Bonus Chapter 2

XML and JSON

WHAT'S IN THIS CHAPTER?

- XML standards
- XmlReader and XmlWriter
- XmlDocument
- XPathNavigator
- LINQ to XML
- The System.Xml.Linq namespace
- Queries in XML documents using LINQ
- JSON
- Object conversion with JSON

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The Wrox.com code downloads for this chapter are found at `www.wrox.com` on the Download Code tab. The source code is also available at `https://github.com/ProfessionalCSharp/ProfessionalCSharp7` in the directory `XMLandJSON`.

The code for this chapter is divided into the following major examples:

- XmlReaderAndWriter
- XmlDocument
- XPathNavigator
- ObjectToXmlSerialization
- ObjectToXmlSerializationWOAttributes
- LinqToXmlSample
- JsonSample

DATA FORMATS

The Extensible Markup Language (XML) has been playing an important part in information technology since 1996. The language is used to describe data, and it's used with configuration files, source code documentation, web services that make use of SOAP, and more. In recent years, it has been replaced in some ways (for example, configuration files and data transfer from REST-based web services) by JavaScript Object Notation (JSON) because this technology has less overhead and can be used easily from JavaScript. However, JSON cannot replace XML in all the scenarios where XML is used today. Both data formats can be used with .NET applications, as covered in this chapter.

For processing XML, different options are available. You can either read the complete document and navigate within the Document Object Model (DOM) hierarchy using the `XmlDocument` class, or you can use `XmlReader` and `XmlWriter`. Using `XmlReader` is more complex to do, but you can read larger documents. With `XmlDocument`, the complete document is loaded in the memory. With the `XmlReader` it is possible to read node by node.

Another way to work with XML is to serialize .NET object trees to XML and deserialize XML data back into .NET objects using the `System.Xml.Serialization` namespace.

When querying and filtering XML content, you can either use an XML standard XPath or use LINQ to XML. Both technologies are covered in this chapter. LINQ to XML also offers an easy way to create XML documents and fragments.

> **NOTE** *If you want to learn more about XML,* Wrox's Beginning XML, *5th Edition (Wiley, 2012) is a great place to start.*

The discussion begins with a brief overview of the status of XML standards.

XML

The first XML examples use the file `books.xml` as the source of data. You can download this file and the other code samples for this chapter from the Wrox website (`www.wrox.com`). The `books.xml` file is a book catalog for an imaginary bookstore. It includes book information such as genre, author name, price, and International Standard Book Number (ISBN).

This is what the `books.xml` file looks like:

```
<?xml version='1.0'?>
<!-- This file represents a fragment of a book store inventory database -->
<bookstore>
  <book genre="autobiography" publicationdate="1991" 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>Melville</last-name>
    </author>
    <price>11.99</price>
  </book>
  <book genre="philosophy" publicationdate="1991" ISBN="1-861001-57-6">
```

```
    <title>The Gorgias</title>
    <author>
      <name>Plato</name>
    </author>
    <price>9.99</price>
  </book>
</bookstore>
```

Let's have a look at the parts of this XML content. An XML document should start with an XML declaration that specifies the XML version number:

```
<?xml version='1.0'?>
```

You can put comments anywhere in an XML document outside of markup. They start with `<!--` and end with `-->`:

```
<!-- This file represents a fragment of a book store inventory database -->
```

A full document can contain only a single root element (whereas an XML fragment can contain multiple elements). With the `books.xml` file, the root element is `bookstore`:

```
<bookstore>
  <!-- child elements here -->
</bookstore>
```

An XML element can contain child elements. The `author` element contains the child elements `first-name` and `last-name`. The `first-name` element itself contains *inner text* `Benjamin`. `first-name` is a *child* element of `author`, which also means `author` is a *parent* element of `first-name`. `first-name` and `last-name` are *sibling* elements:

```
<author>
  <first-name>Benjamin</first-name>
  <last-name>Franklin</last-name>
</author>
```

An XML element can also contain attributes. The `book` element contains the attributes `genre`, `publicationdate`, and `ISBN`. Values for attributes need to be surrounded by quotes.

```
<book genre="novel" publicationdate="1967" ISBN="0-201-63361-2">
</book>
```

> **NOTE** *The HTML5 specification doesn't require quotes with attributes. HTML is not XML; HTML has a more relaxed syntax, whereas XML is strict. HTML documents can also be written using XHTML, which uses XML syntax.*

XML Standards Support in .NET

The World Wide Web Consortium (W3C) has developed a set of standards that give XML its power and potential. Without these standards, XML would not have the impact on the development world that it does. The W3C website (`www.w3.org`) is a valuable source for all things XML.

The .NET Framework supports the following W3C standards:

- XML 1.0 (`www.w3.org/TR/REC-xml`), including DTD support
- XML namespaces (`www.w3.org/TR/REC-xml-names`), both stream level and DOM
- XML schemas (`www.w3.org/XML/Schema`)
- XPath expressions (`www.w3.org/TR/xpath`)
- XSLT transformations (`www.w3.org/TR/xslt`)

- DOM Level 1 Core (`www.w3.org/TR/REC-DOM-Level-1`)
- DOM Level 2 Core (`www.w3.org/TR/DOM-Level-2-Core`)
- SOAP 1.2 (`www.w3.org/TR/SOAP`)

The level of standards support changes as the W3C updates the recommended standards and as Microsoft and the community update .NET Core. Therefore, you need to make sure that you stay up to date with the standards and the level of support provided.

Working with XML in the Framework

The .NET Framework gives you many different options for reading and writing XML. You can directly use the DOM tree to work with `XmlDocument` and classes from the `System.Xml` namespace and the `System.Xml.XmlDocument` NuGet package. This works well and is easy to do with files that fit into the memory.

For fast reading and writing XML, you can use the `XmlReader` and `XmlWriter` classes. These classes allow streaming and make it possible to work with large XML files. These classes are in the `System.Xml` namespace as well, but they're in a different NuGet package: `System.Xml.ReaderWriter`.

For using the XPath standard to navigate and query XML, you can use the `XPathNavigator` class. This is defined in the `System.Xml.XPath` namespace in the NuGet package `System.Xml.XmlDocument`.

.NET also offers another syntax to query XML: LINQ. Although LINQ to XML doesn't support the W3C DOM standard, it provides an easier option to navigate within the XML tree and allows easier creating of XML documents or fragments. The namespace needed here is `System.Xml.Linq`, and the NuGet package `System.Xml.XDocument`.

> **NOTE** *LINQ is covered in Chapter 13, "Language Integrated Query." The specific implementation of LINQ, LINQ to XML, is covered in this chapter.*

To serialize and deserialize .NET objects to XML, you can use the `XmlSerializer`. With .NET Core, the NuGet package needed here is `System.Xml.XmlSerializer` with the namespace `System.Xml.Serialization`.

WCF uses another method for XML serialization: data contract serialization. Although the `XmlSerializer` does allow you to differ serialization between attributes and elements, this is not possible with the `DataContractSerializer` serializing XML.

JSON

JavaScript Object Notation (JSON) came up in recent years because it can be directly used from JavaScript, and it has less overhead compared to XML. JSON is defined by IETF RFC 7159 (`https://tools.ietf.org/html/rfc7159`), and the ECMA standard 404 (`http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`).

For sending JSON documents, there's an official MIME type `"application/json"`. Some frameworks still use older, unofficial MIME types `"text/json"` or `"text/javascript"`.

The same content as the earlier XML file is described here using JSON. Arrays of elements are contained within brackets. In the example, the JSON file contains multiple `book` objects. Curly brackets define objects or dictionaries. The key and value are separated by a colon. The key needs to be quoted; the value is a string:

```
[
  "book": {
    "genre": "autobiography",
    "publicationdate": 1991,
    "ISBN": "1-861003-11-0",
```

```
    "title": "The Autobiography of Benjamin Franklin"
    "author": {
      "first-name": "Benjamin",
      "last-name": "Franklin"
    },
    "price": 8.99
  },
  "book": {
    "genre": "novel",
    "publicationdate": 1967,
    "ISBN": "1-861001-57-6",
    "title": "The Confidence Man"
    "author": {
      "first-name": "Herman",
      "last-name": "Melville"
    },
    "price": 11.99
  },
  "book": {
    "genre": "philosophy",
    "publicationdate": 1991,
    "ISBN": "1-861001-57-6",
    "title": "The Georgias"
    "author": {
      "name": "Plato",
    },
    "price": 9.99
  }
]
```

With .NET, JSON is used in many different places. When you're creating new DNX projects, you can see JSON used as the project configuration file. It's used with web projects to serialize data from and to the client using the Web API (see Chapter 32, "ASP.NET Core Web API.") and used in data stores such as the NoSQL database Azure Cosmos DB.

Different options are available to you when you're using JSON with .NET. One of the JSON serializers is the `DataContractJsonSerializer`. This type derives from the base class `XmlObjectSerializer`, although it doesn't really have a relation to XML. At the time when the data contract serialization technology was invented (which happened with .NET 3.0), the idea was that from now on every serialization is XML (XML in binary format is available as well). As time moved on, this assumption was not true anymore. JSON was widely used. As a matter of fact, JSON was added to the hierarchy to be supported with the data contract serialization. However, a faster, more flexible implementation won the market and is now supported by Microsoft and used with many .NET applications: `Json.NET`. Because this library is the one most used with .NET applications, it is covered in this chapter.

Beside the core JSON standard, JSON grows as well. Features known from XML are added to JSON. Let's get into examples of the JSON improvements and compare them to XML features. The XML Schema Definition (XSD) describes XML vocabularies; at the time of this writing, the JSON Schema with similar features is a work in progress. With WCF, XML can be compacted with a custom binary format. You can also serialize JSON in a binary form that is more compact than the text format. A binary version of JSON is described by BSON (Binary JSON): `http://bsonspec.org`. Sending SOAP (an XML format) across the network makes use of the Web Service Description Language (WSDL) to describe the service. With REST services that are offering JSON data, a description is available as well: Swagger (`http://swagger.io`).

> **NOTE** *ASP.NET Web API and Swagger are covered in Chapter 32.*

Now it's time to get into concrete uses of the .NET classes.

READING AND WRITING STREAMED XML

The `XmlReader` and `XmlWriter` classes provide a fast way to read and write large XML documents. `XmlReader`-based classes provide a very fast, forward-only, read-only cursor that streams the XML data for processing. Because it is a streaming model, the memory requirements are not very demanding. However, you don't have the navigation flexibility and the read or write capabilities that would be available from a DOM-based model. `XmlWriter`-based classes produce an XML document that conforms to the W3C's XML 1.0 (4th edition).

The sample code using `XmlReader` and `XmlWriter` makes use of the following namespaces:

```
System

System.IO

System.Text

System.Xml
```

The application enables you to specify several command-line arguments for all the different sample cases that are defined as `const` value and specifies the filenames to read and write to (code file `XmlReaderAndWriterSample/Program.cs`):

```
class Program
{
  private const string BooksFileName = "books.xml";
  private const string NewBooksFileName = "newbooks.xml";
  private const string ReadTextOption = "-r";
  private const string ReadElementContentOption = "-c";
  private const string ReadElementContentOption2 = "-c2";
  private const string ReadDecimalOption = "-d";
  private const string ReadAttributesOption = "-a";
  private const string WriteOption = "-w";
  //...
}
```

The `Main` method invokes the specific sample method based on the command line that is passed:

```
static void Main(string[] args)
{
  if (args.Length != 1)
  {
    ShowUsage();
    return;
  }

  switch (args[0])
  {
    case ReadTextOption:
      ReadTextNodes();
      break;
    case ReadElementContentOption:
      ReadElementContent();
      break;
    case ReadElementContentOption2:
      ReadElementContent2();
      break;
    case ReadDecimalOption:
      ReadDecimal();
      break;
    case ReadAttributesOption:
      ReadAttributes();
      break;
```

```
      default:
        ShowUsage();
        break;
    }
  }
```

Reading XML with XmlReader

The `XmlReader` enables you to read large XML streams. It is implemented as a pull model parser to pull data into the application that's requesting it.

The following is a very simple example of reading XML data; later you take a closer look at the `XmlReader` class. Because the `XmlReader` is an abstract class, it cannot be directly instantiated. Instead, the factory method `Create` is invoked to return an instance that derives from the base class `XmlReader`. The `Create` method offers several overloads where either a filename, a `TextReader`, or a `Stream` can be supplied with the first argument. The sample code directly passes the filename to the `Books.xml` file. After the reader is created, nodes can be read using the `Read` method. As soon as no node is available, the `Read` method returns `false`. You can debug through the `while` loop to see all the node types returned from the `books.xml` file. Only with the nodes of type `XmlNodeType.Text` is the value written to the console (code file `XMLReaderAndWriterSample/Program.cs`):

```
public static void ReadTextNodes()
{
  using (XmlReader reader = XmlReader.Create(BooksFileName))
  {
    while (reader.Read())
    {
      if (reader.NodeType == XmlNodeType.Text)
      {
        Console.WriteLine(reader.Value);
      }
    }
  }
}
```

Running the application with the `-r` option shows the value of all text nodes:

```
The Autobiography of Benjamin Franklin
Benjamin
Franklin
8.99
The Confidence Man
Herman
Melville
11.99
The Gorgias
Plato
9.99
```

Using Read Methods

Several ways exist to move through the document. As shown in the previous example, `Read` takes you to the next node. You can then verify whether the node has a value (`HasValue`) or, as you see later, whether the node has any attributes (`HasAttributes`). You can also use the `ReadStartElement` method, which verifies whether the current node is the start element and then positions you on the next node. If you are not on the start element, an `XmlException` is raised. Calling this method is the same as calling the `IsStartElement` method followed by a `Read` method.

`ReadElementString` is like `ReadString` except that you can optionally pass in the name of an element. If the next content node is not a start tag, or if the `Name` parameter does not match the current node `Name`, an exception is raised.

Here is an example showing how you can use `ReadElementString`. Notice that it uses `FileStreams`, so you need to ensure that you import the `System.IO` namespace (code file `XMLReaderAndWriterSample/Program.cs`):

```
public static void ReadElementContent()
{
  using (XmlReader reader = XmlReader.Create(BooksFileName))
  {
    while (!reader.EOF)
    {
      if (reader.MoveToContent() == XmlNodeType.Element &&
        reader.Name == "title")
      {
        Console.WriteLine(reader.ReadElementContentAsString());
      }
      else
      {
        // move on
        reader.Read();
      }
    }
  }
}
```

In the `while` loop, the `MoveToContent` method is used to find each node of type `XmlNodeType.Element` with the name `title`. The `EOF` property of the `XmlTextReader` checks the end of the loop condition. If the node is not of type `Element` or not named `title`, the `else` clause issues a `Read` method to move to the next node. When a node is found that matches the criteria, the result is written to the console. This should leave just the book titles written to the console. Note that you don't have to issue a `Read` call after a successful `ReadElementString` because `ReadElementString` consumes the entire `Element` and positions you on the next node.

If you remove && `rdr.Name=="title"` from the `if` clause, you have to catch the `XmlException` when it is thrown. Looking at the XML data file, the first element that `MoveToContent` finds is the `<book-store>` element. Because it is an element, it passes the check in the `if` statement. However, because it does not contain a simple text type, it causes `ReadElementString` to raise an `XmlException`. One way to work around this is to catch the exception and invoke the `Read` method in the exception handler (code file `XmlReaderAndWriterSample/Program.cs`):

```
public static void ReadElementContent2()
{
  using (XmlReader reader = XmlReader.Create(BooksFileName))
  {
    while (!reader.EOF)
    {
      if (reader.MoveToContent() == XmlNodeType.Element)
      {
        try
        {
          Console.WriteLine(reader.ReadElementContentAsString());
        }
        catch (XmlException ex)
        {
          reader.Read();
        }
      }
      else
      {
        // move on
        reader.Read();
      }
    }
  }
}
```

After running this example, the results should be the same as before. The `XmlReader` can also read strongly typed data. There are several `ReadElementContentAs` methods, such as `ReadElementContentAsDouble`, `ReadElementContentAsBoolean`, and so on. The following example shows how to read in the values as a decimal and do some math on the value. In this case, the value from the price element is increased by 25 percent (code file `XmlReaderAndWriterSample/Program.cs`):

```
public static void ReadDecimal()
{
  using (XmlReader reader = XmlReader.Create(BooksFileName))
  {
    while (reader.Read())
    {
      if (reader.NodeType == XmlNodeType.Element)
      {
        if (reader.Name == "price")
        {
          decimal price = reader.ReadElementContentAsDecimal();
          Console.WriteLine($"Current Price = {price}");
          price += price * .25m;
          Console.WriteLine($"New price {price}");
        }
        else if (reader.Name == "title")
        {
          Console.WriteLine(reader.ReadElementContentAsString());
        }
      }
    }
  }
}
```

Retrieving Attribute Data

As you play with the sample code, you might notice that when the nodes are read in, you don't see any attributes. This is because attributes are not considered part of a document's structure. When you are on an element node, you can check for the existence of attributes and optionally retrieve the attribute values.

For example, the `HasAttributes` property returns `true` if there are any attributes; otherwise, it returns `false`. The `AttributeCount` property tells you how many attributes there are, and the `GetAttribute` method gets an attribute by name or by index. If you want to iterate through the attributes one at a time, you can use the `MoveToFirstAttribute` and `MoveToNextAttribute` methods.

The following example iterates through the attributes of the `books.xml` document (code file `XmlReaderAndWriterSample/Program.cs`):

```
public static void ReadAttributes()
{
  using (XmlReader reader = XmlReader.Create(BooksFileName))
  {
    while (reader.Read())
    {
      if (reader.NodeType == XmlNodeType.Element)
      {
        for (int i = 0; i < reader.AttributeCount; i++)
        {
          Console.WriteLine(reader.GetAttribute(i));
        }
      }
    }
  }
}
```

This time you are looking for element nodes. When you find one, you loop through all the attributes and, using the `GetAttribute` method, load the value of the attribute into the list box. In the preceding example, those attributes would be `genre`, `publicationdate`, and `ISBN`.

Using the XmlWriter Class

The `XmlWriter` class enables you to write XML to a stream, a file, a `StringBuilder`, a `TextWriter`, or another `XmlWriter` object. Like `XmlTextReader`, it does so in a forward-only, noncached manner. `XmlWriter` is configurable, enabling you to specify such things as whether to indent content, the amount to indent, what quote character to use in attribute values, and whether namespaces are supported. This configuration is done using an `XmlWriterSettings` object.

Here's a simple example that shows how you can use the `XmlTextWriter` class (code file `XmlReaderAndWriterSample/Program.cs`):

```
public static void WriterSample()
{
  var settings = new XmlWriterSettings
  {
    Indent = true,
    NewLineOnAttributes = true,
    Encoding = Encoding.UTF8,
    WriteEndDocumentOnClose = true
  }

  StreamWriter stream = File.CreateText(NewBooksFileName);
  using (XmlWriter writer = XmlWriter.Create(stream, settings))
  {
    writer.WriteStartDocument();
    //Start creating elements and attributes
    writer.WriteStartElement("book");
    writer.WriteAttributeString("genre", "Mystery");
    writer.WriteAttributeString("publicationdate", "2001");
    writer.WriteAttributeString("ISBN", "123456789");
    writer.WriteElementString("title", "Case of the Missing Cookie");
    writer.WriteStartElement("author");
    writer.WriteElementString("name", "Cookie Monster");
    writer.WriteEndElement();
    writer.WriteElementString("price", "9.99");
    writer.WriteEndElement();
    writer.WriteEndDocument();
  }
}
```

Here, you are writing to a new XML file called `newbook.xml`, adding the data for a new book. Note that `XmlWriter` overwrites an existing file with a new one. (Later in this chapter you read about inserting a new element or node into an existing document.) You are instantiating the `XmlWriter` object by using the `Create` static method. In this example, a string representing a filename is passed as a parameter, along with an instance of an `XmlWriterSettings` class.

The `XmlWriterSettings` class has properties that control how the XML is generated. The `CheckedCharacters` property is a Boolean that raises an exception if a character in the XML does not conform to the W3C XML 1.0 recommendation. The `Encoding` class sets the encoding used for the XML being generated; the default is Encoding.UTF8. The `Indent` property is a Boolean value that determines whether elements should be indented. The `IndentChars` property is set to the character string that it is used to indent. The default is two spaces. The `NewLine` property is used to determine the characters for line breaks. In the preceding example, the `NewLineOnAttribute` is set to `true`. This puts each attribute in a separate line, which can make the generated XML a little easier to read.

`WriteStartDocument` adds the document declaration. Now you start writing data. First is the `book` element; next, you add the `genre`, `publicationdate`, and `ISBN` attributes. Then you write the `title`, `author`, and `price` elements. Note that the `author` element has a child element name.

When you click the button, you produce the `booknew.xml` file, which looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<book
  genre="Mystery"
  publicationdate="2001"
  ISBN="123456789">
  <title>Case of the Missing Cookie</title>
  <author>
    <name>Cookie Monster</name>
  </author>
  <price>9.99</price>
</book>
```

The nesting of elements is controlled by paying attention to when you start and finish writing elements and attributes. You can see this when you add the `name` child element to the `authors` element. Note how the `WriteStartElement` and `WriteEndElement` method calls are arranged and how that arrangement produces the nested elements in the output file.

Along with the `WriteElementString` and `WriteAttributeString` methods, there are several other specialized write methods. `WriteComment` writes out a comment in proper XML format. `WriteChars` writes out the contents of a `char` buffer. `WriteChars` needs a buffer (an array of characters), the starting position for writing (an integer), and the number of characters to write (an integer).

Reading and writing XML using the `XmlReader`- and `XmlWriter`-based classes are flexible and simple to do. Next, you find out how the DOM is implemented in the `System.Xml` namespace through the `XmlDocument` and `XmlNode` classes.

USING THE DOM IN .NET

The DOM implementation in .NET supports the W3C DOM specifications. The DOM is implemented through the `XmlNode` class, which is an abstract class that represents a node of an XML document. Concrete classes are `XmlDocument`, `XmlDocumentFragment`, `XmlAttribute`, and `XmlNotation`. `XmlLinkedNode` is an abstract class that derives from `XmlNode`. Concrete classes that derive from `XmlLinkedNode` are `XmlDeclaration`, `XmlDocumentType`, `XmlElement`, and `XmlProcessingInstruction`.

An `XmlNodeList` class is an ordered list of nodes. This is a live list of nodes, and any changes to any node are immediately reflected in the list. `XmlNodeList` supports indexed access or iterative access.

The `XmlNode` and `XmlNodeList` classes make up the core of the DOM implementation with .NET.

The sample code using `XmlDocument` makes use of the following namespaces:

```
System

System.IO

System.Xml
```

Reading with the XmlDocument Class

`XmlDocument` is a class that represents the XML DOM in .NET. Unlike `XmlReader` and `XmlWriter`, `XmlDocument` provides read and write capabilities as well as random access to the DOM tree.

The example introduced in this section creates an `XmlDocument` object, loads a document from disk, and loads a text box with data from the title elements. This is like one of the examples that you constructed in

the section "Reading XML with XmlReader." The difference is that here you select the nodes you want to work with instead of going through the entire document as in the `XmlReader`-based example.

Here is the code to create an `XmlDocument` object. Note how simple it looks in comparison to the `XmlReader` example (code file `XmlDocumentSample/Program.cs`):

```
public static void ReadXml()
{
  using (FileStream stream = File.OpenRead(BooksFileName))
  {
    var doc = new XmlDocument();
    doc.Load(stream);
    XmlNodeList titleNodes = doc.GetElementsByTagName("title");
    foreach (XmlNode node in titleNodes)
    {
      Console.WriteLine(node.OuterXml);
    }
  }
}
```

If this is all that you wanted to do, using the `XmlReader` would have been a much more efficient way to read the file, because you just go through the document once and then you are finished with it. This is exactly the type of work that `XmlReader` was designed for. However, if you want to revisit a node, using `XmlDocument` is a better way.

Navigating Through the Hierarchy

A big advantage of the `XmlDocument` class is that you can navigate the DOM tree. The following example accesses all `author` elements and writes the outer XML to the console (this is the XML including the `author` element), the inner XML (without the `author` element), the next sibling, the previous sibling, the first child, and the parent (code file `XmlDocumentSample/Program.cs`):

```
public static void NavigateXml()
{
  using (FileStream stream = File.OpenRead(BooksFileName))
  {
    var doc = new XmlDocument();
    doc.Load(stream);
    XmlNodeList authorNodes = doc.GetElementsByTagName("author");
    foreach (XmlNode node in authorNodes)
    {
      Console.WriteLine($"Outer XML: {node.OuterXml}");
      Console.WriteLine($"Inner XML: {node.InnerXml}");
      Console.WriteLine($"Next sibling outer XML: " +
        $"{node.NextSibling.OuterXml}");
      Console.WriteLine($"Previous sibling outer XML: " +
        $"{node.PreviousSibling.OuterXml}");
      Console.WriteLine($"First child outer Xml: {node.FirstChild.OuterXml}");
      Console.WriteLine($"Parent name: {node.ParentNode.Name}");
      Console.WriteLine();
    }
  }
}
```

When you run the application, you can see these values for the first element found:

```
Outer XML: <author><first-name>Benjamin</first-name>
  <last-name>Franklin</last-name></author>
Inner XML: <first-name>Benjamin</first-name><last-name>Franklin</last-name>
Next sibling outer XML: <price>8.99</price>
Previous sibling outer XML:
```

```
  <title>The Autobiography of Benjamin Franklin</title>
First child outer Xml: <first-name>Benjamin</first-name>
Parent name: book
```

Inserting Nodes with XmlDocument

Earlier, you looked at an example that used the `XmlWriter` class that created a new document. The limitation was that it would not insert a node into a current document. With the `XmlDocument` class, you can do just that.

The following code sample creates the element book using `CreateElement`, adds some attributes, adds some child elements, and after creating the complete book element adds it to the root element of the XML document (code file `XmlDocumentSample/Program.cs`):

```
public static void CreateXml()
{
  var doc = new XmlDocument();
  using (FileStream stream = File.OpenRead("books.xml"))
  {
    doc.Load(stream);
  }

  //create a new 'book' element
  XmlElement newBook = doc.CreateElement("book");

  //set some attributes
  newBook.SetAttribute("genre", "Mystery");
  newBook.SetAttribute("publicationdate", "2001");
  newBook.SetAttribute("ISBN", "123456789");

  //create a new 'title' element
  XmlElement newTitle = doc.CreateElement("title");
  newTitle.InnerText = "Case of the Missing Cookie";
  newBook.AppendChild(newTitle);

  //create new author element
  XmlElement newAuthor = doc.CreateElement("author");
  newBook.AppendChild(newAuthor);

  //create new name element
  XmlElement newName = doc.CreateElement("name");
  newName.InnerText = "Cookie Monster";
  newAuthor.AppendChild(newName);

  //create new price element
  XmlElement newPrice = doc.CreateElement("price");
  newPrice.InnerText = "9.95";
  newBook.AppendChild(newPrice);

  //add to the current document
  doc.DocumentElement.AppendChild(newBook);
  var settings = new XmlWriterSettings
  {
    Indent = true,
    IndentChars = "\t",
    NewLineChars = Environment.NewLine
  };

  //write out the doc to disk
  using (StreamWriter streamWriter = File.CreateText(NewBooksFileName))
  using (XmlWriter writer = XmlWriter.Create(streamWriter, settings))
```

```
    {
      doc.WriteContentTo(writer);
    }

    XmlNodeList nodeLst = doc.GetElementsByTagName("title");
    foreach (XmlNode node in nodeLst)
    {
      Console.WriteLine(node.OuterXml);
    }
  }
```

When you run the application, the following book element is added to the bookstore and written to the file `newbooks.xml`:

```
<book genre="Mystery" publicationdate="2001" ISBN="123456789">
  <title>Case of the Missing Cookie</title>
  <author>
    <name>Cookie Monster</name>
  </author>
  <price>9.95</price>
</book>
```

After creating the file, the application writes all title nodes to the console. You can see that the added element is now included:

```
<title>The Autobiography of Benjamin Franklin</title>
<title>The Confidence Man</title>
<title>The Gorgias</title>
<title>Case of the Missing Cookie</title>
```

You should use the `XmlDocument` class when you want to have random access to the document. Use the `XmlReader`-based classes when you want a streaming-type model instead. Remember that there is a cost for the flexibility of the `XmlNode`-based `XmlDocument` class: Memory requirements are higher and the performance of reading the document is not as good as when using `XmlReader`. There is another way to traverse an XML document: the `XPathNavigator`.

USING XPATHNAVIGATOR

An `XPathNavigator` can be used to select, iterate, and find data from an XML document using the XPath syntax. An `XPathNavigator` can be created from an `XPathDocument`. The `XPathDocument` cannot be changed; it is designed for performance and read-only use. Unlike the `XmlReader`, the `XPathNavigator` is not a streaming model, so the document is read and parsed only once. Like `XmlDocument`, it requires the complete document loaded in memory.

The `System.Xml.XPath` namespace defined in the NuGet package `System.Xml.XPath` is built for speed. It provides a read-only view of your XML documents, so there are no editing capabilities. Classes in this namespace are built for fast iteration and selections on the XML document in a cursory fashion.

The following table lists the key classes in `System.Xml.XPath` and gives a short description of the purpose of each class.

CLASS NAME	DESCRIPTION
`XPathDocument`	Provides a view of the entire XML document. Read-only.
`XPathNavigator`	Provides the navigational capabilities to an `XPathDocument`.
`XPathNodeIterator`	Provides iteration capabilities to a node set.
`XPathExpression`	Represents a compiled XPath expression. Used by `SelectNodes`, `SelectSingleNodes`, `Evaluate`, and `Matches`.

The sample code makes use of the following namespaces:

```
System

System.IO

System.Xml

System.Xml.XPath
```

XPathDocument

`XPathDocument` does not offer any of the functionality of the `XmlDocument` class. Its sole purpose is to create `XPathNavigator`s. In fact, that is the only method available on the `XPathDocument` class (other than those provided by `Object`).

You can create an `XPathDocument` in several different ways. You can pass in an `XmlReader`, or a `Stream`-based object to the constructor. This provides a great deal of flexibility.

XPathNavigator

`XPathNavigator` contains methods for moving and selecting elements. Move methods set the current position of the iterator to the element that should be moved to. You can move to specific attributes of an element: the `MoveToFirstAttribute` method moves to the first attribute, the `MoveToNextAttribute` method to the next one. `MoveToAttribute` allows specifying a specific attribute name. You can move to sibling nodes with `MoveToFirst`, `MoveToNext`, `MoveToPrevious`, and `MoveToLast`. It's also possible to move to child elements (`MoveToChild`, `MoveToFirstChild`), to parent elements (`MoveToParent`), and directly to the root element (`MoveToRoot`).

You can select methods using XPath expressions using the `Select` method. To filter the selection based on specific nodes in the tree and the current position, other methods exist. `SelectAncestor` only filters ancestor nodes, and `SelectDescendants` filters all descendants. Only the direct children are filtered with `SelectChildren`. `SelectSingleNode` accepts an XPath expression and returns a single matching node.

The `XPathNavigator` also allows changing the XML tree using one of the Insert methods if the `CanEdit` property returns `true`. When you use the `XmlDocument` class to create an `XPathNavigator`, the `CanEdit` property of the navigator returns `true` and thus allows changes using the `Insert` methods.

XPathNodeIterator

The `XPathDocument` represents the complete XML document, the `XPathNavigator` enables you to select nodes and move the cursor within the document to specific nodes, and the `XPathNodeIterator` enables you to iterate over a set of nodes.

The `XPathNodeIterator` is returned by the `XPathNavigator Select` methods. You use it to iterate over the set of nodes returned by a `Select` method of the `XPathNavigator`. Using the `MoveNext` method of the `XPathNodeIterator` does not change the location of the `XPathNavigator` that created it. However, you can get a new `XPathNavigator` using the `Current` property of an `XPathNodeIterator`. The `Current` property returns an `XPathNavigator` that is set to the current position.

Navigating Through XML Using XPath

The best way to see how these classes are used is to look at some code that iterates through the `books.xml` document. This enables you to see how the navigation works.

The first example iterates all books that define the genre novel. First, an `XPathDocument` object is created that receives the XML filename in the constructor. This object, which holds read-only content of the XML file, offers the `CreateNavigator` method to create an `XPathNavigator`. When you use this

navigator, an XPath expression can be passed to the `Select` method. When you use XPath, you can access element trees using / between hierarchies. `/bookstore/book` retrieves all `book` nodes within the `bookstore` element. `@genre` is a shorthand notation to access the attribute genre. The `Select` method returns an `XPathNodeIterator` that enables you to iterate all nodes that match the expression. The first `while` loop iterates all book elements that match calling the `MoveNext` method. With each iteration, another select method is invoked on the current `XPathNavigator`—`SelectDescendants`. `SelectDescendants` returns all descendants, which means the child nodes, and the children of the child nodes, and the children of those children through the complete hierarchy. With the `SelectDescendants` method, the overload is taken to match only element nodes and to exclude the book element itself. The second `while` loop iterates this collection and writes the name and value to the console (code file `XPathNavigatorSample/Program.cs`):

```
public static void SimpleNavigate()
{
  //modify to match your path structure
  var doc = new XPathDocument(BooksFileName);

  //create the XPath navigator
  XPathNavigator nav = doc.CreateNavigator();

  //create the XPathNodeIterator of book nodes
  // that have genre attribute value of novel
  XPathNodeIterator iterator = nav.Select("/bookstore/book[@genre='novel']");
  while (iterator.MoveNext())
  {
    XPathNodeIterator newIterator = iterator.Current.SelectDescendants(
    XPathNodeType.Element, matchSelf: false);
    while (newIterator.MoveNext())
    {
      Console.WriteLine($"{newIterator.Current.Name}: " +
        $"{newIterator.Current.Value}");
    }
  }
}
```

When you run the application, you can see the content of the only book that matches the novel genre with all its children as you can see with the `first-name` and `last-name` elements that are contained within author:

```
title: The Confidence Man
author: HermanMelville
  first-name: Herman
  last-name: Melville
price: 11.99
```

Using XPath Evaluations

XPath not only allows fast access to XML nodes within a tree, it also defines some functions—for example, `ceiling`, `floor`, `number`, `round`, and `sum`—for numbers. The following sample is somewhat like the previous one; it accesses all book elements instead of only the one matching the novel genre. Iterating the book elements, just the title child element is accessed by moving the current position to the first child title node. From the title node, the name and value are written to the console. The very special piece of code is defined with the last statement. The XPath `sum` function is invoked on the value of `/bookstore/book/price` elements. Such functions can be evaluated by calling the `Evaluate` method on the `XPathNavigator` (code file `XPathNavigatorSample/Program.cs`):

```
public static void UseEvaluate()
{
  //modify to match your path structure
  var doc = new XPathDocument(BooksFileName);
```

```
//create the XPath navigator
XPathNavigator nav = doc.CreateNavigator();

//create the XPathNodeIterator of book nodes
XPathNodeIterator iterator = nav.Select("/bookstore/book");
while (iterator.MoveNext())
{
  if (iterator.Current.MoveToChild("title", string.Empty))
  {
    Console.WriteLine($"{iterator.Current.Name}: {iterator.Current.Value}");
  }
}
Console.WriteLine("========================");
Console.WriteLine($"Total Cost = " +
  $"{nav.Evaluate("sum(/bookstore/book/price)")}");
```

When you run the application, you can see all book titles and the summary price:

```
title: The Autobiography of Benjamin Franklin
title: The Confidence Man
title: The Gorgias
========================
Total Cost = 30.97
```

Changing XML Using XPath

Next, make some changes using XPath. To create a changeable `XPathNavigator`, the `XmlDocument` class is used. The `CanEdit` property of the `XPathNavigator` returns true when you create it via `XmlDocument`, and thus the `InsertAfter` method can be invoked. Using `InsertAfter`, a discount is added as sibling after the price element. The newly created XML document is accessed using the `OuterXml` property of the navigator, and a new XML file is saved (code file `XPathNavigatorSample/Program.cs`):

```
public static void Insert()
{
  var doc = new XmlDocument();
  doc.Load(BooksFileName);

  XPathNavigator navigator = doc.CreateNavigator();
  if (navigator.CanEdit)
  {
    XPathNodeIterator iter = navigator.Select("/bookstore/book/price");
    while (iter.MoveNext())
    {
      iter.Current.InsertAfter("<disc>5</disc>");
    }
  }

  using (var stream = File.CreateText(NewBooksFileName))
  {
    var outDoc = new XmlDocument();
    outDoc.LoadXml(navigator.OuterXml);
    outDoc.Save(stream);
  }
}
```

The newly generated XML contains the `disc` elements:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- This file represents a fragment of a book store inventory database -->
<bookstore>
  <book genre="autobiography" publicationdate="1991" 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>
    <disc>5</disc>
  </book>
  <book genre="novel" publicationdate="1967" ISBN="0-201-63361-2">
    <title>The Confidence Man</title>
    <author>
      <first-name>Herman</first-name>
      <last-name>Melville</last-name>
    </author>
    <price>11.99</price>
    <disc>5</disc>
  </book>
  <book genre="philosophy" publicationdate="1991" ISBN="1-861001-57-6">
    <title>The Gorgias</title>
    <author>
      <name>Plato</name>
    </author>
    <price>9.99</price>
    <disc>5</disc>
  </book>
</bookstore>
```

SERIALIZING OBJECTS IN XML

Serializing is the process of persisting an object to disk. Another part of your application, or even a separate application, can deserialize the object, and it will be in the same state it was in prior to serialization. The .NET Framework includes a couple of ways to do this.

This section looks at the `System.Xml.Serialization` namespace with the NuGet package `System.Xml.XmlSerializer`, which contains classes used to serialize objects into XML documents or streams. This means that an object's public properties and public fields are converted into XML elements, attributes, or both.

The most important class in the `System.Xml.Serialization` namespace is `XmlSerializer`. To serialize an object, you first need to instantiate an `XmlSerializer` object, specifying the type of the object to serialize. Then you need to instantiate a stream/writer object to write the file to a stream/document. The final step is to call the `Serialize` method on the `XMLSerializer`, passing it the stream/writer object and the object to serialize.

Data that can be serialized can be primitive types, fields, arrays, and embedded XML in the form of `XmlElement` and `XmlAttribute` objects. To deserialize an object from an XML document, you reverse the process in the previous example. You create a stream/reader and an `XmlSerializer` object and then pass the stream/reader to the `Deserialize` method. This method returns the deserialized object, although it needs to be cast to the correct type.

> **NOTE** *The XML serializer cannot convert private data—only public data—and it cannot serialize cyclic object graphs. However, these are not serious limitations; by carefully designing your classes, you should be able to easily avoid these issues. If you do need to be able to serialize public and private data as well as an object graph containing many nested objects, you can use the runtime or the data contract serialization mechanisms.*

The sample code makes use of the following namespaces:

```
System

System.IO

System.Xml

System.Xml.Serialization
```

Serializing a Simple Object

Let's start serializing a simple object. The class `Product` has XML attributes from the namespace `System.Xml.Serialization` applied to specify whether a property should be serialized as XML element or attribute. The `XmlElement` attribute specifies the property to serialize as element; the `XmlAttribute` attribute specifies to serialize as attribute. The `XmlRoot` attribute specifies the class to be serialized as the root element (code file `ObjectToXmlSerializationSample/Product.cs`):

```
[XmlRoot]
public class Product
{
  [XmlAttribute(AttributeName = "Discount")]
  public int Discount { get; set; }

  [XmlElement]
  public int ProductID { get; set; }

  [XmlElement]
  public string ProductName { get; set; }

  [XmlElement]
  public int SupplierID { get; set; }

  [XmlElement]
  public int CategoryID { get; set; }

  [XmlElement]
  public string QuantityPerUnit { get; set; }

  [XmlElement]
  public Decimal UnitPrice { get; set; }

  [XmlElement]
  public short UnitsInStock { get; set; }

  [XmlElement]
  public short UnitsOnOrder { get; set; }

  [XmlElement]
  public short ReorderLevel { get; set; }

  [XmlElement]
  public bool Discontinued { get; set; }

  public override string ToString() =>
    $"{ProductID} {ProductName} {UnitPrice:C}";
}
```

With these attributes, you can influence the name, namespace, and type to be generated by using properties of the attribute types.

The following code sample creates an instance of the `Product` class, fills its properties, and serializes it to a file. Creating the `XmlSerializer` requires the type of the class to be serialized to be passed with the constructor. The `Serialize` method is overloaded to accept a `Stream`, `TextWriter`, and `XmlWriter`, and the object to be serialized (code file `ObjectToXmlSerializationSample/Program.cs`):

```
public static void SerializeProduct()
{
  var product = new Product
  {
    ProductID = 200,
    CategoryID = 100,
    Discontinued = false,
    ProductName = "Serialize Objects",
    QuantityPerUnit = "6",
    ReorderLevel = 1,
    SupplierID = 1,
    UnitPrice = 1000,
    UnitsInStock = 10,
    UnitsOnOrder = 0
  };

  FileStream stream = File.OpenWrite(ProductFileName);
  using (TextWriter writer = new StreamWriter(stream))
  {
    XmlSerializer serializer = new XmlSerializer(typeof(Product));
    serializer.Serialize(writer, product);
  }
}
```

The generated XML file lists the Product element with the Discount attribute and the other properties stored as elements:

```
<?xml version="1.0" encoding="utf-8"?>
<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" Discount="0">
  <ProductID>200</ProductID>
  <ProductName>Serialize Objects</ProductName>
  <SupplierID>1</SupplierID>
  <CategoryID>100</CategoryID>
  <QuantityPerUnit>6</QuantityPerUnit>
  <UnitPrice>1000</UnitPrice>
  <UnitsInStock>10</UnitsInStock>
  <UnitsOnOrder>0</UnitsOnOrder>
  <ReorderLevel>1</ReorderLevel>
  <Discontinued>false</Discontinued>
</Product>
```

There is nothing out of the ordinary here. You could use this XML file in any way that you would use an XML document—transform it and display it as HTML, load an `XmlDocument` with it, or, as shown in the example, deserialize it and create an object in the same state that it was in prior to serializing it (which is exactly what you're doing in the next step.

Creating a new object from the file is done by creating an `XmlSerializer` and invoking the `Deserialize` method (code file `ObjectToXmlSerializationSample/Program.cs`):

```
public static void DeserializeProduct()
{
  Product product;
  using (var stream = new FileStream(ProductFileName, FileMode.Open))
  {
    var serializer = new XmlSerializer(typeof(Product));
    product = serializer.Deserialize(stream) as Product;
  }
  Console.WriteLine(product);
}
```

When you run the application, the console shows the product ID, product name, and unit price.

> **NOTE** *To ignore properties from the XML serialization, you can use the* `XmlIgnore` *attribute.*

Serializing a Tree of Objects

What about situations in which you have derived classes and possibly properties that return an array? `XmlSerializer` has that covered as well. The next example is just slightly more complex so that it can deal with these issues.

In addition to the `Product` class, the `BookProduct` (derived from `Product`) and `Inventory` classes are created. The `Inventory` class contains both of the other classes.

The `BookProduct` class derives from Product and adds the `ISBN` property. This property is stored with the XML attribute `Isbn` as defined by the .NET attribute `XmlAttribute` (code file `ObjectToXmlSerializationSample/BookProduct.cs`):

```
public class BookProduct : Product
{
  [XmlAttribute("Isbn")]
  public string ISBN { get; set; }
}
```

The `Inventory` class contains an array of inventory items. An inventory item can be a `Product` or a `BookProduct`. The serializer needs to know all the derived classes that are stored within the array; otherwise it can't deserialize them. The items of the array are defined using the `XmlArrayItem` attribute (code file `ObjectToXmlSerializationSample/Inventory.cs`):

```
public class Inventory
{
  [XmlArrayItem("Product", typeof(Product)),
    XmlArrayItem("Book", typeof(BookProduct))]
  public Product[] InventoryItems { get; set; }

  public override string ToString()
  {
    var outText = new StringBuilder();
    foreach (Product prod in InventoryItems)
    {
      outText.AppendLine(prod.ProductName);
    }
    return outText.ToString();
  }
}
```

In the `SerializeInventory` method after an `Inventory` object is created that is filled with a `Product` and a `BookProduct`, the inventory is serialized (code file `ObjectToXmlSerializationSample/Program.cs`):

```
public static void SerializeInventory()
{
  var product = new Product
  {
    ProductID = 100,
    ProductName = "Product Thing",
    SupplierID = 10
  };

  var book = new BookProduct
```

```
    {
      ProductID = 101,
      ProductName = "How To Use Your New Product Thing",
      SupplierID = 10,
      ISBN = "1234567890"
    };

    Product[] items = { product, book };
    var inventory = new Inventory
    {
      InventoryItems = items
    };

    using (FileStream stream = File.Create(InventoryFileName))
    {
      var serializer = new XmlSerializer(typeof(Inventory));
      serializer.Serialize(stream, inventory);
    }
  }
```

The generated XML file defines an `Inventory` root element and the `Product` and `Book` child elements. The `BookProduct` type is represented as `Book` element because the `XmlItemArray` attribute defined the `Book` name for the `BookProduct` type:

```
<?xml version="1.0"?>
<Inventory xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <InventoryItems>
    <Product Discount="0">
      <ProductID>100</ProductID>
      <ProductName>Product Thing</ProductName>
      <SupplierID>10</SupplierID>
      <CategoryID>0</CategoryID>
      <UnitPrice>0</UnitPrice>
      <UnitsInStock>0</UnitsInStock>
      <UnitsOnOrder>0</UnitsOnOrder>
      <ReorderLevel>0</ReorderLevel>
      <Discontinued>false</Discontinued>
    </Product>
    <Book Discount="0" Isbn="1234567890">
      <ProductID>101</ProductID>
      <ProductName>How To Use Your New Product Thing</ProductName>
      <SupplierID>10</SupplierID>
      <CategoryID>0</CategoryID>
      <UnitPrice>0</UnitPrice>
      <UnitsInStock>0</UnitsInStock>
      <UnitsOnOrder>0</UnitsOnOrder>
      <ReorderLevel>0</ReorderLevel>
      <Discontinued>false</Discontinued>
    </Book>
  </InventoryItems>
</Inventory>
```

To deserialize the objects, you need to invoke the `Deserialize` method of the `XmlSerializer`: (code file `ObjectToXmlSerializationSample/Program.cs`):

```
public static void DeserializeInventory()
{
  using (FileStream stream = File.OpenRead(InventoryFileName))
  {
    var serializer = new XmlSerializer(typeof(Inventory));
```

```
      Inventory newInventory = serializer.Deserialize(stream) as Inventory;
      foreach (Product prod in newInventory.InventoryItems)
      {
        Console.WriteLine(prod.ProductName);
      }
    }
}
```

Serializing Without Attributes

Well, this all works great, but what if you don't have access to the source code for the types that are being serialized? You can't add the attribute if you don't have the source. There is another way: You can use the `XmlAttributes` class and the `XmlAttributeOverrides` class. Together these classes enable you to accomplish the same thing as the previous sample but without adding the attributes. This section demonstrates how this works.

For this example, the `Inventory`, `Product`, and derived `BookProduct` classes could also be in a separate library. As the serialization is independent of that, and to make the sample structure easier, these classes are in the same project as in the previous examples, but note that now there are no attributes added to the `Inventory` class (code file `ObjectToXmlSerializationWOAttributes/Inventory.cs`):

```
public class Inventory
{
  public Product[] InventoryItems { get; set; }
  public override string ToString()
  {
    var outText = new StringBuilder();
    foreach (Product prod in InventoryItems)
    {
      outText.AppendLine(prod.ProductName);
    }
    return outText.ToString();
  }
}
```

The attributes from the `Product` and `BookProduct` classes are removed as well.

The implementation to do the serialization is like what was done before; the difference is that you use a different overload on creating the `XmlSerializer`. This overload accepts `XmlAttributeOverrides`. These overrides are coming from the helper method `GetInventoryXmlAttributes` (code file `ObjectToXmlSerializationWOAttributes/Program.cs`):

```
public static void SerializeInventory()
{
  var product = new Product
  {
    ProductID = 100,
    ProductName = "Product Thing",
    SupplierID = 10
  };

  var book = new BookProduct
  {
    ProductID = 101,
    ProductName = "How To Use Your New Product Thing",
    SupplierID = 10,
    ISBN = "1234567890"
  };
```

```
  Product[] products = { product, book };
  var inventory = new Inventory
  {
    InventoryItems = products
  };

  using (FileStream stream = File.Create(InventoryFileName))
  {
    var serializer = new XmlSerializer(typeof(Inventory),
      GetInventoryXmlAttributes());
    serializer.Serialize(stream, inventory);
  }
}
```

The helper method `GetInventoryXmlAttributes` returns the needed `XmlAttributeOverrides`. Previously, the `Inventory` class had the `XmlArrayItem` attributes applied. They are now done creating `XmlAttributes` and adding `XmlArrayItemAttributes` to the `XmlArrayItems` collection. Another change is that the `Product` and `BookProduct` classes had an `XmlAttribute` applied to the `Discount` and `ISBN` properties. To define the same behavior without applying the attributes to the properties directly, `XmlAttributeAttribute` objects are created and assigned to the `XmlAttribute` property of `XmlAttributes` objects. All these created `XmlAttributes` are then added to the `XmlAttributeOverrides` that contains a collection of `XmlAttributes`. When you invoke the `Add` method of `XmlAttributeOverrides`, you need the type where the attribute should be applied, the name of the property, and the corresponding `XmlAttributes` (code file `ObjectToXmlSerializationWOAttributes/Program.cs`):

```
private static XmlAttributeOverrides GetInventoryXmlAttributes()
{
  var inventoryAttributes = new XmlAttributes();
  inventoryAttributes.XmlArrayItems.Add(new XmlArrayItemAttribute("Book",
    typeof(BookProduct)));
  inventoryAttributes.XmlArrayItems.Add(new XmlArrayItemAttribute("Product",
    typeof(Product)));
  var bookIsbnAttributes = new XmlAttributes();
  bookIsbnAttributes.XmlAttribute = new XmlAttributeAttribute("Isbn");
  var productDiscountAttributes = new XmlAttributes();
  productDiscountAttributes.XmlAttribute =
    new XmlAttributeAttribute("Discount");
  var overrides = new XmlAttributeOverrides();
  overrides.Add(typeof(Inventory), "InventoryItems", inventoryAttributes);
  overrides.Add(typeof(BookProduct), "ISBN", bookIsbnAttributes);
  overrides.Add(typeof(Product), "Discount", productDiscountAttributes);
  return overrides;
}
```

When you run the application, the same XML content is created as before:

```
<?xml version="1.0"?>
<Inventory xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <InventoryItems>
    <Product Discount="0">
      <ProductID>100</ProductID>
      <ProductName>Product Thing</ProductName>
      <SupplierID>10</SupplierID>
      <CategoryID>0</CategoryID>
      <UnitPrice>0</UnitPrice>
      <UnitsInStock>0</UnitsInStock>
      <UnitsOnOrder>0</UnitsOnOrder>
      <ReorderLevel>0</ReorderLevel>
      <Discontinued>false</Discontinued>
    </Product>
```

```
    <Book Discount="0" Isbn="1234567890">
      <ProductID>101</ProductID>
      <ProductName>How To Use Your New Product Thing</ProductName>
      <SupplierID>10</SupplierID>
      <CategoryID>0</CategoryID>
      <UnitPrice>0</UnitPrice>
      <UnitsInStock>0</UnitsInStock>
      <UnitsOnOrder>0</UnitsOnOrder>
      <ReorderLevel>0</ReorderLevel>
      <Discontinued>false</Discontinued>
    </Book>
  </InventoryItems>
</Inventory>
```

> **NOTE** *.NET attribute types typically end with the name* `Attribute`*. This postfix can be ignored when applying the attribute using brackets. The compiler automatically adds the postfix if it is missing. A class that can be used as an attribute derives from the base class* `Attribute`*—directly or indirectly. When you apply the attribute* `XmlElement` *using brackets, the compiler instantiates the type* `XmlElementAttribute`*. This naming becomes especially noticeable when applying the attribute* `XmlAttribute` *using brackets. Behind the scenes, the class* `XmlAttributeAttribute` *is used. How does the compiler differentiate this with the class* `XmlAttribute`*? The class* `XmlAttribute` *is used to read XML attributes from the DOM tree, but it is not a .NET attribute, as it does not derive from the base class* `Attribute`*. You can read more information about attributes in Chapter 16, "Reflection, Metadata, and Dynamic Programming."*

With the deserialization code, the same attribute overrides are needed (code file `ObjectToXmlSerializationWOAttributes/Program.cs`):

```
public static void DeserializeInventory()
{
  using (FileStream stream = File.OpenRead(InventoryFileName))
  {
    XmlSerializer serializer = new XmlSerializer(typeof(Inventory),
      GetInventoryXmlAttributes());
    Inventory newInventory = serializer.Deserialize(stream) as Inventory;

    foreach (Product prod in newInventory.InventoryItems)
    {
      Console.WriteLine(prod.ProductName);
    }
  }
}
```

The `System.Xml.XmlSerialization` namespace provides a very powerful toolset for serializing objects to XML. By serializing and deserializing objects to XML instead of to binary format, you have the option to do something else with this XML, which greatly adds to the flexibility of your designs.

LINQ TO XML

Aren't there already enough options available dealing with XML? Beware, with LINQ to XML another option is available. LINQ to XML allows querying XML code that's like querying object lists and the database. LINQ to Objects are covered in Chapter 12, "Language Integrated Query," and LINQ to Entities are covered in Chapter 26, "Entity Framework Core." Although the DOM tree offered by the `XmlDocument` and

XPath queries offered by the `XPathNavigator` implement a standards-based approach to query XML data, LINQ to XML offers the simple .NET variant for query—a variant that is like querying other data stores. In addition to the methods offered by LINQ to Objects, LINQ to XML adds some XML specifics to this query in the `System.Xml.Linq` namespace. LINQ to XML also offers easier creating of XML content than the standards-based `XmlDocument` XML creation.

The following sections describe the objects that are available with LINQ to XML.

The sample code makes use of the following dependencies and namespaces:

Dependencies

```
NETStandard.Library

System.Xml.XDocument
```

Namespaces

```
System

System.Collections.Generic

System.Linq

System.Xml.Linq

static System.Console
```

XDocument

The `XDocument` represents an XML document like the `XmlDocument` class, but it is easier to work with. The `XDocument` object works with the other new objects in this space, such as the `XNamespace`, `XComment`, `XElement`, and `XAttribute` objects.

One of the more important members of the `XDocument` object is the `Load` method. Here it loads the file `hamlet.xml` that is defined by the constant `HamletFileName` into memory:

```
XDocument doc = XDocument.Load(HamletFileName);
```

You can also pass a `TextReader` or `XmlReader` object into the `Load` method. From here, you can programmatically work with the XML code as shown in the following code snippet to access the name of the root element and check whether the root element has attributes (code file `LinqToXmlSample/Program.cs`):

```
XDocument doc = XDocument.Load(HamletFileName);
Console.WriteLine($"root name: {doc.Root.Name}");
Console.WriteLine($"has root attributes? {doc.Root.HasAttributes}");
```

This produces the following results:

```
root name: PLAY
has root attributes? False
```

Another important member to be aware of is the `Save` method, which, like the `Load` method, enables you to save to a physical disk location or to a `TextWriter` or `XmlWriter` object:

```
XDocument doc = XDocument.Load(HamletFileName);
doc.Save(SaveFileName);
```

XElement

One object that you will work with frequently is the `XElement` object. With `XElement` objects, you can easily create single-element objects that are XML documents themselves, as well as fragments of XML. You

can use the `Load` method with the `XElement` similarly to how you use the `Load` method with the `XDocument`. The following code snippet shows writing an XML element with its corresponding value to the console:

```
var company = new XElement("Company", "Microsoft Corporation");
Console.WriteLine(company);
```

In the creation of an `XElement` object, you can define the name of the element as well as the value used in the element. In this case, the name of the element is `<Company>`, and the value of the `<Company>` element is `Microsoft Corporation`. Running this in a console application produces the following result:

```
<Company>Microsoft Corporation</Company>
```

You can create an even more complete XML document using multiple `XElement` objects, as shown in the following example (code file `LinqToXmlSample/Program.cs`):

```
public static void CreateXml()
{
  var company =
    new XElement("Company",
      new XElement("CompanyName", "Microsoft Corporation"),
      new XElement("CompanyAddress",
        new XElement("Address", "One Microsoft Way"),
        new XElement("City", "Redmond"),
        new XElement("Zip", "WA 98052-6399"),
        new XElement("State", "WA"),
        new XElement("Country", "USA")));
  Console.WriteLine(company);
}
```

What's extremely nice with this API is that the hierarchy of the XML is represented by the API. The first instantiation of the `XElement` passes the string `"Company"` to the first parameter. This parameter is of type `XName` that represents the name of the XML element. The second parameter is another `XElement`. This second `XElement` defines the XML child element of the `Company`. This second element defines `"CompanyName"` as `XName`, and `"Microsoft Corporation"` as its value. The `XElement` specifying the company address is another child of the Company element. All the other `XElement` objects that follow are direct child objects of `CompanyAddress`. The constructor allows passing any number of objects as defined by the type `params object[]`. All these objects are treated as children.

Running this application produces this result:

```
<Company>
  <CompanyName>Microsoft Corporation</CompanyName>
  <CompanyAddress>
    <Address>One Microsoft Way</Address>
    <City>Redmond</City>
    <Zip>WA 98052-6399</Zip>
    <State>WA</State>
    <Country>USA</Country>
  </CompanyAddress>
</Company>
```

> **NOTE** *The constructor syntax of* `XElement` *allows easy creation of hierarchical XML. This makes it easy to create XML out of LINQ queries (transforming object trees to XML), as is shown later in this section, and you can also transform one XML syntax to another XML syntax.*

XNamespace

`XNamespace` is an object that represents an XML namespace, and it is easily applied to elements within your document. For instance, you can take the previous example and easily apply a namespace to the root element by creating an `XNamespace` object (code file `LinqToXmlSample/Program.cs`):

```
public static void WithNamespace()
{
  XNamespace ns = "http://www.cninnovation.com/samples/2018";
  var company =
    new XElement(ns + "Company",
      new XElement("CompanyName", "Microsoft Corporation"),
      new XElement("CompanyAddress",
        new XElement("Address", "One Microsoft Way"),
        new XElement("City", "Redmond"),
        new XElement("Zip", "WA 98052-6399"),
        new XElement("State", "WA"),
        new XElement("Country", "USA")));
  Console.WriteLine(company);
}
```

In this case, an `XNamespace` object is created by assigning it a value of `http://www.cninnovation.com/samples/2018`. From there, it is used in the root element `<Company>` with the instantiation of the `XElement` object.

This produces the following result:

```
<Company xmlns="http://www.cninnovation.com/samples/2018">
<CompanyName xmlns="">Microsoft Corporation</CompanyName>
  <CompanyAddress xmlns="">
    <Address>One Microsoft Way</Address>
    <City>Redmond</City>
    <Zip>WA 98052-6399</Zip>
    <State>WA</State>
    <Country>USA</Country>
  </CompanyAddress>
</Company>
```

> **NOTE** *The* `XNamespace` *allows creation by assigning a string to the* `XNamespace` *instead of using the new operator because this class implements an implicit cast operator from string. It's also possible to use the + operator with the* `XNamespace` *object by having a string on the right side because of an implementation of the + operator that returns an* `XName`*. Operator overloading is explained in Chapter 6, "Operators and Casts."*

In addition to dealing with only the root element, you can also apply namespaces to all your elements, as shown in the following example (code file `LinqToXmlSample/Program.cs`):

```
public static void With2Namespace()
{
  XNamespace ns1 = "http://www.cninnovation.com/samples/2018";
  XNamespace ns2 = "http://www.cninnovation.com/samples/2018/address";
  var company =
    new XElement(ns1 + "Company",
      new XElement(ns2 + "CompanyName", "Microsoft Corporation"),
      new XElement(ns2 + "CompanyAddress",
        new XElement(ns2 + "Address", "One Microsoft Way"),
        new XElement(ns2 + "City", "Redmond"),
        new XElement(ns2 + „Zip", „WA 98052-6399"),
```

```
            new XElement(ns2 + "State", "WA"),
            new XElement(ns2 + "Country", "USA")));
  Console.WriteLine(company);
}
```

which produces the following result:

```
<Company xmlns="http://www.cninnovation.com/samples/2018">
<CompanyName xmlns="http://www.cninnovation.com/samples/2018/address">
  Microsoft Corporation</CompanyName>
  <CompanyAddress xmlns="http://www.cninnovation.com/samples/2018/address">
    <Address>One Microsoft Way</Address>
    <City>Redmond</City>
    <Zip>WA 98052-6399</Zip>
    <State>WA</State>
    <Country>USA</Country>
  </CompanyAddress>
</Company>
```

In this case, you can see that the subnamespace was applied to everything you specified except for the `<Address>`, `<City>`, `<State>`, and `<Country>` elements because they inherit from their parent, `<CompanyAddress>`, which has the namespace declaration.

XComment

The `XComment` object enables you to easily add XML comments to your XML documents. The following example shows the addition of a comment to the top of the document and within the `Company` element (code file `LinqToXmlSample/Program.cs`):

```
public static void WithComments()
{
  var doc = new XDocument();
  XComment comment = new XComment("Sample XML for Professional C#.");
  doc.Add(comment);
  var company =
    new XElement("Company",
      new XElement("CompanyName", "Microsoft Corporation"),
      new XComment("A great company"),
      new XElement("CompanyAddress",
        new XElement("Address", "One Microsoft Way"),
        new XElement("City", "Redmond"),
        new XElement("Zip", "WA 98052-6399"),
        new XElement("State", "WA"),
        new XElement("Country", "USA")));
  doc.Add(company);
  Console.WriteLine(doc);
}
```

When you run the application and call the `WithComments` method, you can see the generated XML comments:

```
<!--Sample XML for Professional C#.-->
<Company>
  <CompanyName>Microsoft Corporation</CompanyName>
  <!-A great company->
  <CompanyAddress>
    <Address>One Microsoft Way</Address>
    <City>Redmond</City>
    <Zip>WA 98052-6399</Zip>
    <State>WA</State>
    <Country>USA</Country>
  </CompanyAddress>
</Company>
```

XAttribute

In addition to elements, another important factor of XML is attributes. You add and work with attributes by using the `XAttribute` object. The following example shows the addition of an attribute to the root `<Company>` node (code file `LinqToXmlSample/Program.cs`):

```
public static void WithAttributes()
{
  var company =
    new XElement("Company",
      new XElement("CompanyName", "Microsoft Corporation"),
      new XAttribute("TaxId", "91-1144442"),
      new XComment("A great company"),
      new XElement("CompanyAddress",
        new XElement("Address", "One Microsoft Way"),
        new XElement("City", "Redmond"),
        new XElement("Zip", "WA 98052-6399"),
        new XElement("State", "WA"),
       new XElement("Country", "USA")));
  Console.WriteLine(company);
}
```

The attribute shows up as shown with the `Company` element:

Now that you can get your XML documents into an `XDocument` object and work with the various parts of this document, you can also use LINQ to XML to query your XML documents and work with the results.

Querying XML Documents with LINQ

You will notice that querying a static XML document using LINQ to XML takes almost no work at all. The following example makes use of the `hamlet.xml` file and queries to get all the players (actors) who appear in the play. Each of these players is defined in the XML document with the `<PERSONA>` element. The `Descendants` method of the `XDocument` class returns an `IEnumerable<XElement>` that contains all the `PERSONA` elements within the tree. With every `PERSONA` element of this tree, the `Value` property is accessed with the LINQ query and written to the resulting collection (code file `LinqToXmlSample/Program.cs`):

```
public static void QueryHamlet()
{
  XDocument doc = XDocument.Load(HamletFileName);
  IEnumerable<string> persons = (from people in doc.Descendants("PERSONA")
                                 select people.Value).ToList();
  Console.WriteLine($"{persons.Count()} Players Found");
  Console.WriteLine();

  foreach (var item in persons)
  {
    Console.WriteLine(item);
  }
}
```

When you run the application, you can see the following result from the play *Hamlet*. You can't say you're not learning literature from a C# programming book:

```
26 Players Found
CLAUDIUS, king of Denmark.
HAMLET, son to the late king, and nephew to the present king.
POLONIUS, lord chamberlain.
HORATIO, friend to Hamlet.
LAERTES, son to Polonius.
LUCIANUS, nephew to the king.
VOLTIMAND
CORNELIUS
```

```
ROSENCRANTZ
GUILDENSTERN
OSRIC
A Gentleman
A Priest.
MARCELLUS
BERNARDO
FRANCISCO, a soldier.
REYNALDO, servant to Polonius.
Players.
Two Clowns, grave-diggers.
FORTINBRAS, prince of Norway.
A Captain.
English Ambassadors.
GERTRUDE, queen of Denmark, and mother to Hamlet.
OPHELIA, daughter to Polonius.
Lords, Ladies, Officers, Soldiers, Sailors, Messengers, and other Attendants.
Ghost of Hamlet's Father.
```

Querying Dynamic XML Documents

A lot of dynamic XML documents are available online these days. You can find blog feeds, podcast feeds, and more that provide an XML document by sending a request to a specific URL endpoint. You can view these feeds either in the browser, through an RSS aggregator, or as pure XML. The next example demonstrates how to work with an Atom feed directly from your code.

Here, you can see that the `Load` method of the `XDocument` points to a URL where the XML is retrieved. With the Atom feed, the root element is a `feed` element that contains direct children with information about the feed and a list of `entry` elements for every article. What might not be missed when accessing the elements is the Atom namespace `http://www.w3.org/2005/Atom`, otherwise the results will be empty.

With the sample code, first the values of the title and subtitle elements are accessed that are defined as child elements of the root element. The Atom feed can contain multiple link elements. When you use a LINQ query, only the first `link` element that contains the `rel` attribute with the value `alternate` is retrieved. After writing overall information about the feed to the console, all entry elements are retrieved to create an anonymous type with `Title`, `Published`, `Summary`, `Url`, and `Comments` properties (code file `LinqToXmlSample/Program.cs`):

```
public static void QueryFeed()
{
  try
  {
    var httpClient = new HttpClient();
    using (Stream stream = await httpClient.GetStreamAsync(
      "http://csharp.christiannagel.com/feed/atom/"))
    {
      XNamespace ns = "http://www.w3.org/2005/Atom";
      XDocument doc = XDocument.Load(stream);

      Console.WriteLine($"Title: {doc.Root.Element(ns + "title").Value}");
      Console.WriteLine($"Subtitle: {doc.Root.Element(
        ns + "subtitle").Value}");
      string url = doc.Root.Elements(ns + "link")
        .Where(e => e.Attribute("rel").Value == "alternate")
        .FirstOrDefault()
        ?.Attribute("href")?.Value;

      Console.WriteLine($"Link: {url}");
      Console.WriteLine();

      var queryPosts =
```

```
        from myPosts in doc.Descendants(ns + "entry")
        select new
        {
          Title = myPosts.Element(ns + "title")?.Value,
          Published = DateTime.Parse(
            myPosts.Element(ns + "published")?.Value),
          Summary = myPosts.Element(ns + "summary")?.Value,
          Url = myPosts.Element(ns + "link")?.Value,
          Comments = myPosts.Element(ns + "comments")?.Value
        };

      foreach (var item in queryPosts)
      {
        Console.string shortTitle = item.Title.Length > 50 ?
          item.Title.Substring(0, 50) + "..." : item.Title;
        Console.WriteLine(shortTitle);
      }
    }
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Run the application to see this overall information for the feed:

```
Title: csharp.christiannagel.com
Subtitle: Christian Nagel's Professional C# with UWP, .NET Core, and more
Link: http://csharp.christiannagel.com
```

and the results of the query showing all titles:

```
Local Functions &#8211; What&#8217;s the Value?
Array Pool
Windows Template Studio
.NET Core with csproj
C# 7.0 &#8211; What&#8217;s New
View Components with ASP.NET Core 1.1
Workshop with Updates for C# 7.0 and Visual Studio...
20 Years Visual Studio and 16 Years Professional C...
C# 7.0 &#8211; Pattern Matching
C# 7.0 and ASP.NET Core at BASTA! In Frankfurt, Ge...
```

Transforming to Objects

Using LINQ to SQL, it's easy to transform an XML document to an object tree. The Hamlet file contains all personas of the play. Some personas that belong to groups are grouped within `PGROUP` elements. A group contains the name of the group within the `GRPDESC` element, and personas of the group within `PERSONA` elements. The following sample creates objects for every group and adds the group name and personas to the object. The code sample makes use of the LINQ method syntax instead of the LINQ query for using an overload of the `Select` method that offers the index parameter. The index goes into the newly created object as well. The `Descendants` method of the `XDocument` filters all the `PGROUP` elements. Every group is selected with the `Select` method, and there an anonymous object is created that fills the `Number`, `Description`, and `Characters` properties. The `Characters` property itself is a list of all values of the `PERSONA` elements within the group (code file `LinqToXmlSample/Program.cs`):

```
public static void TransformingToObjects()
{
  XDocument doc = XDocument.Load(HamletFileName);
  var groups =
    doc.Descendants("PGROUP")
      .Select((g, i) =>
```

```
        new
        {
          Number = i + 1,
          Description = g.Element("GRPDESCR").Value,
          Characters = g.Elements("PERSONA").Select(p => p.Value)
        });

    foreach (var group in groups)
    {
      Console.WriteLine(group.Number);
      Console.WriteLine(group.Description);

      foreach (var name in group.Characters)
      {
        Console.WriteLine(name);
      }
      Console.WriteLine();
    }
  }
```

Run the application to invoke the `TransformingToObjects` method and see two groups with their personas:

```
1
courtiers.
VOLTIMAND
CORNELIUS
ROSENCRANTZ
GUILDENSTERN
OSRIC

2
officers.
MARCELLUS
BERNARDO
```

Transforming to XML

Because it's easy to create XML with the `XElement` class and its flexible constructor to pass any number of child elements, the previous example can be changed to create XML instead of an object list. The query is the same as in the previous code sample. What's different is that a new `XElement` passing the name `hamlet` is created. `hamlet` is the root element of this generated XML. The child elements are defined by the result of the `Select` method that follows the `Descendants` method to select all `PGROUP` elements. For every group, a new `group XElement` gets created. Every `group` contains an attribute with the group `number`, an attribute with the `description`, and a `characters` element that contains a list of `name` elements (code file `LinqToXmlSample/Program.cs`):

```
public static void TransformingToXml()
{
  XDocument doc = XDocument.Load(HamletFileName);
  var hamlet =
    new XElement("hamlet",
      doc.Descendants("PGROUP")
        .Select((g, i) =>
          new XElement("group",
            new XAttribute("number", i + 1),
            new XAttribute("description", g.Element("GRPDESCR").Value),
            new XElement("characters",
              g.Elements("PERSONA").Select(p => new XElement("name", p.Value))
  ))));
  Console.WriteLine(hamlet);
}
```

When you run the application, you can see this generated XML fragment:

```
<hamlet>
  <group number="1" description="courtiers.">
    <characters>
      <name>VOLTIMAND</name>
      <name>CORNELIUS</name>
      <name>ROSENCRANTZ</name>
      <name>GUILDENSTERN</name>
      <name>OSRIC</name>
    </characters>
  </group>
  <group number="2" description="officers.">
    <characters>
      <name>MARCELLUS</name>
      <name>BERNARDO</name>
    </characters>
  </group>
</hamlet>
```

JSON

After taking a long tour through many XML features of the .NET Framework, let's get into the JSON data format. *Json.NET* offers a large API where you can use JSON to do many aspects you've seen in this chapter with XML, and some of these will be covered here.

The sample code makes use of the following dependency and namespaces:

Dependency

`Newtonsoft.Json`

Namespaces

`Newtonsoft.Json`

`Newtonsoft.Json.Linq`

`System`

`System.IO`

`System.Xml.Linq`

Creating JSON

To create JSON objects manually with JSON.NET, several types are available in the `Newtonsoft.Json.Linq` namespace. A `JObject` represents a JSON object. `JObject` is a dictionary with strings for the key (property names with .NET objects), and `JToken` for the value. This way `JObject` offers indexed access. An array of JSON objects is defined by the `JArray` type. Both `JObject` and `JArray` derive from the abstract base class `JContainer` that contains a list of `JToken` objects.

The following code snippet creates the `JObject book1` and `book2` objects by filling `title` and `publisher` values using indexed dictionary access. Both book objects are added to a `JArray` (code file `JsonSample/Program.cs`):

```
public static void CreateJson()
{
  var book1 = new JObject();
  book1["title"] = "Professional C# 7 and .NET Core 2.0";
  book1["publisher"] = "Wrox Press";
```

```
    var book2 = new JObject();
    book2["title"] = "Professional C# 6 and .NET Core 1.0";
    book2["publisher"] = "Wrox Press";

    var books = new JArray();
    books.Add(book1);
    books.Add(book2);

    var json = new JObject();
    json["books"] = books;
    Console.WriteLine(json);
  }
```

Run the application to see this JSON code generated:

```
{
  "books": [
    {
      "title": "Professional C# 7 and .NET Core 2.0",
      "publisher": "Wrox Press"
    },
    {
      "title": "Professional C# 6 and .NET Core 1.0",
      "publisher": "Wrox Press"
    }
  ]
}
```

Converting Objects

Instead of using `JsonObject` and `JsonArray` to create JSON content, you can also use the `JsonConvert` class. `JsonConvert` enables you to create JSON from an object tree and convert a JSON string back into an object tree.

With the sample code in this section, you create an `Inventory` object from the helper method `GetInventoryObject` (code file `JsonSample/Program.cs`):

```
public static Inventory GetInventoryObject() =>
  new Inventory
  {
    InventoryItems = new Product[]
    {
      new Product
      {
        ProductID = 100,
        ProductName = "Product Thing",
        SupplierID = 10
      },
      new BookProduct
      {
        ProductID = 101,
        ProductName = "How To Use Your New Product Thing",
        SupplierID = 10,
        ISBN = "1234567890"
      }
    }
  };
```

The method `ConvertObject` retrieves the `Inventory` object and converts it to JSON using `JsonConvert.SerializeObject`. The second parameter of `SerializeObject` allows formatting to be defined `None` or Indented. `None` is best for keeping whitespace to a minimum; `Indented` allows for better readability. The JSON string is written to the console before it is converted back to an object tree using `JsonConvert`

`.DeserializeObject`. `DeserializeObject` has a few overloads. The generic variant returns the generic type instead of an object, so a cast is not necessary:

```
public static void ConvertObject()
{
  Inventory inventory = GetInventoryObject();
  string json = JsonConvert.SerializeObject(inventory, Formatting.Indented);
  Console.WriteLine(json);
  Console.WriteLine();
  Inventory newInventory = JsonConvert.DeserializeObject<Inventory>(json);
  foreach (var product in newInventory.InventoryItems)
  {
    Console.WriteLine(product.ProductName);
  }
}
```

Running the application shows the generated console output of the JSON generated Inventory type:

```
{
  "InventoryItems": [
    {
      "Discount": 0,
      "ProductID": 100,
      "ProductName": "Product Thing",
      "SupplierID": 10,
      "CategoryID": 0,
      "QuantityPerUnit": null,
      "UnitPrice": 0.0,
      "UnitsInStock": 0,
      "UnitsOnOrder": 0,
      "ReorderLevel": 0,
      "Discontinued": false
    },
    {
      "ISBN": "1234567890",
      "Discount": 0,
      "ProductID": 101,
      "ProductName": "How To Use Your New Product Thing",
      "SupplierID": 10,
      "CategoryID": 0,
      "QuantityPerUnit": null,
      "UnitPrice": 0.0,
      "UnitsInStock": 0,
      "UnitsOnOrder": 0,
      "ReorderLevel": 0,
      "Discontinued": false
    }
  ]
}
```

Converting back JSON to objects, the product names are shown:

```
Product Thing
How To Use Your New Product Thing
```

Serializing Objects

Like the `XmlSerializer`, you can also stream the JSON string directly to a file. The following code snippet retrieves the Inventory object and writes it to a file stream using the `JsonSerializer` (code file `JsonSample/Program.cs`):

```
public static void SerializeJson()
{
  using (StreamWriter writer = File.CreateText(InventoryFileName))
```

```
  {
    JsonSerializer serializer = JsonSerializer.Create(
      new JsonSerializerSettings { Formatting = Formatting.Indented });
    serializer.Serialize(writer, GetInventoryObject());
  }
}
```

You can convert JSON from a stream by calling the `Deserialize` method on the `JsonSerializer`:

```
public static void DeserializeJson()
{
  using (StreamReader reader = File.OpenText(InventoryFileName))
  {
    JsonSerializer serializer = JsonSerializer.Create();
    var inventory = serializer.Deserialize(reader, typeof(Inventory))
      as Inventory;

    foreach (var item in inventory.InventoryItems)
    {
      Console.WriteLine(item.ProductName);
    }
  }
}
```

Iterating Through JSON Nodes

To access information about all the JSON nodes, you can use the `JsonTextReader` and iterate through the nodes to invoke the `Read` method. When you use the `JsonTextReader`, you can see the type of the node with the `TokenType` property, access path, value, and lines and character positions in the JSON file (code file `JsonSample/Program.cs`):

```
public static void ReaderSample()
{
  StreamReader textReader = File.OpenText(InventoryFileName);
  using (JsonTextReader jsonReader = new JsonTextReader(textReader)
  { CloseInput = true })
  {
    while (jsonReader.Read())
    {
      Console.Write($"token: {jsonReader.TokenType}, ");
      if (!string.IsNullOrEmpty(jsonReader.Path))
      {
        Console.Write($"path: {jsonReader.Path}, ");
      }
      if (!string.IsNullOrEmpty(jsonReader.Value?.ToString()))
      {
        Console.Write($"value: {jsonReader.Value}");
      }
      Console.WriteLine();
    }
  }
}
```

When you run the application, an extract of the output is shown—with token types such as `StartObject`, `Propertyname`, `StartArray`, and `Integer` tokens containing a value:

```
token: StartObject,
token: PropertyName, path: InventoryItems, value: InventoryItems
token: StartArray, path: InventoryItems,
token: StartObject, path: InventoryItems[0],
token: PropertyName, path: InventoryItems[0].Discount, value: Discount
token: Integer, path: InventoryItems[0].Discount, value: 0
token: PropertyName, path: InventoryItems[0].ProductID, value: ProductID
token: Integer, path: InventoryItems[0].ProductID, value: 100
```

SUMMARY

This chapter explored many aspects of the `System.Xml` namespace. You looked at how to read and write XML documents using the very fast `XmlReader`- and `XmlWriter`-based classes. You saw how the DOM is implemented in .NET and how to use the power of DOM, with the `XmlDocument` class. In addition, you visited XPath, serialized objects to XML, and were able to bring them back with just a couple of method calls.

By using LINQ to XML, you've seen how to easily create XML documents and fragments and create queries using XML data.

Aside of XML, you've seen how to serialize objects using JSON with `Json.NET`, and you've parsed JSON strings to build .NET objects.