XML.NET

based on:
T. Thangarathinam, Professional ASP.NET 2.0 XML, 2006, Wrox
MSDN

Krzysztof Mossakowski
Faculty of Mathematics and Information Science
http://www.mini.pw.edu.pl/~mossakow

Contents

- XML
- XML classes in the .NET Framework
- XmlReader
- XmlWriter
- XML DOM in the .NET Framework
- XSLT transformations
- XML in ADO.NET

XML

XML – Extensible Markup Language

- XML is a language defined by the World Wide Web Consortium
  - It is a metamarkup language which allows to create custom markup languages, e.g. RSS, MathML, MusicXML
  - XML is a simplified subset of the Standard Generalized Markup Language (SGML)
- XML 1.0 (W3C Recommendation) was published in 1998

XML Features

- Advantages:
  - Human-readable and machine-readable
  - Platform independence, no licences or restrictions
  - Support for Unicode
  - The strict syntax and parsing requirements
  - Ability to represent general data structures: records, lists, and trees
  - The self-documenting format
  - Widespread industry support and a large family of XML support technologies
- Disadvantages:
  - Verbose and redundant syntax
  - The hierarchical model (which is limited compared to the relational model)

Sample XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<recipe name="bread" prep_time="5 mins" cook_time="3 hours">
  <title>Basic bread</title>
  <ingredient amount="3" unit="cups">Flour</ingredient>
  <ingredient amount="0.25" unit="ounce">Yeast</ingredient>
  <ingredient amount="1.5" unit="cups"
              state="warm">Water</ingredient>
  <ingredient amount="1" unit="teaspoon">Salt</ingredient>
  <instructions>
    <step>Mix all ingredients together, and knead
          thoroughly.</step>
    <step>Cover with a cloth, and leave for one hour in warm
          room.</step>
    <step>Knead again, place in a tin, and then bake in the
          oven.</step>
  </instructions>
</recipe>
```

Well Formed XML Documents

- A well-formed document must meet the following constraints:
  - The document contains one or more elements
  - The document consists of exactly one root element (also known as the document element)
  - The name of an element's end tag matches the name defined in the start tag
  - No attribute may appear more than once within an element
  - Attribute values cannot contain a left-angle bracket (<)
  - Elements delimited with start and end tags must nest properly within each other

CDATA, PCDATA, and Entity References

- CDATA – Character Data
  - The textual data that appears between **<![CDATA[** and **]]>**
  - Ignored by the parser and not processed at all

```
<range><![CDATA[0 < counter < 100]]></range>
```

- PCDATA – Parsed Character Data
  - All other textual data
  - Special characters must be replaced with entity references

<	<	>	>	&	&
‘	'	"	"		

```
<range>0 &lt; counter &lt; 100</range>
```

Namespaces

- A namespace groups elements together by partitioning elements and their attributes into logical areas and providing a way to identify the elements and attributes uniquely
- Namespaces are also used to reference a particular DTD or XML Schema
- Namespaces are identified with Universal Resource Indicators (URIs)
  - Namespace names are URLs usually, but they don't have to point to anything

```
http://www.develop.com/student
http://www.ed.gov/elementary/students
urn:www-develop-com:student
urn:www.ed.gov:elementary.students
urn:uuid:E7F73B13-05FE-44ec-81CE-F898C4A6CDB4
```

Parts of a Namespace Declaration

- **`xmlns`** – identifies the value as an XML namespace
  - It is required to declare a namespace
  - It can be attached to any XML element
- **`prefix`** – identifies a namespace prefix
  - If it's used, any element found in the document that uses the prefix (**`prefix:element`**) is then assumed to fall under the scope of the declared namespace
- **`namespaceURI`** – the unique identifier
  - It is a symbolic link

Examples of Using Namespaces

```
<x:transform version='1.0'
             xmlns:x='http://www.w3.org/1999/XSL/Transform'>
    <x:template match='/'>
        <hello_world/>
    </x:template>
</x:transform>
```

```
<d:student xmlns:d='http://www.develop.com/student'>
    <d:id>3235329</d:id>
    <d:name>Jeff Smith</d:name>
    <d:language>C#</d:language>
    <d:rating>9.5</d:rating>
</d:student>
```

DTD

- The DTD (Document Type Definition) is one of several SGML and XML schema languages
  - It is often referred to as a **`doctype`**
- DTDs consist of a series of declarations for elements and associated attributes that may appear in the documents they validate
- The DTD also includes information about data types, whether values are required, default values, and number of allowed occurrences

DTD Example

```
<!ELEMENT people_list (person*)>
<!ELEMENT person (name, birthdate?, gender?,socialsecuritynumber?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT birthdate (#PCDATA)>
<!ELEMENT gender (#PCDATA)>
<!ELEMENT socialsecuritynumber (#PCDATA)>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE people_list SYSTEM "example.dtd">
<people_list>
    <person>
        <name>Fred Bloggs</name>
        <birthdate>27/11/2008</birthdate>
        <gender>Male</gender>
    </person>
</people_list>
```

DTD Limitations

- The lack of strong type-checking
- A strange and seemingly archaic syntax
- Only a limited capability in describing the document structure in terms of how many elements can nest within other elements

XSD

- XSD (XML Schema Definition) is a formal definition for defining a schema for a class of XML documents
- Features of the XSD:
  - More expressive than XML DTDs
  - Expressed in XML
  - Self-describing

XSD Example

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="country" type="Country"/>
    <xs:complexType name="Country">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="population" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

```
<country
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="country.xsd">
    <name>France</name>
    <population>59.7</population>
</country>
```

XSLT

- XSLT (Extensible Stylesheet Language Transformations) is a language for processing XML data
- An XSLT style sheet is an XML document
- There are two primary uses of XSLT:
  - Documents conversion, from XML into HTML
    - It is a fine way to separate content from appearance
  - Documents conversion, from XML into other XML

XSLT Example

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
    <xsl:template match="/hello-world">
        <HTML><HEAD><TITLE></TITLE></HEAD><BODY>
        <H1><xsl:value-of select="greeting"/></H1>
        <xsl:apply-templates select="greeter"/>
        </BODY></HTML>
    </xsl:template>
    <xsl:template match="greeter">
        <DIV>from <I><xsl:value-of select="."/></I></DIV>
    </xsl:template>
</xsl:stylesheet>
```

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="hello.xsl"?>
<hello-world>
  <greeter>An XSLT Programmer</greeter>
  <greeting>Hello, World!</greeting>
</hello-world>
```

XML DOM

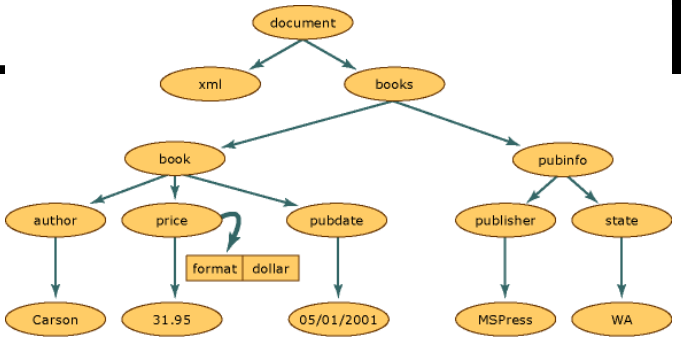
- The XML DOM (Document Object Model) provides a navigable set of classes
  - These classes allow to construct an XML document in memory
  - It is possible to compile and validate XML documents against a DTD or schema
- The DOM provides a programmatic representation of XML documents
  - It is well suited for traversing and modifying an XML document
  - It provides little support for finding an arbitrary element or attribute in a document

XML DOM Capabilities

- Find the root node in an XML document
- Find a list of elements with a given tag name
- Get a list of children of a given node
- Get the parent of a given node
- Get the tag name of an element
- Get the data associated with an element
- Get a list of attributes of an element
- Get the tag name of an attribute
- Get the value of an attribute
- Add, modify, or delete an element in the document
- Add, modify, or delete an attribute in the document
- Copy a node into a document (including subnodes)

Example of the XML DOM

```
<?xml version="1.0"?>
<books>
  <book>
      <author>Carson</author>
      <price format="dollar">31.95</price>
      <pubdate>05/01/2001</pubdate>
  </book>
  <pubinfo>
      <publisher>MSPress</publisher>
      <state>WA</state>
  </pubinfo>
</books>
```



XPath

- XPath is a navigational query language specified by the W3C for locating data within an XML document
- XPath can be used to query an XML document
  - An XPath query expression can select on document parts, or types, such as document elements, attributes, and text
- XPath also provides functions to do numeric evaluations, such as summations and rounding
- XPath includes over 100 built-in functions, there are functions for string values, numeric values, date and time comparison, node and QName manipulation, sequence manipulation, Boolean values, and more

XPath Capabilities

- Find all children of the current node
- Find all ancestor elements of the current context node with a specific tag
- Find the last child element of the current node with a specific tag
- Find the nth child element of the current context node with a given attribute
- Find the first child element with a tag of <tag1> or <tag2>
- Get all child nodes that do not have an element with a given attribute
- Get the sum of all child nodes with a numeric element
- Get the count of all child nodes

SAX

- SAX (Simple API for XML) is a publicly developed standard for the events-based parsing of XML documents
- It is a popular alternative to the Document Object Model (DOM)
  - It requires significantly less memory to process an XML document than the DOM
  - It allows to retrieve small amounts of information
  - SAX can be used to build DOM trees
  - Developers can traverse DOM trees and emit SAX streams
- SAX facilitates the search of large documents to extract small pieces of information and allows to abort processing after the information is located

XML Classes in the .NET Framework

Supported Standards

- XML 1.0 - http://www.w3.org/TR/1998/REC-xml-19980210
  - including DTD support
- XML Namespaces - http://www.w3.org/TR/REC-xml-names/
  - both stream level and DOM
- XSD Schemas - http://www.w3.org/2001/XMLSchema
- XPath expressions - http://www.w3.org/TR/xpath
- XSLT transformations - http://www.w3.org/TR/xslt
- DOM Level 1 Core – http://www.w3.org/TR/REC-DOM-Level-1/
- DOM Level 2 Core - http://www.w3.org/TR/DOM-Level-2/

XML Namespaces

- **`System.Xml`**
  - The core of the whole ofl XML functionality
- **`System.Xml.Schema`**
  - Support for XML Schema Definition Language (XSD) schemas
- **`System.Xml.Serialization`**
  - Classes that allows to serialize and deserialize objects into XML formatted documents
- **`System.Xml.XPath`**
  - Support for the XPath parser and evaluation functionality
- **`System.Xml.Xsl`**
  - Support for XSLT transformations

XML Parsing

- A validating parser can use a DTD or schema to verify that a document is properly constructed
  - A DTD or XML schema can also specify default values for the attributes of various elements
- A non-validating parser only requires that the document must be well-formed
  - It's possible to parse well-formed documents without referring to a DTD or XSD schema

XML Parsing in the .NET Framework

- Fast, non-cached, forward-only access
  - The **`XmlReader`** class allows to access XML data from a stream or the XML document
- Random access via an in-memory DOM tree
  - The **`XmlDocument`** class is a representation of the XML document in memory
- SAX is not supported in the .NET Framework

Writing XML

- The **`XmlWriter`** class is the core class that allows to create XML streams and write data to well-formed XML documents
- Sample tasks:
  - Writing multiple documents into one output stream
  - Writing valid names and tokens into the stream
  - Encoding binary data
  - Writing text output
  - Managing output
  - Flushing and closing the output stream

XPath Support

- The XPath functionality in the .NET Framework is encapsulated in the **`System.Xml.XPath`** namespace
  - The **`XPathDocument`** class provides a read-only cache for a fast and highly optimized processing of XML documents using XSLT
  - The **`XPathExpression`** class encapsulates a compiled XPath expression
  - The **`XPathNavigator`** class provides a cursor model for navigating and editing XML data
  - The **`XPathNodeIterator`** class allows to iterate a set of nodes selected by the XPath methods

XML Schema Object Model

- The Schema Object Model consists of a set of classes that allows to read the schema definition from a file
  - The **`System.Xml.Schema`** namespace
- Features:
  - Loading and saving valid XSD schemas
  - Creating in-memory schemas
  - Caching and retrieving schemas using the **`XmlSchemaSet`** class
  - Validating XML instance documents against the schemas using the **`XmlReader`** and **`XmlReaderSettings`** classes
  - Building a scheme programmatically

Transforming XML using XSLT

- The **`System.Xml.Xsl`** namespace provides support for the Extensible Stylesheet Transformation (XSLT)
  - The **`XslTransform`** class is marked as obsolete
  - The **`XslCompiledTransofm`** class is the new XSLT processor
    - This class includes many performance improvements
    - It compiles the XSLT style sheet down to a common intermediate format
    - Once the style sheet is compiled, it can be cached and reused

XML and ADO.NET

- A typed **`DataSet`** is a class that is derived from a **`DataSet`** class and has an associated XML schema
  - Changes made to the XSD file are reflected in the underlying **`DataSet`**
- On loading an XML document into a **`DataSet`**, XML schema validates the data
- The **`XmlDataDocument`** class allows to use ADO.NET and XML together
  - The **`DataSet`** and **`XmlDataDocument`** objects provide a synchronized view of the same data using a relational and hierarchical model, respectively

XmlReader

The XmlReader Class

- It is an abstract class that provides access to the XML data
  - Non-cached
  - Forward-only
  - Read-only
- It supports reading XML data from a stream or file
- The **`XmlReader`** class allows to:
  - Verify that the characters are legal XML characters, and that element and attribute names are valid XML names
  - Verify that the XML document is well formed
  - Validate the data against a DTD or schema
  - Retrieve data from the XML stream or skip unwanted records using a pull model

Creating XML Readers

- **`XmlReader`** instances are created using the **`Create()`** method
- The **`XmlReaderSettings`** class is used to specify the set of features to enable on the **`XmlReader`** object
- The **`XmlTextReader`**, **`XmlNodeReader`**, and **`XmlValidatingReader`** classes are concrete implementations of the **`XmlReader`** class
  - Using the **`XmlReader.Create()`** method is recommended

Examples of Creating XML Readers

- Instantiating an **`XmlReader`** object

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ConformanceLevel = ConformanceLevel.Fragment;
settings.IgnoreWhitespace = true;
settings.IgnoreComments = true;
XmlReader reader = XmlReader.Create("books.xml", settings);
```

- Wrapping a reader instance within another reader

```
XmlTextReader txtReader = new XmlTextReader("bookOrder.xml");
XmlReaderSettings settings = new XmlReaderSettings();
settings.Schemas.Add("urn:po-schema", "PO.xsd");
settings.ValidationType = ValidationType.Schema;
XmlReader reader = XmlReader.Create(txtReader, settings);
```

Examples of Creating XML Readers cont.

- Chaining readers to add additional settings

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.DTD;
XmlReader inner = XmlReader.Create("book.xml", settings); // DTD
settings.Schemas.Add("urn:book-schema", "book.xsd");
settings.ValidationType = ValidationType.Schema;
XmlReader outer = XmlReader.Create(inner, settings);   // Schema
```

- Accessing external resources

```
// Create a resolver with default credentials
XmlUrlResolver resolver = new XmlUrlResolver();
resolver.Credentials =
    System.Net.CredentialCache.DefaultCredentials;
// Set the reader settings object to use the resolver
settings.XmlResolver = resolver;
// Create the XmlReader object
XmlReader reader =
    XmlReader.Create("http://www.a.com/data.xml", settings);
```

XML Validation Using XmlReaderSettings

- Settings on the **`XmlReaderSettings`** object used by the **`Create()`** method determine what type of data validation, if any, the **`XmlReader`** object supports
- Validation settings on the **`XmlReaderSettings`** class
  - **`ProhibitDtd`** – the default value is **`true`**
  - **`ValidationType`** – **`DTD, None, Schema`**
  - **`ValidationEventHandler`** – an event handler for receiving information (if not provided, an **`XmlException`** is thrown on the first validation error)
  - **`ValidationFlags`**
  - **`XmlResolver`** – used to resolve and access any external resources (**`xs:include`** or **`xs:import`** elements)

Using the XmlSchemaSet

- The **`XmlSchemaSet`** is a cache or library where the XML Schema definition language (XSD) schemas can be stored
  - It improves performance by caching schemas in memory

```
// Create the XmlSchemaSet class
XmlSchemaSet sc = new XmlSchemaSet();
// Add the schema to the collection
sc.Add("urn:bookstore-schema", "books.xsd");
// Set the validation settings
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas = sc;
settings.ValidationEventHandler +=
    new ValidationEventHandler (ValidationCallBack);
// Create the XmlReader object
XmlReader reader = XmlReader.Create(
    "booksSchemaFail.xml", settings);
// Parse the file
while (reader.Read()) ;
```

Using an Inline XML Schema

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.ValidationFlags |=
    XmlSchemaValidationFlags.ProcessInlineSchema;
settings.ValidationFlags |=
    XmlSchemaValidationFlags.ReportValidationWarnings;
settings.ValidationEventHandler +=
    new ValidationEventHandler (ValidationCallBack);

XmlReader reader = XmlReader.Create(
    "xmlWithInlineSchema.xml", settings);
while (reader.Read());


private static void ValidationCallBack (object sender,
                                        ValidationEventArgs args) {
    if (args.Severity == XmlSeverityType.Warning)
        Console.WriteLine("\tWarning:" + args.Message);
     else
        Console.WriteLine("\tError: " + args.Message);
}
```

Validation Using a DTD

```
// Set the validation settings
XmlReaderSettings settings = new XmlReaderSettings();
settings.ProhibitDtd = false;
settings.ValidationType = ValidationType.DTD;
settings.ValidationEventHandler +=
    new ValidationEventHandler (ValidationCallBack);

// Create the XmlReader object
XmlReader reader = XmlReader.Create("itemDTD.xml", settings);

// Parse the file
while (reader.Read());
```

Validation Using a Wrapped XmlReader

```
// Create and load the XML document
XmlDocument doc = new XmlDocument();
doc.Load("booksSchema.xml");
// Make changes to the document
XmlElement book = (XmlElement) doc.DocumentElement.FirstChild;
book.SetAttribute("publisher", "Worldwide Publishing");
// Create an XmlNodeReader using the XML document
XmlNodeReader nodeReader = new XmlNodeReader(doc);

// Set the validation settings on the XmlReaderSettings object
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add("urn:bookstore-schema", "books.xsd");
settings.ValidationEventHandler +=
    new ValidationEventHandler (ValidationCallBack);
// Create a reader that wraps the XmlNodeReader object
XmlReader reader = XmlReader.Create(nodeReader, settings);
// Parse the XML file
while (reader.Read());
```

Reading XML Data

- The **`XmlReader`** class works by starting at the beginning of the file and reading one node at a time
  - As each node is read, you can either ignore the node or access the node information
- The steps for using the **`XmlReader`** class are as follows:
  1. Create an instance of the class using the **`Create()`** method of the **`XmlReader`** class
  2. Set up a loop that calls the **`Read()`** method repeatedly
     - This method starts with the first node in the file and then reads all remaining nodes, one at a time, as it is called
     - It returns **`true`** if there is a node to read, **`false`** when the end of the file has been reached
  3. In the loop, examine the properties and methods of the **`XmlReader`** object to obtain information about the current node

Example of Reading XML Data

```
using (XmlReader reader = XmlReader.Create("C:\\data.xml")) {
  while (reader.Read()) {
    // Process only the elements
    if (reader.NodeType == XmlNodeType.Element) {
      // Reset the variable for a new element
      Console.WriteLine("Element - depth: {0}, name: {1}\n",
          reader.Depth, reader.Name);
      // Check if the element has any attributes
      if (reader.HasAttributes) {
        Console.WriteLine("Attributes: ");
        for (int i = 0; i < reader.AttributeCount; i++) {
          // Read the current attribute
          reader.MoveToAttribute(i);
          Console.WriteLine("{0} ", reader.Name);
        }
        Console.WriteLine("\n");
        // Instruct the parser to go back the element
        reader.MoveToElement();
    }
  }
}
```

Reading Elements

- The methods and properties of the **`XmlReader`** class:
  - **`ReadStartElement()`** – checks that the current node is an element and advances the reader to the next node
  - **`ReadEndElement()`** – checks that the current node is an end tag and advances the reader to the next node
  - **`ReadElementString()`** – reads a text-only element
  - **`ReadToDescendant()`** – advances the **`XmlReader`** to the next descendant element with the specified name
  - **`ReadToNextSibling()`** – advances the **`XmlReader`** to the next sibling element with the specified name
  - **`IsStartElement()`** – checks if the current node is a start tag or an empty element tag
  - **`IsElementEmpty`** – checks if the current element has an empty element tag

Reading Attributes

- Reading attributes on elements
  - **`HasAttributes, AttributeCount`**
  - **`GetAttribute(), Item()`**
  - **`IsDefault`** – check if the current node is an attribute that was generated from the default value defined in the DTD or schema
  - **`MoveToFirstAttribute(), MoveToNextAttribute(), MoveToAttribute()`**
  - **`MoveToElement()`** – moves to the element that owns the current attribute node
  - **`ReadAttributeValue()`** – parses the attribute value into one or more **`Text`**, **`EntityReference`**, or **`EndEntity`** nodes

Reading Content

- The **`Value`** property can be used to get the text content of the current node
  - The value returned depends on the node type of the current node
- The **`ReadString()`** method returns the content of a element or text node as a string
  - This method concatenates all text, significant white space, white space, and CDATA section nodes together
  - It stops on processing instructions and comments - it does not ignore them
- The **`ReadInnerXml()`** method returns all the content of the current node, including the markup
- The **`ReadOuterXml()`** method returns all the XML content, including markup, of the current node and all its children

Reading Typed Data

- The **`XmlReader`** class permits callers to read XML data and return values as simple-typed CLR values
- Reading typed content:
  - **`ReadContentAs()`** – the type is specified as a parameter
  - **`ReadContentAsObject()`**
    - If the content is typed, the reader returns a boxed CLR of the most appropriate type
    - If the content is a list type, the reader returns an array of boxed objects of the appropriate type
  - **`ReadContentAsBoolean(), ReadContentAsDateTime()`** **`ReadContentAsDouble(), ReadContentAsLong(),`** **`ReadContentAsInt(), ReadContentAsString(),`** **`ReadContentAsBase64(), ReadContentAsBinHex()`**
- Similar methods for reading typed element content are also provided (**`ReadElementContentAs`**…)

XmlReader Helper Methods

- The **`Skip()`** method is used to skip the subnodes of the current node
  - If the reader is currently positioned on a leaf node, calling **`Skip()`** is the same as calling **`Read()`**
- The **`MoveToContent()`** method is used to skip non-content nodes
- The **`ReadSubtree()`** method can be used to read an element and all its descendants
  - This method returns a new **`XmlReader`** instance set to the **`Initial`** state
  - Do not perform any operations on the original reader until the new reader has been closed

XmlWriter

The XmlWriter Class

- The XmlWriter class is an abstract base class that provides a forward-only, write-only, non-cached way of generating XML streams
- It allows to:
  - Verify that the characters are legal XML characters and that element and attribute names are valid XML names
  - Verify that the XML document is well-formed
  - Encode binary bytes as Base64, or BinHex, and write out the resulting text
  - Pass values using CLR types rather than strings
  - Write multiple documents to one output stream
  - Write valid names, qualified names, and name tokens

Creating XML Writers

- **`XmlWriter`** instances are created using the static **`XmlWriter.Create()`** method
- The **`XmlWriterSettings`** class is used to specify the set of features you want to enable on the new **`XmlWriter`** object
- The **`XmlTextWriter`** class is a concrete implementation of the **`XmlWriter`** class
  - Using the **`Create()`** method is recommended

Examples of Creating XML Writer

```
XmlWriterSettings settings = new XmlWriterSettings();
settings.Indent = true;
settings.IndentChars = ("    ");
using (XmlWriter writer = XmlWriter.Create("b.xml", settings)) {
   writer.WriteStartElement("book");
   writer.WriteElementString("price", "19.95");
   writer.WriteEndElement();
   writer.Flush();
}
```

```
XmlWriterSettings settings = new XmlWriterSettings();
settings.OmitXmlDeclaration = true;
settings.ConformanceLevel = ConformanceLevel.Fragment;
settings.CloseOutput = false;
MemoryStream strm = new MemoryStream();
XmlWriter writer = XmlWriter.Create(strm, settings);
writer.WriteElementString("orderID", "1-456-ab");
writer.WriteElementString("orderID", "2-36-00a");
writer.Flush();
writer.Close();
```

Formatting the Output of the XmlWriter

- Properties of the **`XmlWriterSettings`** class which control how the **`XmlWriter`** content is formatted
  - **`Encoding`** – the default is **`Encoding.UTF8`**
  - **`Indent`** – whether to indent elements (the default is **`false`**)
  - **`IndentChars`** – a character string used when indenting (the default is two spaces)
  - **`NewLineChars`** – the default is **`\r\n`**
  - **`NewLineHandling`** – **`Entitize`**, **`Replace`**, or **`None`**
  - **`NewLineOnAttributes`** – whether to write attributes in a new line
  - **`OmitXmlDeclaration`** – the default is **`false`**

Data Conformance

- The **`XmlWriterSettings.CheckCharacters`** property instructs the writer to check characters and throw an **`XmlException`** if any characters are outside the range of legal XML characters
  - It does not include checking for illegal characters in XML names
- The **`XmlWriterSettings.ConformanceLevel`** property configures the **`XmlWriter`** to check and guarantee that the stream being written complies with a certain set of rules
  - **`Document`** – the output conforms to the rules for a well-formed XML 1.0 document
  - **`Fragment`** – the XML data conforms to the rules for a well-formed XML 1.0 document fragment
  - **`Auto`** – the writer decides which level of conformance to apply based on the incoming data

Namespace Handling in the XmlWriter

- Declare namespaces manually

```
writer.WriteStartElement("root");
writer.WriteAttributeString("xmlns", "x", null, "urn:1");
writer.WriteStartElement("item","urn:1");
writer.WriteEndElement();
writer.WriteStartElement("item","urn:1");
writer.WriteEndElement();
writer.WriteEndElement();
```

```
<root xmlns:x="urn:1">
  <x:item/>
  <x:item/>
</root>
```

- Override current namespaces

```
writer.WriteStartElement("x","root","123");
writer.WriteStartElement("item");
writer.WriteAttributeString("xmlns","x",null,"abc");
writer.WriteEndElement();
writer.WriteEndElement();
```

```
<x:root xmlns:x="123">
  <item xmlns:x="abc" />
</x:root>
```

Writing Elements

- The **`WriteElementString()`** is used to write an entire element node, including a string value

```
writer.WriteElementString("price", "19.95");
```

```
<price>19.95</price>
```

- The **`WriteStartElement()`** allows to write the element value using multiple method calls

```
writer.WriteStartElement("bk", "book", "urn:books");
writer.WriteAttributeString("genre", "urn:books", "mystery");
writer.WriteElementString("price", "19.95");
writer.WriteEndElement();
```

```
<bk:book bk:genre="mystery"
         xmlns:bk="urn:books">
  <price>19.95</price>
</bk:book>
```

Writing Elements cont.

- The **`WriteNode()`** method allows to copy an entire element node found at the current position of the supplied **`XmlReader`** or **`XPathNavigator`** object

```
// Create a reader and position it on the book node
XmlReader reader = XmlReader.Create("books.xml");
reader.ReadToFollowing("book");
// Write out the book node
XmlWriter writer = XmlWriter.Create("newBook.xml");
writer.WriteNode(reader, true);
writer.Flush();
```

```
<?xml version="1.0" encoding="utf-8"?>
<book genre="autobiography" year="1996"
      ISBN="0-486-29073-5">
  <title>The Autobiography of Benjamin Franklin
  </title>
  <author>
    <first-name>Benjamin</first-name>
    <last-name>Franklin</last-name>
  </author>
  <price>8.99</price>
</book>
```

Writing Attributes

- The **`WriteAttributeString()`** method is used to write an entire attribute node, including a string value

```
writer.WriteAttributeString("supplierID", "A23-1");
```

```
supplierID='A23-1'
```

- The **`WriteStartAttribute()`** method allows to write the attribute value using multiple method calls

```
writer.WriteStartAttribute("review-date");
writer.WriteValue(hireDate.AddMonths(6));
writer.WriteEndAttribute();
```

Writing Attributes cont.

- **`WriteAttributes()`** allows to copy all the attributes found at the current position of the supplied **`XmlReader`**

```
XmlReader reader = XmlReader.Create("test1.xml");
XmlWriterSettings settings = new XmlWriterSettings();
settings.Indent = true;
XmlWriter writer = XmlWriter.Create(Console.Out);

while (reader.Read()) {
    if (reader.NodeType == XmlNodeType.Element) {
        writer.WriteStartElement(reader.Name.ToUpper());
        writer.WriteAttributes(reader, false);
        if (reader.IsEmptyElement)
            writer.WriteEndElement();
    } else if (reader.NodeType == XmlNodeType.EndElement) {
        writer.WriteEndElement();
    }
}
writer.Close();
reader.Close();
```

Writing Typed Data

- The **`WriteValue()`** method accepts CLR simple-typed values
  - When **`WriteValue()`** is called, the typed value is serialized to text using the **`XmlConvert`** class rules for that schema type

Boolean	xsd:boolean
Byte[]	xsd:base64Binary
DateTime	xsd:dateTime
Decimal	xsd:decimal
Double	xsd:double
Int32	xsd:integer
Int64	xsd:integer
Single	xsd:float
String	xsd:string

  - After conversion to string, the **`WriteValue()`** method writes the data out using the **`WriteString()`** method

Other Writing Possibilities

- **`WriteBase64(), WriteBinHex()`** – encode the specified binary bytes as base64 or BinHex resp., and writes out the resulting text
- **`WriteCData()`** – writes out a CData section containing the specified text
- **`WriteCharEntity()`** – writes out the Unicode character in hexadecimal character entity reference format
- **`WriteChars()`** – used to write large amounts of text one buffer at a time
- **`WriteComment()`** – writes out a comment **`<!--...-->`** containing the specified text
- **`WriteDocType()`** – writes out the DOCTYPE declaration with the specified name and optional attributes
- **`WriteEntityRef()`** - writes out an entity reference

Example of Writing XML File

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!--This XML file represents the details of an employee-->
<employees>
<employee id="1">
<name>
<firstName>Nancy</firstName>
<lastName>Edwards</lastName>
</name>
<city>Seattle</city>
<state>WA</state>
<zipCode>98122</zipCod
</employee>
</employees>
```

```
using (XmlWriter writer = XmlWriter.Create("C:\\a.xml")) {
  writer.WriteStartDocument(false);
  writer.WriteComment("The details of an employee");
  writer.WriteStartElement(employees");
    writer.WriteStartElement(employee");
    writer.WriteAttributeString("id", "1");
      writer.WriteStartElement("name");
        writer.WriteElementString("firstName", "Nancy");
        writer.WriteElementString("lastName", "Edwards");
      writer.WriteEndElement();
      writer.WriteElementString("city", "Seattle");
      writer.WriteElementString("state", "WA");
      writer.WriteElementString("zipCode", "98122");
    writer.WriteEndElement();
  writer.WriteEndElement();
  writer.WriteEndDocument();
  writer.Flush();
}
```

Example of Embedding an Image

```
using (XmlWriter writer = XmlWriter.Create("C:\\a.xml")) {
  writer.WriteStartDocument(false);
  writer.WriteStartElement("employee");
  writer.WriteAttributeString("id", "1");
  writer.WriteStartElement("image");
  writer.WriteAttributeString("fileName", imageFileName);

  FileInfo fi = new FileInfo(imageFileName);
  int size = (int)fi.Length;
  FileStream stream = new FileStream(imageFileName,FileMode.Open);
  BinaryReader reader = new BinaryReader(stream);
  byte[] imgBytes = reader.ReadBytes(size);
  reader.Close();

  writer.WriteBinHex(imgBytes, 0, size);
  writer.WriteEndElement();
  writer.WriteEndElement();
  writer.WriteEndDocument();
  writer.Flush();
}
```

XML DOM
in the .NET Framework

.NET Framework Classes

- **`XmlDocument`** – the container of all nodes
- **`XmlDocumentFragment`** – one or more nodes (without any tree structure)
- **`XmlDocumentType`** – represents <!DOCTYPE> node
- **`XmlEntityReference`** – non-expanded entity reference text
- **`XmlElement`** – an element node
- **`XmlAttribute`** – an attribute of an element
- **`XmlProcessingInstruction`** – a processing instruction node
- **`XmlComment`** – a comment node
- **`XmlText`** – text belonging to an element or attribute
- **`XmlCDataSection`** – represents CDATA
- **`XmlEntity`** – represents <!ENTITY> declaration
- **`XmlNotation`** – represents a notation declared in the DTD

.NET DOM Hierarchy

- The following classes do not inherit from the **`XmlNode`**:
  - **`XmlImplementation`**
  - **`XmlNamedNodeMap`**
  - **`XmlNodeList`**
  - **`XmlNodeChangedEventArgs`**

Reading an XML Document into the DOM

- To read XML information into memory, use:
  - The **`XmlDocument.Load()`** method to read from a stream, URL, text reader, or a class derived from the **`XmlReader`**
  - The **`XmlDocument.LoadXml()`** method to read from a string
- Reading from an **`XmlReader`** provides special possibilities
  - If the reader is in its initial state, **`Load()`** consumes the entire contents from the reader
  - If the reader is already positioned on a node, **`Load()`** attempts to read the current node and all of its siblings, up to the end tag that closes the current depth into memory
  - By default, **Load()** does not verify whether the XML is valid (it only verifies whether the XML is well-formed)
    - To force validation use the **`ValidationType`** property in the **`XmlReaderSettings`** applied for the **`XmlReader`**

Adding New Nodes

- The **`XmlDocument`** has a create method for all of the node types:

  `CreateCDataSection, CreateComment, CreateDocumentFragment, CreateDocumentType, CreateElement, CreateNode, CreateProcessingInstruction, CreateSignificantWhitespace, CreateTextNode, CreateWhitespace, CreateXmlDeclaration`

- Once new nodes are created, one of the following method can be used to add them to the document:
  - **`XmlNode.InsertBefore()`**
  - **`XmlNode.InsertAfter()`**
  - **`XmlNode.AppendChild()`** – as the last child
  - **`XmlNode.PrependChild()`** – as the first child
- The **`XmlDocument.ImportNode()`** method can be used to copy a node or an entire node subtree from one **`XmlDocument`** to another

Removing the Data

- Removing nodes:
  - **`XmlNode.RemoveChild(), XmlNode.RemoveAll()`**
- Removing attributes:
  - **`XmlAttributeCollection`** methods: **`Remove(), RemoveAt(), RemoveAll()`**
  - **`XmlElement`** methods: **`RemoveAtrribute(), XmlElement.RemoveAt (), RemoveAllAttributes()`**
- Removing node content:
  - **`XmlCharacterData.DeleteData()`** (**`XmlComment, XmlText, XmlCDataSection, XmlWhitespace, XmlSignificantWhiteSpace`**)
  - To remove content completely, remove the node that contains the content
  - To clear the content, modify it in the node

Modifying the Data

- **`XmlNode.Value`** – changing the value of nodes
  - For an attribute – the value
  - For a CDATA section, comment, processing instruction, or text – the content
  - For an XmlDeclaration – the content, excluding **`<?xml`** and **`?>`** markup
  - For a Whitespace, SignificantWhitespace – the value
- **`XmlNode.InnerXml`** – modifying an entire set of nodes
  - Setting this property replaces the child nodes with the parsed contents of the given string
- **`XmlCharacterData`** methods: **`AppendData(), InsertData(), ReplaceData(), DeleteData()`**

Events of the XmlDocument

- Events of the **`XmlDocument`** class:
  - **`NodeInserting, NodeInserted`**
  - **`NodeRemoving, NodeRemoved`**
  - **`NodeChanging, NodeChanged`** – when the **`Value`** property of a node belonging to this document has been changed (is about to be changed)
- Properties of the **`XmlNodeChangedEventArgs`** structure (used for all events)
  - **`Node`**
  - **`OldParent, NewParent`**
  - **`OldValue, NewValue`**
  - **`Action`** – **`Insert, Remove, Change`**

Validating an XML Document in the DOM

- The **`Validate`** method of the **`XmlDocument`** class validates the XML data loaded in the DOM against the schemas in the **`XmlDocument`** object's **`Schemas`** property
- After successful validation:
  - Schema defaults are applied
  - Text values are converted to atomic values as necessary
  - Type information is associated with validated information items
  - As a result, typed XML data replaces previously untyped XML data
- The **`ValidationEventHandler`** object (assigned to the **`XmlReaderSettings`** or passed to the **`Validate()`** method) will handle schema validation errors
  - If no **`ValidationEventHandler`** is used, **`XmlSchemaValidationException`** can be thrown

Saving a Document

- The **`XmlDocument.Save()`** method can write the XML document to a stream, string, **`TextWriter`**, or **`XmlWriter`**

```
XmlDocument doc = new XmlDocument();
doc.Load("text.xml");
XmlTextWriter tw = new XmlTextWriter("out.xml", null);
doc.Save(tw);
```

```
XmlDocument doc = new XmlDocument();
doc.AppendChild(doc.CreateElement("item","urn:1"));
doc.Save(Console.Out);
```

```
<item
    xmls="urn:1"/>
```

- The saved document may differ from the original
  - Whitespaces and a quoting character can be changed and numeric character entities can be expanded

Navigating Through the Hierarchy

- Properties of the **`XmlNode`** class:
  - **`NextSibling, PreviousSibling`**
  - **`ChildNodes`**
    - The type of this property is the **`XmlNodeList`** collection
    - Changes to the children of the node object that it was created from are immediately reflected in the nodes returned by the **`XmlNodeList`** properties and methods
  - **`FirstChild, LastChild`**
  - **`ParentNode`**
  - **`OwnerDocument`**

```
void DisplayNodes(XmlNode node) {
    // Print the node type, node name and node value of the node
    if (node.NodeType == XmlNodeType.Text) {
        Console.WriteLine("Type= [" + node.NodeType +
                          "] Value=" + node.Value + "\n");
    } else {
        Console.WriteLine("Type= [" + node.NodeType +
                          "] Name=" + node.Name + "\n");
    }
    // Print attributes of the node
    if (node.Attributes != null) {
        XmlAttributeCollection attrs = node.Attributes;
        foreach (XmlAttribute attr in attrs) {
            Console.WriteLine("Attribute Name =" + attr.Name +
                              "Attribute Value =" + attr.Value);
        }
    }
    //Print individual children of the node
    XmlNodeList children = node.ChildNodes;
    foreach (XmlNode child in children) {
        DisplayNodes(child);
    }
}
```

Finding Nodes

- **`XmlDocument.GetElementsByTagName()`, `XmlElement.GetElementsByTagName()`**
  - These methods return an **`XmlNodeList`** object containing references to nodes that have a given name
  - They may return nodes at different levels of the subtree
  - The **`XmlElement`** version searches the document subtree rooted at the element
- **`XmlDocument.GetElementById()`**
  - It returns the first node with a specified ID attribute, that has been found
  - The DOM implementation has to include information which defines which attributes are of type ID
    - In current version only DTD can be used

Selecting Nodes

- The **`XmlNode.SelectNodes()`** method returns an **`XmlNodeList`** containing references to nodes that match a specified XPath expression
  - XPath allows to specify a very complex set of node criteria to select nodes
  - The **`XmlNodeList`** should not be expected to be connected "live" to the XML document (that is, changes that appear in the XML document may not appear in the **`XmlNodeList`**, and vice versa)
- The **`XmlNode.SelectSingleNode()`** method returns only the first node that matches an XPath expression

XPath Expression Syntax

- `/` - starts an absolute path that selects from the root node
  - **`/bookstore/book/title`** - all title elements that are children of the book element, which is itself a child of the root
- `//` - starts a relative path that selects nodes anywhere
  - **`//book/title`** - all of the title elements that are children of a book element, regardless of where they appear in the document
- `@` - selects an attribute of a node
  - **`/book/@genre`** - selects the attribute named genre from the root book element
- `*` - any element in the path
  - **`/book/*`** - selects all nodes that are contained by a root book element
- `|` - union operator
  - **`/bookstore/book/title | bookstore/book/author`** - selects the title nodes and the author nodes

XPath Expression Syntax cont.

- `[]` - defines selection criteria that can test a contained node or attribute value
  - `/book[@genre="autobiography"]` - selects the book elements with the indicated attribute value
- `starts-with` - retrieves elements based on what text a contained element starts with
  - `/bookstore/book/author[starts-with(first-name, "B")]`
- `position` - retrieves elements based on position
  - `/bookstore/book[position()=2]` - selects the second book element.
- `count` - counts elements
  - `/bookstore/book/author[count(first-name) = 1]` - retrieves author elements that have exactly one nested first-name element

XPath Expression Syntax cont.

- The Axis Specifier indicates navigation direction within the tree representation of the XML document
  - **`child`** (the default in abbreviated syntax)
  - **`attribute`** (@ in abbreviated syntax)
  - **`descendant`**
  - **`descendant-or-self`** (//)
  - **`parent`** (..)
  - **`ancestor`**
  - **`ancestor-or-self`**
  - **`following`**
  - **`preceding`**
  - **`following-sibling`**
  - **`preceding-sibling`**
  - **`self`** (.)
  - **`namespace`**

Examples of Selecting Nodes

```
<?xml version='1.0'?>
<bookstore xmlns="urn:newbooks-schema">
  <book genre="novel" style="other">
    <title>Pride and Prejudice</title>
    <author>
      <first-name>Jane</first-name>
      <last-name>Austen</last-name>
    </author>
    <price>11.99</price>
  </book>
</bookstor
```

```
XmlDocument doc = new XmlDocument();
doc.Load("booksort.xml");
XmlNodeList nodeList;
XmlNode root = doc.DocumentElement;
nodeList = root.SelectNodes(
       "descendant::book[author/last-name='Austen']");
// Change the price on the books
foreach (XmlNode book in nodeList) {
    book.LastChild.InnerText="15.95";
}
doc.Save(Console.Out);
```

The XPathDocument Class

- The **`XPathDocument`** class provides a fast, read-only, in-memory representation of an XML document using the XPath data model
- The **`CreateNavigator()`** method creates a read-only **`XPathNavigator`** object for navigating through nodes in this **`XPathDocument`**

The XPathNavigator Class

- It is an abstract class which defines a cursor model for navigating and editing XML information items as instances of the XQuery 1.0 and XPath 2.0 Data Model
  - **`XPathNavigator`** objects created by **`XPathDocument`** objects are read-only
  - **`XPathNavigator`** objects created by **`XmlDocument`** objects can be edited
- The **`XPathNavigator`** class allows to optimize performance when working with XPath queries
  - The classes from the **`System.Xml.XPath`** namespace can be used

Examples of Using XPathNavigator

```
XPathDocument document = new XPathDocument("books.xml");
XPathNavigator navigator = document.CreateNavigator();
XPathNodeIterator nodes = navigator.Select("/bookstore/book");
while(nodes.MoveNext()) {
    Console.WriteLine(nodes.Current.Name);
}
```

```
XPathDocument document = new XPathDocument("books.xml");
XPathNavigator navigator = document.CreateNavigator();
XPathExpression query = navigator.Compile("sum(//price/text())");
Double total = (Double)navigator.Evaluate(query);
Console.WriteLine(total);
```

```
XPathDocument document = new XPathDocument("input.xml");
XPathNavigator navigator = document.CreateNavigator();
navigator.Matches("b[@c]");
```

Examples of Using XPathNavigator cont.

```
XmlDocument document = new XmlDocument();
document.Load("contosoBooks.xml");
XPathNavigator nav = document.CreateNavigator();

nav.MoveToChild("bookstore","http://www.contoso.com/books");
nav.MoveToChild("book", "http://www.contoso.com/books");
nav.MoveToChild("price", "http://www.contoso.com/books");

nav.InsertBefore("<pages>100</pages>");
// or any other change, e.g.
// nav.DeleteSelf();

nav.MoveToParent();
Console.WriteLine(nav.OuterXml);
```

XSLT Transformations

XSLT

- The Extensible Stylesheet Language Transformation (XSLT) allows to transform the content of a source XML document into another document that is different in format or structure
- The transformation process needs two input files:
  - The XML document, which makes up the source tree
  - The XSLT file, which consists of elements used to transform data to the required format
- In the transformation process, XSLT uses XPath to define parts of the source document that match one or more predefined templates

.NET Classes

- The **`XslCompiledTransform`** class is the Microsoft .NET Framework XSLT processor
  - This class is used to compile style sheets and execute XSLT transformations
- Other classes involved in XSL transformations:
  - **`XsltSettings`** – allows to specify the XSLT features to support during execution of the XSLT style sheet
  - **`XsltArgumentList`** – allows to pass a variable number of parameters and extensions objects to an XSL style sheet
  - **`XsltCompileException`**
- The **`XslTransform`** class is obsolete in the Microsoft .NET Framework version 2.0

XsltCompiledTransform.Transform()

- The **`Transform()`** method accepts three input types for the source document:
  - An object that implements the **`IXPathNavigable`**
    - The **`XmlNode`** and **`XPathDocument`** classes implement the **`IXPathNavigable`** interface
    - **`XPathDocument`** is the recommended class for XSLT processing, it provides faster performance when compared to the **`XmlNode`** class
    - Transformations apply to the document as a whole
      - To transform a node fragment, an object containing just the node fragment must be created and passed to the method
  - An **`XmlReader`** that reads the source document
  - A string URI

XsltCompiledTransform.Transform() cont.

- The following list describes the output types available on the **`Transform()`** command:
  - **`XmlWriter`** – also the **`XmlWriterSettings`** class can be used to specify output options
    - If the style sheet uses the **`xsl:output`** element, use the **`XslCompiledTransform.OutputSettings`** to get the **`XmlWriterSettings`** object
  - **`String`** – the URI of the output file
  - **`Stream`**
  - **`TextWriter`** – useful for output to a string

```
// Load the style sheet
XslCompiledTransform xslt = new XslCompiledTransform();
xslt.Load("output.xsl");
// Execute the transform and output the results to a file
xslt.Transform("books.xml", "books.html");
```

XSLT Extension Objects

- Extension objects are used to extend the functionality of style sheets
  - They are maintained by the **`XsltArgumentList`** class
- XSLT parameters are added to the **`XsltArgumentList`** using the **`AddParam()`** method
- The **`XslCompiledTransform`** class supports embedded scripts using the **`msxsl:script`** element
  - When the style sheet is loaded, any defined functions are compiled to MSIL

Example of Extending XSLT

```
<!--Represents a customer order-->
<order>
  <book ISBN='10-861003-324'>
    <title>The Handmaid's Tale</title>
    <price>19.95</price>
  </book>
  <cd ISBN='2-3631-4'>
    <title>Americana</title>
    <price>16.95</price>
  </cd>
</order>
```

Output

```
<?xml version="1.0"
      encoding="utf-8"?>
<order>
  <total>36.9</total>
     15% discount if paid by:
     2/4/2004 12:00:00 AM
</order>
```

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:param name="discount"/>
  <xsl:template match="/">
    <order>
      <xsl:variable name="sub-total" select="sum(//price)"/>
      <total><xsl:value-of select="$sub-total"/></total>
           15% discount if paid by:
           <xsl:value-of select="$discount"/>
    </order>
  </xsl:template>
</xsl:stylesheet>
```

Example of Extending XSLT cont.

```
// Create the XslCompiledTransform and load the style sheet
XslCompiledTransform xslt = new XslCompiledTransform();
xslt.Load("discount.xsl");

// Create the XsltArgumentList
XsltArgumentList argList = new XsltArgumentList();

// Calculate the discount date
DateTime orderDate = new DateTime(2004, 01, 15);
DateTime discountDate = orderDate.AddDays(20);
argList.AddParam("discount", "", discountDate.ToString());

// Create an XmlWriter to write the output
XmlWriter writer = XmlWriter.Create("orderOut.xml");

// Transform the file
xslt.Transform(new XPathDocument("order.xml"), argList, writer);
writer.Close();
```

Transforming a Node Fragment

```
// Load an XPathDocument
XPathDocument doc = new XPathDocument("books.xml");

// Locate the node fragment
XPathNavigator nav = doc.CreateNavigator();
XPathNavigator myBook = nav.SelectSingleNode(
                  "descendant::book[@ISBN = '0-201-63361-2']");

// Create a new object with just the node fragment
XmlReader reader = myBook.ReadSubtree();
reader.MoveToContent();

// Load the style sheet
XslCompiledTransform xslt = new XslCompiledTransform();
xslt.Load("single.xsl");

// Transform the node fragment
xslt.Transform(reader,
            XmlWriter.Create(Console.Out, xslt.OutputSettings));
```

XML and ADO.NET

The ReadXml() Method

- The contents of an ADO.NET DataSet can be created from an XML stream or document
- The **`DataSet.ReadXml()`** method fills a DataSet with data from XML
  - It can read the XML data from a file, a stream, or an **`XmlReader`**
- If the DataSet already contains data, the new data from the XML is added to the data already present in the DataSet
  - **`ReadXml()`** does not merge any row information with matching primary keys from the XML into the DataSet
  - To overwrite existing row information with new information from XML, use **`ReadXml()`** to create a new DataSet, and then use the **`Merge()`** method

The XmlReadMode Argument

- Possible values of the **`XmlReadMode`** argument (optionally passed to the **`ReadXml()`** method):
  - **`Auto`** – the default
  - **`ReadSchema`** – reads an inline schema and loads it
  - **`IgnoreSchema`** – ignores any inline schema, any data that does not match the existing schema is discarded
  - **`InferSchema`** – ignores any inline schema and infers the schema from the XML data, then loads the data
  - **`DiffGram`** – reads a DiffGram and adds the data to the current schema
  - **`Fragment`** – continues reading multiple XML fragments until the end of the stream, fragments that do not match the DataSet schema are discarded

DiffGram

- A DiffGram is an XML format that identifies current and original versions of data elements
  - The DataSet uses the DiffGram format to load and persist its contents, and to serialize its contents for transport across a network connection

```
<?xml version="1.0"?>
<diffgr:diffgram
        xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
        xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <DataInstance>
  </DataInstance>

  <diffgr:before>
  </diffgr:before>

  <diffgr:errors>
  </diffgr:errors>
</diffgr:diffgram>
```

```
<diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-
msdata" xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
  <CustomerDataSet>
    <Customers diffgr:id="Customers1" msdata:rowOrder="0"
               diffgr:hasChanges="modified">
      <CustomerID>ALFKI</CustomerID>
      <CompanyName>New Company</CompanyName>
    </Customers>
    <Customers diffgr:id="Customers2" msdata:rowOrder="1"
               diffgram:hasErrors="true">
      <CustomerID>ANATR</CustomerID>
      <CompanyName>Ana Trujillo Emparedados</CompanyName>
    </Customers>
    <Customers diffgr:id="Customers3" msdata:rowOrder="2">
      <CustomerID>ANTON</CustomerID>
      <CompanyName>Antonio Moreno Taquera</CompanyName>
    </Customers>
   </CustomerDataSet>
  <diffgr:before>
    <Customers diffgr:id="Customers1" msdata:rowOrder="0">
      <CustomerID>ALFKI</CustomerID>
      <CompanyName>Alfreds Futterkiste</CompanyName>
    </Customers>
  </diffgr:before>
  <diffgr:errors>
    <Customers diffgr:id="Customers2" diffgr:Error="An optimistic
              concurrency violation has occurred for this row."/>
  </diffgr:errors>
</diffgr:diffgram>
```

Schema Inference Rules

- When inferring a schema from an XML document, a DataSet follows these rules:
  - Elements with attributes become tables
  - Elements with child elements become tables
  - Repeating elements become columns in a single table
  - Attributes become columns
  - If the document element has no attributes and no child elements that can be inferred to be columns, it is inferred to be a DataSet; otherwise, the document element becomes a table
  - For elements inferred to be tables that have no child elements and contain text, a new column called Tablename_Text is created for the text of each of the elements
  - If an element with both child nodes and text is inferred to be a table, the text is ignored
  - For elements that are inferred to be tables nested within other elements inferred to be tables, a nested **`DataRelation`** is created between the two tables

ReadXmlSchema() and InferXmlSchema()

- The **`ReadXmlSchema()`** method can be used to load only DataSet schema information (and no data) from an XML document
  - It takes a stream, an **`XmlReader`**, or a file name
  - When a DataSet already contains a schema, the existing schema is extended
- The **`InferXmlSchema()`** method can be used to load the DataSet schema from an XML document
  - This method has the same functionality as that of the **`ReadXml()`** method that uses the **`XmlReadMode`** enumeration value set to **`InferSchema`**
  - The second parameter is a string array of the namespaces that need to be ignored when inferring the schema

XmlWrite() and GetXml()

- The **`DataSet.WriteXml()`** method allows to write the XML representation of the DataSet to a file, stream, an **`XmlWriter`** object, or a string
- The **`XmlWriteMode`** enumeration (an optional parameter):
  - **`IgnoreSchema`** - writes the current contents of the DataSet as XML data, without an XML Schema (the default)
  - **`WriteSchema`** - writes the current contents of the DataSet as XML data with the relational structure as inline XML Schema
  - **`DiffGram`** - writes the entire DataSet as a DiffGram, including original and current values
- The **`DataSet.GetXml()`** method returns the XML representation of the data stored in the DataSet (as a string)
  - It works like the **`WriteXml()`** method with the **`IgnoreSchema`** option

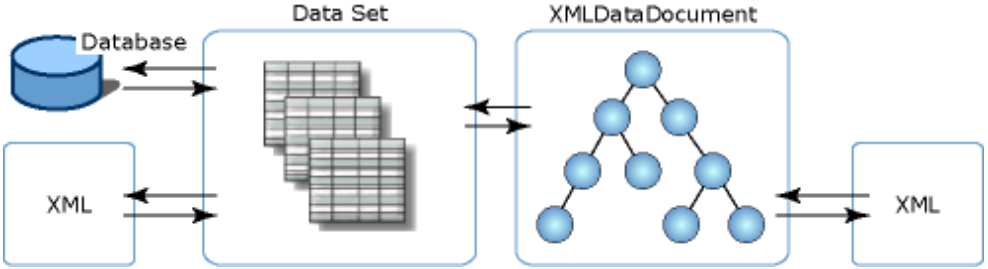
WriteXmlSchema() and GetXmlSchema()

- The **`DataSet.WriteXmlSchema()`** method writes the DataSet structure as an XML schema
  - It can write the schema to a stream, string, or an **`XmlWriter`** object
- The **`DataSet.GetXmlSchema()`** method returns the XML schema for the XML representation of the data stored in the DataSet
  - Calling this method is identical to calling **`WriteXmlSchema()`**, except that only the primary schema is written

```
DataSet originalDataSet = new DataSet("dataSet");

// ... creating tables

// Write the schema and data to XML file with FileStream
string xmlFilename = "XmlDocument.xml";
System.IO.FileStream streamWrite = new System.IO.FileStream(
    xmlFilename, System.IO.FileMode.Create);

// Use WriteXml to write the XML document
originalDataSet.WriteXml(streamWrite);

// Close the FileStream
streamWrite.Close();

// Dispose of the original DataSet
originalDataSet.Dispose();

// Create a new DataSet
DataSet newDataSet = new DataSet("New DataSet");

// Read the XML document back in
// Create new FileStream to read schema with
System.IO.FileStream streamRead = new System.IO.FileStream(
    xmlFilename,System.IO.FileMode.Open);
newDataSet.ReadXml(streamRead);
```

The XmlDataDocument Class

- The **`XmlDataDocument`** class is a derived class of the **`XmlDocument`**, and contains XML data
- The **`XmlDataDocument.DataSet`** property returns a reference to the DataSet that provides a relational representation of the data in the XmlDataDocument



The Benefits of Using XmlDataDocument

- The structured portion of an XML document can be mapped to a dataset, and be efficiently stored, indexed, and searched
- Transformations, validation, and navigation can be done efficiently through a cursor model over the XML data that is stored relationally
- It allows to use XPath queries
- The order of the elements is preserved
- White spaces and formatting in the source XML are also preserved

Example of Using XmlDataDocument

```
// Create an XmlDataDocument
XmlDataDocument doc = new XmlDataDocument();
// Load the schema file
doc.DataSet.ReadXmlSchema("store.xsd");
// Load the XML data
doc.Load("2books.xml");
// Update the price on the first book using the DataSet methods
DataTable books = doc.DataSet.Tables["book"];
books.Rows[0]["price"] = "12.95";
// Display the modified XML data
doc.Save(Console.Out);
```

```
<!--sample XML fragment-->
<bookstore>
  <book genre='novel' ISBN='10-861003-324'>
    <title>The Handmaid's Tale</title>
    <price>19.95</price>
  </book>
  <book genre='novel' ISBN='1-861001-57-5'>
    <title>Pride And Prejudice</title>
    <price>24.95</price>
  </book>
</bookstore>
```