Programming XML with C#

Programming XML with C# is a book written in step-by-step tutorial format for beginners and students who want to learn XML programming using C# language. It is recommended that you have some programming experience using any of the object-oriented languages such as C++, Pascal, or Java. It is also recommended that you are familiar with C# language syntaxes and programming. If you are not a C# programmer, I recommend to read Programming C# for Beginners before this book. This book can be found in C# Programming section of C# Corner.

In this book, you will learn the basic elements of XML and classes and objects available in .NET Framework to work with XML. After that, you will learn how to read, write, updated, and transform XML using C#. .NET also provides support for relationships between data (via ADO.NET) and XML. In this chapter, I also discuss how you can take advantages of classes found on ADO.NET and connect data with XML.

Table of Contents

Introduction to XML

Note: If you are familiar with HTML and XML, you may skip this section and jump to XML Representation in .NET World section.

The ADO.NET and XML.NET Framework Application Programming Interface (API) combination provides a unified way to work with XML in the Microsoft .NET Framework. There are two ways to represent data using XML: in a tagged-text format metalanguage similar to HTML and in a relational table format. You use ADO .NET to access relational table formats. You would use DOM to access the text format.

Before talking about the role of XML in the .NET Framework and how to work with it, it's important you understand the basic building blocks of XML and its related terminology. You'll learn the basic definitions of Standard Generalized Markup Language (SGML) and HTML in the following sections. If you're already familiar with these languages, you can skip to the "XML Overview" section.

Standard Generalized markup Language (SGML)

In 1986, Standard Generalized Markup Language (SGML) because the international standards for representing electronic documents in a unified way. SGML provides a standard format for designing your own markup schemes. **Markup** is a way to represent some information about data.

Later Hypertext Markup Language (HTML) became the international standard for representing documents on the Web in a unified way.

Hyper text Markup Language (HTML)

The HTML file format is text format that contains, rather heavily. Markup tags. A tag is a section of a program that starts with < and ends with > such as <name>. (An **element** consists of a pair of tags, starting with <name> and ending with </name>). The language defines all of the markup tags. All browsers support HTML tags, which tell a browser how to display the text of an HTML document. You can create an HTML file using a simple text editor such as Notepad. After typing text in a text editor, you save the file with `an.htm` or `.html` extension.

NOTE: An HTML document is also called HTML pages or HTML file.

Listing 6-1 shows an example of an HTML file, type the following in a text editor, and save it `myfile.htm`.

Listing 6-1. A simple HTML file

```
<html>

<head>
<title> A Test HTML Page </title>
</head>

<body>
Here is the body part.
</body>

</html>
```

If you view this field in a browser, you'll see the text Here is the body part. In Listing 6-1, your HTML file starts with the <html> tag and ends with the </html> tag. The <html> tag tells a browser that this is the starting point of an HTML document. The </html> tag tells a browser that this is the ending point of an HTML documents. These tags are required in all HTML documents. The <head> tag is header information of a document and is not displayed in the browser. The <body> and</body> tags, which are required, makeup the main content of a document. As you can see, all tags ends with a<\> tag.

NOTE: HTML tags are not case sensitive. However, the World Wide Web Consortium (W3C) recommends using lowercase tags in HTML4. The next generation of HTML, XHTML, doesn't support uppercase tags. (The W3C promotes the web worldwide and makes it more it more useful. You can find more information on the W3C at http://www.w3c.org.)

Tags can have **attributes**, which provide additional information about the tags. Attributes are part of the starting tag. For example:

```
<table border ="0">
```

In this example the <table> tag has an attribute border and its value is 0. This value applies to the entire <table> tag, ending with the </table> tag. Table 6-1 describes some common HTML tags.

Table 6-1 Common HTML Tags

TAG	DESCRIPTION
<html>	Indicates start and end of an HTML document
<title>	Contains the title of the page
<body>	Contains the main content, or body, of the page
<h1…h6>	Creates headings (from level 1 to 6)
<p>	Starts a new paragraph
 	Insert a single line break
<hr>	Defines a horizontal rule
<!-->	Defines a comment tag in a document
<b>	Defines bold text
<I>	Defines italic text
<Strong>	Defines strong text
<table>	Defines a table
<tr>	Defines a row of a table
<td>	Defines a cell of a table row
<font>	Defines a font name and size

There are comes tags beyond those described in table 6-1. In fact the W3C's HTML 4 specification is quite extensive. However, discussing all of the HTML tags is beyond the scope of this article. Before moving to the next topic, you'll take a look at one more HTML example using the tags discussed in the table. Listing 6-2 shows you another HTML document example.

Listing 6-2. HTML tag their usage

```
<html>
<head>
<title> A Test HTML Page</title>
</head>
<!- – This is a comment - ->
```

```
<body>
<h1> Heading 1</h1>
<h2 Heading 2</h2>
<p><b><i><font size = "4">Bold and Italic Text. </font></i></b></p>
<table border = "1" width ="43%">
<tr>
<td width = "50%">Row1, Column1</td>
<td width = "50%">Row1, column2</td>
</tr>
<tr>
<td width = "50%"> Row2, Column1</td>
<td width = "50%"> Row2, Column2</td>
</tr>
</table>
</body>
</html>
```

NOTE: In Listing 6-2, the `<font>` and `<td>` tags contain size and width attributes, respectively. The `size` attribute tells the browser to display the size of the font, which is 4 in this example, and the `width` attribute tells the browser to display the table cell as 50 percent of the browser window.

XML Overview

I'll now cover some XML-related terminology. So what exactly is XML? XML stands for Extensible Markup Language. It's family member of SGML and an extended version of HTML. If you've ever gotten your hands dirty with HTML, then XML will be piece of cake.

Essentially XML extends the power and flexibility of HTML. You don't have to work a limited number of tags as you do in HTML. You can define your own tags. And you can store your data in structured format.

Unlike HTML, XML stores and exchanges data. By contrast, HTML represents the data. You can create separate XML files to store data, which can be used as a data source for HTML and other applications.

You'll now see an XML example. Listing 6-3 shows a simple XML file: `books.Xml`. By default, this file comes with Visual Studio (VS).NET if you have VS .NET or the .NET Framework installed on your machine; you probably have this file in your sample folder.

You'll create this XML file called `books.xml`, which will store data about books in a bookstore. You'll create a tag for each of these properties, such as a <title> tag that will store the title of the book and so on.

You can write an XML file in any XML editor or text editor. Type the code shown in listing 6-3 and save the file as `books.xml`.

This file stores information about a bookstore. The root node of the document is `<bookstore>`. Other tags follow the `<bookstore>` tag, and the document ends with the`</bookstore>` tag. Other tags defined inside the `<bookstore>` tag are `<book>`, `<title>`, `<author>`, and `<price>`. The tags store information on the store name, book publication date, book ISBN number, book title, author's name and price.

Listing 6-3. Your first XML file sample

```
<?xml version ='1.0'?>
<bookstore>
<book>
<title>The Autobiography of Benjamin Franklin</title>
<author>
<first-name>Benjamin</first-name>
<last-name>Franklin</last-name>
</author>
<price>8.99</price>
</book>
<book>
<title> The Confidence Man</title>
<author>
<first-name>Herman</first-name>
<last-name>Melville</last-name>
</author>
<price>11.99</price>
</book>
<book>
<title>The Gorgias</title>
<author>
<name>Plato</name>
</author>
<price>9.99</price>
</book>
</bookstore>
```

The first line of an XML file looks like this: `<? Xml version ="1.0"? >`. This line defines the XML version of the document. This tag tells the browser to start executing the file. You may have noticed that `<?>` doesn't have an ending `</?>` tag. Like HTML, other tags in an XML document start with < and are followed by a/> tag. For example, the`<title>` tag stores the book's title like this: `<title>` The Gorgias`</title>`.

In Listing 6-3, <`bookstore`> is the root node. Every XML document must start with a root node with the starting tag and end with the root node ending tag; otherwise the XML passer gives an error. (I'll discuss XML parsers shortly.)

Now, if you view this document in a browser, the output looks like Listing 6-4.

Listing 6-4. Output of books.xml in the browser

```
<?xml version="1.0" ?>
-<bookstore>
  -<book>
      <title>The Autobiography of Benjamin Franklin</title>
    -<author>
        <first-name>Benjamin</first-name>
        <last-name>Franklin</last-name>
      </author>
      <price>8.99</price>
    </book>
  -<book>
      <title>The Confidence Man</title>
    - <author>
```

```
        <first-name>Herman</first-name>
        <last-name>Melville</last-name>
      </author>
      <price>11.99</price>
    </book>
  -<book>
      <title>The Gorgias</title>
    - <author>
        <name>Plato</name>
      </author>
      <price>9.99</price>
    </book>
  </bookstore>
```

Your browser recognizes the XML and colors it appropriately.

Important Characteristics of XML

There are few things you need to know about XML. Unlike HTML, XML is case sensitive. In XML, `<Books>` and `<books>` are two different tags. All tag in xml must be well formed and must have a closing tag. A language is **well formed** only if it follows exact language syntaxes the way they are defined.

Improper nesting of tags in XML won't the document property. For example:

```
<b><i>Bold and Italic Text.</b></i>
```

is not well-formed. The well- formed version of the same code is this:

```
<b><i>Bold and Italic Text.</i></b>
```

Another difference between HTML and Xml is that attributes must use double quotes in XML. Attributes function like HTML attributes and are extra information you can add to a tag. (I'll discuss attributes in the "An XML Document and its Items" section later in this article.) Having attributes without double quotes is improper in XML. For example, Listing 6-5 is a correct example of using the attributes `ISBN`, `genre`, and `Publication date inside the <book>`tag.

Listing 6-5 Attributes in XML files

```
?xml version ='1.0'?>
<!-- This file represents a fragment of a book store inventory database -->
<bookstore>
<book genre = "autobiography" publicationdate = "1981" ISBN ="1-861003-11- 0">
<title>The Autobiography of Benjamin Franklin</title>
<author>
<first-name>Benjamin</first-name>
<last-name>Franklin</last-name>
</author>
<price>8.99</price>
</book>
</bookstore>
```

The `genre, publicationdate, and ISBN` attributes store information about the category, publication date, and ISBN number of the book, respectively. Browsers won't have a problem parsing the code in listing 6-5, but if you remove the double quotes the attributes like this:

```
<book genre = autobiography publicationdate = 1981  ISBN =1-861003-11-0>
```

then the browser will give the error message shown in F**igure 6-1.**

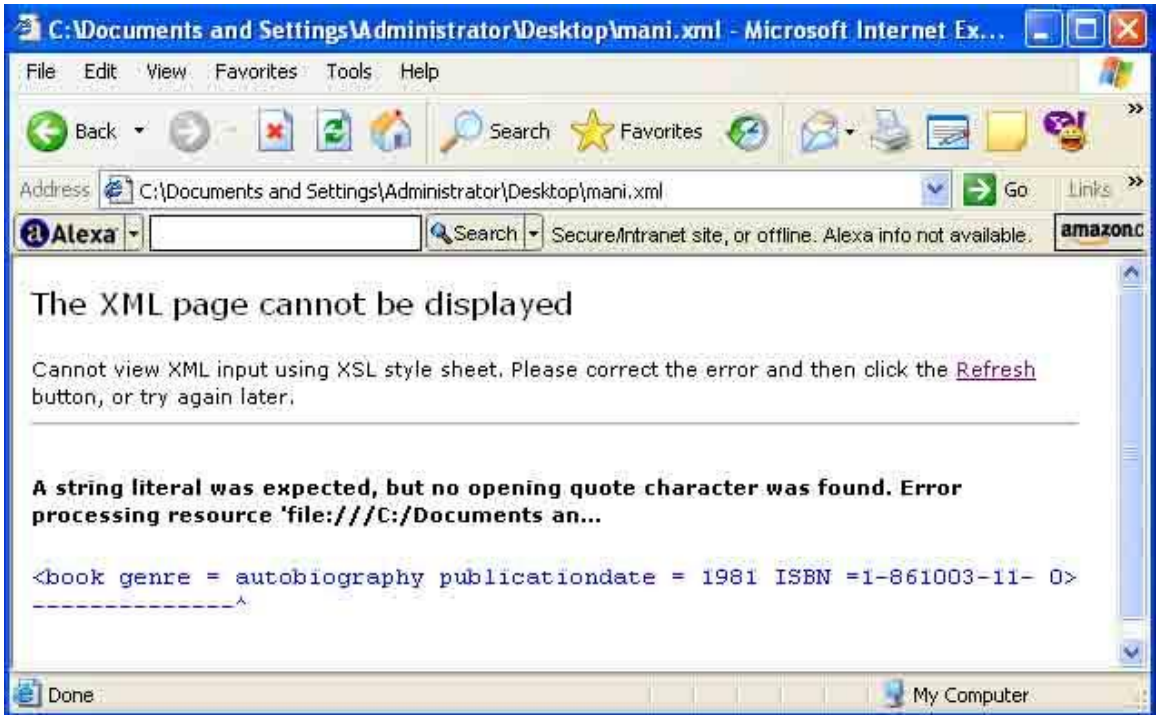

Figure 6-1. XML attribute definition error message

Another character you might notice in Listing 6-5 is the ! - -, which represents a comment in XML document. (I'll cover comments in a moment. See the "Comments" section.)

Unlike HTML, XML preserves spaces, which means you'll see the white space in your document displayed in the browser.

XML Parser

An XML parser is a program that sits between XML documents and the application using the document. The job of a parser is to make sure the document meets the define structures, validation, and constraints. You can define validation rules and constraints in a Document type Definition (DTD) or schema.

An XML parser comes with Internet Explorer (IE) 4 or later and can read XML data process it, generate a structured tree, and expose all data elements as DOM objects. The parser then makes the data available for further manipulation through scripting. After that, another application can handle this data.

MSXML parser comes with IE 5 or later and resides in the `MSXML.DLL` library. MSXML parser supports the W3C XML 1.0 and XML DOM recommendations, DTDs, schemas, and validations.

You can use MSXML programmatically from languages such as JavaScript, VBScript, Visual Basic, Perl, and C++.

Universal Resource Identifier (URI)

A Universal Resource Identifier (URI) is a resource name available on the Internet. A URI contains three parts: the naming schema (a protocol used to access the resource), the name of the machine (in the form of an Internet Protocol) upon which the resource reside, and the name of the resource (the file name). For Example, `http://www.csharpcorner.com/Images/cshea1.gif is a` URI name where `http://` is a protocol, `www.csharpcorner.com` is the address of the machine (which is actually a conceptual name for the address), and `Images/afile.gift` is the filename location on that machine.

XML Namespaces

Because users define an XML document's element names, it's possible that many developers will user the same names. XML **namespaces** allow developers to write a unique name and avoid conflicts between element names with other developers. With the help of URI, a namespace ensures the uniqueness of XML elements, tags, and attributes.

To declare namespaces, you can use default or explicit names. When you define your own namespace. The W3C recommends you control the URI and point to the same location consistently.

The scope of a document's elements depends on the URI. Listing 6-6 shows an example of XML document with namespace. In this example, <book> and its attributes and tags belong to the http://www.c-sharpcorner.com/Images URI.

Listing 6-6. XML namespace declaration example

```
<?xml version ='1.0'?>
<book xmlns = "http://www.c-sharpcorner.com/Images" >
<title> the autobiography of Benjamin Franklin</title>
<author>
<first-name>Benjamin</first-name>
<last-name>Franklin</last-name>
</author>
<price>8.99</price>
</book>
```

Document type Definition (DTD) and schemas

A Documents Type Definition (DTD) defines a document structure with a list of legal elements. You can declare DTDs inline or as a link to an external file. You can also use DTDs to validate XML documents. This is an example of a DTD:

```
<!ELEMENT Two (#PCDATA)>
<!ELEMENT one (B)>
<!ATTLIST one c CDATA # REQUIRED>
```

This DTD defines a format of a data. The following XML is valid because the tag`<Two>` is inside the tag `<one>`:

```
<One c="Attrib">
<Two> Something here</Two>
</One>
```

An XML **schema** describes the relationship between a document's elements and attributes. XML schemas describe the rules, which can be applied to any XML document, for elements and attributes. If an XML document references a schema and it doesn't meet the criteria XML parser will give an error during parsing.

You need a language to write schemas. These languages describe the syntaxes for each schema (XML document) you write. There are many schema languages, including DTD, XML Data Reduced (XDR), and simple object XML (SOX).

Similar to an XML document, an XML schema always starts with statement `<?xml version ="1.0" ?>`, which specifies the XML version.

The next statement of a schema contains an xsd:schema statement, xmlns, and target namespace. The xsd: schema indicates that file is a schema.

A schema starts with a `<xsd:schema>` tag and ends with a `</xsd:schema>tag`. All schema items have the prefix xsd. The `xmlns ="http://www.w3.org/2001/XMLschema"` is a http://www.W3c.org URI, which indicates the schema should be interpreted according to the default, namespace of the W3C. The next piece of this line is the target namespace, which indicates the location of a machine (a URI). Listing 6-7.is a schema representation for the document in Listing 6-5.

Listing 6-7. XML schema example

```
<xsd:schema xmlns:xsd ="http://www.w3.org/2001/XML Schema">

<xsd:element name = "bookstore" type = "bookstoreType"/>

<xsd: ComplexType name ="bookstoreType">
<xsd: squence maxOccurs = "unbounded">
<sdx: element name = "book" type = "bookType"/>
</xsd: sequence>
</xsd: complexType>

<xsd: ComplexType name = "bookType">
<xsd: sequence>
<xsd: element name = "title" type = "xsd:string:"/>
<xsd: element name = "author" type = "authorName"/>
<xsd: element name = "price"  type = "xsd:decimal"/>
</xsd: sequence>
</xsd: attribute name = "genre" type = "xsd:string"/>
</xsd: complexType>

<xsd: complexType name = "authorName">
<xsd: sequence>
<xsd: element name ="first-name" type ="xsd:string"/>
<xsd: element name = "last–name" type= "xsd:string"/>
</xsd: sequence>
</xsd: complexType>
</xsd:schema>
```

In this listing, <ComplexType>, <sequence> and<element> are schema elements. An element is a simple item with a single element. The ComplexType element is a set of attributes that denotes that element has children. Some other schema items are <all>, <annotation>, <any>, <anyAttribute>, <attribute>, <choice>, <documentation>, <field>, <group>, <include>, <key>, <length>, <maxLength>, <minLegth>, <selection>, <pattern>, <simpleType>, <unique>, and so on.

Elements and attributes are basic building block of a schema. An *element* is a tag with data. An element can have nested elements and attributes. Elements with one or more elements or attributes are ComplexType elements. An element contains a name and a data type. For example, the element price is of type decimal in the following line:

```
<xsd:element name ="price" type = "xsd:decimal"/>
```

This definition of the element price makes sure that it can only store a decimal type of a value. Other types of values are invalid values. For example, this is valid:

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

But this example is invalid:

```
<price>something</price>
```

Schema attributes are similar to XML attributes, but you can also define them using an xsd:attribute item. For example:

```
<xsd: ComplexType name= "bookstoreType">
```

or

```
<xsd: attribute name = " bookstoreType" type ="xsd:string"/>
```

A full discussion of these items is beyond the scope of this article: however, I'll describe any items I use in any of the samples.

Extensible Hypertext markup language (XHTML)

Extensible Hypertext Markup Language (XHTML) is a next-generation language of HTML. In January 2000, XHTML 1.0 became a W3C recommendation. XHTML is a better and improved version of HTML; however, it does have some restrictions.

XHTML is a combination of XML and HTML. XHTML uses elements of HTML 4.01 and rules of XML to provide a more consistent, well-formed and organized language.

An XML Document and its Items

An XML document is a set of elements in a well-formed and valid standard format. A document is valid if it has DTD associated with it and if it complies with the DTD. As mentioned earlier, a document is well formed if it contains one or more elements and if it follows the exact syntaxes of the language. An XML parser will only parse a document that is a well formed, but the document doesn't necessarily have to be valid. This means that a document must have at least one element (a root element) in it, but it doesn't matter whether it uses DTDs.

An XML document has the following parts, each described in the sections that follow:

- Prolog
- DOCTYPE declaration
- Start and end tags
- Comments
- Character and entity references
- Empty elements
- Processing instructions
- CDATA section
- Attributes
- White spaces

Prolog

The prolog part of a document appears before the root tag. The prolog information applies to the entire document. It can have character encoding, stylesheets, comments, and processing instructions. This is an example of a prolog:

```
<?xml version ="1.0"  ?>
<?xml-stylesheet type="text/xsl"  href ="books.xsl" ?>
<!DOCTYPE StudentRecord SYSTEM "mydtd.dtd">
<!=my comments - - - ->
```

DOCTYPE Declaration

With the help of a DOCTYPE declaration, you can read the structure of your root element and DTD from external files. A DOCTYPE declaration can contain a root element or a DTD (used for document validation). In a validating environment, a DOCTYPE declaration is must. In a DOCTYPE reference, you can even use a URI reference. For example:

```
<!DOCTYPE rootElement>

or

<!DOCTYPE rootElement SYSTEM "URIreference">

or

<!DOCTYPE StudentRecord SYSTEM "mydtd.dtd">
```

Start and End tags

Start and end tags are the heart of XML language. As mentioned earlier in the article, XML is nothing but a text file start and end tags. Each tag starts with `<TAG>` and ends with `</TAG>`. If you want to add a tag called `<book>` to your XML file, it must start with <book> and end the `</book>`, as shown in this example:

```
<?xml version ="1.0"  ?>
<book xmlns = "http://www.c-sharpcorner.com/xmlNet">
<title> The Autobiography of Benjamin Franklin</title>
<author>
<first-name>Benjamin</ First-name>
<last-name>Franklin</ last- name>
</author>
<price>8.99</ price>
</book>
```

NOTE: Empty elements don't have to heed this < >….</ > criteria. I'll discuss empty tags later in the "Empty Elements" section.

NOTE: An element is another name a starting and ending tag pair

Comments

Using `comments` in your code is good programming practice. They help you understand your code, as well as help others to understand your code, by explaining certain code lines. You use the <! - - and - - > pair to write comments in an XML document:

```
<!-- My comments here -->
<!-- This is a comment -->
```

XML parsers ignore comments.

CDATA Sections

What if you want to use < and > characters in your XML file but not as part of a tag? Well, you can't use them because the XML parser will interpret them as start and end tags. CDATE provides the following solution. So you can use XML markup characters in your documents and have the XML parser ignore them. If you use the following line:

```
<! [CDATA [I want to use < and >, characters]]>
```

the parser will treat those characters as data.
Another good example of CDATA is the following example:

```
<! [CDATA [< Title>This is the title of a page</ Title>
```

In this case, the parser will treat the second title as data as data, not as a mark up tag.

Character and entity reference

In some cases, you can't use a character directly in a document because of some limitations, such as character being treated as markup character or a device or processor limitation.

By using character and entity references, you can include information in a document by reference rather than the character.

A character reference is a hexadecimal code for a character. You use the hash symbol (#) before the hexadecimal value. The XML parser takes care of the rest. For example, the character reference for the Return Key is# `x000d`.

The reference start with an ampersand (&) and a #, and it ends with a semicolon (;). The syntax for decimal and hexadecimal references is & # value; and &#xvalue; respectively. XML has some built-in entities. Use the lt, gt, and amp entities for less than, greater than, and ampersand, respectively. Table 6-2 shows five XML built-in entities and their references. For example, if you want to write a > b or Jack & Jill, you can do that by using these entities:

```
A&gt;b and Jack&amp; Jill
```

Table 6-2. XML Build- in Entities

ENTITY	REFERENCE	DESCRIPTION
Lt	&lt	Less than: <
Gt	&gt	Greater than: >
Amp	&amp	Ampersand: &
Apos	&apos	Single quote: ‘
Auot	&quot	Double quote: “

Empty elements

Empty elements start and end with the same tag. They start with < and end with >. The text between these two symbols is the text data. For example:

```
<Name> </Name>
<IMG SRC= "img.jpg" />
<tagname/>
```

are all empty element example. The <IMG> specifies an inline image, and the SRC attribute specifies the image’s location. The image can be any format, though browsers generally support only GIF, JPEG, and PNG images.

Processing Instructions

Processing instructions (PIs) play a vital role in XML parsing. A PI holds the parsing instructions, which are read by the parser and other programs. If you noticed the first line of any of the XML samples discussed earlier, a PI starts like this:

```
<?xml version ="1.0" ?>
All PIs start with <? And end with ?>. This is another example of PI:

<?xml-stylesheet type ="text/ xsl" href = "myxsl.xsl"?>
```

This PI tells a parser to apply a stylesheet on the document.

Attributes

Attributes let you add extra information to an element without creating another element. An attribute is a name and value pair. Both the name and value must be present in an attribute. The attribute value must be in double quotes; otherwise the parser will give an error. Listing 6-8 is an example of an attribute in a <table> tag. In the example, the <table> tag has border and width attributes, and the <td> tag a width attribute.

Listing 6-8. Attributes in the < table> tag

```
<table border ="1" width = "43%">
```

```
<tr>
<td width ="50%">Row1, Column1</td>
<td width ="50%">Row1, Column2</td>
</tr>
<tr>
<td width = "50%">Row2, Column1</td>
<td width = "50%">Row2, Column2</td>
</tr>
</table>
```

White spaces

XML preserves white spaces except in attribute values. That means white space in your document will be displayed in the browser. However, white spaces are not allowed before the XML declaration. The XML parser reports all white spaces available in the document. If white spaces appear before declaration, the parser treats them as PI.

In element, XML 1.0 standard defines the `xml: space` attribute to insert spaces in a document. The XML:space attribute accepts only two values: `default and preserve`. The `default` value is the same as not specifying an xml:space attribute. It allows the parser to treat spaces as in a normal document. The `Preserve` value tells the parser to preserve space in the document. The parser preserves space in attributes, but it converts line break into single spaces.

DOM overview

Document object model (DOM) is a platform- and language- neutral interface that allows programs and scripts to dynamically access and update XML and HTML documents. The DOM API is a set of language- independent, implementation- neutral interfaces and objects based on the Object Management Group (OMG) Interface Definition Language (IDL) specification (not the COM) version of IDL). Set http://www.w3.org/TR/DOM-Level-2/ for more detail.

DOM defines the logical structure of a document's data. You can access the document in a structured format (generally through a tree format). Tree nodes and entities represent the document's data. DOM also helps developers build XML and HTML documents, as well as to add, modify, delete, and navigate the document's data. Figure 6-2 shows you various contents of DOM in a tree structure.

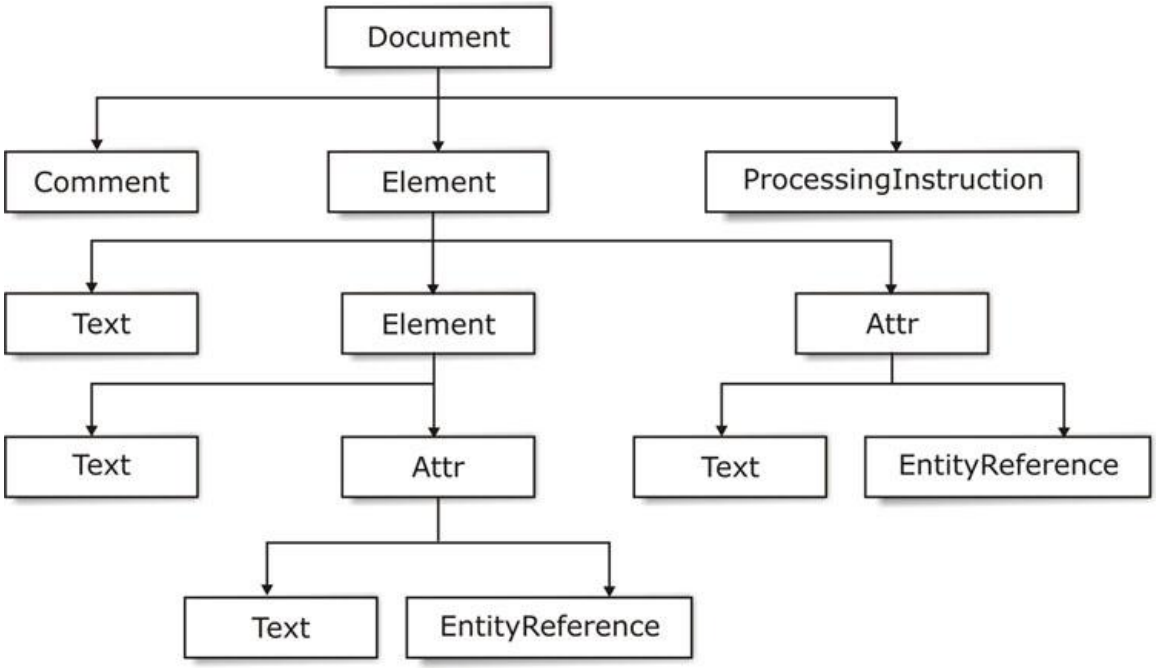

Figure 6-2. DOM tree structure

This is the tree structure implementation of an XML file.

```
<table>
<tr>
<td>Mahesh </td>
<td>Testing</td>
</tr>
<tr>
<td> Second Line</td>
<td> Tested</td>
</tr>
</table>
```

Figure 6-3 shows the DOM tree representation of this XML.

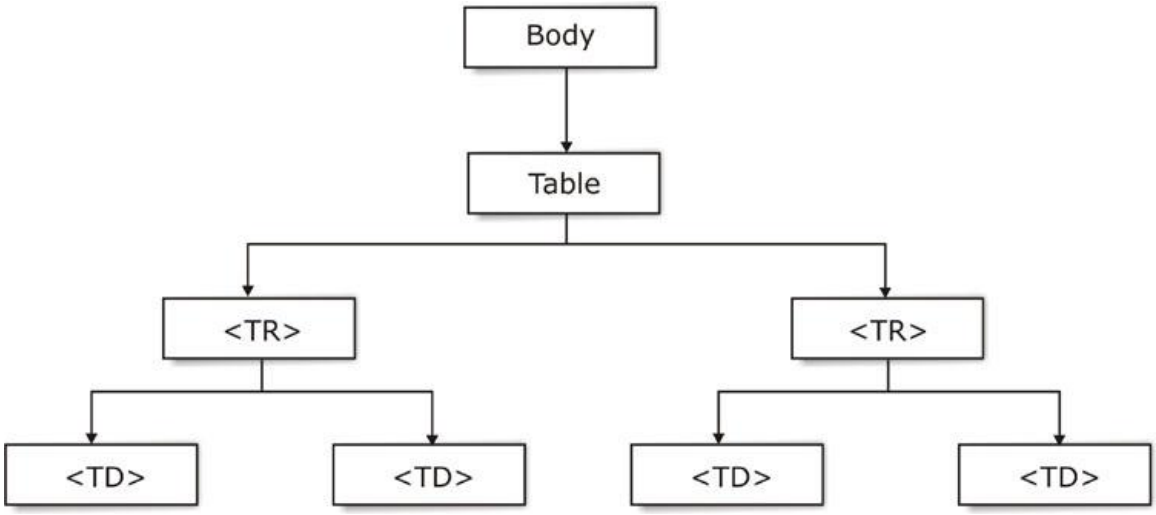

Figure 6-3. XML DOM tree representation

In DOM, a document takes a hierarchical structure, which is similar to a tree structure. The document has a root node, and the rest of the document has branches and leaves.

These nodes are defines as interfaces object. You use the interfaces to access and manipulate document objects. The DOM core API also allows you to create and populate documents load documents and save them.

Table 6-3 defines some XML document nodes and node contents.

Table 6-3. XML Nodes

NODE	DESCRIPTION	CHILDREN
Document	Represent an HTML or XML document and root of the document tree	Element, ProcessingInstruction, DocumentType, Comment
DocumentType	Represent the document type attribute of a document	No children
Element	An element of the document	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attr	An attribute	Text, EntityReference
ProcessingInstruction	Represent a processing instruction; used in XML	No children
Comment	Represent comments in an XML or HTML document; characters between the starting <! - - and ending - - >	No children
Text	Text of a node	No children
Entity	An entity type item	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference

XML Representation in .NET World

Microsoft's .NET Framework utilizes XML features to internally and externally transfer data between applications. In this section, you'll see XML namespaces and classes, which I'll be using in the examples through out this article. In the .NET Framework Library, the `System.Xml` and its four supportive namespaces define the functionality to work with XML data and documents. These namespaces are `System.Xml, system.Xml.Schema, System.Xml.Serialization`, `System.Xml.Xpath, and System.Xml.Xsl.` These namespaces reside in the `System.Xml.dll` assembly.

Before moving to the next topic, I'll describe these namespaces and their classes. I'll also discuss some of these classes in more detail through out this article.

The System.Xml Namespace

The `System.Xml namespace` defines common and major XML functionality. It defines classes for XML 1.0 XML namespaces and schemas. XPath, XSL Transformations (XSLT), DOM Level 2 core and SOAP 1.1.

The following sections define some of the `System.Xml` namespace classes.

The xml Node Class

The `XmlNode class`, an abstract base class for `XmlDocument` and `XmlDataDocument`, represents a single node in a document. This class implements methods for adding, removing, and inserting nodes into a document. This class also implements properties to get data from a node such as name, child nodes, siblings, parents, and so on.

Document classes

The `System.Xml` namespace also contains classes to a deal with XML documents. The `XmlDocument and XmlDocument` Fragment classes represent an entire XML document and a fragment of a document, respectively. The `XmlDocumentFragment` class is useful when you deal a small fragment of a document.

`The XmlDataDocument` class allows you to work with relational data using the `DataSet` object. It provides functionality to store, retrieve, and manipulate data. The `XmlDocumentType` class represents the type of document.

The `XmlDocument and XmlDataDocument` classes come form `XmlNode.` Besides the methods contained in `XmlNode`, this class implements a series of `Createxxx` methods to create a document's contents such as Comment, Element, Text and all the other contents discussed in the "DOM Overview" section of this article. You can even load an XML document by using its Load and `LoadXml` methods.

Each content type of an XML document has corresponding class defined in this namespace. The classes are `XmlAttribute, XmlCDataSection, XmlComment, XmlDeclaration, XmlEntity, XmlEntityReference, XmlProcessingInstruction, XmlText, and XmlWhitespace.` All of these classes are self–explanatory. For example, the `Attribute and XmlComment` classes represent an attribute and comment of a document. You'll see these classes in the examples.

Reader and Writer classes

Six classes (`XmlReader`, `XmlWriter`, `XmlTextWriter`, `XmlTextReader` `XmlValidatingReader`, and `XmlNodeReader`) represent the reading and writing XML documents.

`XmlReader and XmlWriter` are abstract base classes representing a reader that provides fast, non-cached, forward-only stream access to XML documents. `XmlReader` has three classes: `XmlTextReader, XmlValidatingReader, and XmlNodeReader`. As their node imply, `XmlTextReader` is for reading text XML documents, `XmlNodeReader` is for reading XML DOM trees, and `XmlvalidatingReader` can validate data using DTDs or schemas. This reader also expands general entities and supports default attributes. Xml writer is an abstract base class that defines functionality to write XML. It implements methods and properties to write XML contents. `XmlTextWriter` class comes from the `XmlWrinter` class.

Other classes

The `XmlConvert` class provides conversion in XML. It defines methods for converting Common Language Runtime (CLR), or .NET data types, and XML schema Definition (XSD) types.

- `XmlException` defines functionality to represent detailed exceptions
- `XmlNamespaceManager` resolves, Adds, and removes namespace to a collection and provides scope management for these namespaces.
- `XmlLinkedNode` returns the node immediately preceding or following this node.
- `XmlNodeList` represents a collection of nodes.

The System. Xml. Schema Namespace

The `System.Xml.Schema` namespace contains classes to work with XML schemas. These classes support XML schemas for structure and xml schemas for data types.

This namespace defines many classes to work with schemas. The discussion of these classes is beyond the scope of this book. Some of these namespace classes are `XmlSchema, XmlSchemaAll, XmlSchemaPath, and XmlSchemaType`.

The System.Xml.Serialization Namespace

This namespace contains classes to serialize objects into XML format documents or streams. Serialization is the process of reading and writing an object to or from a persistent storage medium such as a hard drive.

You can use the main class.`XmlSerializer, with TextWriter or XmlWriter to` write the data to document. Again this namespace also defines many classes. The discussion of these classes is beyond the scope of this article.

The System.Xml.XPath Namespace

This namespace is pretty small in comparison to the previous three namespaces. This namespace contains only four classes: `XpathDocument, XpathExression. XPathNavigator`, and `XPathNodeIterator.`

The `XPathDocument` class provides fast XML document processing using XSLT. This class is optimized for XSLT processing and the XPATH data model. The `CreateNavigator` method of this class creates an instance of `XpathNavigator`.

The XpathNavigator class reads data and treats a document as a tree and provides methods to traverse through a document as a tree. Its `Movexxx` methods let you traverse through a document.

Two other classes of this namespace are `XpathExpression and XpathIterator`. `XpathExpressio`n encapsulates an Xpath expression, and XpathIterator provides an Iterator over the set of selected nodes.

The System.Xml.Xsl Namespace

The last namespace, `System.Xml. Xsl`, defines functionality for XSL/T transformations. It suppports XSLT 1.0. The XsltTransform class defines functionality to transform data using an XSLT stylesheet.

DOM Interfaces

As you've seen in the previous discussion, you can represent an XML document in a tree structure using DOM interfaces and objects (shown in figure 6-3).

Microsoft .NET provides a nice wrapper around these interfaces: the DOM API. This wrapper has a class for almost every interface. These classes hide all the complexity of interface programming and provide a high-level programming model for developers. For example, the .NET class `XmlDocument` provides a wrapper for the Document interface.

Besides DOM, the Microsoft .NET XML API also provides corresponding classes for the XPath, XSD and XSLT industry standards. These classes are well coupled with the .NET database models (ADO.NET) to interact with databases.

The XML.NET Architecture

The XML.NET API is a nice wrapper around the XML DOM interfaces and provides a higher-level of programming over XML documents. The heart of the XML .NET architecture consists of three classes: `XmlDocument, XmlReader, and XmlWriter.`

The `XmlRader and XmlWriter` classes are abstrct base classes that provide fast, non-cached, forward- only cursors to read/ write XML data. `XmlTextReader, XmlValidatingReader`, and `XmlNodeReader` are concrete implementations of the `XmlReade`r class. The `XmlWriter and XmlNodeWriter` classes come from `the XmlWriter` class. `XmlDocument` represents an XML document in a tree structure with the help of the `XmlNode, XmlElement, and XmlAttribute` classes.

Figure 6-4 shows a relationship between these classes and the XML.NET architecture.

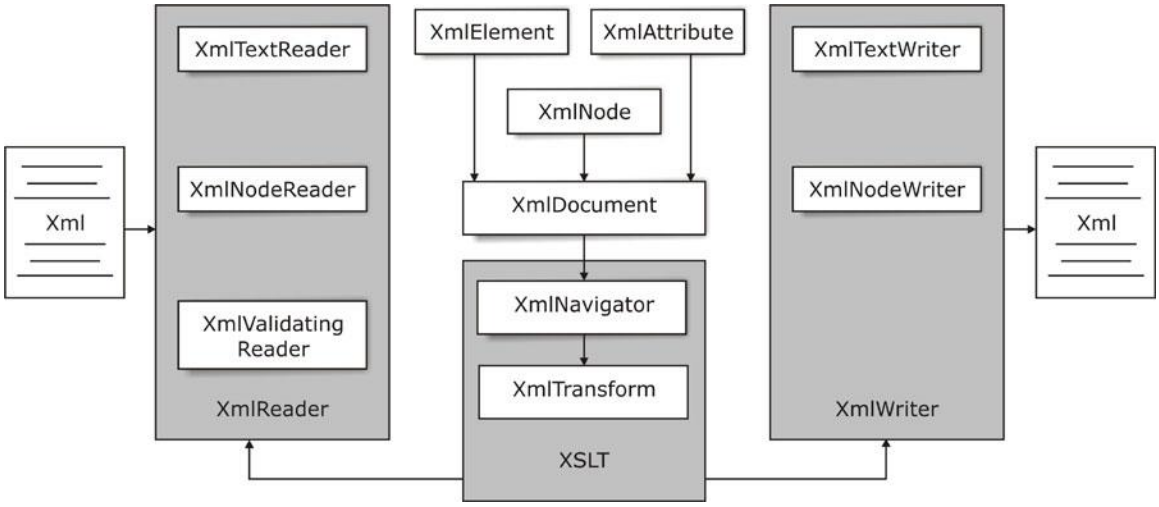

Figure 6-4. XML.NET architecture

The `System.Xml.Xsl` interface provides classes that implement XSLT. (I'll discuss XSLT in more detail later in this article.) The `XmlTransform` class implements XSLT. This class reads and writes XML data with the help of the `XmlReader and XmlWriter` classes.

The `XPathDocument and the XPathNavigator` classes provide read/ write and navigation access to the XML documents.

Associated with these classes are some more powerful classes for working with XML. I'll discuss these classes in "Navigation in XML" and other sections of this article.

Adding System.Xml Namespace Reference

You're probably aware of this, but before using `System.Xml` classes in your application, you may need to add a reference to the `System.Xml.dll` assembly using Project > Add Reference (see **figure 6-5**) and include the `System.Xml namespace:`

```
using System.Xml;
```

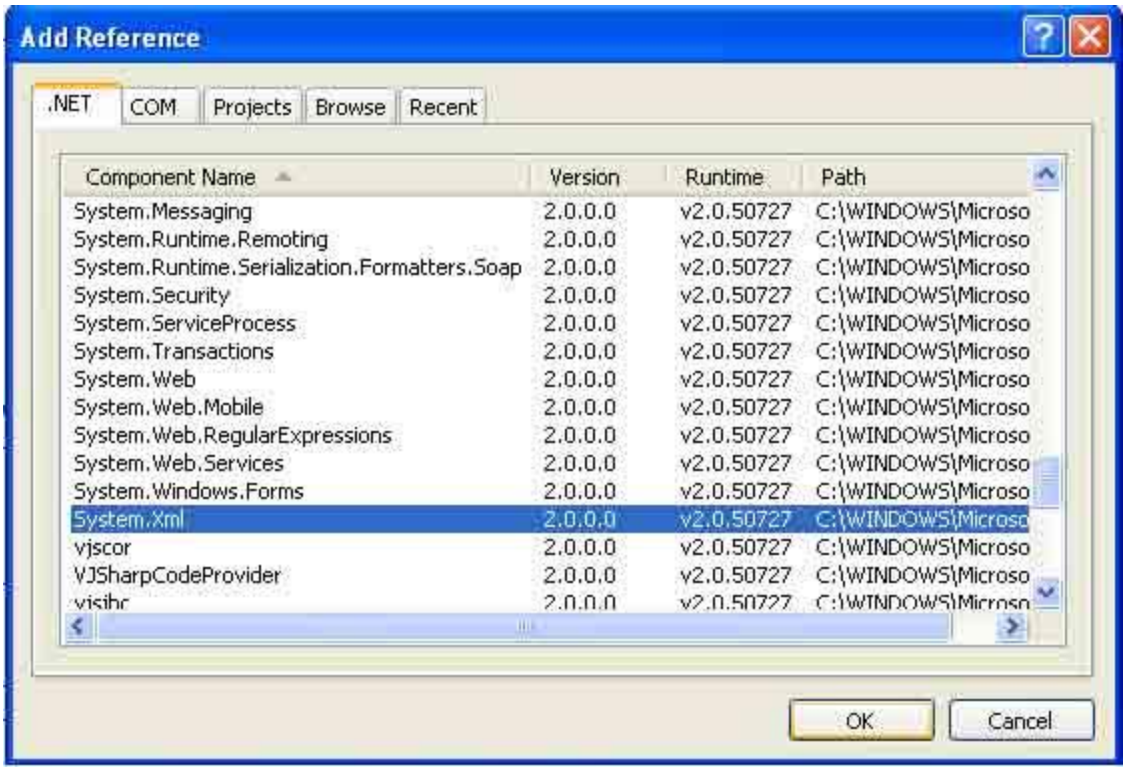

Figure 6-5. Adding a reference to the System.Xml.dlll assembly

The abstract base classes `XmlReader and XmlWriter` support reading and writing XML documents in the .NET Framework.

Reading XML

The `XmlReader` is an abstract base class for XML reader classes. This class provides fast, non-cached forward-only cursors to read XML documents.

The `XmlTextReader, XmlNodeReader, and XmlValidatingReader` classes are defined from the `XmlReader` class. **Figure 6-6** shows `XmlReader` and its derived classes.

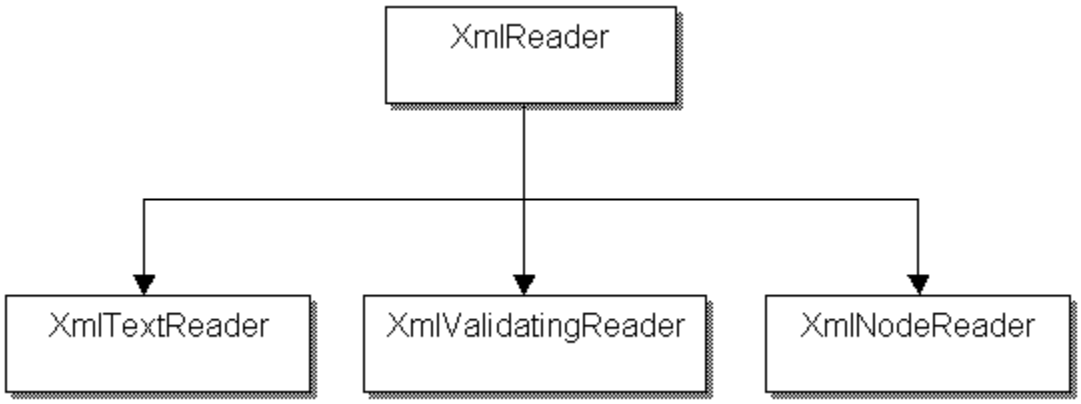

Figure 6-6. XmlReader classes

You use the `XmlTextReader, XmlNodeReader, and XmlValidatingReader` classes to read XML documents. These classes define overloaded constructors to read XML files, strings, streams, `TextReader objects, XmlNameTable`, and combinations of these. After creating an instance, you simply call the Read method of the class to read the document. The Read method starts reading the document from the root node and continues until Read returns false, which indicates there is no node left to read in the document. Listing 6-9 reads an XML file and displays some information about the file. In this example I'll use the books.xml file. You can use any XML by replacing the string name.

Listing 6-9. Reading an XML file

```
XmlTextReader reader = new XmlTextReader(@"C:/books.Xml");
Console.WriteLine ("General Information");
Console.WriteLine ("= = = = = = = = = ");
Console.WriteLine(reader.Name);
Console.WriteLine(reader.BaseURI);
Console.WriteLine(reader.LocalName);
```

Getting Node Information

The Name Property returns the name of the node with the namespace prefix, and the `LocalName` property returns the name of the node without the prefix.

The Item is the indexer. The Value property returns the value of a current node. you can even get the level of the node by using the Depth property, as shown in this example:

```
XmlTextReader reader = new XmlTextReader(@"C:/books.Xml");
while (reader.Read())
{
if (reader.HasValue)
```

```
{
Console.WriteLine("Name : "+ reader. Name);
Console.WriteLine("Node Depth: " + reader.Depth.ToString( ));
Console.WriteLine("Value : " + reader.Value);
}
}
```

The Node Type property returns the type of the current node in the form of `XmlNodeType` enumeration:

```
XmlNodeType type = reader.NodeType;
```

Which defines the type of a node. The `XmlNodeType` enumeration members are Attribute, CDATA, Comment, Document, Element, WhiteSpace, and so on. These represent XML document node types.

In Listing 6-10, you read a document's nodes one by one and count them. Once reading and counting are done, you see how many comments, processing instructions, CDATAs, elements, whitespaces, and so on that a document has and display them on the console. The `XmlReader`.NodeType property returns the type of node in the form of `XmlNodeType` enumeration. The `XmlNodeType` enumeration contains a member corresponding to each node types. You can compare the return value with `XmlNode` Type memebers to find out the type of a node.

Listing 6-10. Getting node information

```
static void Main(string[] args)
 {
int DecCounter = 0, PICounter = 0, DocCounter = 0, CommentCounter = 0;
int ElementCounter = 0, AttributeCounter = 0, TextCounter = 0,
WhitespaceCounter = 0;
XmlTextReader reader = new XmlTextReader(@"C:/books.Xml");
while (reader.Read())
{
    XmlNodeType nodetype = reader.NodeType;
    switch (nodetype)
    {
        case XmlNodeType.XmlDeclaration:
            DecCounter++;
            break;
        case XmlNodeType.ProcessingInstruction:
            PICounter++;
            break;
        case XmlNodeType.DocumentType:
            DocCounter++;
            break;
        case XmlNodeType.Comment:
            CommentCounter++;
            break;
        case XmlNodeType.Element:
            ElementCounter++;
            if (reader.HasAttributes)
                AttributeCounter += reader.AttributeCount;
            break;
        case XmlNodeType.Text:
            TextCounter++;
            break;
```

```
        case XmlNodeType.Whitespace:
            WhitespaceCounter++;
            break;
    }
}
// print the info
Console.WriteLine("White Spaces:" + WhitespaceCounter.ToString());
Console.WriteLine("Process Instruction:" + PICounter.ToString());
Console.WriteLine("Declaration:" + DecCounter.ToString());
Console.WriteLine("White Spaces:" + DocCounter.ToString());
Console.WriteLine("Comments:" + CommentCounter.ToString());
Console.WriteLine("Attributes:" + AttributeCounter.ToString());
}
```

The case statement can have values `XmlNodeType.XmlDeclaration`, `XmlNodeType.ProcessingInstruction`, `XmlNodeType.DocumentType`, `XmlNodeType.Comment`, `XmlNodeType.Element`, `XmlNodeType.Text`, `XmlNodeType.Whitespace`, and so on.

The `XmlNodeType` enumeration specifies the type of node. Table 6-4 describes its members.

Table 6-4. the xml Node Type Enumeration's members

MEMBER NAME	DESCRIPTION
Attribute	Attribute node
CDATA	CDATA section
Comment	Comment node
Document	Document object
DocumentFragment	Document Fragment
DocumentType	The DTD, indicated by the <! DOCTYPE> tag
Element	Element node
EndElement	End of element
EndEntity	End of an entity
Entity	Entity declaration
EntityReference	Reference to an entity
None	Returned if `XmlReader` is not called yet
Notation	Returned if `XmlReader` is not called yet
ProcessingInstruction	Represents a processing instruction (PI) node
SignificationWhitespace	Represents white space between markup in a mixed content model
Text	Represent the text content of an element
Whitespace	Represents white space between markup
XmlDeclaration	Represents an XML declaration node

Moving to a Content

You can use the `MoveToMethod` to move from the current to the next content node of an XML document. A content's node is an item of the following type: text CDATA, Element, EntityReference, or Entity. So if you call the `MoveToContent` method, it skips other types of nodes besides the content type nodes. For example if the next node of the current node is DxlDeclaration, or DocumentType, it will skip these nodes until it finds a content type node. See the following example:

```
XmlTextReader reader = new XmlTextReader(@"c:\books.xml");
```

```
if (reader.Read())
{
Console.WriteLine(reader.Name);
reader.MoveToContent();
Console.WriteLine(reader.Name);
}
```

The Get Attributes of a Node

The `GetAttribute` method is an overloaded method. You can use this method to return attributes with the specified name, index, local name, or namespace URI. You use the `HasAttributes` property to check if a node has attributes, and `AttributesCount` returns the number of attributes on the node. The local name is the name of the current node without prefixes. For example, if `<bk:book>` represents a name of a node, where bk is a namespace and: is used to refer to the namespace, the local name for the `<bk:book>` element is book. `MoveToFirstAttributes` moves to the first attribute. The `MoveToElement` method moves to the element that contains the current attributes node (see listing 6-11).

Listing 6-11. Get Attributes of a node

```
using System;
using System.Xml;

class XmlReaderSamp
{
static void Main(string[] args)
{
XmlTextReader reader = new XmlTextReader(@"C:\books. Xml");
reader.MoveToContent();
reader.MoveToFirstAttribute();
Console.WriteLine("First Attribute value" + reader.Value);
Console.WriteLine("First Attribute Name" +reader.Name);
while (reader.Read())
{
    if (reader.HasAttributes)
    {
        Console.WriteLine(reader.Name + "Attribute");
        for (int i = 0; i < reader.AttributeCount; i++)
        {
            reader.MoveToAttribute(i);
            Console.WriteLine("Nam: " + reader.Name + ", value: " +
reader.Value);
        }
        reader.MoveToElement();
    }
}
}
}
```

You can move to attributes by using `MoveToAttribute`, `MoveToFirstAttribute`, and `MoveToNextAttribute`. `MoveToFirstAttribute` and `MoveToNextAttribute` move to the first and next attributes, respectively. After calling `MoveToAttribute`, `the Name`, `Namespace`, and `Prefix` property will reflect the properties of the specified attribute.

Searching for a Node

The `Skip` method skips the current node. It's useful when you're looking for a particular node and want to skip other nodes. In listing 6-12, you read your books.xml document and compare its `XmlReader.Name(through XmlTextReader)` to look for a node with name bookstore and display the name, level, and value of that node using `XmlReader's Name, Depth, and Value` properties.

Listing 6-12. Skip Method

```
XmlTextReader reader = new XmlTextReader(@"c:\books.xml");
while (reader.Read())
{
// Look for a Node with name bookstore
if (reader.Name != "bookstore")
reader.Skip();
else
{
Console.WriteLine("Name: " + reader.Name);
Console.WriteLine("Level of the node:" + reader.Depth.ToString());
Console.WriteLine("Value: " + reader.Value);
}
}
```

Closing the Document

Finally, use `Close` to close the opened XML document.

Table 6-5 and 6-6 list the `XmlReader` class properties and methods. I've discussed some of them already.

Table 6-5 xml Reader properties

PUBLIC INSTANCE PROPERTY	DESCRIPTION
`AttributeCount`	Returns the number of attributes on the current node
`BaseURI`	Returns the base URI of the current node
`Depth`	Returns the level of the current node
`EOF`	Indicates whether its pointer is at the end of the stream
`HasAttributes`	Indicates if a node has attributes or not
`HasValue`	Indicates if a node has a value or not
`IsDefault`	Indicates whether the current node is an attributes generated from the default value defined in the DTD or schema
`IsEmptyTag`	Returns if the current node is empty or not
`Item`	Returns if value of the attribute
`LocalName`	Name of the current node without the namespace prefix
`Name`	Name of the current node with the namespaces prefix
`NamespaceURI`	Namespace uniform Resource Name (URN) of the current namespace scope
`NameTable`	Returns the `XmlNameTable` associated with this implementation
`NodeType`	Returns the type of node
`Prefix`	Returns the namespace associated with a node
`ReadState`	Read state
`Value`	Returns the value of a node
`XmlLang`	Returns the current `xml:lang` scope

`XmlSpace`	Returns the current `xml:space` scope

Table 6-6. xml Reader Methods

PUBLIC INSTANCE METHOD	DESCRIPTION
`Close`	Close the stream and changes `ReadState` to Closed
`GetAttribute`	Returns the value of an attribute
`IsStartElement`	Checks if a node has start tag
`LookupNamespace`	Resolves a namespace prefix in the current element's scope
`MoveToAttribute, MoveToContent, MoveToElement,`	Moves to specified attributes, content, and element
`MoveToFirstAttribute, MoveToNextAttribute`	Moves to the first and next attributes
`Read`	Reads a node
`ReadAttributeValue`	Parses the attributes value into one or more Text and/or EntityReference node types
`ReadXXXX (ReadChar, ReadBoolean, ReadDate, ReadIn32, and so on)`	Reads the contents of an element into the specified type including char, double, string, date, and so on
`ReadInnerXml`	Reads all the content as a string
`Skip`	Skips the current element

Writing XML

The `XmlWriter` class contains methods and properties to write to XML documents, and `XmlTextWriter and XmlNodeWriter come` from the `XmlWriter` class (see figure 6-7).

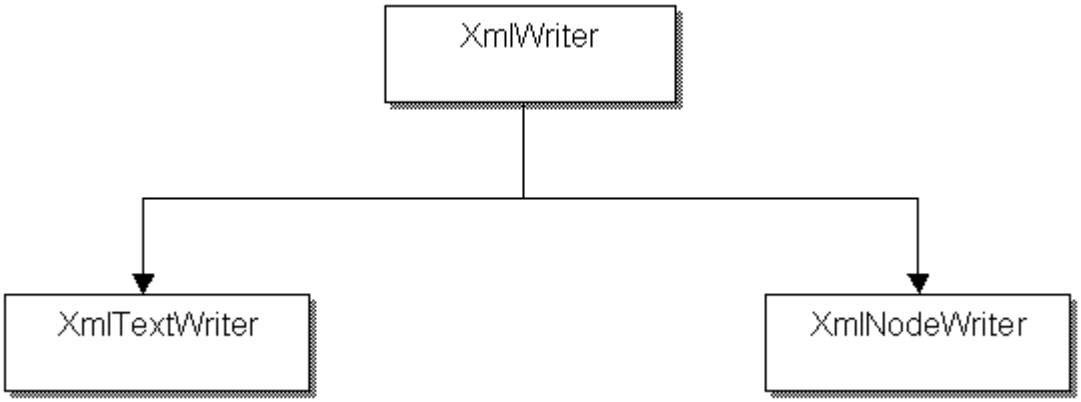

Figure 6-7. XmWriter classes

Besides providing a constructor and three properties (`WriteState, XmlLang, and XmlSpace`), the `XmlWriter` classes have many writexxx methods to write to XML documents. This section discusses some of these class methods and properties and uses them in examples of the `XmlTextWriter and XmlNodeWriter classes`. `XmlTextWriter` creates a write object and writes to the document. The `XmlTextWriter` constructor can take three types of inputs: a string, a stream, or a TextWriter.

XmlWriter properties

The `XmlWriter`r class contains three properties: `WriterState, XmlLang, and XmlSpace`. The `WriteState` property gets the current state of the `XmlWriter` class. The values could be Attributes, Start, Element, Content, closed, or Prolog. The return value `WriteState`.Start means the Write method is not yet called. In otnher cases, it represents what is being written. For example, the return value `WriteState`.Attribute means the Attribute value has written. `WriteState`.Close represents that the stream has closed by calling Close method.

Writing XML Items

As discussed earlier, an XML document can have any types of items including elements, Comments, attributes, and white spaces. Although it's not possible to describe all the W`ritexxx` methods here. I'll cover some of them.

The `WriteState`Document and `WriteEndDocument` methods open and close a document for writing, respectively. You must open a document before you start writing to it. The `WriteComment` method writes comment to a document. It takes only one string type of argument. The WriteString method writes a string to a document. With the help of `WriteString`, you can use the `WriteStartElement and WriteEndElement` method pair to write an element to a document. The `WriteStartAttribute` and `WriteEndAttribute` pair writes an attribute. WriteNode is another write method, which writes XmlReader to a document as a node of the document. The following example summarizes all these methods and creates a new XML document with some items in it such as elements, attributes, strings, comments, and so on. (See listing 6-13 in the next section.)

In this example, you create a new XML file, c:\xmlWriterText.xml, using XmlTextWriter:

```
// Create a new file c:\ xmlWriterTest.Xml
XmlTextWriter writer = new XmlTextWriter("C:\\xmlWriterTest.xml",
null);
```

After that, add comments and elements to the document using Writexxx methods. After that you can read the `books.xml` xml file using `Xml TextReader` and add its elements to `xmlWriterTest.xml` using XmlTextWriter:

```
// Create an XmlText Reader to read books. xml
XmlTextReader reader = new XmlTextReader ("@c:\books.xml");
while (reader.Read())
{
if (reader.NodeType == XmlNodeType.Element)
{
// Add node.xml to xmlWriterTest .xml using WriteNode
writer.WriteNode(reader, true);
}
}
```

Listing 6-13 shows an example of using `XmlWriter` to create a new document and write its items. This program creates a new XML document, xml writer Test, in the C:\root directory.

Listing 6-13 XmlWriter example

```
static void Main(string[]args)
{


// Create a new File c:\xmlWriterTest.xml
XmlTextWriter writer = new XmlTextWriter("C:\\ xmlWriterTest.xml",
null);

// opens the document
writer.WriteStartDocument();

// write comments
writer.WriteComment("This Program uses XmlTextWriter.");
writer.WriteComment("Developed by :Mahesh Chand.");
writer.WriteComment("= = = = = = = = = = = = = = =");

// write first element
writer.WriteStartElement("root");
writer.WriteStartElement("r", "RECORD", "urn: record");

// write next element
writer.WriteStartElement ("FirstName"," ");
writer.WriteString("Mahesh");
writer.WriteEndElement();

// write one more element
writer.WriteStartElement("LastName", " ");
writer.WriteString("Chand");
writer.WriteEndElement();
```

```
// Create an XmlTextReader to read books.xml
XmlTextReader reader = new XmlTextReader(@"c:\books. Xml");
while (reader.Read())
{
if (reader.NodeType == XmlNodeType.Element)
{
// Add node.xml to xmlWriterTest.xml using WriteNode
writer.WriteNode(reader, true);
}
}
// Ends the document.
writer.WriteEndDocument();
writer.Close();
return;
}
```

NOTE: In Listing 6-13 you write output of the program to a file. If you want to write your output directly on the console, pass `Console.Out` as the file name when create an `XmlTextWriter` object. For example: `XmlTextWriter writer = new XmlTextWriter (Console.Out);`

When you open `C: \ xmlWriterTest.Xml` in a browser, the output of the program looks like Listing 6-14.

Listing 6-14. Output of XmlWriterSample.cs class

```
<?xml version="1.0" ?>
- <!-- This program uses xmlTextWriter. -->
- <!-- Developed by: Mahesh chand. -->
- <!-- ======= ====== -->
- <root>
  - <r:RECORD xmlns:r="urn:record">
      <FirstName>Mahesh</FirstName>
      <LastName>Chand</LastName>
    - <bookstore>
      - <book genre="autobiography" publicationdate="1981"
          ISBN="1-861003-11-0">
          <title>the Autobiography of Benjamin
            Franklin</title>
        - <author>
            <First-name>Benjamin</First-name>
            <last-name>Franklin</last-name>
          </author>
          <price>8.99</price>
        </book>
      -<book genre="novel" publicationdate="1967" ISBN="0-201-
          63361-2">
          <title>The confidence man</title>
        - <author>
            <first-name>Herman</first-name>
            <last-name>Malville</last-name>
          </author>
          <price>11.99</price>
```

```
        </book>
      -<book genre="Philosophy" publicationdate="1991" ISBN="1-
          861001-56-6">
          <title>The Gorgias</title>
        - <author>
            <name>Plato</name>
          </author>
          <price>9.99</price>
        </book>
      </bookstore>
    </r:RECORD>
  </root>
```

The close method

You use the `Close` method when you're done with the `XmlWriter` object, which closes the stream.

The XmlConvert class

There are some characters that are not valid in XML documents. XML documents use XSD types, which are different than CRL (.NET) data types. The `XmlConvert` class contains methods to convert from CLR types to XSD types and vice versa. The `DecodeName` method transfers an XML name into an ADO.NET object such as `DataTable`. The `EncodeName` Method is the reverse of `DecodeName`: it converts an ADO.NET object to valid XSD name. It takes any invalid character and replaces it with an escape string. Another method, `EncodeLocalNAme`, converts unpermitted names to valid names.

Besides these three methods, the `XmlConvert` class has many methods to convert from a string object to Boolean. Byte, integer, and so on. Listing 6-15 shows the conversion from Boolean and Date Time object to XML values.

Listing 6-15 xml convert example

```
XmlTextWriter writer = new XmlTextWriter(@"c:\test. Xml", null);
writer.WriteStartElement("MyTestElements");
bool b1 = true;
writer.WriteElementString("TestBoolean", XmlConvert.ToString(b1));
DateTime dt = new DateTime(2000, 01, 01);
writer.WriteElementString("test date", XmlConvert.ToString(dt));
writer.WriteEndElement();
writer.Flush();
writer.Close();
```

Understanding DOM Implementation

Microsoft.NET supports the W3C DOM Level 1 and Core DOM Level 2 specifications. The .NET Framework provides DOM implementation through many classes. `XmlNode and XmlDocument` are two of them. By using these two classes, you can easily traverse though XML documents in the same manner you do in a tree.

The XmlNode class

The `XmlNode` class is an abstract base class. It represents a tree node in a document. This tree node can be the entire document. This class defines enough methods and properties to represent a document node as a tree node and traverse though it. It also provides methods to insert, replace, and remove document nodes.

The `ChildNodes` property returns all the children nodes of current node. You can treat an entire document as node and use `ChildNode`s to get all nodes in a document. You can use the `FirstChild, LastChild, and HasChildNodes` triplet to traverse from a document's first node to the last node. The `ParentNode, PreviousSibling, and NextSibling` properties return the parent and next sibling node of the current node. Other common properties are Attributes, Base URI, `InnerXml, Inner Text, Item Node Type, Name, Value,` and so on.

You can use the `CreateNavigator` method of this class to create an Xpath Navigator object, which provides fast navigation using xpath. The `Appendchilds, InsertAfter, and InsertBefore methods add nodes to the document. The Remove All, Remove Child, and` ReplaceChild methods remove or replace document nodes, respectively. You'll implement these methods and properties in the example after discussing a few more classes.

The xml Document Class

The `XmlDocument` class represents an XML document. Before it's derived from the `XmlNode` class, it supports all tree traversal, insert, remove, and replace functionality. In spite of `XmlNode` functionaality, this class contains many useful methods.

Loading a Document

DOM is a cache tree representation of an XML document. The `Loads and LoadXml` methods of this class load XML data and documents, and the `Save` method saves a document.

The `Load` Method can load a document from a string, stream, `TextReader`, or `XmlReader`. This code example loads the document `books.xml` from a string:

```
XmlDocument xmlDoc = new XmlDocument();
string filename = @"c:\ books. Xml";
xmlDoc.Load(filename);
xmlDoc.Save(Console.Out);
```

This example uses the Load method to load a document from an XmlReader:

```
XmlDocument xmlDoc = new XmlDocument();
XmlTextReader reader = new XmlTextReader("c:\\books.xml");
xmlDoc.Load(reader);
```

```
xmlDoc.Save(Console.Out);
```

The LoadXml method loads a documernt from the specified string. For example

```
xmlDoc.LoadXml("<Record> write something</ Record>");
```

Saving a Document

The Save methods saves a document to a specified location. The Save method takes a paramenter of XmlWriter, XmlTextWriter or string type:

```
string filename = @"C:\ books.xml";
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load(filename);

XmlTextWriter writer = new XmlTextWriter("c:\\ domtest.Xml", null);
writer.Formatting = Formatting.Indented;
xmlDoc.Save(writer);
```

You can also use a filename or Console.Out to save output as file or on the console:

```
xmlDoc.Save("c:\\ domtest. Xml");
xmlDoc.Save(Console.Out);
```

The XmlDocumentFragment class

Usually, you would use this class when you need to insert a small fragment of an XML document or node into a document. This class also comes from XmlNode. Because this class is derived from XmlNode, it has the same tree node traverse, insert, remove, and replace capabilities.

You usually create this class instance by calling Xml Document's CreateDocumentFragment method. The InnerXml represents the children of this node. Listing 6-16 shows an example of how to create XmlDocumentFragment and load a small piece of XML data by setting its InnerXml property.

Listing 6-16. XmlDocumentFragment sample

```
//open an XML file
string filename = @"c:\ books.xml";
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load(filename);
// Create a document fragment.
XmlDocumentFragment docFrag = xmlDoc.CreateDocumentFragment();
// Set the contents of the document Fragment.
docFrag.InnerXml = "<Record> write something</ Record>";
// Display the document fragment.
Console.WriteLine(docFrag.InnerXml);
```

You can use XmlNode methods to add, remove, and replace data. Listing 6-17 appends a node in the document fragment.

Listing 6-17. Appending in an XML document fragment

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

```
doc.LoadXml("<book genre = 'programming'> " +
"<title> ADO.NET programming </ title> " + "</book>");
// Get the root node
XmlNode root = doc.DocumentElement;
// Create a new node.
XmlElement newbook = doc.CreateElement("price");
newbook.InnerText = "44.95";
// Add the node to the document.
root.AppendChild(newbook);
doc.Save(Console.Out);
```

The Xml Element Class

An `XmlElement` class object represents an element in a document. This class comes from the `XmlLinkedNode` class, which comes from `XmlNode` (see figure 6-8).

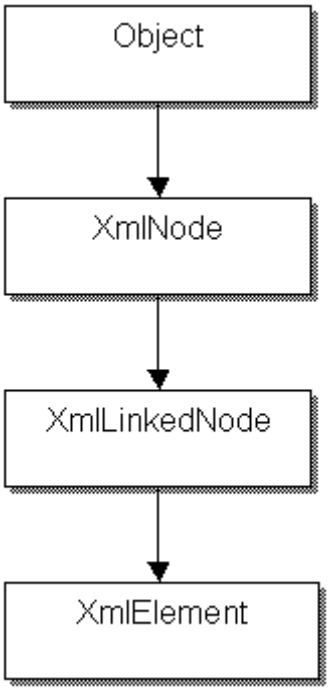

Figure 6-8. xml element inheritance

The `XmlLinkedNode` has two useful properties: `NextSibing` and `previousSibling`. As their names indicate, these properties return the next and previous nodes of an XML document's current node.

The `XmlElement` class implements and overrides some useful methods for adding and removing attributes and element (see table 6-7).

Table 6-7. Some xml element methods

METHOD	DESCRIPTION
`GetAttribute`	Returns the attribute value
`HasAttribute`	Checks if a node has the specified attribute
`RemoveAll`	Removes all the children and attributes of the current node
`RemoveAllAttributes,`	Removes all attributes and specified attributes from an element

RemoveAttribute	respectively
RemoveAttributeAt	Removes the attribute node with the specified index from the attribute collection
RemoveAttributeNode	Removes an XmlAttribute
SetAttribute	Sets the value of the specified attribute
SetAttribute Node	Adds a new xml Attribute

In the later examples. I'll show you how you can use these methods in your programs to get and set XML element attributes.

Adding Nodes to a Document

You can use the AppendChild method to add to an existing document. The AppendChild method takes a single parameter of XmlNode type. The XmlDocument's Createxxx methods can create different types of nodes. For example, the CreateComment and CreateElement methods create comment and element node types. Listing 6-18 shows an example of adding two nodes to a document.

Listing 6-18. Adding nodes to a document

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<Record> some value </Record>");
// Adding a new comment node to the document
XmlNode node1 = xmlDoc.CreateComment("DOM Testing sample");
xmlDoc.AppendChild(node1);
// Adding a First Name to the documentt
node1 = xmlDoc.CreateElement("First Name");
node1.InnerText = "Mahesh";
xmlDoc.DocumentElement.AppendChild(node1);
xmlDoc.Save(Console.Out);
```

Getting the Root Node

The DocumentElement method of the XmlDocument class (inherited from XmlNode) returns the root node of a document. The following example shows you how to get the root of a document (see listing 6-19).

Listing 6-19. Getting root node of a document

```
string filename = @"c: \books.xml";
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load(filename);
XmlElement root = xmlDoc.DocumentElement;
```

Removing and Replacing Nodes

The RemoveAll method of the XmlNode class can remove all elements and attributes of a node. The RemoveChild removes the specified child only. The following example calls RemoveAll to remove all elements had attributes. Listing 6-20 calls RemoveAll to remove all item of a node.

Listing 6-20. Removing all item of a node

```
public static void Main()
```

```
{
// Load a document fragment
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<book genre ='programming'>" +
"<title> ADO.NET programming </title> </book>");
XmlNode root = xmlDoc.DocumentElement;
Console.WriteLine("XML Document Fragment");
Console.WriteLine("= = = = = = = = = = ");
xmlDoc.Save(Console.Out);
Console.WriteLine();
Console.WriteLine("-----------");
Console.WriteLine("XML Document Fragment Remove All");
Console.WriteLine("= = = = = = = = = = =");
// Remove all attribute and child nodes.
root.RemoveAll();
// Display the contents on the console after
// Removing elements and attributes
xmlDoc.Save(Console.Out);
}
```

NOTE: You can apply the Remove All method on the books.xml files to delete all the data, but make sure to have backup copy first!

Listing 6-21 shows how to delete all the item of books. Xml

Listing 6-21.CcallingRemoveAll for books.Xml

```
public static void Main()
{
string filename = "c:\\ books.Xml";
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load(filename);
XmlNode root = xmlDoc.DocumentElement;
Console.WriteLine("XML Document Fragment");
Console.WriteLine("= = = = = = = = = = ");
xmlDoc.Save(Console.Out);
Console.WriteLine();
Console.WriteLine("- - - - - - - - - ");
Console.WriteLine("XML Document Fragment After RemoveAll");
Console.WriteLine("= = = = = = = = = = = ");
//Remove all attribute and child nodes.
root.RemoveAll();
// Display the contents on the console after
// Removing elements and attributes
xmlDoc.Save(Console.Out);
}
```

The `ReplaceChild` method replaces an old child with a new child node. In Listing 6-22, `ReplaceChild` replaces root Node; `Last Child with xmlDocFrag`.

Listing 6-22 Replace Child method sample

```
string filename = @"C:\books.xml";
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load(filename);
XmlElement root = xmlDoc.DocumentElement;
```

```
XmlDocumentFragment xmlDocFragment = xmlDoc.CreateDocumentFragment();
xmlDocFragment.InnerXml =
"<Fragment><SomeData>Fragment Data</SomeData></ Fragment>";
XmlElement rootNode = xmlDoc.DocumentElement;
//Replace xmlDocFragment with rootNode.LastChild
rootNode.ReplaceChild(xmlDocFragment, rootNode. LastChild);
xmlDoc.Save(Console.Out);
```

Inserting XML Fragments into an XML Document

As discussed previously, the `XmlNode` class is useful for navigating through the nodes of a document. It also provides other methods to insert XML fragments into a document. For instance, the `InsertAfter` method inserts a document or element after the current node. This method takes two arguments. The first argument is an `XmlDocumentFragment` object, and the second argument is the position of where you want to insert the fragment. As discussed earlier in this article, you create an `XmlDocumentFragment` class object by using the `CreateDocumentFragment` method of the `XmlDocument` class. Listing 6-23 inserts an XML fragment into a document after the current node using `InsertAfter`.

Listing 6-23. Inserting an XML fragment into a document

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load(@"C:\\ books.Xml");
XmlDocumentFragment xmlDocFragment = xmlDoc.CreateDocumentFragment();
xmlDocFragment.InnerXml =
"< Fragment >< Some Data> Fragment Data</ Some Data> </ Fragment>";
XmlNode aNode = xmlDoc.DocumentElement.FirstChild;
aNode.InsertAfter(xmlDocFragment, aNode.LastChild);
xmlDoc.Save(Console.Out);
```

Adding Attributes to a Node

You use the `SetAttributeNode` method of `xmlElement` to add attributes to an element, which is a Node. The `XmlAttribute` represents an XML attribute. You create an instance of `XmlAttribute` by calling `CreateAttribute of XmlDocument`. After that you call an `xml Element's Set Attribute` method to set the attribute of an element. Finally, you append this new item to the document (see listing 6-24).

Listing 6-24. Adding a node with attributes

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load (@"c:\\books.Xml");
XmlElement newElem = xmlDoc.CreateElement("NewElement");
XmlAttribute newAttr = xmlDoc.CreateAttribute("NewAttribute");
newElem.SetAttributeNode(newAttr);
// add the new element to the document
XmlElement root = xmlDoc.DocumentElement;
root.AppendChild(newElem);
xmlDoc.Save(Console.Out);
```

Transformation and XSLT

Extensible Stylesheet Language (XSL) is a language for expressing `stylesheets`. Stylesheets format XML documents in a way so that the XML data can be presented in a certain structure in a browser or other media such as catalogs books and so on.

The XML stylesheet processor reads an XML document (Called an XML source tree) and stylesheet, and it presents the document data in an XML tree format. This processing is XSL Transformation (XSLT). See figure 6-9.

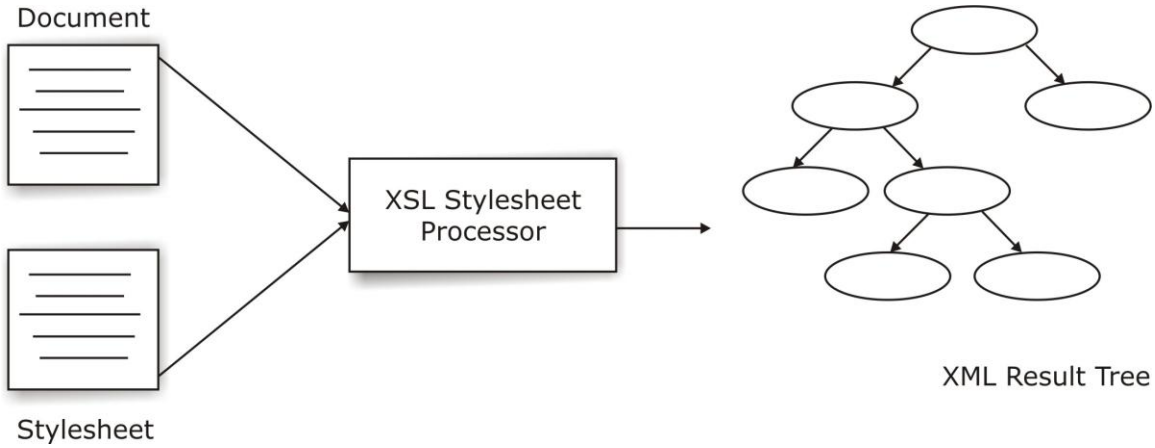

Figure 6-9. XSL transformation

The result tree generated after XML transformation contains element and attribute nodes. The result tree is also called an element –attribute or tree. In this tree, an object is an XML element, and properties are attribute- value pairs.

The XSL stylesheet plays a vital role in the XSLT process. A stylesheet contains a set of tree construction rules, which have two parts. The first part is a pattern of elements in the source tree, and the second is a template for the result tree. The XSL parser reads the pattern and elements from the source tree and then generates results according to the result tree template.

XSLT in .NET

In the .NET Framework, the `XslTransform` class implements the XSLT specification. This class you defined in a separate namespace called `System.Xml.Xsl`. Make sure you add a reference to this namespace before using the `XslTransform` class. You can use the XsltException class to handle exceptions thrown by an XSLT transformation.

The Transform Method

The `Transform Method` of `XslTransaforms` data using loaded stylesheet and outputs the result depending on the argument. This method has eight overloaded forms. You can write output of `Transform` in the form of `XmlWriter, stream, TextWriter, or XPathNavigator`. (I'll discuss `XPathNavigator` later in this article.)

Transforming a Document

Follow these steps to perform the transformation:

1. First you need to create an `xslTransform` object:

```
XslTransform xslt = new XslTransform();
```

2. Now, you load the stylesheet using the Load method:

```
xslt.Load("stylesheetFrmt. xsl");
```

3. Finally, call the Transform method of XslTransform:

```
xslt.Transform("xmlfile.xml", "file.html");
```

Example

Before you use `XslTransform` in your application, you need to add couple of namespace references to your application. These namespace are `Sysem.Xml`, `System.Xml.XPath`, and `System.Xml.Xsl`. (I'll discuss the `X path` namespace in more detail in the "Navigation in HTML" section of this article.) This example uses the books.xsl schema file that comes with the .NET SDK sample (see listing 6-25).

Listing 6-25. XSL Transformation sample code

```
// Create a new XslTransform object and load the stylesheet
XslTransform xslt = new XslTransform();
xslt.Load(@"c:\books.Xsl");
// Create a new XPathDocument and load the XML data to be transformed.
XPathDocument mydata = new XPathDocument(@"c:\ books .xml");
// Create an XmlTextWriter which output to the console.
XmlWriter writer = new XmlTextWriter(Console.Out);
// Transform the data and send the output to the console.
xslt.Transform(mydata, null, writer);
```

Connecting Data and XML via ADO .NET

So far in this article, you've seen how to work with XML documents. In this section, you'll now learn how to work with XML documents with the help of ADO.NET. There are two approaches to work with XML and ADO. First, you can use ADO.NET to access XML documents. Second, you can use XML and ADO.NET to access XML. Additionally, you can access a relational database using ADO.NET and XML.NET.

Reading XML using a DataSet

In ADO.NET, you can access the data using the `DataSet` class. The `DataSet` class implements methods and properties to work with XML documents. The following sections discuss methods that read XML data.

The ReadXml Method

`ReadXml` is an overloaded method; you can use it to read a data stream, `TextReader`, `XmlReader`, or an XML file and to store into a `DataSet` object, which can later be used to display the data in a tabular format. The `ReadXml` method has eight overloaded forms. It can read a text, string, stream, `TextReader`, `XmlReader`, and their combination formats. In the following example, create a new `DataSet` object.

In the following example, create a new `DataSet` object and call the `DataSet.` ReadXml method to load the books.xml file in a `DataSet` object:

```
//Create a DataSet object
DataSet ds = new DataSet();
// Fill with the data
ds.ReadXml("books.xml ");
```

Once you've a `DataSet` object, you know how powerful it is. Make sure you provide the correct path of books.xml.

NOTE: Make sure you add a reference to `System.Data and the System.Data.Common` namespace before using DataSet and other common data components.

The ReadXmlSchema method

The `ReadXMLSchema` method reads an XML schema in a `DataSet` object. It has four overloaded forms. You can use a `Text Reader`, string, stream, and `XmlReader`. The following example shows how to use a file as direct input and call the `ReadXmlSchema` method to read the file:

```
DataSet ds = new DataSet( );
ds.ReadSchema (@"c:\books. xml");
```

The following example reads the `file XmlReader and uses XmlTextReader` as the input of `ReadXmlSchema`:

```
//Create a dataset object
DataSet ds = new DataSet("New DataSet");
// Read xsl in an XmlTextReader
```

```
XmlTextReader myXmlReader = new XmlTextReader(@"c:\books.Xml");
// Call Read xml schema
ds.ReadXmlSchema(myXmlReader);
myXmlReader.Close();
```

Writing XML using a DataSet

Not only reading, the `DataSet` class contains methods to write XML file from a `DataSet` object and fill the data to the file.

The WriteXml Method

The `WriteXml` method writes the current data (the schema and data) of a `DataSet` object to an XML file. This is overloaded method. By using this method, you can write data to a file, stream, `TextWriter, or XmlWriter`. This example creates a DataSet, fills the data for the DataSet, and writes the data to an XML file.

Listing 6-26. WriteXml Method

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

namespace XmlAndDataSetsampB2
{
      class XmlAndDataSetSampCls
{
      public static void Main()
{

try
{
// Create a DataSet, namespace and Student table
// with Name and Address columns
DataSet ds = new DataSet("DS");
ds.Namespace = "StdNamespace";
DataTable stdTable = new DataTable("Student");
DataColumn col1 = new DataColumn("Name");
DataColumn col2 = new DataColumn("Address");
stdTable.Columns.Add(col1);
stdTable.Columns.Add(col2);
ds.Tables.Add(stdTable);
//Add student Data to the table
DataRow newRow; newRow = stdTable.NewRow();
newRow["Name"] = "Mahesh Chand";
newRow["Address"] = "Meadowlake Dr, Dtown";
stdTable.Rows.Add(newRow);
newRow = stdTable.NewRow();
newRow["Name"] = "Mike Gold";
newRow["Address"] = "NewYork";
stdTable.Rows.Add(newRow);
newRow = stdTable.NewRow();
newRow["Name"] = "Mike Gold";
newRow["Address"] = "New York";
stdTable.Rows.Add(newRow);
```

```
ds.AcceptChanges();
// Create a new StreamWriter
// I'll save data in stdData.Xml file
System.IO.StreamWriter myStreamWriter = new
System.IO.StreamWriter(@"c:\stdData.xml");
// Writer data to DataSet which actually creates the file
ds.WriteXml(myStreamWriter);
myStreamWriter.Close();
}
catch (Exception e)
{
Console.WriteLine("Exception: {0}", e.ToString());
}
return;

}
}
}
```

You wouldn't believe the `WriteXml` method does for you. If you see the output `stdData.xml` file, it generates a standard XML file that looks like listing 6-27.

Listing 6-27. WriteXml method output

```
-<DS xmlns="StdNamespace">
  - <Student>
      <Name>Mahesh Chand</Name>
      <Address>Meadowlake Dr, Dtown</Address>
    </Student>
  - <Student>
      <Name>Mike Gold</Name>
      <Address>NewYork</Address>
    </Student>
  - <Student>
      <Name>Mike Gold</Name>
      <Address>New York</Address>
    </Student>
  </DS>
```

The Write xml schema method

This method writes `DataSet` structure to an XML schema. `WriteXmlSchema` has four overloaded methods. You can write the data to a stream, text, `TextWriter, or Xmlwriter`. Listing 6-28 uses `XmlWriter` for the output.

Listing 6-28. write xml schema sample

```
DataSet ds = new DataSet("DS");
ds.Namespace = "StdNamespace";
DataTable stdTable = new DataTable("Students");
DataColumn col1 = new DataColumn("Name");
DataColumn col2 = new DataColumn("Address");
stdTable.Columns.Add(col1);
stdTable.Columns.Add(col2);
```

```
ds.Tables.Add(stdTable);
// Add student Data to the table
DataRow newRow; newRow = stdTable.NewRow();
newRow["Name"] = "Mahesh chand";
newRow["Address"] = "Meadowlake Dr, Dtown";
stdTable.Rows.Add(newRow);
newRow = stdTable.NewRow();
newRow["Name"] = "Mike Gold";
newRow["Address"] = "NewYork";
stdTable.Rows.Add(newRow);
ds.AcceptChanges();
XmlTextWriter writer = new XmlTextWriter(Console.Out);
ds.WriteXmlSchema(writer);
```

Refer to the previous section to see how to create an `XmlTextWriter` object.

XmlDataDocument and XML

As discussed earlier in this article, the `XmlDocument` class provides DOM tree structure of XML documents. The `XmlDataDocument` class comes from `XmlDocument`, which is comes from `XmlNode`.

Figure 6-10 shows the XmlDataDocument hierarchy.

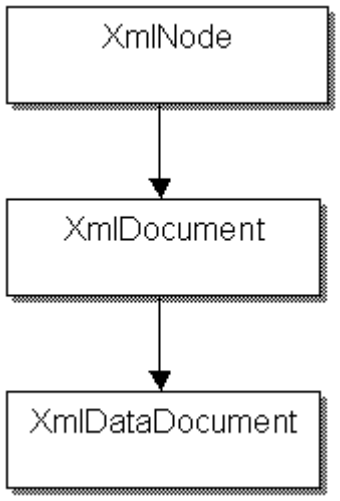

Figure 6-10. Xml Data Document hierarchy

Besides overriding the methods of `XmlNode and XmlDocument, XmlDataDocument` also implements its own methods. The `XmlDataDocument` class lets you lead relational data using the `DataSet` object as well as XML documents using the `Load and LoadXml` methods. As figure 6-11 indicates, you can use a `DataSet` to load relational data to an `XmlDataDocument` object and use the `Load or LoadXml` methods to read an XML document. Figure 6-11 shows a relationship between a `Reader, Writer, DataSet, and XmlDataDocument.`

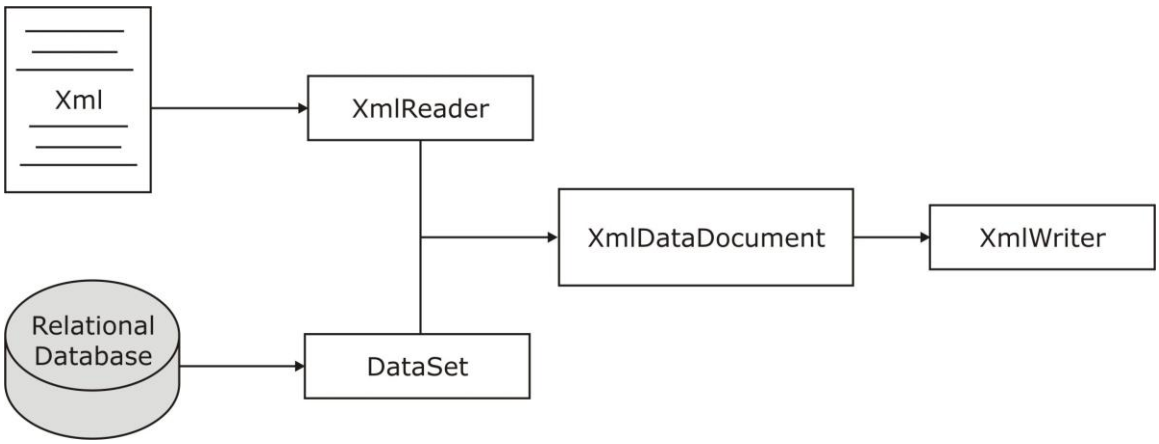

Figure 6-11. Reading and writing data using xml Data Document

The `XmlDataDocument` class extends the functionality of `XmlDocument` and synchronizes it with DataSet. As you know a `DataSet` is a powerful object in ADO.NET. As figure 6-11 shows, you can take data from two different sources. First, you can load data from an XML document with the help of `XmlReader`, and second, you can load data from relational data sources with the help of database provides and `DataSet`. The neat thing is the data synchronization between these two objects. That means if you update data in a `DataSet` object, you see results in `the XmlDataDocument` object and vice versa. For example, if you add a record to a `DataSet` object, the action will add one node to the `XmlDataDocument` object representing the newly added record.

Once the data is loaded, you're allowed to use any operations that you were able to use on XmlDocument objects. You can also use `XmlReader and XmlWriter` objects to read and write the data.

The `xmlData Documet` class has property called `DataSet`. It returns the attached `DataSet` object with `XmlDataDocument`. The `DataSet` property provides you a relational representation of an XML document. Once you've a `DataSet` object, you can do anything with it such as attaching to a DataGrid.

You Can use all XML read and write methods of the `DataSet` object through the `DataSet` property such as `ReadXml, ReadXmlSchema, WriteXml, and WriteXml` schema. Refer to the `DataSet` read write methods in the previous section to see how these methods are used.

Loading Data using `Load and LoadXml` from the XmlDataDocument

You can use either the `Load` method or the `LoadXml` method to load an XML document. The `Load` method takes a parameter of a filename string, a `TextReader, or an XmlReader`. Similarly, you can use the `LoadXml` method. This method passes an XML file name to load the XML file for example:

```
XmlDataDocument doc = new XmlDataDocument( );
doc.Load("c:\\ Books.xml");
```

Or you can load an XML fragment, as in the following example:

```
XmlDataDocument doc = new XmlDataDocument( );
doc.LoadsXml ("<Record> write something </Record>");
```

Loading Data Using a DataSet

A DataSet object has methods to read XML documents. These methods are ReadXmlSchema and LoadXml. You use the Load or LoadXml methods to load an XML document the same way you did directly from the XMLDataDocument. Again the Load method takes a parameter of a filename string, TextReader, or XmlReader. Similarly, use the LoadXml method to pass an XML filename through the dataset. For example:

```
XmlDataDocument doc = new XmlDataDocument( );
doc.DataSet.ReadXmlSchema ("test. Xsd");

Or

doc.DataSet.ReadXml ("<Record> write something </Record>");
```

Displaying XML Data In a DataSet Format

As mentioned previously, you can get DataSet object from an XmlDataDocument object by using its DataSet property. OK, now it's time to see how to do that. The next sample will show you how easy is to display an XML document data in a DataSet format.

To read XML document in a dataset, first you read to document. You can read a document using the ReadXml method of the DataSet object. The DataSet property of XmlDataDocument represents the dataset of XmlDataDocument. After reading a document in a dataset, you can create data views from the dataset, or you can also use a DataSet'sDefaultViewManager property to bind to data-bound controls, as you can see in the following code:

```
XmlDataDocument xmlDatadoc = new XmlDataDocument();
xmlDatadoc.DataSet.ReadXml ("c:\\ xmlDataDoc.xml");
dataGrid1.DataSource = xmlDatadoc.DataSet.DefaultViewManager;
```

Listing 6-29 shows the complete code. As you can see from Listing 6-29, I created a new dataset, Books, fill from the books.xml and bind to a DataGrid control using its DataSource property. To make Listing 6-29 work, you need to create a Windows application and drag a DataGrid control to the form. After doing that, you need to write the Listing 6-29 code on the Form1 constructor or Form load event.

Listing 6-29. XmlDataDocumentSample.cs

```
public Form1( )
            {
            // Initialize Component and other code here
            // Create an XmlDataDocument object and read an XML
            XmlDataDocument xmlDatadoc = new XmlDataDocument();
            xmlDatadoc.DataSet.ReadXml("C:\\books.xml");
            // Create a DataSet object and fill with the dataset
            // of XmlDataDocument
            DataSet ds = new DataSet("Books DataSet");
            ds = xmlDatadoc.DataSet;
            // Attach dataset view to the Data Grid control
            dataGrid1.DataSource = ds.DefaultViewManager;
            }
```

The output of this program looks like **figure 6-12**. Only a few lines code, and you're all set. Neat huh?

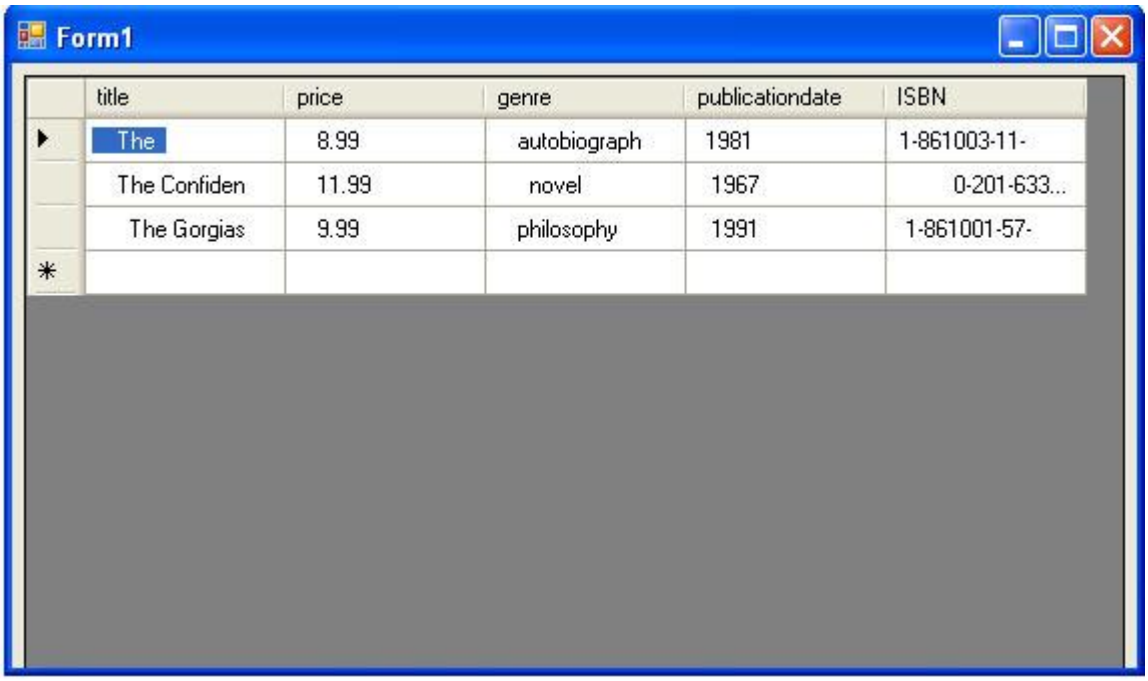

Figure 6-12. XmlDataDocumentSample.cs output

Saving Data from a DataSet to XML

You can save a `DataSet` data as an XML document using the `Save` method of `XmlDataDocument`. Actually, `XmlDataDocument` comes from `XmlDocument`., and the `XmlDocument` class defines the Save method. I've already discussed that you can use Save method to save your data in a string, stream, `TextWriter, and XmlWriter.`

First, you create a `DataSe`t object and fill it using a `DataAdapte`r. The following example reads the Customers table from the Northwind Access database and fills data from the read to the `DataSet:`

```
string SQLStmt = "SELECT * FROM Customers";
string ConnectionString =
"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C: \\ Northwind.mdb";
// Create data adapter
OleDbDataAdapter da = new OleDbDataAdapter(SQLStmt, ConnectionString);
// create a new dataset object and fill using data adapter's fill
method
DataSet ds = new DataSet();
da.Fill(ds);
```

Now, you create an instance of `XmlDataDocument` with the `DataSet` as an argument and call the `Save` method to save the data as an XML document:

```
XmlDataDocument doc = new XmlDataDocument(ds);
doc.Save("C:\\XmlDataDoc.xml");
```

Listing 6-30 shows a complete program listing. You create an `XmlDataDocument` object with dataset and call the save method to save the dataset data in an XML file.

Listing 6-30. Saving the dataset data to an XML document

```
using System;
using System.Data;
using System.Data.OleDb;
using System.Xml;

namespace DataDocsampB2
{
      class Class1
      {
            static void Main(string[] args)
            {
            // create SQL Query
            string SQLStmt = "SELECT * FROM Customers";
            // Connection string
            string ConnectionString =
            "Provider = Microsoft.Jet.OLEDB.4.0;Data Source = C:\\
            Northwind.mdb";
            // Create data adapter
            OleDbDataAdapter da = new OleDbDataAdapter(SQLStmt,
            ConnectionString);
            // create a new dataset object and fill using data
            adapter's fill method
            DataSet doc = new DataSet();
            // Now use SxlDataDocument's Save method to save data as an
            XML file XmlDataDocument doc = new XmlDataDocument(ds);
            doc.Save("C:\\ XmlDataDoc.xml");
            }
      }
}
```

Traversing XML Documents

As you've seen, `XmlNode` provides a way to navigate DOM tree with the help of its `FirstChild`, `ChildNodes`, `LastChild`, `PreviousNode`, `NextSibling`, and `PreviousSibling` methods.

Besides `XmlNode`, the XML.NET has two more classes, which help you navigate XML documents. These classes are `XPathDocument` and `XPathNavigator`. The `System.Xml.Xpath` namespace defines both these classes.

The `XPath` namespace contains classes to provide read-only, fast access to documents. Before using these classes, you must add a reference of the `System.Xml.Xpath` namespace to your application.

`XPathNodeIterator`, `XPathExpression`, and `XPathException` are other classes defined in this namespace. The `XPathNodeIterator` class provides iteration capabilities to a node. `XPathExpression` provides selection criteria to select a set of nodes from a document based on those criteria, and the `XPathExection` class is an exception class. The `XPathDocument` class provides a fast cache for XML document processing using XSLT and XPath.

You use the `XPathDocument` constructor to create an instance of `XmlPathDocument`. It has many overloaded constructors. You can pass an `XmlReader`, `TextReader`, or even direct XML filenames.

The XPathNavigator class

The `XPathNavigator` class implements the functionality to navigate through a document. It has easy-to-use and self-explanatory methods. You create an `XPathNavigator` instance by calling `XPpathDocument's CreateNavigator` method.

You can also create a `XPathNavigator` object by calling `XmlDocument's CreateNavigator method`. For example, the following code calls `XmlDocument 's CreateNavigator` method to create a `XPathNavigator` object:

```
// Load books.xml document
XmlDocument xmlDoc = new XmlDocument();
XmlDoc.Load(@"c:\ books.xml");
// Create XPathNavigator object by calling create Navigator of
XmlDocument
XPathNavigator nav = xmlDoc.CreateNavigator();
```

NOTE: Don't forget to add a reference of the `System.Xml .XPath` to your project before using any of its classes.

`XPathNavigator` contain methods and properties to move to the first, next, child, parent, and root nodes to the document.

XPathNavigator move methods

Table 6-8 describes the `XPathNavigator` class's move methods. Some of these methods are `MoveToFirst`, `moveToNext` , `MoveToroot`, `MoveToFirstAttribute`,

MoveToFirstChild, MoveToId, MoveToNamespace, MoveToPrevious, MoveToParent and so on.

Table 6-8. XPathNavigator Memebers

MEMBER	DESCRPITION
MoveToAttribute	Moves to an attribute
MoveToFirst	Moves to the first sibling of the current node
MoveToFirstAttribute	Moves to the first attribute
MoveToFirstChild	Moves to the first child of the current node
MoveToFirstNamespace	Moves the X Path Navigator to the first namespace node of the current element
MoveToId	Moves to the node with specified ID
MoveToNamespace	Moves to the specified namespace
MoveToNext	Moves to the next node of the current node
MoveToNextAttribute	Moves to the next Attribute
MoveToNextNamespace	Moves to the Next namespace
MoveToParent	Moves to the parent of the current node
MoveToPrevious	Moves to the previous sibling of the current node
MoveToRoot	Moves to the root node

So, with the help of these methods, you can move through a document as a DOM tree. Listing 6-31 uses the MoveToRoot and MoveToFirstChild methods to move to the root node and first child of the root node. Once you have a root, you can display corresponding information such as name, value, node type, and so on.

Listing 6-31. Moving to root and first child nodes using XpathNavigator

```
// Load books.xml document
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load(@"c:\ books.xml");
// Create XPathNavigator object by calling CreateNavigator of
XmlDocument
XPathNavigator nav = xmlDoc.CreateNavigator();
//Move to root node
nav.MoveToRoot();
string name = nav.Name;
Console.WriteLine("Root node info: ");
Console.WriteLine("Base URI" + nav.BaseURI.ToString());
Console.WriteLine("Name:" + nav.Name.ToString());
Console.WriteLine("Node Type: " + nav.NodeType.ToString());
Console.WriteLine("Node Value: " + nav.Value.ToString());
if (nav.HasChildren)
{
nav.MoveToFirstChild();
}
```

Now, using the MoveToNext and MoveToParent methods, you can move through the entire document. Listing 6-32 Moves though an entire document and displays the data on the console. The GetNodeInfo method displays a node's information, and you call it recursively.

Listing 6-32. Reading a document using XpathNavigator

```
static void Main(string[] args)
{
```

```
// Load books.xml document
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load(@"c:\ books.xml");
// Create XPathNavigator object by calling CreateNavigator of
XmlDocument
XPathNavigator nav = xmlDoc.CreateNavigator( );
// move to root node
nav.MoveToRoot();
string name = nav.Name;
Console.WriteLine("Root node info: ");
Console.WriteLine("Base URI" + nav.BaseURI.ToString());
Console.WriteLine("Name: " + nav.NodeType.ToString());
Console.WriteLine("Node Type: " + nav.NodeType.ToString());
Console.WriteLine("Node Value: " + nav.Value.ToString());
if (nav.HasChildren)
{
nav.MoveToFirstChild();
GetNodeInfo(nav);
}
}
private static void GetNodeInfo(XPathNavigator nav1)
{
Console.WriteLine("Name: " + nav1.Name.ToString());
Console.WriteLine("Node Type: " + nav1.NodeType.ToString());
Console.WriteLine("Node value: " + nav1.Value.ToString());
// If node has children, move to first child.
if (nav1.HasChildren)
{
nav1.MoveToFirstChild();
while (nav1.MoveToNext())
{
GetNodeInfo(nav1);
nav1.MoveToParent();
}
}
else /* Else move to next sibling */
{
nav1.MoveToNext();
GetNodeInfo(nav1);
}
}
```

Searching using XPathNavigator

`Select`, `SelectChildren`, `SelectAncestors`, and `SelectDescendents` are other useful methods. Specifically, these methods are useful when you need to select a document's items based on an XPath expression. For example, you could use one when selecting nodes for the author tag only and so on. Now, say you want to search and display all `<first- name>` tag nodes in the `books.xml` document.

In listing 6-33, you use `XPathNavigator's Select` method to apply a criteria (all elements with the author-name tag) to read and display all nodes.

Listing 6-33. Use `of XPathIterator` and `Select`

```
// Load books.xml document
```

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load (@" c:\ books.xml");

// Create XPathNavigator object by calling CreateNavigator of
XmlDocument
XPathNavigator nav = xmlDoc.CreateNavigator();
// Look for author's first name
Console.WriteLine("Author First Name");
XPathNodeIterator itrator= nav.Select("descendant : : first-name");
while(itrator.MoveNext())
{
Console.WriteLine(itrator.Current.Value.ToString());
}
```

XML Designer in Visual Studio .NET

XML schemas play major in the .NET Framework, and visual studio .NET provides many tools and utilities to work with XML. The.NET Framework uses XML to transfer data from one application to another. XML schemas define the structure and validation rules of XML document. You use XML schemas definition (XSD) language to define XML schemas

VS.NET provides an XML designer to work with schemas. In this section you'll see how you can take advantage of the VS.NETXML designer and wizard features to work with XML Documents and database.

Generating a New Schema

To generate a new schema, create a new Windows application using File > New > Project > Visual C# Projects > Window Application. Just follow the steps outlined in the following sections.

Adding an Empty Schema

First, right-click on the project select Add > Add New Item (see Figure 6-13).

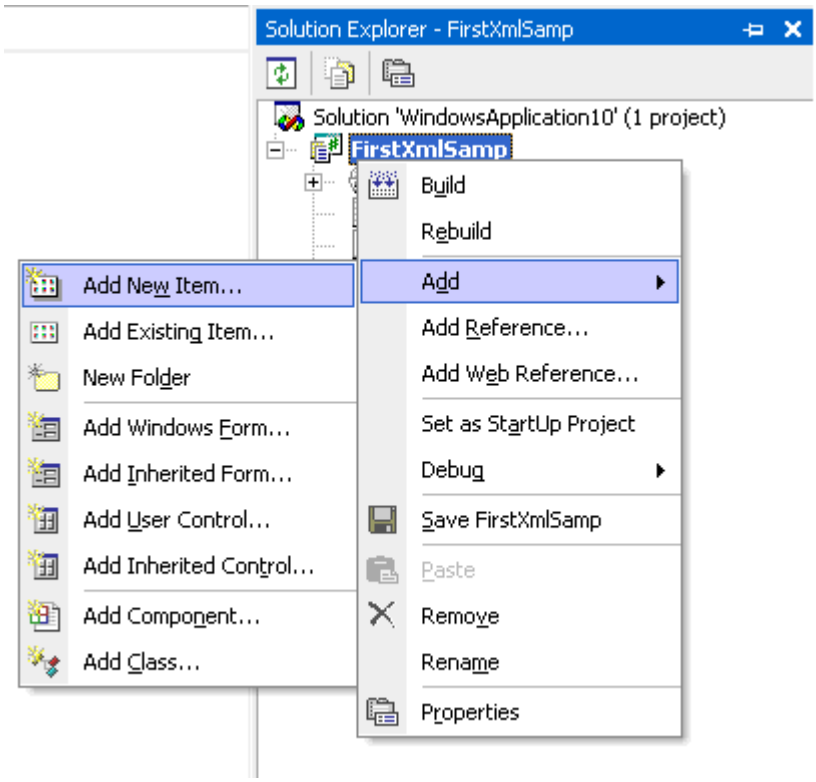

Figure 6-13. Adding a new item to the project

Now, from Templates, select the XML schema option, type your schema name, and click open (see figure 6-14).

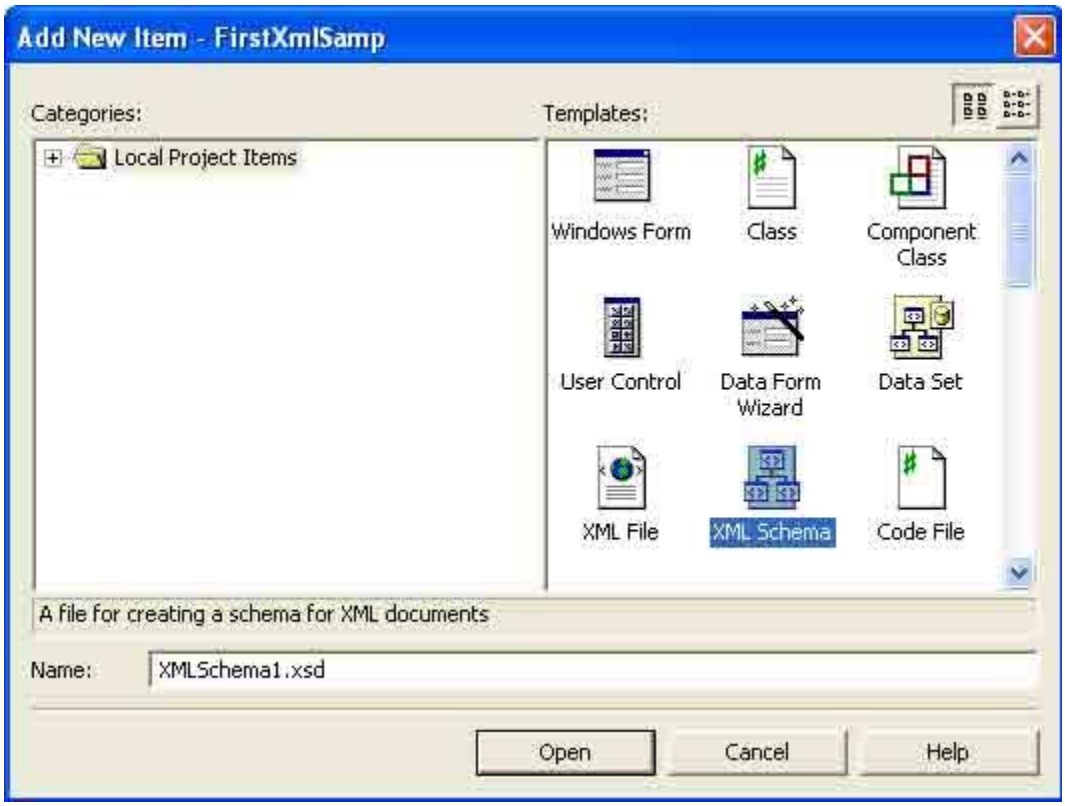

Figure 6-14. Selecting the XML schema template to add a schema to the project

This action launches XML Designer, now you'll see your XmlSchema1.xsd file, as shown in figure 6-15.

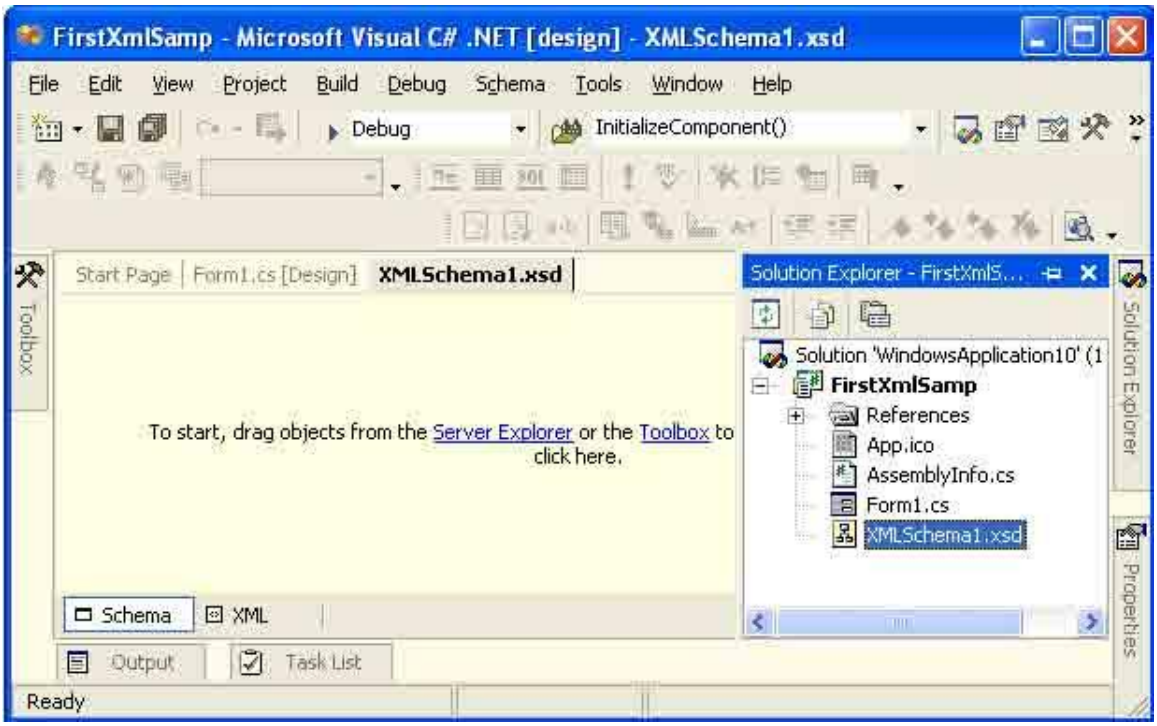

Figure 6-15. XML Designer

This action adds an empty XML schema to your project. If you click on the XML option at the button of screen, you'll see your XML looks like the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="XMLSchema1"
targetNamespace="http://tempuri.org/XMLSchema1.xsd"
elementFormDefault="qualified"
xmlns="http://tempuri.org/XMLSchema1.xsd"
xmlns:mstns="http://tempuri.org/XMLSchema1.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
</xs:schema>
```

As you see in figure 6-15, there are two options (blue links): Server Explorer and Toolbox.

Adding Schema Items

You can add schema items using the Toolbox option. Clicking the Toolbox link launches the toolbox, as shown in figure 6-16.

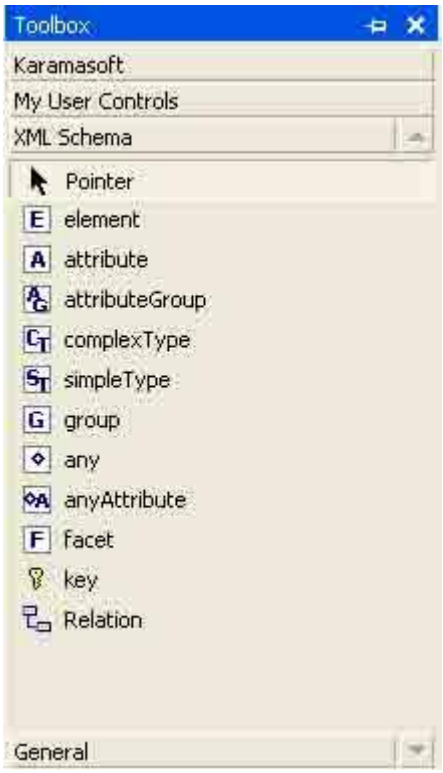

Figure 6-16. XML schema toolbox

As you can see in figure 6-16, you can add an element, attribute, complexType, and other schema items to the form by just dragging an item to XML Designer.

OK, now you'll learn how to add XML schema items to the schema and set their properties with the help of XML Designer. First, add an element. To add an element to the schema, drag an element from the toolbox. Now you can set its name and type in the designer. The default

element looks like figure 6-17, if you click on the Right-side column of the grid, you'll see a drop-down list with element types. You can either select the type of an item from the list or define your own type. Your type is called a user-defined type.

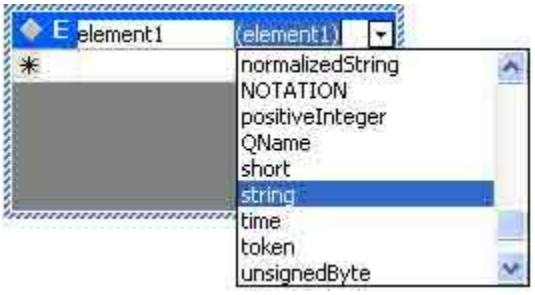

Figure 6-17. Adding a schema element and its type

Define your first element as `bookstore` with a custom type of `bookstoretype`. Figure 6-18 shows the bookstore element of `bookstoretype`.

Figure 6-18. Adding a new bookstore element

Now add a `complexType` by dragging a `comlpexType` to XML Designer (see figure 6-19).

Figure 6-19. A complex Type item

A complex Type item can contain other types, too. You can add items to a complexType in many ways. You can either drag an item from the toolbox to the complexType or right-click on a complexType and use the Add option and its sub options to add an item. Figure 6-20 shows different items you can add to a complex type.

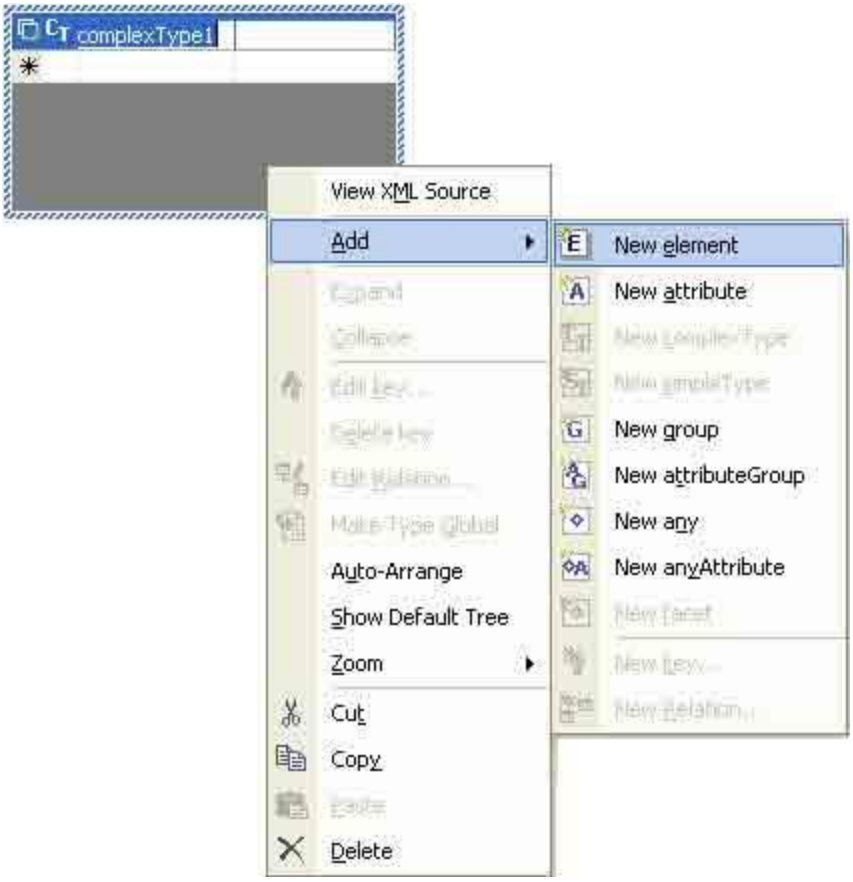

Figure 6-20. An item list can be added to a complex Type

You can delete items by right clicking and selecting Delete. You can also delete the entire complexType or other schema items by right clicking on the header of an item or on the left side of the item.

Now rename the added complexType name to book and add four element types: title, author, price, and category. Now your complexType book looks like Figure 6-21.

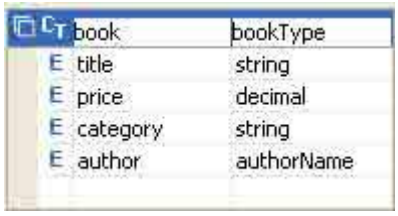

Figure 6-21. The book complex Type and its elements

After that, adds one more complexType author with two elements: `first-name and last-name`. Your final schema look like **figure 6-22.**

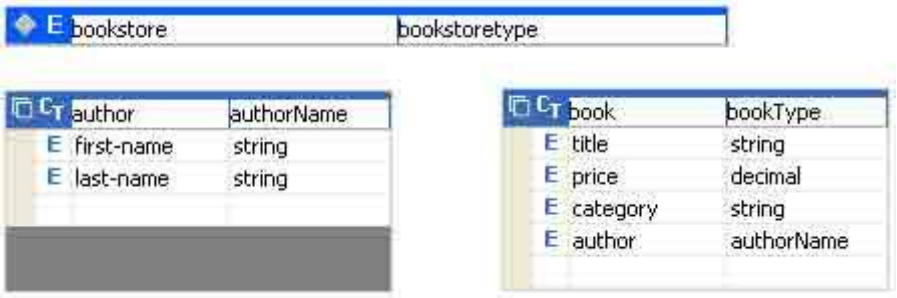

Figure 6-22. The author and book complexType in an XML schema

Now you can see XML code for this schema by clicking on the left-bottom XML button shown in figure 6-23.

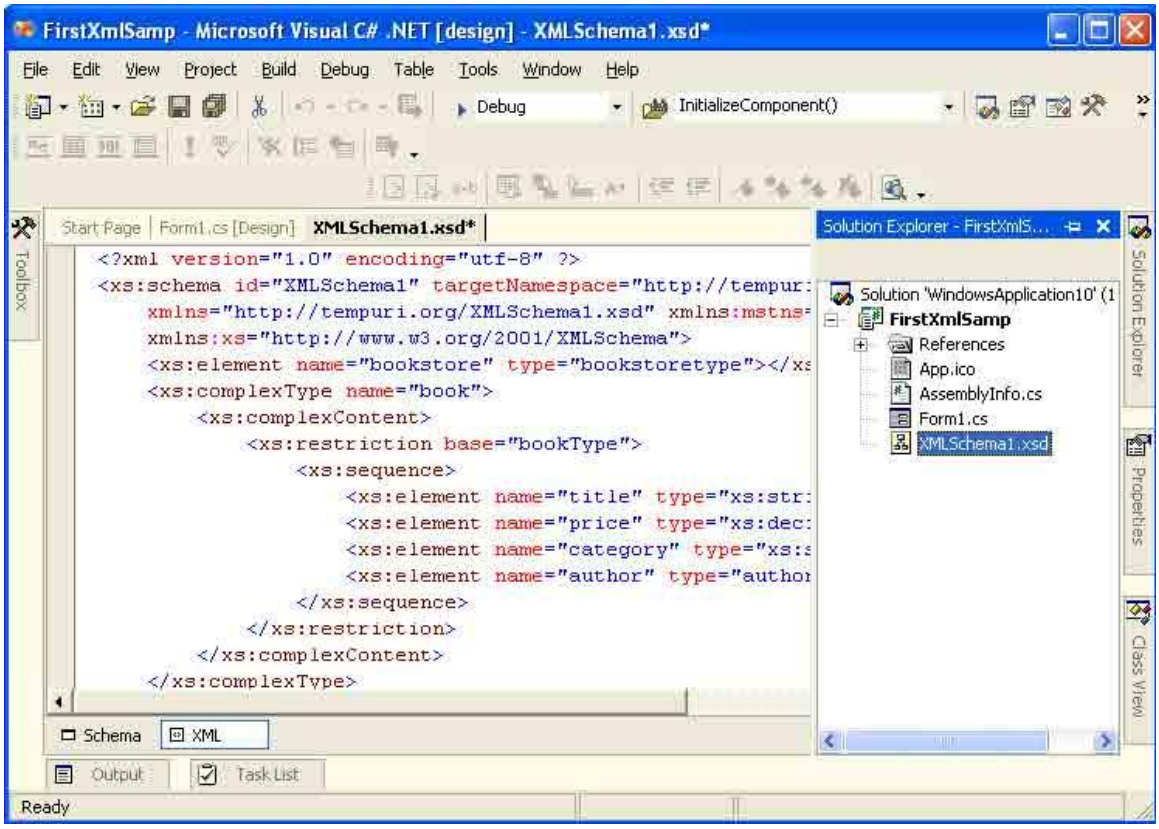

Figure 6-23. Viewing the XML for a schema

Listing 6-34 shows the schema XML code.

Listing 6-34. Xml generated using XML Designer

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="XMLSchema1"
targetNamespace="http://tempuri.org/XMLSchema1.xsd"
elementFormDefault="qualified"
xmlns="http://tempuri.org/XMLSchema1.xsd"
xmlns:mstns="http://tempuri.org/XMLSchema1.xsd"
```

```
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="bookstore" type="bookstoretype"></xs:element>
<xs:complexType name="book">
<xs:complexContent>
<xs:restriction base="bookType">
<xs:sequence>
<xs:element name="title" type="xs:string"></xs:element>
<xs:element name="price" type="xs:decimal"></xs:element>
<xs:element name="category" type="xs:string"></xs:element>
<xs:element name="author" type="authorName"></xs:element>
</xs:sequence>
</xs:restriction>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="author">
<xs:complexContent>
<xs:restriction base="authorName">
<xs:sequence>
<xs:element name="first-name" type="xs:string" />
<xs:element name="last-name" type="xs:string" />
</xs:sequence>
</xs:restriction>
</xs:complexContent>
</xs:complexType>
</xs:schema>
```

Working with `DataSets`

Now you'll look at the Server Explorer option of XML Designer. Clicking on server Explorer launches Server Explorer (see figure 6-24).

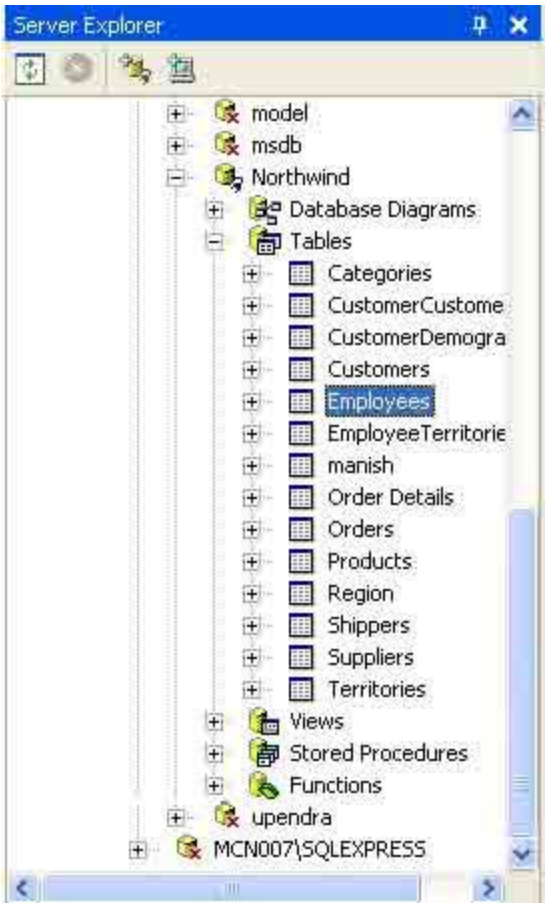

Figure 6-24. Server Explorer

In figure 6-24, you see that you can expand a database connection and see its tables and views. You can drag these data objects (tables, views, stored procedures, columns) onto XML Designer. For this example, drag the Employee table onto the designer. After dragging, your XML Designer generates a schema for the table, which looks like figure 6-25.

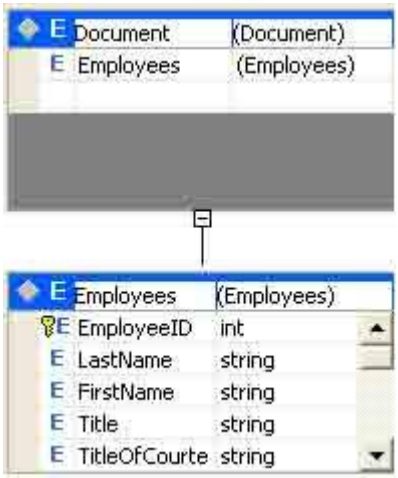

Figure 6-25. XML Designer – generated schema

Listing 6-35 shows the generated XML code.

Listing 6-35. XML Schema generated for a database table

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="XMLSchema1"
targetNamespace="http://tempuri.org/XMLSchema1.xsd"
elementFormDefault="qualified"
xmlns="http://tempuri.org/XMLSchema1.xsd"
xmlns:mstns="http://tempuri.org/XMLSchema1.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-
microsoft-com:xml-msdata">
<xs:element name="bookstotre">
<xs:complexType>
<xs:sequence />
</xs:complexType>
</xs:element>
<xs:complexType name="book">
<xs:complexContent>
<xs:restriction base="booktype">
<xs:sequence>
<xs:element name="titleelement1" type="xs:string"></xs:element>
<xs:element name="author" type="authername"></xs:element>
<xs:element name="price" type="xs:decimal"></xs:element>
<xs:element name="categary" type="xs:string"></xs:element>
</xs:sequence>
</xs:restriction>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="author">
<xs:complexContent>
<xs:restriction base="authorName">
<xs:sequence>
<xs:element name="first-name" type="xs:string"></xs:element>
<xs:element name="last-name" type="xs:string" />
</xs:sequence>
</xs:restriction>
</xs:complexContent>
</xs:complexType>
<xs:element name="Document">
<xs:complexType>
<xs:choice maxOccurs="unbounded">
<xs:element name="Employees">
<xs:complexType>
<xs:sequence>
<xs:element name="EmployeeID" msdata:ReadOnly="true"
msdata:AutoIncrement="true" type="xs:int" />
<xs:element name="LastName" type="xs:string" />
<xs:element name="FirstName" type="xs:string" />
<xs:element name="Title" type="xs:string" minOccurs="0" />
<xs:element name="TitleOfCourtesy" type="xs:string" minOccurs="0" />
<xs:element name="BirthDate" type="xs:dateTime" minOccurs="0" />
<xs:element name="HireDate" type="xs:dateTime" minOccurs="0" />
<xs:element name="Address" type="xs:string" minOccurs="0" />
<xs:element name="City" type="xs:string" minOccurs="0" />
<xs:element name="Region" type="xs:string" minOccurs="0" />
<xs:element name="PostalCode" type="xs:string" minOccurs="0" />
```

```
<xs:element name="Country" type="xs:string" minOccurs="0" />
<xs:element name="HomePhone" type="xs:string" minOccurs="0" />
<xs:element name="Extension" type="xs:string" minOccurs="0" />
<xs:element name="Photo" type="xs:base64Binary" minOccurs="0" />
<xs:element name="Notes" type="xs:string" minOccurs="0" />
<xs:element name="ReportsTo" type="xs:int" minOccurs="0" />
<xs:element name="PhotoPath" type="xs:string" minOccurs="0" />
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:choice>
</xs:complexType>
<xs:unique name="DocumentKey1" msdata:PrimaryKey="true">
<xs:selector xpath=".//mstns:Employees" />
<xs:field xpath="mstns:EmployeeID" />
</xs:unique>
</xs:element>
</xs:schema>
```

Generating ADO.NET Typed DataSet from a Schema

There may be occasions when other applications will generate XML schemas and your application needs to use them to access databases. You can generate a typed dataset from an existing schema. But before generating Dataset option generates a typed `DataSet` for an XML schema. But before generating a `DataSet` you need to add schema to the project.

Adding an Existing schema to project

Now you'll see how you can generate a `DataSet` object from an existing schema. To test this. I created a new Windows application project. You can use the Employee table schema generated in the previous section. To add an existing schema to the project, right-click on the project and select Add > Add Existing Item and browse for the schema (see figure 6-26).

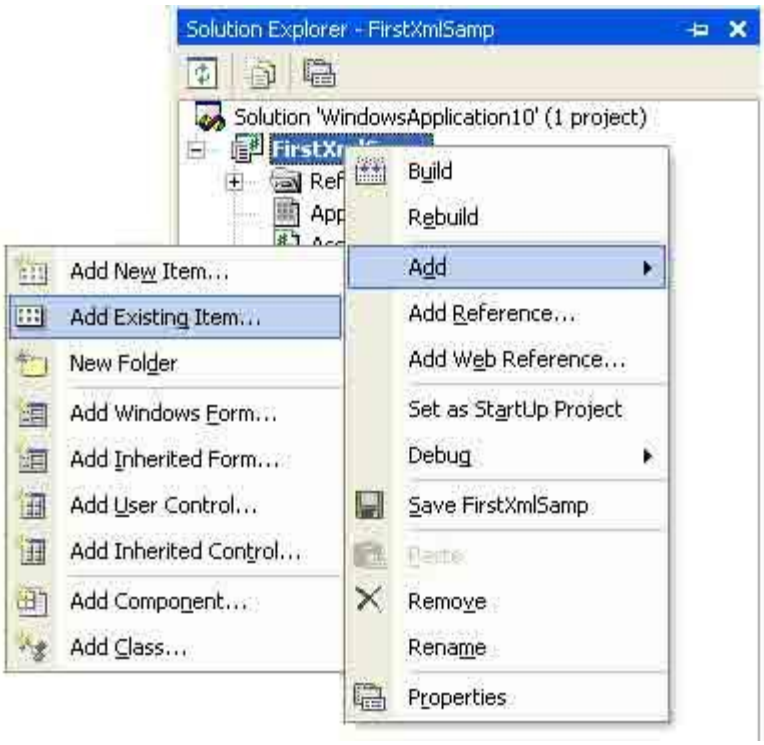

Figure 6-26. Adding an existing schema to a project

If your schema name was different, select that schema and click open (see figure 6-27).

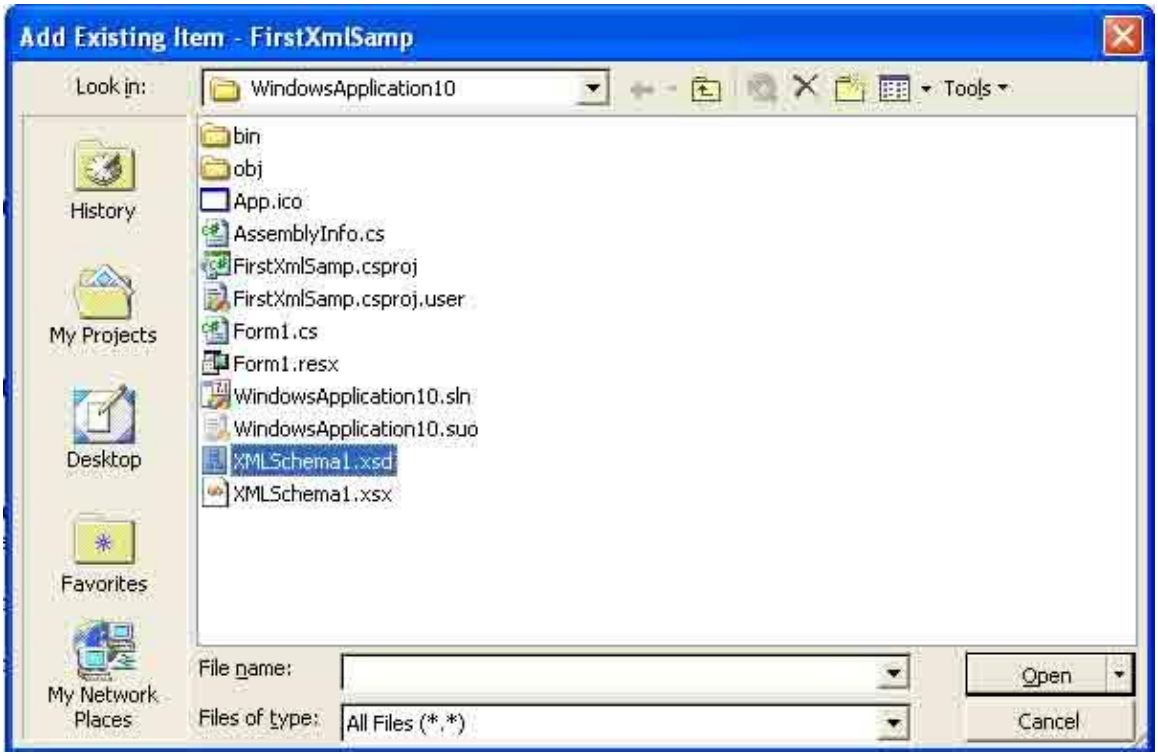

Figure 6-27. Browsing for schema

This action adds a schema to the current project. You can also add an XML schema by dragging a database table onto XML Designer.

Generating a Typed Data Set from a schema

Generating a typed dataset from a schema is pretty simple. Right–click on XML Designer and select the Generate Dataset option (see figure 6-28).

Figure 6-28. Generate Data set option of XML Designer

This action generates a `DataSet` class and adds it to your project. If you look in your Class Wizard, you use the Document class derived from `DataSet` and its members. The Document class looks like figure 6-29 in the Class View.

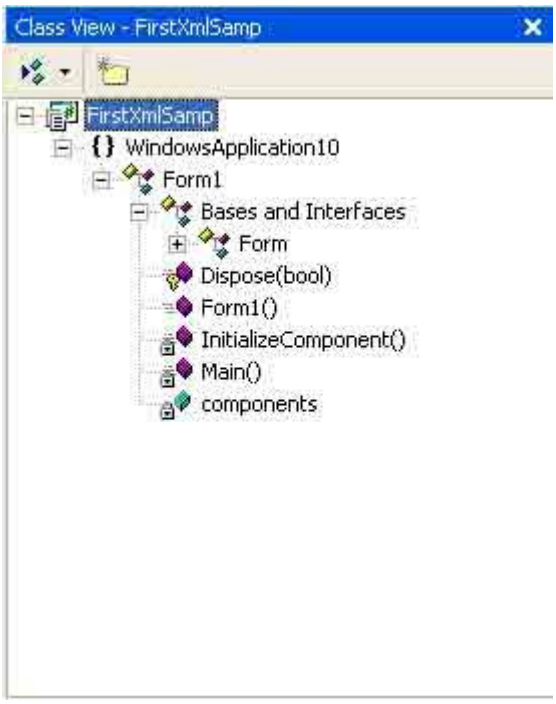

Figure 6-29. DataSet-derived class in the Class View

NOTE: The Generate Data Set option may not generate a Data Set if the XML schema is not designed properly.

Once you've `a DataSet` object, you can use it the way you want.

Summary

This article covered XML syntax as well the uses of XML on the .NET platform. You learned about the DOM structure and DOM node types. You learned how XML is represented in .NET through classes such as `XmlNode`, `XmlAttribute`, `XmlElement`, and `XmlDocument`. You also learned how to read and write to these structures using the `XmlReader` and `XmlWriter` classes. Also discussed was the navigation in an XML node structure using `XmlPathNavigator`. Most importantly, you learned how XML applies to ADO.NET and how to use a `DataSe`t to read and write data with XML. Visual Studio .NET provides XML Designer to work with XML. Using XML Designer, you can generate XML schema, which later can be used to generate typed datasets.