following code demonstrates an object of type `FilmOrder_Multiple` being deserialized (or rehydrated) from a file, `Filmorama.xml`. This object is deserialized using this file in conjunction with the `Deserialize` method of `XmlSerializer`:

```
' Open file, ..\Filmorama.xml
Dim dehydrated As FileStream = _
   New FileStream("..\Filmorama.xml", FileMode.Open)

' Create an XmlSerializer instance to handle deserializing, ' FilmOrder_Multiple
Dim serialize As XmlSerializer = _
   New XmlSerializer(GetType(FilmOrder_Multiple))

' Create an object to contain the deserialized instance of the object.
Dim myFilmOrder As FilmOrder_Multiple = _
   New FilmOrder_Multiple

' Deserialize object
myFilmOrder = serialize.Deserialize(dehydrated)
```

Once deserialized, the array of prescriptions can be displayed:

```
Dim SingleFilmOrder As FilmOrder

For Each SingleFilmOrder In myFilmOrder.multiFilmOrders
   Console.Out.WriteLine("{0}, {1}, {2}", _
      SingleFilmOrder.name, _
      SingleFilmOrder.filmId, _
      SingleFilmOrder.quantity)
Next

Console.ReadLine()
```

This example is just code that serializes an instance of type, `FilmOrder_Multiple`. The output generated by displaying the deserialized object containing an array of film orders is:

```
Grease, 101, 10
Lawrence of Arabia, 102, 10
Star Wars, 103, 10
```

`XmlSerializer` also implements a `CanDeserialize` method. The prototype for this method is:

```
Public Overridable Function CanDeserialize(ByVal xmlReader As XmlReader) _
   As Boolean
```

If `CanDeserialize` returns `True`, then the XML document specified by the `xmlReader` parameter can be deserialized. If the return value of this method is `False`, then the specified XML document cannot be deserialized.

The `FromTypes` method of `XmlSerializer` facilitates the creation of arrays that contain `XmlSerializer` objects. This array of `XmlSerializer` objects can be used in turn to process arrays of the type to be serialized. The prototype for `FromTypes` is:

```
Public Shared Function FromTypes(ByVal types() As Type) As XmlSerializer()
```

Before we further explore the `System.Xml.Serialization` namespace, we need to take a moment to consider the various uses of the term "attribute."

Source Code Style Attributes

Thus far you have seen attributes applied to a specific portion of an XML document. Visual Basic has its own flavor of attributes, as do C# and each of the other .NET languages. These attributes refer to annotations to the source code that specify information (or metadata) that can be used by other applications without the need for the original source code. We will call such attributes Source Code Style attributes.

In the context of the `System.Xml.Serialization` namespace, Source Code Style attributes can be used to change the names of the elements generated for the data members of a class or to generate XML attributes instead of XML elements for the data members of a class. To demonstrate this, we will use a class called `ElokuvaTilaus`, which contains data members named `name`, `filmId`, and `quantity`. It just so happens that the default XML generated when serializing this class is not in a form that can be readily consumed by an external application. As an example of this, assume that a Finnish development team has written this external application, and hence the XML element and attribute names are in Finnish (minus the umlauts) rather than in English.

To rename the XML generated for a data member, `name`, a Source Code Style attribute will be used. This Source Code Style attribute would specify that when `ElokuvaTilaus` is serialized, the `name` data member would be represented as an XML element, `<Nimi>`. The actual Source Code Style attribute that specifies this is:

```
<XmlElementAttribute("Nimi")> Public name As String
```

`ElokuvaTilaus` also contains other Source Code Style attributes:

- `<XmlAttributeAttribute("ElokuvaId")>` — Specifies that `filmId` is to be serialized as an XML attribute named `ElokuvaId`
- `<XmlAttributeAttribute("Maara")>` — Specifies that `quantity` is to be serialized as an XML attribute named `Maara`

`ElokuvaTilaus` is defined as follows:

```
Imports System.Xml.Serialization

Public Class ElokuvaTilaus

  ' These are Public because we have yet to implement
  ' properties to provide program access.

  <XmlElementAttribute("Nimi")> Public name As String
  <XmlAttributeAttribute("ElokuvaId")> Public filmId As Integer
  <XmlAttributeAttribute("Maara")> Public quantity As Integer

  Public Sub New()
  End Sub

  Public Sub New(ByVal name As String, _
                 ByVal filmId As Integer, _
                 ByVal quantity As Integer)
      Me.name = name
      Me.filmId = filmId
      Me.quantity = quantity
  End Sub

End Class
```

`ElokuvaTilaus` can be serialized as follows:

```
Dim serialize As XmlSerializer = _
    New XmlSerializer(GetType(ElokuvaTilaus))
Dim MyMovieOrder As ElokuvaTilaus = _
    New ElokuvaTilaus("Grease", 101, 10)

serialize.Serialize(Console.Out, MyMovieOrder)
```

The output generated by this code reflects the Source Code Style attributes associated with class `ElokuvaTilaus`:

```
<?xml version="1.0" encoding="IBM437"?>
<ElokuvaTilaus xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 ElokuvaId="101" Maara="10">
    <Nimi>Grease</Nimi>
</ElokuvaTilaus>
```

The value of `filmId` is contained in an XML attribute, `ElokuvaId`, and the value of `quantity` is contained in an XML attribute, `Maara`. The value of `name` is contained in an XML element, `Nimi`.

The example has only demonstrated the Source Code Style attributes exposed by the `XmlAttributeAttribute` and `XmlElementAttribute` classes in the `System.Xml .Serialization` namespace. A variety of other Source Code Style attributes exist in this namespace that also control the

form of XML generated by serialization. The classes associated with such Source Code Style attributes include `XmlTypeAttribute`, `XmlTextAttribute`, `XmlRootAttribute`, `XmlIncludeAttribute`, `XmlIgnoreAttribute`, and `XmlEnumAttribute`.

System.Xml Document Support

The `System.Xml` namespace implements a variety of objects that support standards-based XML processing. The XML-specific standards facilitated by this namespace include XML 1.0, Document Type Definition (DTD) support, XML namespaces, XML schemas, XPath, XQuery, XSLT, DOM Level 1 and DOM Level 2 (Core implementations), as well as SOAP 1.1, SOAP 1.2, SOAP Contract Language, and SOAP Discovery. The `System.Xml` namespace exposes over 30 separate classes in order to facilitate this level of XML standard's compliance.

With respect to generating and navigating XML documents, there are two styles of access:

- **Stream-based** — `System.Xml` exposes a variety of classes that read XML from and write XML to a stream. This approach tends to be a fast way to consume or generate an XML document because it represents a set of serial reads or writes. The limitation of this approach is that it does not view the XML data as a document composed of tangible entities, such as nodes, elements, and attributes. An example of where a stream could be used is when receiving XML documents from a socket or a file.
- **Document Object Model (DOM)–based** — `System.Xml` exposes a set of objects that access XML documents as data. The data is accessed using entities from the XML document tree (nodes, elements, and attributes). This style of XML generation and navigation is flexible but may not yield the same performance as stream-based XML generation and navigation. DOM is an excellent technology for editing and manipulating documents. For example, the functionality exposed by DOM might make merging your checking, savings, and brokerage accounts simpler.

XML Stream-Style Parsers

When demonstrating XML serialization, you alluded to XML stream-style parsers. After all, when an instance of an object was serialized to XML, it had to be written to a stream, and when it was deserialized, it was read from a stream. When an XML document is parsed using a stream parser, the parser always points to the current node in the document. The basic architecture of stream parsers is shown in Figure 12-1.

The classes that access a stream of XML (read XML) and generate a stream of XML (write XML) are contained in the `System.Xml` namespace and are

- `XmlWriter` — This abstract class specifies a noncached, forward-only stream that writes an XML document (data and schema).
- `XmlReader` — This abstract class specifies a noncached, forward-only stream that reads an XML document (data and schema).

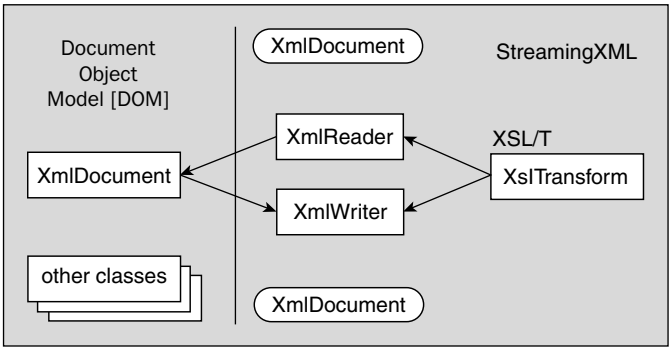

Figure 12-1

Your diagram of the classes associated with the XML stream-style parser referred to one other class, `XslTransform`. This class is found in the `System.Xml.Xsl` namespace and is not an XML stream-style parser. Rather, it is used in conjunction with `XmlWriter` and `XmlReader`. This class will be reviewed in detail later.

The `System.Xml` namespace exposes a plethora of additional XML manipulation classes in addition to those shown in the architecture diagram. The classes shown in the diagram include

- `XmlResolver` — This abstract class resolves an external XML resource using a Uniform Resource Identifier (URI). `XmlUrlResolver` is an implementation of an `XmlResolver`.
- `XmlNameTable` — This abstract class provides a fast means by which an XML parser can access element or attribute names.

Writing an XML Stream

An XML document can be created programmatically in .NET. One way to perform this task is by writing the individual components of an XML document (schema, attributes, elements, and so on) to an XML stream. Using a unidirectional write-stream means that each element and its attributes must be written in order — the idea is that data is always written at the head of the stream. To accomplish this, you use a writable XML stream class (a class derived from `XmlWriter`). Such a class ensures that the XML document you generate correctly implements the W3C Extensible Markup Language (XML) 1.0 specification and the Namespaces in XML specification.

But why would this be necessary since you have XML serialization? You need to be very careful here to separate interface from implementation. XML serialization worked for a specific class, `ElokuvaTilaus`. The class is a proprietary implementation and not the format in which data is exchanged. For this one specific case, the XML document generated when `ElokuvaTilaus` is serialized just so happens to be the XML format used when placing an order for some movies. `ElokuvaTilaus` was given a little help from Source Code Style attributes so that it would conform to a standard XML representation of a film order summary.

In a different application, if the software used to manage an entire movie distribution business wants to generate movie orders, it will have to generate a document of the appropriate form. The movie distribution management software will achieve this by using the `XmlWriter` object.

Before reviewing the subtleties of `XmlWriter`, it is important to note that this class exposes over 40 methods and properties. The example presented in this section will provide an overview that touches on a subset of these methods and properties. This subset will allow an XML document that corresponds to a movie order to be generated.

For this example, let's build a module that generates an XML document corresponding to a movie order. You will use an instance of `XmlWriter`, `FilmOrdersWriter`, which will actually be a file on disk. This means that the XML document generated is streamed to this file. Since the `FilmOrdersWriter` variable represents a file, it must be

- **Created** — The instance of `XmlWriter` `FilmOrdersWriter` is created using the `Create` method as well as by assigning all the properties of this object with the `XmlWriterSettings` object.
- **Opened** — The file the XML is streamed to, `FilmOrdersProgrammatic.xml`, is opened by passing the file name to the constructor associated with `XmlWriter`.
- **Generated** — The process of generating the XML document is described in detail at the end of this section.
- **Closed** — The file (the XML stream) is closed using the `Close` method of `XmlWriter` or by simply using the `Using` keyword.

Before you go about creating the `XmlWriter` object, you will first need to customize how the object will operate by using the `XmlWriterSettings` object. This object, which is new to .NET 2.0, allows you to configure the behavior of the `XmlWriter` object before you instantiate it.

```
Dim myXmlSettings As New XmlWriterSettings
myXmlSettings.Indent = True
myXmlSettings.NewLineOnAttributes = True
```

The `XmlWriterSettings` object allows for a few settings on how the XML creation will be handled by the `XmlWriter` object. The following table details the properties of the `XmlWriterSettings` class.

Property	Initial Value	Description
`CheckCharacters`	`True`	This property, if set to `True`, will perform a character check upon the contents of the `XmlWriter` object. Legal characters can be found at `www.w3.org/TR/REC-xml#charsets`.
`CloseOutput`	`False`	This property will get or set a value indicating whether the `XmlWriter` should also close the underlying stream or `System.IO.TextWriter` when the `XmlWriter.Close` method is called.

Table continued on following page

Property	Initial Value	Description
`ConformanceLevel`	`ConformanceLevel Document`	Allows the XML to be checked to make sure that it follows certain specified rules. Possible conformance level settings include `Document`, `Fragment`, and `Default`.
`Encoding`	`Encoding.UTF8`	Defines the encoding of the XML generated.
`Indent`	`False`	Defines whether the XML generated should be indented or not. Setting this value to `True` will properly indent child nodes from parent nodes.
`IndentChars`	`Two spaces`	Specifies the number of spaces by which child nodes will be indented from parent nodes. This setting only works when the `Indent` property is set to `True`.
`NewLineChars`	`\r\n`	Assigns the characters that are used to define line breaks.
`NewLineHandling`	`System.Xml .NewLineHandling .Replace`	This property gets or sets a value indicating whether to normalize line breaks in the output.
`NewLineOnAttributes`	`False`	Defines whether a node's attributes should be written to a new line in the construction. This will occur if set to `True`.
`OmitXmlDeclaration`	`False`	Defines whether an XML declaration should be generated in the output. This omission only occurs if set to `True`.
`OutputMethod`	`System.Xml .XmlOutputMethod .Xml`	This property gets the method used to serialize the `System.Xml.XmlWriter` output.

Once the `XmlWriterSettings` object has been instantiated and assigned the values you deem necessary, the next steps are to invoke the `XmlWriter` object as well as make the association between the `XmlWriterSettings` object and the `XmlWriter` object.

The basic infrastructure for managing the file (the XML text stream) and applying the settings class is:

```
Dim FilmOrdersWriter As XmlWriter = _
   XmlWriter.Create("..\FilmOrdersProgrammatic.xml", myXmlSettings)

FilmOrdersWriter.Close()
```

or the following, if you are utilizing the `Using` keyword, which is new to the .NET Framework 2.0 and highly recommended:

```
Using FilmOrdersWriter As XmlTextWriter = _
   XmlWriter.Create("..\FilmOrdersProgrammatic.xml", myXmlSettings)

End Using
```

With the preliminaries completed (file created and formatting configured), the process of writing the actual attributes and elements of your XML document can begin. The sequence of steps used to generate your XML document is:

- Write an XML comment using the `WriteComment` method. This comment describes from whence the concept for this XML document originated and generates the following code:

```
<!-- Same as generated by serializing, ElokuvaTilaus -->
```

- Begin writing the XML element, `<ElokuvaTilaus>`, by calling the `WriteStartElement` method. You can only begin writing this element because its attributes and child elements must be written before the element can be ended with a corresponding `</ElokuvaTilaus>`. The XML generated by the `WriteStartElement` method is:

```
<ElokuvaTilaus>
```

- Write the attributes associated with `<ElokuvaTilaus>` by calling the `WriteAttributeString` method twice. The XML generated by calling the `WriteAttributeString` method twice adds to the `ElokuvaTilaus`XML element that is currently being written to:

```
<ElokuvaTilaus ElokuvaId="101" Maara="10">
```

- Using the `WriteElementString` method, write the child XML element `<Nimi>` contained in the XML element, `<ElokuvaTilaus>`. The XML generated by calling this method is:

```
<Nimi>Grease</Nimi>
```

- Complete writing the `<ElokuvaTilaus>` parent XML element by calling the `WriteEndElement` method. The XML generated by calling this method is:

```
</ElokuvaTilaus>
```

Let's now put all this together in the `Module1.vb` file shown here:

```
Imports System.Xml
Imports System.Xml.Serialization
Imports System.IO

Module Module1

    Sub Main()

        Dim myXmlSettings As New XmlWriterSettings
```

```
            myXmlSettings.Indent = True
            myXmlSettings.NewLineOnAttributes = True

            Using FilmOrdersWriter As XmlWriter = _
                XmlWriter.Create("..\FilmOrdersProgrammatic.xml", myXmlSettings)

                FilmOrdersWriter.WriteComment(" Same as generated " & _
                   "by serializing, ElokuvaTilaus ")
                FilmOrdersWriter.WriteStartElement("ElokuvaTilaus")
                FilmOrdersWriter.WriteAttributeString("ElokuvaId", "101")
                FilmOrdersWriter.WriteAttributeString("Maara", "10")
                FilmOrdersWriter.WriteElementString("Nimi", "Grease")
                FilmOrdersWriter.WriteEndElement() ' End ElokuvaTilaus

            End Using

        End Sub

    End Module
```

Once this is run, you will then find the XML file `FilmOrdersProgrammatic.xml` created in the same folder as the `Module1.vb` file. The content of this file is:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Same as generated by serializing, ElokuvaTilaus -->
<ElokuvaTilaus
  ElokuvaId="101"
  Maara="10">
  <Nimi>Grease</Nimi>
</ElokuvaTilaus>
```

The previous XML document is the same in form as the XML document generated by serializing the `ElokuvaTilaus` class. Notice how in the previous XML document the `<Nimi>` element is indented two characters and that each attribute is on a different line in the document? This was achieved using the `XmlWriterSettings` class.

The sample application covered only a small portion of the methods and properties exposed by the XML stream-writing class, `XmlWriter`. Other methods implemented by this class include methods that manipulate the underlying file, such as the `Flush` method, and methods that allow XML text to be written directly to the stream, such as the `WriteRaw` method.

The `XmlWriter` class also exposes a variety of methods that write a specific type of XML data to the stream. These methods include `WriteBinHex`, `WriteCData`, `WriteString`, and `WriteWhiteSpace`.

You can now generate the same XML document in two different ways. You have used two different applications that took two different approaches to generating a document that represents a standardized movie order. However, there are even more ways to generate XML, depending on the circumstances. For example, you could receive a movie order from a store, and this order would have to be transformed from the XML format used by the supplier to your own order format.

Reading an XML Stream

In .NET, XML documents can be read from a stream as well. The way a readable stream works is that data is traversed in the stream in order (first XML element, second XML element, and so on). This traversal is very quick because the data is processed in one direction, and features, such as write and move backward in the traversal, are not supported. At any given instance, only data at the current position in the stream can be accessed.

Before exploring how an XML stream can be read, you need to understand why it should be read in the first place. To answer this question, let's return to your movie supplier example. Imagine that the application that manages the movie orders can generate a variety of XML documents corresponding to current orders, preorders, and returns. All the documents (current orders, preorders, and returns) can be extracted in stream form and processed by a report-generating application. This application prints up the orders for a given day, the preorders that are going to be due, and the returns that are coming down back to the supplier. The report-generating application processes the data by reading in and parsing a stream of XML.

One class that can be used to read and parse such an XML stream is `XmlReader`. Other classes in the .NET Framework are derived from `XmlReader`, such as `XmlTextReader`, which can read XML from a file (specified by a string corresponding to the file's name), a `Stream`, or an `XmlReader`. For demonstration purposes, you will use an `XmlReader` to read an XML document contained in a file. Reading XML from a file and writing it to a file is not the norm when it comes to XML processing, but a file is the simplest way to access XML data. This simplified access allows you to focus more on XML-specific issues.

In creating a sample, the fist step is to make the proper imports into the `Module1.vb` file:

```
Imports System.Xml
Imports System.Xml.Serialization
Imports System.IO
```

From there, the next step in accessing a stream of XML data is to create an instance of the object that will open the stream (the `readMovieInfo` variable of type `XmlReader`) and then to open the stream itself. Your application performs this as follows (where `MovieManage.xml` will be the name of the file containing the XML document):

```
Dim myXmlSettings As New XmlReaderSettings()
Using readMovieInfo As XmlReader = XmlReader.Create(fileName, myXmlSettings)
```

You will notice that like the `XmlWriter` has a settings class, the `XmlReader` also has a settings class. Though you can make assignments to the `XmlReaderSettings` object, in this case you do not. Later, this chapter will detail the `XmlReaderSettings` object.

The basic mechanism for traversing each stream is to traverse from node to node using the `Read` method. Node types in XML include element and white space. Numerous other node types are defined, but for the sake of this example you will focus on traversing XML elements and the white space that is used to make the elements more readable (carriage returns, linefeeds, and indentation spaces). Once the stream is positioned at a node, the `MoveToNextAttribute` method can be called to read each attribute contained in an element. The `MoveToNextAttribute` method will only traverse attributes for nodes that contain attributes (nodes of type element). An example of an `XmlReader` traversing each node and then traversing the attributes of each node follows:

```
      While readMovieInfo.Read()
         ' Process node here.
         While readMovieInfo.MoveToNextAttribute()
            ' Process attribute here.
         End While
      End While
```

This code, which reads the contents of the XML stream, does not utilize any knowledge of the stream's contents. However, a great many applications know exactly how the stream they are going to traverse is structured. Such applications can use `XmlReader` in a more deliberate manner and not simply traverse the stream without foreknowledge.

Once the example stream has been read, it can be cleaned up using the `End Using` call:

```
   End Using
```

This `ReadMovieXml` subroutine takes the file name containing the XML to read as a parameter. The code for the subroutine is as follows and is basically the code just outlined:

```
   Private Sub ReadMovieXml(ByVal fileName As String)
      Dim myXmlSettings As New XmlReaderSettings()
      Using readMovieInfo As XmlReader = XmlReader.Create(fileName, myXmlSettings)
         While readMovieInfo.Read()
            ShowXmlNode(readMovieInfo)
            While readMovieInfo.MoveToNextAttribute()
               ShowXmlNode(readMovieInfo)
            End While
         End While
      End Using

      Console.ReadLine()
   End Sub
```

For each node encountered after a call to the `Read` method, `ReadMovieXml` calls the `ShowXmlNode` subroutine. Similarly, for each attribute traversed, the `ShowXmlNode` subroutine is called. This subroutine breaks down each node into its subentities.

- **Depth** — The `Depth` property of `XmlReader` determines the level at which a node resides in the XML document tree. To understand depth, consider the following XML document composed solely of elements: `<A><B></B><C><D></D></C></A>`. Element `<A>` is the root element and when parsed would return a `Depth` of `0`. Elements `<B>` and `<C>` are contained in `<A>` and are hence a `Depth` value of `1`. Element `<D>` is contained in `<C>`. The `Depth` property value associated with `<D>` (depth of `2`) should, therefore, be one more than the `Depth` property associated with `<C>` (depth of `1`).
- **Type** — The type of each node is determined using the `NodeType` property of `XmlReader`. The node returned is of enumeration type, `XmlNodeType`. Permissible node types include `Attribute`, `Element`, and `Whitespace`. (Numerous other node types can also be returned including `CDATA`, `Comment`, `Document`, `Entity`, and `DocumentType`.)
- **Name** — The type of each node is retrieved using the `Name` property of `XmlReader`. The name of the node could be an element name, such as `<ElokuvaTilaus>`, or an attribute name, such as `ElokuvaId`.

- **Attribute Count** — The number of attributes associated with a node is retrieved using the `AttributeCount` property of `XmlReader`'s `NodeType`.
- **Value** — The value of a node is retrieved using the `Value` property of `XmlReader`. For example, the element node `<Nimi>` contains a value of `Grease`.

Subroutine `ShowXmlNode` is implemented as follows:

```
Private Sub ShowXmlNode(ByVal reader As XmlReader)

  If reader.Depth > 0 Then
     For depthCount As Integer = 1 To reader.Depth
        Console.Write(" ")
     Next
  End If

  If reader.NodeType = XmlNodeType.Whitespace Then

     Console.Out.WriteLine("Type: {0} ", reader.NodeType)

  ElseIf reader.NodeType = XmlNodeType.Text Then

     Console.Out.WriteLine("Type: {0}, Value: {1} ", _
                           reader.NodeType, _
                           reader.Value)

  Else

     Console.Out.WriteLine("Name: {0}, Type: {1}, " & _
                           "AttributeCount: {2}, Value: {3} ", _
                           reader.Name, _
                           reader.NodeType, _
                           reader.AttributeCount, _
                           reader.Value)
  End If

End Sub
```

Within the `ShowXmlNode` subroutine, each level of node depth adds two spaces to the output generated:

```
If reader.Depth > 0 Then
  For depthCount As Integer = 1 To reader.Depth
    Console.Write(" ")
  Next
End If
```

You add these spaces in order to make the output generated human-readable (so you can easily determine the depth of each node displayed). For each type of node, `ShowXmlNode` displays the value of the `NodeType` property. The `ShowXmlNode` subroutine makes a distinction between nodes of type `Whitespace` and other types of nodes. The reason for this is simple: A node of type `Whitespace` does not contain a name or attribute count. The value of such a node is any combination of white-space characters (space, tab, carriage return, and so on). Therefore, it does not make sense to display the properties if the `NodeType` is `XmlNodeType.WhiteSpace`. Nodes of type `Text` have no name associated with them

and so for this type, subroutine `ShowXmlNode` only displays the properties `NodeType` and `Value`. For all other node types, the `Name`, `AttributeCount`, `Value`, and `NodeType` properties are displayed.

For the finalization of this module, add a `Sub Main` as follows:

```
Sub Main(ByVal args() As String)
   ReadMovieXml("..\MovieManage.xml")
End Sub
```

An example construction of the `MovieManage.xml` file is:

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

 <FilmOrder_Multiple>
    <multiFilmOrders>
       <FilmOrder>
          <name>Grease</name>
          <filmId>101</filmId>
          <quantity>10</quantity>
       </FilmOrder>
       <FilmOrder>
          <name>Lawrence of Arabia</name>
          <filmId>102</filmId>
          <quantity>10</quantity>
       </FilmOrder>
       <FilmOrder>
          <name>Star Wars</name>
          <filmId>103</filmId>
          <quantity>10</quantity>
       </FilmOrder>
    </multiFilmOrders>
 </FilmOrder_Multiple>

 <PreOrder>
    <FilmOrder>
       <name>Shrek III - Shrek Becomes a Programmer</name>
       <filmId>104</filmId>
       <quantity>10</quantity>
    </FilmOrder>
 </PreOrder>

 <Returns>
    <FilmOrder>
       <name>Star Wars</name>
       <filmId>103</filmId>
       <quantity>2</quantity>
    </FilmOrder>
 </Returns>

</MovieOrderDump>
```

Running this module produces the following output (a partial display since it would be rather lengthy):

```
Name: xml, Type: XmlDeclaration, AttributeCount: 2, Value: version="1.0"
encoding="utf-8"
Name: version, Type: Attribute, AttributeCount: 2, Value: 1.0
Name: encoding, Type: Attribute, AttributeCount: 2, Value: utf-8
Type: Whitespace
Name: MovieOrderDump, Type: Element, AttributeCount: 0, Value:
 Type: Whitespace
 Name: FilmOrder_Multiple, Type: Element, AttributeCount: 0, Value:
  Type: Whitespace
  Name: multiFilmOrders, Type: Element, AttributeCount: 0, Value:
   Type: Whitespace
   Name: FilmOrder, Type: Element, AttributeCount: 0, Value:
    Type: Whitespace
    Name: name, Type: Element, AttributeCount: 0, Value:
     Type: Text, Value: Grease
```

This example managed to use three methods and five properties of `XmlReader`. The output generated was informative but far from practical. `XmlReader` exposes over 50 methods and properties, which means that you have only scratched the surface of this highly versatile class. The remainder of this section will look at the `XmlReaderSettings` class, introduce a more realistic use of `XmlReader`, and demonstrate how the classes of `System.Xml` handle errors.

The XmlReaderSettings Class

Just like the `XmlWriter` object, the `XmlReader` object requires settings to be applied for instantiation of the object. This means that you can apply settings for how the `XmlReader` object behaves for when it is reading whatever XML that you might have for it. This includes settings for how to deal with white space, schemas, and more. The following table details these settings.

Property	Initial Value	Description
`CheckCharacters`	`True`	This property, if set to `True`, will perform a character check upon the contents of the retrieved object. Legal characters can be found at `www.w3.org/TR/REC-xml#charsets`.
`CloseInput`	`False`	This property gets or sets a value indicating whether the underlying stream or `System.IO.TextReader` should be closed when the reader is closed.
`ConformanceLevel`	`ConformanceLevel.Document`	Allows for the XML to be checked to make sure that it follows certain specified rules. Possible conformance level settings include `Document`, `Fragment`, and `Default`.
`DtdValidate`	`False`	Defines whether the `XmlReader` should perform a DTD validation.

Table continued on following page

Property	Initial Value	Description
`IgnoreComments`	`False`	Defines whether comments should be ignored or not.
`IgnoreInlineSchema`	`True`	Defines whether any inline schemas should be ignored or not.
`IgnoreProcessing Instructions`	`False`	Defines whether processing instructions contained within the XML should be ignored.
`IgnoreSchema Location`	`True`	Defines whether the `xsi: schemaLocation` or `xsi: noNamespaceSchemaLocation` attributes should be ignored or not.
`IgnoreValidation Warnings`	`True`	Defines whether the `XmlReader` object should ignore all validation warnings.
`IgnoreWhitespace`	`False`	Defines whether the `XmlReader` object should ignore all insignificant white space.
`LineNumberOffset`	`0`	Defines the line number which the `LineNumber` property starts counting within the XML file.
`LinePositionOffset`	`0`	Defines the line number which the `LineNumber` property starts counting with the XML file.
`NameTable`	An empty `XmlNameTable` object	Allows the `XmlReader` to work with a specific `XmlNameTable` object that is used for atomized string comparisons.
`ProhibitDtd`	`True`	This property gets or sets a value indicating whether to prohibit document type definition (DTD) processing.
`Schemas`	An empty `XmlSchemaSet` object	Allows the `XmlReader` to work with an instance of the `XmlSchemaSet` class.
`ValidationFlags`		This property gets or sets a value indicating the schema validation settings.
`ValidationType`	`ValidationType .None`	This property gets or sets a value indicating whether the `System.Xml .XmlReader` will perform validation or type assignment when reading.
`XmlResolver`	A new `XmlResolver` with no credentials	This property sets the `XmlResolver` to access external documents.

An example of using this setting class to modify the behavior of the XmlReader class is:

```
Dim myXmlSettings As New XmlReaderSettings()
myXmlSettings.IgnoreWhitespace = True
myXmlSettings.IgnoreComments = True

Using readMovieInfo As XmlReader = XmlReader.Create(fileName, myXmlSettings)
   ' Use XmlReader object here.
End Using
```

In this case, the `XmlReader` object that is created will behave in that it will ignore the white space that it encounters as well as ignoring any of the XML comments. These settings, once established with the `XmlReaderSettings` object are then associated to the `XmlReader` object through its `Create` method.

Traversing XML Using XmlTextReader

An application can easily use `XmlReader` to traverse a document that is received in a known format. The document can thus be traversed in a deliberate manner. You implemented a class that serialized arrays of movie orders. The next example will take an XML document containing multiple XML documents of that type and traverse them. Each movie order will be forwarded to the movie supplier by sending a fax. The document will be traversed as follows:

```
Read root element: <MovieOrderDump>
    Process each <FilmOrder_Multiple> element
        Read <multiFilmOrders> element
            Process each <FilmOrder>
                Send fax for each movie order here
```

The basic outline for the program's implementation is to open a file containing the XML document to parse and to traverse it from element to element.

```
Dim myXmlSettings As New XmlReaderSettings()

Using readMovieInfo As XmlReader = XmlReader.Create(fileName, myXmlSettings)
      readMovieInfo.Read()
      readMovieInfo.ReadStartElement("MovieOrderDump")

      Do While (True)

         '*************************************************
         '* Process FilmOrder elements here               *
         '*************************************************

      Loop

      readMovieInfo.ReadEndElement()  '  </MovieOrderDump>

End Using
```

The previous code opened the file using the constructor of `XmlReader`, and the `End Using` statement takes care of shutting everything down for you. The previous code also introduced two methods of the `XmlReader` class:

- `ReadStartElement(String)` — This verifies that the current in the stream is an element and that the element's name matches the string passed to method `ReadStartElement`. If the verification is successful, the stream is advanced to the next element.
- `ReadEndElement()` — This verifies that the current element is an end tab, and if the verification is successful the stream is advanced to the next element.

The application knows that an element, `<MovieOrderDump>`, will be found at a specific point in the document. The `ReadStartElement` method verifies this foreknowledge of the document format. Once all the elements contained in element `<MovieOrderDump>` have been traversed, the stream should point to the end tag `</MovieOrderDump>`. The `ReadEndElement` method verifies this.

The code that traverses each element of type `<FilmOrder>` similarly uses the `ReadStartElement` and `ReadEndElement` methods to indicate the start and end of the `<FilmOrder>` and `<multiFilmOrders>` elements. The code that ultimately parses the list of prescription and faxes the movie supplier (using the `FranticallyFaxTheMovieSupplier` subroutine) is:

```
Dim myXmlSettings As New XmlReaderSettings()

Using readMovieInfo As XmlReader = XmlReader.Create(fileName, myXmlSettings)
      readMovieInfo.Read()
      readMovieInfo.ReadStartElement("MovieOrderDump")

      Do While (True)
         readMovieInfo.ReadStartElement("FilmOrder_Multiple")
         readMovieInfo.ReadStartElement("multiFilmOrders")

         Do While (True)
            readMovieInfo.ReadStartElement("FilmOrder")
            movieName = readMovieInfo.ReadElementString()
            movieId = readMovieInfo.ReadElementString()
            quantity = readMovieInfo.ReadElementString()
            readMovieInfo.ReadEndElement() ' clear </FilmOrder>

            FranticallyFaxTheMovieSupplier(movieName, movieId, quantity)

            ' Should read next FilmOrder node
            ' else quits
            readMovieInfo.Read()

            If ("FilmOrder" <> readMovieInfo.Name) Then
               Exit Do
            End If
         Loop

         readMovieInfo.ReadEndElement() ' clear </multiFilmOrders>
         readMovieInfo.ReadEndElement() ' clear </FilmOrder_Multiple>

         ' Should read next FilmOrder_Multiple node
         ' else you quit
         readMovieInfo.Read() ' clear </MovieOrderDump>

         If ("FilmOrder_Multiple" <> readMovieInfo.Name) Then
            Exit Do
         End If
```

```
        Loop

        readMovieInfo.ReadEndElement()  '  </MovieOrderDump>

    End Using
```

Three lines within the previous code contain a call to the `ReadElementString` method:

```
movieName = readMovieInfo.ReadElementString()
movieId = readMovieInfo.ReadElementString()
quantity = readMovieInfo.ReadElementString()
```

While parsing the stream, it was known that an element named `<name>` existed and that this element contained the name of the movie. Rather than parsing the start tag, getting the value, and parsing the end tag, it was easier just to get the data using the `ReadElementString` method. This method retrieves the data string associated with an element and advances the stream to the next element. The `ReadElementString` method was also used to retrieve the data associated with the XML elements `<filmId>` and `<quantity>`.

The output of this example was a fax, which we won't show because the emphasis of this example is on showing that it is simpler to traverse a document when its form is known. The format of the document is still verified by `XmlReader` as it is parsed.

The `XmlReader` class also exposes properties that give more insight into the data contained in the XML document and the state of parsing: `IsEmptyElement`, `EOF`, and `IsStartElement`. This class also allows data in a variety of forms to be retrieved using methods such as `ReadBase64`, `ReadHex`, and `ReadChars`. The raw XML associated with the document can also be retrieved, using `ReadInnerXml` and `ReadOuterXml`. Once again, you have only scratched the surface of the `XmlReader` class. You will find this class to be quite rich in functionality.

Handling Exceptions

XML is text and could easily be read using mundane methods such as `Read` and `ReadLine`. A key feature of each class that reads and traverses XML is inherent support for error detection and handling. To demonstrate this, consider the following malformed XML document found in the file named `malformed.XML`:

```
<?xml version="1.0" encoding="IBM437" ?>
<ElokuvaTilaus ElokuvaId="101", Maara="10">
   <Nimi>Grease</Nimi>
<ElokuvaTilaus>
```

This document may not immediately appear to be malformed. By wrapping a call to the method you developed (`movieReadXML`), you can see what type of exception is raised when `XmlReader` detects the malformed XML within this document:

```
Try
    movieReadXML("..\Malformed.xml")
Catch xmlEx As XmlException
    Console.Error.WriteLine("XML Error: " + xmlEx.ToString())
Catch ex As Exception
    Console.Error.WriteLine("Some other error: " + ex.ToString())
End Try
```

The methods and properties exposed by the `XmlReader` class raise exceptions of type `System.Xml.XmlException`. In fact, every class in the `System.Xml` namespace raises exceptions of type `XmlException`. Although this is a discussion of errors using an instance of type `XmlReader`, the concepts reviewed apply to all errors generated by classes found in the `System.Xml` namespace.

The properties exposed by `XmlException` include

- `LineNumber` — The number of the line within an XML document where the error occurred.
- `LinePosition` — The position within the line specified by `LineNumber` where the error occurred.
- `Message` — The error message that corresponds to the error that occurred. This error took place at the line in the XML document specified by `LineNumber` and within the line at the position specified by `LinePostion`.
- `SourceUri` — Provides the URI of the element or document in which the error occurred.

The error displayed when subroutine `movieReadXML` processes `malformed.xml` is:

```
XML Error: System.Xml.XmlException: The ',' character, hexadecimal value 0x2C,
 cannot begin a name. Line 2, position 49.
```

Looking closely at the document, there is a comma separating the attributes in element, `<FilmOrder>(ElokuvaTilaus="101", Maara="10")`. This comma is invalid. Removing the comma and running the code again gives the following output:

```
XML Error: System.Xml.XmlException: This is an unexpected token. Expected
'EndElement'. Line 5, position 27.
```

Once again, you can recognize the precise error. In this case, you do not have an end element, `</ElokuvaTilaus>`, but you do have an opening element, `<ElokuvaTilaus>`.

The properties provided by the `XmlException` class (`LineNumber`, `LinePosition`, and `Message`) provide a useful level of precision when tracking down errors. The `XmlReader` class also exposes a level of precision with respect to the parsing of the XML document. This precision is exposed by the `XmlReader` through properties such as `LineNumber` and `LinePosition`.

Using the MemoryStream Object

A very useful class that can greatly help you when working with XML is `System.IO.MemoryStream`. Rather than needing a network or disk resource backing the stream (as in `System.Net.Sockets.NetworkStream` and `System.IO.FileStream`), `MemoryStream` backs itself onto a block of memory. Imagine that you want to generate an XML document and email it. The built-in classes for sending email rely on having a `System.String` containing a block of text for the message body. But, if you want to generate an XML document, you need a stream.

If the document is reasonably sized, you should write the document directly to memory and copy that block of memory to the email. This is good from a performance and reliability perspective because you don't have to open a file, write it, rewind it, and read the data back in again. However, you must consider scalability in this situation because if the file is very large, or you have a great number of smaller files, you could run out of memory (in which case you'll have to go the "file" route).

In this section, you'll see how to generate an XML document to a `MemoryStream` object. You'll read the document back out again as a `System.String` value and email it. What you'll do is create a new class called `EmailStream` that extends `MemoryStream`. This new class will contain an extra method called `CloseAndSend` that, as its name implies, will close the stream and send the email message.

First, you'll create a new console application project called `EmailStream`. The first job is to create a basic `Customer` object that contains a few basic members and that can be automatically serialized by .NET through use of the `SerializableAttribute` attribute:

```
<Serializable()> Public Class Customer

  ' members...
  Public Id As Integer
  Public FirstName As String
  Public LastName As String
  Public Email As String

End Class
```

The fun part now is the `EmailStream` class itself. This needs access to the `System.Web.Mail` namespace, so you'll need to add a reference to the `System.Web` assembly. The new class should also extend `System.IO.MemoryStream`, as shown here:

```
Imports System.IO
Imports System.Web.Mail

Public Class EmailStream
    Inherits MemoryStream
```

The first job of `CloseAndSend` is to start putting together the mail message. This is done by creating a new `System.Web.Mail.MailMessage` object and configuring the sender, recipient, and subject.

```
 ' CloseAndSend - close the stream and send the email...
Public Sub CloseAndSend(ByVal fromAddress As String, _
                        ByVal toAddress As String, _
                        ByVal subject As String)

   ' Create the new message...
   Dim message As New MailMessage
   message.From = fromAddress
   message.To = toAddress
   message.Subject = subject
```

This method will be called once the XML document has been written to the stream, so you can assume at this point that the stream contains a block of data. To read the data back out again, you have to rewind the stream and use a `System.IO.StreamReader`. Before you do this, the first thing you should do is call `Flush`. Traditionally, streams have always been buffered, that is, the data is not sent to the final destination (the memory block in this case, but a file in the case of a `FileStream` and so on) each and every time the stream is written. Instead, the data is written in (pretty much) a nondeterministic way. Because you need all the data to be written, you call `Flush` to ensure that all the data has been sent to the destination and that the buffer is empty.

In a way, `EmailStream` is a great example of buffering. All of the data is held in a memory "buffer" until you finally send the data on to its destination in a response to an explicit call to this method:

```
    ' Flush and rewind the stream...

    Flush()
    Seek(0, SeekOrigin.Begin)
```

Once you've flushed and rewound the stream, you can create a `StreamReader` and dredge all the data out into the `Body` property of the `MailMessage` object:

```
    ' Read out the data...

    Dim reader As New StreamReader(Me)
    message.Body = reader.ReadToEnd()
```

After you've done that, you close the stream by calling the base class method:

```
    ' Close the stream...

    Close()
```

Finally, you send the message:

```
    ' Send the message...

    SmtpMail.Send(message)

End Sub
```

To call this method, you need to add some code to the `Main` method. First, you create a new `Customer` object and populate it with some test data:

```
Imports System.Xml.Serialization

Module Module1

  Sub Main()

    ' Create a new customer...
    Dim customer As New Customer
    customer.Id = 27
    customer.FirstName = "Bill"
    customer.LastName = "Gates"
    customer.Email = "bill.gates@microsoft.com"
```

After you've done that, you can create a new `EmailStream` object. You then use `XmlSerializer` to write an XML document representing the newly created `Customer` instance to the block of memory that `EmailStream` is backing to:

```
        ' Create a new email stream...
        Dim stream As New EmailStream

        ' Serialize...
        Dim serializer As New XmlSerializer(customer.GetType())
        serializer.Serialize(stream, customer)
```

At this point, the stream will be filled with data, and after all the data has been flushed, the block of memory that `EmailStream` backs on to will contain the complete document. Now, you can call `CloseAndSend` to email the document.

```
        ' Send the email...
        stream.CloseAndSend("evjen@yahoo.com", _
           "evjen@yahoo.com", "XML Customer Document")

    End Sub

End Module
```

You probably already have Microsoft SMTP Service properly configured — this service is necessary to send email. You also need to make sure that the email addresses used in your code goes to your email address! Run the project, check your email, and you should see something, as shown in Figure 12-2.

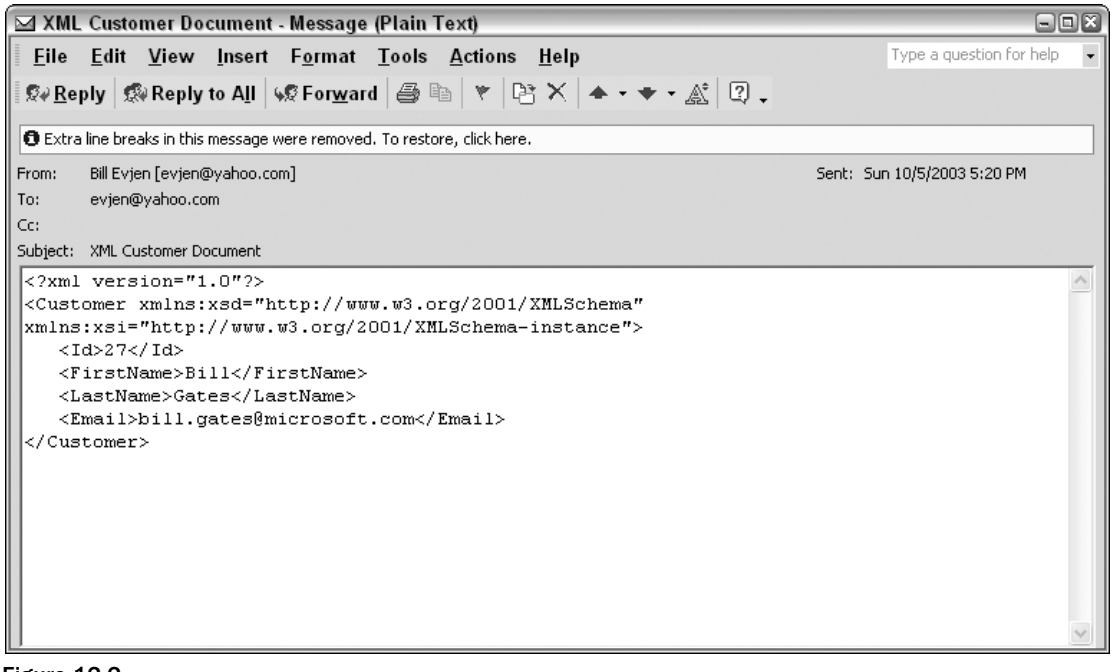

Figure 12-2

Document Object Model (DOM)

The classes of the `System.Xml` namespace that support the Document Object Model (DOM) interact as illustrated in Figure 12-3.

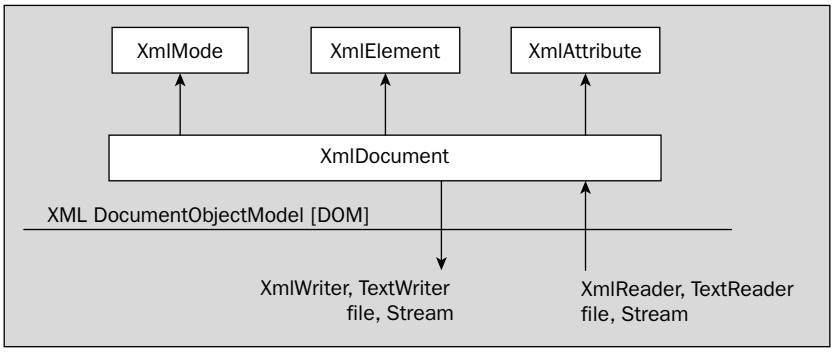

Figure 12-3

Within this diagram, an XML document is contained in a class named `XmlDocument`. Each node within this document is accessible and managed using `XmlNode`. Nodes can also be accessed and managed using a class specifically designed to process a specific node's type (`XmlElement`, `XmlAttribute`, and so on). XML documents are extracted from `XmlDocument` using a variety of mechanisms exposed through such classes as `XmlWriter`, `TextWriter`, `Stream`, and a file (specified by file name of type `String`). XML documents are consumed by an `XmlDocument` using a variety of load mechanisms exposed through the same classes.

Where a DOM-style parser differs from a stream-style parser is with respect to movement. Using DOM, the nodes can be traversed forward and backward. Nodes can be added to the document, removed from the document, and updated. However, this flexibility comes at a performance cost. It is faster to read or write XML using a stream-style parser.

The DOM-specific classes exposed by `System.Xml` include

- `XmlDocument` — Corresponds to an entire XML document. A document is loaded using the `Load` method. XML documents are loaded from a file (the file name specified as type `String`), `TextReader`, or `XmlReader`. A document can be loaded using `LoadXml` in conjunction with a string containing the XML document. The `Save` method is used to save XML documents. The methods exposed by `XmlDocument` reflect the intricate manipulation of an XML document. For example, the following self-documenting creation methods are implemented by this class: `CreateAttribute`, `CreateDataSection`, `CreateComment`, `CreateDocumentFragment`, `CreateDocumentType`, `CreateElement`, `CreateEntityReference`, `CreateNode`, `CreateProcessingInstruction`, `CreateSignificantWhitespace`, `CreateTextNode`, `CreateWhitespace`, and `CreateXmlDeclaration`. The elements contained in the document can be retrieved. Other methods support the retrieving, importing, cloning, loading, and writing of nodes.
- `XmlNode` — Corresponds to a node within the DOM tree. This class supports datatypes, namespaces, and DTDs. A robust set of methods and properties are provided to create, delete, and replace nodes: `AppendChild`, `CloneNode`, `InsertAfter`, `InsertBefore`, `PrependChild`,

`RemoveAll`, `RemoveChild`, and `ReplaceChild`. The contents of a node can similarly be traversed in a variety of ways: `FirstChild`, `LastChild`, `NextSibling`, `ParentNode`, and `PreviousSibling`.

- `XmlElement` — Corresponds to an element within the DOM tree. The functionality exposed by this class contains a variety of methods used to manipulate an element's attributes: `GetAttribute`, `GetAttributeNode`, `RemoveAllAttributes`, `RemoveAttributeAt`, `RemoveAttributeNode`, `SetAttribute`, and `SetAttributeNode`.
- `XmlAttribute` — Corresponds to an attribute of an element (`XmlElement`) within the DOM tree. An attribute contains data and lists of subordinate data. For this reason, it is a less complicated object than an `XmlNode` or an `XmlElement`. An `XmlAttribute` can retrieve its owner document (property, `OwnerDocument`), retrieve its owner element (property, `OwnerElement`), retrieve its parent node (property, `ParentNode`), and retrieve its name (property, `Name`). The value of an `XmlAttribute` is available via a read-write property named `Value`.

Given the diverse number of methods and properties (and there are many more than those listed here) exposed by `XmlDocument`, `XmlNode`, `XmlElement`, and `XmlAttribute`, it should be clear that any XML 1.0–compliant document can be generated and manipulated using these classes. In comparison to their XML stream counterparts, these classes afford more flexible movement within and editing of XML documents.

A similar comparison could be made between DOM and data serialized and deserialized using XML. Using serialization, the type of node (for example, attribute or element) and the node name are specified at compile time. There is no on-the-fly modification of the XML generated by the serialization process.

Other technologies that generate and consume XML are not as flexible as DOM. This includes ADO.NET and ADO, which generate XML of a particular form. Out of the box, SQL Server 2000 does expose a certain amount of flexibility when it comes to the generation (`FOR XML` queries) and consumption of XML (`OPENXML`) . SQL Server 2005 has more support from XML and even supports an XML datatype. SQL Server 2005 also expands upon the `FOR XML` query with `FOR XML TYPE`. The choice between using classes within DOM and a version of SQL Server is a choice between using a language, such as Visual Basic, to manipulate objects or installing SQL Server and performing most of the XML manipulation in SQL.

DOM Traversing Raw XML Elements

The first DOM example will load an XML document into an `XmlDocument` object using a string that contains the actual XML document. This scenario is typical of an application that uses ADO.NET to generate XML but then uses the objects of DOM to traverse and manipulate this XML. ADO.NET's `DataSet` object contains the results of ADO.NET data access operations. The `DataSet` class exposes a `GetXml` method. This method retrieves the underlying XML associated with the `DataSet`. The following code demonstrates how the contents of the `DataSet` are loaded into the `XmlDocument`:

```
Dim xmlDoc As New XmlDocument
Dim ds As New DataSet

' Set up ADO.NET DataSet() here
xmlDoc.LoadXml(ds.GetXml())
```

This example will simply traverse each XML element (`XmlNode`) in the document (`XmlDocument`) and display the data accordingly. The data associated with this example will not be retrieved from a `DataSet` but will instead be contained in a string, `rawData`. This string is initialized as follows:

```
Dim rawData As String = _
    "<multiFilmOrders>" & _
    "  <FilmOrder>" & _
    "    <name>Grease</name>" & _
    "    <filmId>101</filmId>" & _
    "    <quantity>10</quantity>" & _
    "  </FilmOrder>" & _
    "  <FilmOrder>" & _
    "    <name>Lawrence of Arabia</name>" & _
    "    <filmId>102</filmId>" & _
    "    <quantity>10</quantity>" & _
    "  </FilmOrder>" & _
    "</multiFilmOrders>"
```

The XML document in `rawData` is a portion of the XML hierarchy associated with a prescription written at your dental office. The basic idea in processing this data is to traverse each `<FilmOrder>` element in order to display the data it contains. Each node corresponding to a `<FilmOrder>` element can be retrieved from your `XmlDocument` using the `GetElementsByTagName` method (specifying a tag name of `FilmOrder`). The `GetElementsByTagName` method returns a list of `XmlNode` objects in the form of a collection of type `XmlNodeList`. Using the `For Each` statement to construct this list, the `XmlNodeList` (`movieOrderNodes`) can be traversed as individual `XmlNode` elements (`movieOrderNode`). The code for handling this is as follows:

```
Dim xmlDoc As New XmlDocument
Dim movieOrderNodes As XmlNodeList
Dim movieOrderNode As XmlNode

xmlDoc.LoadXml(rawData)

' Traverse each <FilmOrder>
movieOrderNodes = xmlDoc.GetElementsByTagName("FilmOrder")

For Each movieOrderNode In movieOrderNodes

    '*******************************************************
    ' Process <name>, <filmId> and <quantity> here
    '*******************************************************

Next
```

Each `XmlNode` can then have its contents displayed by traversing the children of this node using the `ChildNodes` method. This method returns an `XmlNodeList` (`baseDataNodes`) that can be traversed one `XmlNode` list element at a time:

```
Dim baseDataNodes As XmlNodeList
Dim bFirstInRow As Boolean

baseDataNodes = movieOrderNode.ChildNodes
bFirstInRow = True
For Each baseDataNode As XmlNode In baseDataNodes
```

```
    If (bFirstInRow) Then
      bFirstInRow = False
    Else
      Console.Out.Write(", ")
    End If
    Console.Out.Write(baseDataNode.Name & ": " & baseDataNode.InnerText)
  Next
  Console.Out.WriteLine()
```

The bulk of the previous code retrieves the name of the node using the `Name` property and the `InnerText` property of the node. The `InnerText` property of each `XmlNode` retrieved contains the data associated with the XML elements (nodes) `<name>`, `<filmId>`, and `<quantity>`. The example displays the contents of the XML elements using `Console.Out`. The XML document is displayed as follows:

```
name: Grease, filmId: 101, quantity: 10
name: Lawrence of Arabia, filmId: 102, quantity: 10
```

Other, more practical, methods for using this data could have been implemented, including:

- The contents could have been directed to an ASP.NET `Response` object. The data retrieved could have been used to create an HTML table (`<table>` table, `<tr>` row, and `<td>` data) that would be written to the `Response` object.
- The data traversed could have been directed to a `ListBox` or `ComboBox` Windows Forms control. This would allow the data returned to be selected as part of a GUI application.
- The data could have been edited as part of your application's business rules. For example, you could have used the traversal to verify that the `<filmId>` matched the `<name>`. For example, if you really wanted to validate the data entered into the XML document in any manner.

The example in its entirety is:

```
Dim rawData As String = _
    "<multiFilmOrders>" & _
    "  <FilmOrder>" & _
    "    <name>Grease</name>" & _
    "    <filmId>101</filmId>" & _
    "    <quantity>10</quantity>" & _
    "  </FilmOrder>" & _
    "  <FilmOrder>" & _
    "    <name>Lawrence of Arabia</name>" & _
    "    <filmId>102</filmId>" & _
    "    <quantity>10</quantity>" & _
    "  </FilmOrder>" & _
    "</multiFilmOrders>"

Dim xmlDoc As New XmlDocument
Dim movieOrderNodes As XmlNodeList
Dim movieOrderNode As XmlNode
Dim baseDataNodes As XmlNodeList
Dim bFirstInRow As Boolean

xmlDoc.LoadXml(rawData)
' Traverse each <FilmOrder>
movieOrderNodes = xmlDoc.GetElementsByTagName("FilmOrder")
```

```
For Each movieOrderNode In movieOrderNodes
  baseDataNodes = movieOrderNode.ChildNodes
  bFirstInRow = True
  For Each baseDataNode As XmlNode In baseDataNodes
    If (bFirstInRow) Then
      bFirstInRow = False
    Else
      Console.Out.Write(", ")
    End If
    Console.Out.Write(baseDataNode.Name & ": " & baseDataNode.InnerText)
  Next
  Console.Out.WriteLine()
Next
```

DOM Traversing XML Attributes

This next example will demonstrate how to traverse data contained in attributes and how to update the attributes based on a set of business rules. In this example, the `XmlDocument` object is populated by retrieving an XML document from a file. After the business rules edit the object, the data will be persisted back to the file.

```
Dim xmlDoc As New XmlDocument

xmlDoc.Load("..\MovieSupplierShippingListV2.xml")

'*****************************************
' Business rules process document here

'*****************************************
xmlDoc.Save("..\MovieSupplierShippingListV2.xml")
```

The data contained in the file, `MovieSupplierShippingListV2.xml`, is a variation of the dental prescription. You have altered your rigid standard (for the sake of example) so that the data associated with individual movie orders is contained in XML attributes instead of XML elements. An example of this movie order data is:

```
<FilmOrder name="Grease" filmId="101" quantity="10" />
```

You have already seen how to traverse the XML elements associated with a document, so let's assume that you have successfully retrieved the `XmlNode` associated with the `<FilmOrder>` element.

```
Dim attributes As XmlAttributeCollection
Dim filmId As Integer
Dim quantity As Integer

attributes = node.Attributes()
For Each attribute As XmlAttribute In attributes
  If 0 = String.Compare(attribute.Name, "filmId") Then
    filmId = attribute.InnerXml
  ElseIf 0 = String.Compare(attribute.Name, "quantity") Then
    quantity = attribute.InnerXml
  End If
Next
```

The previous code traverses the attributes of an `XmlNode` by retrieving a list of attributes using the `Attributes` method. The value of this method is used to set the attributes object (datatype, `XmlAttributeCollection`). The individual `XmlAttribute` objects (variable, `attribute`) contained in attributes are traversed using a `For Each` loop. Within the loop, the contents of the `filmId` and the `quantity` attribute are saved for processing by your business rules.

Your business rules execute an algorithm that ensures that the movies in the company's order are provided in the correct quantity. This rule is that the movie associated with `filmId=101` must be sent to the customer in batches of six at a time due to packaging. In the event of an invalid quantity, the code for enforcing this business rule keeps removing a single order from the `quantity` value until the number is divisible by six. Then this number is assigned to the `quantity` attribute. The `Value` property of the `XmlAttribute` object is used to set the correct value of the order's quantity. The code performing this business rule is:

```
If filmId = 101 Then
  ' This film comes packaged in batches of six.
  Do Until (quantity / 6) = True
    quantity -= 1
  Loop

  Attributes.ItemOf("quantity").Value = quantity
End If
```

What is elegant about this example is that the list of attributes was traversed using `For Each`. Then `ItemOf` was used to look up a specific attribute that had already been traversed. This would not have been possible by reading an XML stream with an object derived from the XML stream reader class, `XmlReader`.

You can use this code as follows:

```
Sub TraverseAttributes(ByRef node As XmlNode)
    Dim attributes As XmlAttributeCollection
    Dim filmId As Integer
    Dim quantity As Integer
        attributes = node.Attributes()
    For Each attribute As XmlAttribute In attributes
        If 0 = String.Compare(attribute.Name, "filmId") Then
            filmId = attribute.InnerXml
        ElseIf 0 = String.Compare(attribute.Name, "quantity") Then
            quantity = attribute.InnerXml
        End If
    Next

    If filmId = 101 Then
       ' This film comes packaged in batches of six
       Do Until (quantity / 6) = True
          quantity -= 1
       Loop

       Attributes.ItemOf("quantity").Value = quantity
```

```
        End If

    End Sub

    Sub WXReadMovieDOM()

        Dim xmlDoc As New XmlDocument
        Dim movieOrderNodes As XmlNodeList
        xmlDoc.Load("..\MovieSupplierShippingListV2.xml")

        ' Traverse each <FilmOrder>
        movieOrderNodes = xmlDoc.GetElementsByTagName("FilmOrder")

        For Each movieOrderNode As XmlNode In movieOrderNodes
            TraverseAttributes(movieOrderNode)
        Next

        xmlDoc.Save("..\MovieSupplierShippingListV2.xml")

    End Sub
```

XSLT Transforms

XSLT is a language that is used to transform XML documents so that they can be presented visually. You have performed a similar task before. When working with XML serialization, you rewrote the `FilmOrder` class. This class was used to serialize a movie order object to XML using nodes that contained English-language names. The rewritten version of this class, `ElokuvaTilaus`, serialized XML nodes containing Finnish names. Source Code Style attributes were used in conjunction with the `XmlSerializer` class to accomplish this transformation. Two words in this paragraph send chills down the spine of any experienced developer: rewrote and rewritten. The point of an XSL Transform is to use an alternate language (XSLT) to transform the XML rather than rewriting the source code, SQL commands, or some other mechanism used to generate XML.

Conceptually, XSLT is straightforward. A file with an .xslt extension describes the changes (transformations) that will be applied to a particular XML file. Once this is completed, an XSLT processor is provided with the source XML file and the XSLT file, and performs the transformation. The `System.Xml.Xsl.XslTransform` class is such an XSLT processor. A new processor in .NET 2.0 is the `XslCompiledTransform` object found at `System.Xml.XslCompiledTransform`. You will take a look at using both of these processors.

The XSLT file is itself an XML document, although certain elements within this document are XSLT-specific commands. There are dozens of XSLT commands that can be used in writing an XSLT file. In the first example, you will explore the following XSLT elements (commands):

- `stylesheet` — This element indicates the start of the style sheet (XSL) in the XSLT file.
- `template` — This element denotes a reusable template for producing specific output. This output is generated using a specific node type within the source document under a specific context. For example, the text `<xsl: template match="/">` selects all root notes (`"/"`) for the specific transform template.

- `for-each` — This element applies the same template to each node in the specified set. Recall that you demonstrated a class (`FilmOrder_Multiple`) that could be serialized. This class contained an array of prescriptions. Given the XML document generated when a `FilmOrder_Multiple` is serialized, each prescription serialized could be processed using `<xsl:for-each select = "FilmOrder_Multiple/multiFilmOrders/FilmOrder">`.
- `value-of` — This element retrieves the value of the specified node and inserts it into the document in text form. For example, `<xsl:value-of select="name" />` would take the value of XML element, `<name>`, and insert it into the transformed document.

The `FilmOrder_Multiple` class when serialized generates XML such as the following (where . . . indicates where additional `<FilmOrder>` elements may reside):

```
<?xml version="1.0" encoding="you-ascii" ?>
<FilmOrder_Multiple>
    <multiFilmOrders>
        <FilmOrder>
            <name>Grease</name>
            <filmId>101</filmId>
            <quantity>10</quantity>
        </FilmOrder>
        ...
    </multiFilmOrders>
</FilmOrder_Multiple>
```

The previous XML document is used to generate a report that is viewed by the manager of the movie supplier. This report is in HTML form, so that it can be viewed via the Web. The XSLT elements you previously reviewed (`stylesheet`, `template`, and `for-each`) are all the XSLT elements required to transform the XML document (in which data is stored) into an HTML file (show that the data can be displayed). An XSLT file `DisplayThatPuppy.xslt` contains the following text that is used to transform a serialized version, `FilmOrder_Multiple`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
 <xsl:template match="/">
      <HTML>
      <TITLE>What people are ordering</TITLE>
      <BODY>
          <TABLE BORDER="1">
            <TR>
              <TD><B>Film Name</B></TD>
              <TD><B>Film ID</B></TD>
              <TD><B>Quantity</B></TD>
            </TR>
            <xsl:for-each select=
             "FilmOrder_Multiple/multiFilmOrders/FilmOrder">
            <TR>
               <TD><xsl:value-of select="name" /></TD>
               <TD><xsl:value-of select="filmId" /></TD>
               <TD><xsl:value-of select="quantity" /></TD>
            </TR>
            </xsl:for-each>
```

```
        </TABLE>
      </BODY>
    </HTML>
  </xsl:template>
</xsl:stylesheet>
```

In the previous XSLT file, the XSLT elements are marked in boldface. These elements perform operations on the source XML file containing a serialized `FilmOrder_Multiple` object and generate the appropriate HTML file. Your file contains a table (marked by the table tag, `<TABLE>`) that contains a set of rows (each row marked by a table row tag, `<TR>`). The columns of the table are contained in table data tags, `<TD>`. The previous XSLT file contains the header row for the table:

```
<TR>
    <TD><B>Film Name</B></TD>
    <TD><B>Film ID</B></TD>
    <TD><B>Quantity</B></TD>
</TR>
```

Each row containing data (an individual prescription from the serialized object, `FilmOrder_Multiple`) is generated using the XSLT element, `for-each`, to traverse each `<FilmOrder>` element within the source XML document:

```
<xsl:for-each select=
    "FilmOrder_Multiple/multiFilmOrders/FilmOrder">
```

The individual columns of data are generated using the `value-of` XSLT element, in order to query the elements contained within each `<FilmOrder>` element (`<name>`, `<filmId>`, and `<quantity>`):

```
<TR>
    <TD><xsl:value-of select="name" /></TD>
    <TD><xsl:value-of select="filmId" /></TD>
    <TD><xsl:value-of select="quantity" /></TD>
</TR>
```

The code to create a displayable XML file using the `XslTransform` object is:

```
Dim myXslTransform As XslTransform = New XslTransform

Dim destFileName As String = "..\ShowIt.html"

myXslTransform.Load("..\DisplayThatPuppy.xslt")
myXslTransform.Transform("..\FilmOrders.xml", destFileName)

System.Diagnostics.Process.Start(destFileName)
```

This consists of only seven lines of code with the bulk of the coding taking place in the XSLT file. Your previous code snippet created an instance of a `System.Xml.Xsl.XslTransform` object named `myXslTransform`. The `Load` method of this class is used to load the XSLT file you previously reviewed, `DisplayThatPuppy.xslt`. The `Transform` method takes a source XML file as the first parameter, which in this case was a file containing a serialized `FilmOrder_Multiple` object. The second parameter is the

destination file that will be created by the transform (file name `ShowIt.html`). The `Start` method of the `Process` class is used to display the HTML file. The `Start` method launches a process that is most suitable for displaying the file provided. Basically, the extension of the file dictates which application will be used to display the file. On a typical Windows machine, the program used to display this file is Internet Explorer, as shown in Figure 12-4.

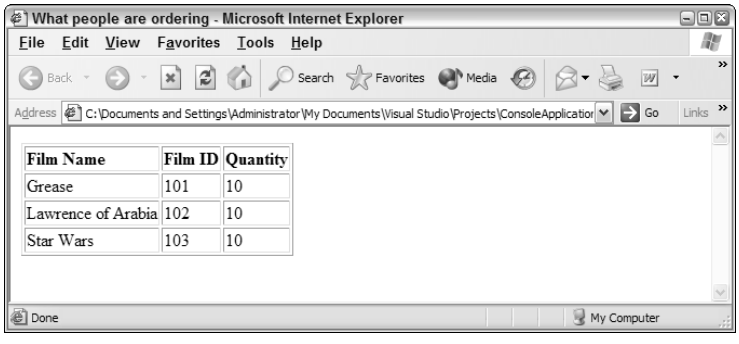

Figure 12-4

Do not confuse displaying this HTML file with ASP.NET. Displaying an HTML file in this manner takes place on a single machine without the involvement of a Web server. Using ASP.NET is more complex than displaying an HTML page in the default browser.

As was demonstrated, the backbone of the `System.Xml.Xsl` namespace is the `XslTransform` class. This class uses XSLT files to transform XML documents. `XslTransform` exposes the following methods and properties:

- `XmlResolver` — This get/set property is used to specify a class (abstract base class, `XmlResolver`) that is used to handle external references (import and include elements within the style sheet). These external references are encountered when a document is transformed (method, `Transform`, is executed). The `System.Xml` namespace contains a class, `XmlUrlResolver`, which is derived from `XmlResolver`. The `XmlUrlResolver` class resolves the external resource based on a URI.
- `Load` — This overloaded method loads an XSLT style sheet to be used in transforming XML documents. It is permissible to specify the XSLT style sheet as a parameter of type `XPathNavigator`, file name of XSLT file (specified as parameter type, `String`), `XmlReader`, or `IXPathNavigable`. For each of type of XSLT supported, an overloaded member is provided that allows an `XmlResolver` to also be specified. For example, it is possible to call `Load(String, XmlResolver)` where `String` corresponds to a file name and `XmlResolver` is an object that handles references in the style sheet of type `xsl:import` and `xsl:include`. It would also be permissible to pass in a value of `Nothing` for the second parameter of the `Load` method (so that no `XmlResolver` would be specified). Note that there have been considerable changes to the parameters that the `Load` method takes between versions 1.0 and 1.1 of the .NET Framework. Look at the SDK documentation for details on the breaking changes that you might encounter when working with the `XslTransform` class.

- `Transform` — This overloaded method transforms a specified XML document using the previously specified XSLT style sheet and an `XmlResolver`. The location where the transformed XML is to be output is specified as a parameter to this method. The first parameter of each overloaded method is the XML document to be transformed. This parameter can be represented as an `IXPathNavigable`, XML file name (specified as parameter type, `String`), or `XPathNavigator`. Note that there have been considerable changes to the parameters that the `Transform` method takes between versions 1.0 and 1.1 of the .NET Framework. Look at the SDK documentation for details on the breaking changes that you might encounter when working with the `XslTransform` class.

The most straightforward variant of the `Transform` method is `Transform(String, String, XmlResolver)`. In this case, a file containing an XML document is specified as the first parameter, and a file name that receives the transformed XML document is specified as the second parameter, and the `XmlResolver` used as the third parameter. This is exactly how the first XSLT example utilized the `Transform` method:

```
myXslTransform.Transform("..\FilmOrders.xml", destFileName)
```

The first parameter to the `Transform` method can also be specified as `IXPathNavigable` or `XPathNavigator`. Either of these parameter types allows the XML output to be sent to an object of type `Stream`, `TextWriter`, or `XmlWriter`. When these two flavors of input are specified, a parameter containing an object of type `XsltArgumentList` can be specified. An `XsltArgumentList` object contains a list of arguments that are used as input to the transform.

When working with a .NET 2.0 project, it is preferable to use the `XslCompiledTransform` object instead of the `XslTransform` object, because the `XslTransform` object is considered obsolete. When using the new `XslCompiledTransform` object, you construct the file using the following code:

```
Dim myXsltCommand As New XslCompiledTransform()
Dim destFileName As String = "..\ShowIt.html"
myXsltCommand.Load("..\DisplayThatPuppy.xslt")
myXsltCommand.Transform("..\FilmOrders.xml", destFileName)
System.Diagnostics.Process.Start(destFileName)
```

Just like the `XslTransform` object, the `XslCompiledTransform` object uses the `Load` and `Transform` methods. The `Load` method provides the following signatures:

```
XslCompiledTransform.Load (IXPathNavigable)
XslCompiledTransform.Load (String)
XslCompiledTransform.Load (XmlReader)
XslCompiledTransform.Load (IXPathNavigable, XsltSettings, XmlResolver)
XslCompiledTransform.Load (String, XsltSettings, XmlResolver)
XslCompiledTransform.Load (XmlReader, XsltSettings, XmlResolver)
```

In this case, `String` is a representation of the .xslt file that should be used in the transformation. `XmlResolver` has already been explained and `XsltSettings` is an object that allows you to define which XSLT additional options to permit. The previous example used a single parameter, `String`, which showed the location of the style sheet:

```
myXsltCommand.Load("..\DisplayThatPuppy.xslt")
```

The `XslCompiledTransform` object's `Transform` method provides the following signatures:

```
XslCompiledTransform.Transform (IXPathNavigable, XmlWriter)
XslCompiledTransform.Transform (String, String)
XslCompiledTransform.Transform (String, XmlWriter)
XslCompiledTransform.Transform (XmlReader, XmlWriter)
XslCompiledTransform.Transform (IXPathNavigable, XsltArgumentList, Stream)
XslCompiledTransform.Transform (IXPathNavigable, XsltArgumentList, TextWriter)
XslCompiledTransform.Transform (IXPathNavigable, XsltArgumentList, XmlWriter)
XslCompiledTransform.Transform (String, XsltArgumentList, Stream)
XslCompiledTransform.Transform (String, XsltArgumentList, TextWriter)
XslCompiledTransform.Transform (String, XsltArgumentList, XmlWriter)
XslCompiledTransform.Transform (XmlReader, XsltArgumentList, Stream)
XslCompiledTransform.Transform (XmlReader, XsltArgumentList, TextWriter)
XslCompiledTransform.Transform (XmlReader, XsltArgumentList, XmlWriter)
XslCompiledTransform.Transform (XmlReader, XsltArgumentList, XmlWriter, XmlResolver)
```

In this case, `String` represents the location of specific files (whether it is source files or output files). Some of the signatures also allow for output to `XmlWriter` objects, streams, and `TextWriter` objects. These can be done by also providing additional arguments using the `XsltArgumentList` object. In the previous example, you used the second signature `XslCompiledTransform.Transform(String, String)`, which asked for the source file and the destination file (both string representations of the location of said files).

```
myXsltCommand.Transform("..\FilmOrders.xml", destFileName)
```

The `XslCompiledTransform` object example will produce the same table as was generated using the `XslTransform` object.

XSLT Transforming between XML Standards

The first example used four XSLT elements to transform an XML file into an HTML file. Such an example has merit, but it does not demonstrate an important use of XSLT. Another major application of XSLT is to transform XML from one standard into another standard. This may involve renaming elements/attributes, excluding elements/attributes, changing datatypes, altering the node hierarchy, and representing elements as attributes and vice versa.

A case of differing XML standards could easily happen to your software that automates movie orders coming into a supplier. Imagine that the software, including its XML representation of a movie order, is so successful that you sell 100,000 copies. However, just as you're celebrating, a consortium of the largest movie supplier chains announces that they will no longer be accepting faxed orders and that they are introducing their own standard for the exchange of movie orders between movie sellers and buyers.

Rather than panic, you simply ship an upgrade that comes complete with an XSLT file. This upgrade (a bit of extra code plus the XSLT file) transforms your XML representation of a movie order into the XML representation dictated by the consortium of movie suppliers. Using an XSLT file allows you to ship the upgrade immediately. If the consortium of movie suppliers revises their XML representation, you are not obliged to change your source code. Instead, you can simply ship the upgraded XSLT file that will ensure that each movie order document is compliant.

Using XML in Visual Basic 2005

The specific source code that executes the transform is:

```
Dim myXsltCommand As New XslCompiledTransform()
myXsltCommand.Load("..\ConvertLegacyToNewStandard.xslt")
myXsltCommand.Transform("..\MovieOrdersOriginal.xml", "..\MovieOrdersModified.xml")
```

The three lines of code are

1. Create an `XslCompiledTransform` object.
2. Use the `Load` method to load an XSLT file (`ConvertLegacyToNewStandard.xslt`).
3. Use the `Transform` method to transform a source XML file (`MovieOrdersOriginal.xml`) into a destination XML file (`MovieOrdersModified.xml`).

Recall that the input XML document (`MovieOrdersOriginal.xml`) does not match the format required by your consortium of movie supplier chains. The content of this source XML file is:

```
<?xml version="1.0" encoding="you-ascii" ?>
<FilmOrder_Multiple>
    <multiFilmOrders>
        <FilmOrder>
            <name>Grease</name>
            <filmId>101</filmId>
            <quantity>10</quantity>
        </FilmOrder>
        ...
    </multiFilmOrders>
</FilmOrder_Multiple>
```

The format exhibited in the previous XML document does not match the format of the consortium of movie supplier chains. To be accepted by the collective of suppliers, you must transform the document as follows:

- Rename element `<FilmOrder_Multiple>` to `<Root>`.
- Remove element `<multiFilmOrders>`.
- Rename element `<FilmOrder>` to `<DvdOrder>`.
- Remove element `<name>` (the film's name is not to be contained in the document).
- Rename element `<quantity>` to `HowMuch` and make `HowMuch` an attribute of `<DvdOrder>`.
- Rename element `<filmId>` to `FilmOrderNumber` and make `FilmOrderNumber` an attribute of `<DvdOrder>`.
- Display attribute `HowMuch` before attribute `FilmOrderNumber`.

A great many of the steps performed by the transform could have been achieved using an alternative technology. For example, you could have used Source Code Style attributes with your serialization to generate the correct XML attribute and XML element name. If you had known in advance that a consortium of suppliers was going to develop a standard, you could have written your classes to be

serialized based on the standard. The point was that you didn't know and now one standard (your legacy standard) has to be converted into a newly adopted standard of the movie suppliers' consortium. The worst thing you could do would be to change your working code and then force all users working with the application to upgrade. It is vastly simpler to add an extra transformation step to address the new standard.

The XSLT file that facilitates the transform is named `ConvertLegacyToNewStandard.xslt`. A portion of this file is implemented as follows:

```
<xsl:template match="FilmOrder">
    <!-- rename <FilmOrder> to <DvdOrder> -->
    <xsl:element name="DvdOrder">
       <!-- Make element 'quantity' attribute HowMuch
            Notice attribute HowMuch comes before attribute FilmOrderNumber -->
        <xsl:attribute name="HowMuch">
            <xsl:value-of select='quantity'></xsl:value-of>
        </xsl:attribute>
        <!-- Make element filmId attribute FilmOrderNumber -->
        <xsl:attribute name="FilmOrderNumber">
            <xsl:value-of select='filmId'></xsl:value-of>
        </xsl:attribute>
    </xsl:element>
    <!-- end of DvdOrder element -->
</xsl:template>
```

In the previous snippet of XSLT, the following XSLT elements are used to facilitate the transformation:

- `<xsl:template match="FilmOrder">` — All operations in this `template` XSLT element will take place on the original document's `FilmOrder` node.
- `<xsl:element name="DvdOrder">` — The element corresponding to the source document's `FilmOrder` element will be called `DvdOrder` in the destination document.
- `<xsl:attribute name="HowMuch">` — An attribute named `HowMuch` will be contained in the previously specified element. The previously specified element is `<DvdOrder>`. This `attribute` XSLT element for `HowMuch` comes before the `attribute` XSLT element for `FilmOrderNumber`. This order was specified as part of your transform to adhere to the new standard.
- `<xsl:value-of select='quantity'>` — Retrieve the value of the source document's `<quantity>` element and place it in the destination document. This instance of XSLT element, `value-of`, provides the value associated with attribute `HowMuch`.

Two new XSLT elements have crept into your vocabulary: `element` and `attribute`. Both of these XSLT elements live up to their names. Specifying the XSLT element named `element` places an element in the destination XML document. Specifying the XSLT element named `attribute` places an attribute in the destination XML document. The XSLT transform found in `ConvertLegacyToNewStandard.xslt` is too long to review completely. When reading this file in its entirety, you should remember that this XSLT file contains inline documentation to specify precisely what aspect of the transformation is being performed at which location in the XSLT document. For example, the following XML code comments inform you about what the XSLT element `attribute` is about to do:

```
<!-- Make element 'quantity' attribute HowMuch
      Notice attribute HowMuch comes before attribute FilmOrderNumber -->
<xsl:attribute name="HowMuch">
```

```
        <xsl:value-of select='quantity'></xsl:value-of>
    </xsl:attribute>
```

The previous example spanned several pages but contained just three lines of code. This demonstrates that there is more to XML than learning how to use it in Visual Basic and the .NET Framework. Among other things, you also need a good understanding of XSLT, XPath, and XQuery.

Other Classes and Interfaces in System.Xml.Xsl

We just took a good look at XSLT and the `System.Xml.Xsl` namespace, but there is a lot more to it than that. The other classes and interfaces exposed by `System.Xml.Xsl` namespace include

- `IXsltContextFunction` — This interface accesses at runtime a given function defined in the XSLT style sheet.
- `IXsltContextVariable` — This interface accesses at runtime a given variable defined in the XSLT style sheet.
- `XsltArgumentList` — This class contains a list of arguments. These arguments are XSLT parameters or XSLT extension objects. The `XsltArgumentList` object is used in conjunction with the `Transform` method of `XslTransform` and `XslCompiledTransform`.
- `XsltContext` — This class contains the state of the XSLT processor. This context information allows XPath expressions to have their various components resolved (functions, parameters, and namespaces).
- `XsltException`, `XsltCompileException` — These classes contain the information pertaining to an exception raised while transforming data. `XsltCompileException` is derived from `XsltException`.

ADO.NET

ADO.NET allows Visual Basic applications to generate XML documents and to use such documents to update persisted data. ADO.NET natively represents its `DataSet`'s underlying datastore in XML. ADO.NET also allows SQL Server — specific XML support to be accessed. In this chapter, your focus is on those features of ADO.NET that allow the XML generated and consumed to be customized. ADO.NET is covered in detail in Chapter 11.

The `DataSet` properties and methods that are pertinent to XML include `Namespace`, `Prefix`, `GetXml`, `GetXmlSchema`, `InferXmlSchema`, `ReadXml`, `ReadXmlSchema`, `WriteXml`, and `WriteXmlSchema`. An example of code that uses the `GetXml` method is:

```
Dim adapter As New _
    SqlDataAdapter("SELECT ShipperID, CompanyName, Phone " & _
                   "FROM Shippers", _
                   "SERVER=localhost;UID=sa;PWD=sa;Database=Northwind;")
Dim ds As New DataSet

adapter.Fill(ds)
Console.Out.WriteLine(ds.GetXml())
```

The previous code uses the sample `Northwind` database (which comes with SQL Server and MSDE) and retrieves all rows from the `Shippers` table. This table was selected because it contains only three rows of data. The XML returned by `GetXml` is as follows (where `...` signifies that `<Table>` elements were removed for the sake of brevity):

```
<NewDataSet>
  <Table>
    <ShipperID>1</ShipperID>
    <CompanyName>Speedy Express</CompanyName>
    <Phone>(503) 555-9831</Phone>
  </Table>
  ...
</NewDataSet>
```

What you are trying to determine from the previous XML document is how to customize the XML generated. The more customization you can perform at the ADO.NET level, the less need there will be later. With this in mind, you notice that the root element is `<NewDataSet>` and that each row of the `DataSet` is returned as an XML element, `<Table>`. The data returned is contained in an XML element named for the column in which the data resides (`<ShipperID>`, `<CompanyName>`, and `<Phone>`, respectively).

The root element, `<NewDataSet>`, is just the default name of the `DataSet`. This name could have been changed when the `DataSet` was constructed by specifying the name as a parameter to the constructor:

```
Dim ds As New DataSet("WeNameTheDataSet")
```

If the previous version of the constructor was executed, then the `<NewDataSet>` element would be renamed `<WeNameTheDataSet>`. After the `DataSet` has been constructed, you can still set the property `DataSetName`, thus changing `<NewDataSet>` to a name such as `<WeNameTheDataSetAgain>`:

```
ds.DataSetName = "WeNameTheDataSetAgain"
```

The `<Table>` element is actually the name of a table in the `DataSet`'s `Tables` property. Programmatically, you can change `<Table>` to `<WeNameTheTable>`.

```
ds.Tables("Table").TableName = "WeNameTheTable"
```

You can customize the names of the data columns returned by modifying the SQL to use alias names. For example, you could retrieve the same data but generate different elements using the following SQL code:

```
SELECT ShipperID As TheID, CompanyName As CName, Phone As TelephoneNumber FROM
Shippers
```

Using the previous SQL statement, the `<ShipperID>` element would become the `<TheID>` element. The `<CompanyName>` element would become `<CName>` and `<Phone>` would become `<TelephoneNumber>`. The column names can also be changed programmatically by using the `Columns` property associated with the table in which the column resides. An example of this follows, where the XML element `<TheID>` is changed to `<AnotherNewName>`.

```
ds.Tables("WeNameTheTable").Columns("TheID").ColumnName = "AnotherNewName"
```

This XML could be transformed using `System.Xml.Xsl`. This XML could be read as a stream (`XmlTextReader`) or written as a stream (`XmlTextWriter`). The XML returned by ADO.NET could even be deserialized and used to create an object or objects using `XmlSerializer`. What is important is to recognize what ADO.NET-generated XML looks like. If you know its format, then you can transform it into whatever you like.

ADO.NET and SQL Server 2000's Built-In XML Features

Those interested in fully exploring the XML-specific features of SQL Server should take a look at *Professional SQL Server 2000 Programming* (Wrox Press, ISBN 0764543792). However, since the content of that book is not .NET-specific, the next example will form a bridge between *Professional SQL Server 2000 Programming* and the .NET Framework.

Two of the major XML-related features exposed by SQL Server are

- `FOR XML` — The `FOR XML` clause of an SQL `SELECT` statement allows a rowset to be returned as an XML document. The XML document generated by a `FOR XML` clause is highly customizable with respect to the document hierarchy generated, per-column data transforms, representation of binary data, XML schema generated, and a variety of other XML nuances.
- `OPENXML` — The `OPENXML` extension to Transact-SQL allows a stored procedure call to manipulate an XML document as a rowset. Subsequently, this rowset can be used to perform a variety of tasks, such as `SELECT`, `INSERT INTO`, `DELETE`, and `UPDATE`.

SQL Server's support for `OPENXML` is a matter of calling a stored procedure call. A developer who can execute a stored procedure call using Visual Basic in conjunction with ADO.NET can take full advantage of SQL Server's support for `OPENXML`. `FOR XML` queries have a certain caveat when it comes to ADO.NET. To understand this caveat, consider the following `FOR XML` query:

```
SELECT ShipperID, CompanyName, Phone FROM Shippers FOR XML RAW
```

Using SQL Server's Query Analyzer, this `FOR XML RAW` query generated the following XML:

```
<row ShipperID="1" CompanyName="Speedy Express" Phone="(503) 555-9831"/>
<row ShipperID="2" CompanyName="United Package" Phone="(503) 555-3199"/>
<row ShipperID="3" CompanyName="Federal Shipping" Phone="(503) 555-9931"/>
```

The same `FOR XML RAW` query can be executed from ADO.NET as follows:

```
Dim adapter As New _
    SqlDataAdapter("SELECT ShipperID, CompanyName, Phone " & _
                   "FROM Shippers FOR XML RAW", _
                   "SERVER=localhost;UID=sa;PWD=sa;Database=Northwind;")
Dim ds As New DataSet

adapter.Fill(ds)
Console.Out.WriteLine(ds.GetXml())
```

The caveat with respect to a `FOR XML` query is that all data (the XML text) must be returned via a result set containing a single row and a single column named `XML_F52E2B61-18A1-11d1-B105- 00805F49916B`.

The output from the previous code snippet demonstrates this caveat (where . . . represents similar data not shown for reasons of brevity):

```
<NewDataSet>
  <Table>
    <XML_F52E2B61-18A1-11d1-B105-00805F49916B>
      &lt;row ShipperID="1" CompanyName= "Speedy Express" Phone="(503)
       555-9831"/&gt;
      ...
    </XML_F52E2B61-18A1-11d1-B105-00805F49916B>
  </Table>
</NewDataSet>
```

The value of the single row and single column returned contains what looks like XML, but it contains `/<` instead of the less-than character, and `/>` instead of the greater-than character. The symbols < and > cannot appear inside XML data. For this reason, they must be entity-encoded (that is, represented as `/>` and `/<`). The data returned in element `<XML_F52E2B61-18A1-11d1-B105- 00805F49916B>` is not XML but is data contained in an XML document.

To fully utilize `FOR XML` queries, the data must be accessible as XML. The solution to this quandary is the `ExecuteXmlReader` method of the `SQLCommand` class. When this method is called, an `SQLCommand` object assumes that it is executed as a `FOR XML` query and returns the results of this query as an `XmlReader` object. An example of this follows (again found in `VBNetXML05`):

```
Dim connection As New _
    SqlConnection("SERVER=localhost;UID=sa;PWD=sa;Database=Northwind;")
Dim command As New _
    SqlCommand("SELECT ShipperID, CompanyName, Phone " & _
                    "FROM Shippers FOR XML RAW")
Dim memStream As MemoryStream = New MemoryStream
Dim xmlReader As New XmlTextReader(memStream)

connection.Open()
command.Connection = connection
xmlReader = command.ExecuteXmlReader()
' Extract results from XMLReader
```

The `XmlReader` created in this code is of type `XmlTextReader`, which derives from `XmlReader`. The `XmlTextReader` is backed by a `MemoryStream`, hence it is an in-memory stream of XML that can be traversed using the methods and properties exposed by `XmlTextReader`. Streaming XML generation and retrieval has been discussed earlier.

Using the `ExecuteXmlReader` method of the `SQLCommand` class, it is possible to retrieve the result of `FOR XML` queries. What makes `FOR XML` style of queries so powerful is that it can configure the data retrieved. The three types of `FOR XML` queries support the following forms of XML customization:

- `FOR XML RAW` — This type of query returns each row of a result set inside an XML element named `<row>`. The data retrieved is contained as attributes of the `<row>` element. The attributes are named for the column name or column alias in the `FOR XML RAW` query.
- `FOR XML AUTO` — By default, this type of query returns each row of a result set inside an XML element named for the table or table alias contained in the `FOR XML AUTO` query. The data

retrieved is contained as attributes of this element. The attributes are named for the column name or column alias in the `FOR XML AUTO` query. By specifying `FOR XML AUTO, ELEMENTS` it is possible to retrieve all data inside elements rather than inside attributes. All data retrieved must be in attribute or element form. There is no mix-and-match capability.

- `FOR XML EXPLICIT` — This form of the `FOR XML` query allows the precise XML type of each column returned to be specified. The data associated with a column can be returned as an attribute or an element. Specific XML types, such as `CDATA` and `ID`, can be associated with a column returned. Even the level in the XML hierarchy in which data resides can be specified using a `FOR XML EXPLICIT` query. This style of query is fairly complicated to implement.

`FOR XML` queries are flexible. Using `FOR XML EXPLICIT` and the dental database, it would be possible to generate any form of XML medical prescription standard. The decision that needs to be made is where XML configuration takes place. Using Visual Basic, a developer could use `XmlTextReader` and `XmlTextWriter` to create any style of XML document. Using the XSLT language and an XSLT file, the same level of configuration can be achieved. SQL Server and, in particular, `FOR XML EXPLICIT` allow the same level of XML customization, but this customization takes place at the SQL level and may even be configured to stored procedure calls.

XML and SQL Server 2005

As a representation for data, XML is ideal in that it is a self-describing data format that allows you to provide your datasets as complex datatypes as well as providing order to your data. SQL Server 2005 embraces this direction.

More and more developers are turning to XML as a means of data storage. For instance, Microsoft Office allows documents to be saved and stored as XML documents. As more and more products and solutions are turning toward XML as a means of storage, this allows for a separation between the underlying data and the presentation aspect of what is being viewed. XML is also being used as a means of communicating datasets across platforms and across the enterprise. The entire XML Web services story is a result of this new capability. Simply said, XML is a powerful alternative to your data storage solutions.

Just remember that the power of using XML isn't only about storing data as XML somewhere (whether that is XML files or not), but is also about the ability to quickly get at this XML data and to be able to query the data that is retrieved.

SQL Server 2005 makes a big leap toward XML in adding an XML datatype. This allows you to unify the relational data aspects of the database and the new desires to work with XML data.

`FOR XML` has also been expanded from within this latest edition of SQL Server. This includes a new `TYPE` directive which returns an XML datatype instance. Also, from the Framework, .NET 2.0 adds a new namespace — `System.Data.SqlXml` — which allows you to easily work with the XML data that comes from SQL Server 2005. The `SqlXml` object is an `XmlReader`-derived type. Another addition is the use of the `SqlDataReader` object's `GetXml` method.

Summary

Ultimately, XML could be the underpinnings of electronic commerce, banking transactions, and data exchange of almost every conceivable kind. The beauty of XML is that it isolates data representation from data display. Technologies, such as HTML, contain data that is tightly bound to its display format. XML does not suffer this limitation, yet at the same time has the readability of HTML. Accordingly, the XML facilities available to a Visual Basic application are vast, and there are a large number of XML-related features, classes, and interfaces exposed by the .NET Framework.

In this chapter, you saw how to use `System.Xml.Serialization.XmlSerializer` to serialize classes. Source Code Style attributes were introduced in conjunction with serialization. This style of attributes allows the customization of the XML serialized to be extended to the source code associated with a class. What is important to remember about the direct of serialization classes is that a required change in the XML format becomes a change in the underlying source code. Developers should resist the temptation to rewrite the serialized classes in order to conform to some new XML data standard (such as the prescription format endorsed by your consortium of pharmacies). Technologies, such as XSLT, exposed via the `System.Xml.Query` namespace should be examined first as alternatives. You saw how to use XSLT style sheets to transform XML data using the classes found in the `System.Xml.Xsl` namespace.

The most useful classes and interfaces in the `System.Xml` namespace were reviewed, including those that support document-style XML access: `XmlDocument`, `XmlNode`, `XmlElement`, and `XmlAttribute`. The `System.Xml` namespace also contains classes and interfaces that support stream-style XML access: `XmlReader` and `XmlWriter`.

Finally, you looked at using XML with Microsoft's SQL Server.