Beginning XML with C# 2008: From Novice to Professional

Copyright © 2008 by Bipin Joshi

ISBN-13 (pbk): 978-1-4302-0997-3

ISBN-13 (electronic): 978-1-4302-0998-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Matthew Moodie
Technical Reviewer: Fabio Claudio Ferracchiati
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Senior Project Manager: Beth Christmas
Copy Editor: Heather Lang
Associate Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor: Susan Glinert
Proofreader: Linda Seifert
Indexer: Brenda Miller
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at http://www.apress.com/info/bulksales.

The source code for this book is available to readers at http://www.apress.com.

CHAPTER 1

Introducing XML and the .NET Framework

XML has emerged as the de facto standard for data representation and transportation. No wonder that Microsoft has embraced it fully in the NET Framework. This chapter provides an overview of what XML is and how it is related to the .NET Framework. Many of the topics discussed in this chapter might be already familiar to you. Nevertheless, I will cover them briefly here so as to form a common platform for further chapters. Specifically, this chapter includes the following:

- Features and benefits of XML
- Rules of XML grammar
- Brief introduction to allied technologies such as DTD, XML schema, parsers, XSLT, and XPath
- Overview of the .NET Framework
- Use of XML in the .NET Framework
- Introduction to Visual Studio

If you find these concepts highly familiar, you may want to skip ahead to Chapter 2.

What Is XML?

XML stands for *Extensible Markup Language* and is a markup language used to describe data. It offers a standardized way to represent textual data. Often the XML data is also referred to as an XML document. The XML data doesn't perform anything on its own; to process that data, you need to use a piece of software called a *parser*. Unlike Hypertext Markup Language (HTML), which focuses on how to present data, XML focuses on how to represent data. XML consists of user-defined tags, which means you are free to define and use your own tags in an XML document.

XML was approved as a recommendation by the World Wide Web Consortium (W3C) in February 1998. Naturally, this very fact contributed a lot to such a wide acceptance and support of XML in the software industry.

Now that you have brief idea about XML, let's see a simple XML document, as illustrated in Listing 1-1.

Listing 1-1. *A Simple XML Document*

```
<?xml version="1.0"?>
<customers>
   <customer ID="C001">
      <name>Acme Inc.</name>
      <phone>12345</phone>
   </customer>
   <customer ID="C002">
      <name>Star Wars Inc.</name>
      <phone>23456</phone>
   </customer>
</customers>
```

Many rules govern the creation of such XML documents. But we will save them for later discussion.

Benefits of XML

Why did XML become so popular? Well, this question has many answers, and I will present some of the important ones in this section.

XML Is an Industry Standard

As you learned previously, XML is a W3C recommendation. This means it is an industry standard governed by a vendor-independent body. History shows that vendor-specific proprietary standards don't get massive acceptance in the software industry. This nonacceptance affects overall cross-platform data sharing and integration. Being an industry standard has helped XML gain huge acceptance.

XML Is Self-Describing

XML documents are self-describing. Because of markup tags, they are more readable than, say, comma-separated values (CSV) files.

XML Is Extensible

Markup languages such as HTML have a fixed set of tags and attributes—you cannot add your own tags. XML, on the other hand, allows you to define your own markup tags.

XML Can Be Processed Easily

Traditionally, the CSV format was a common way to represent and transport data. However, to process such data, you need to know the exact location of the commas (`,`) or any other delimiter used. This makes reading and writing the document difficult. The problem becomes severe when you are dealing with a number of altogether different and unknown CSV files.

As I said earlier, XML documents can be processed by a piece of software called a parser. Because XML documents use markup tags, a parser can read them easily. Parsers are discussed in more detail later in this chapter.

XML Can Be Used to Easily Exchange Data

Integrating cross-platform and cross-vendor applications is always difficult and challenging. Exchanging data in heterogeneous systems is a key problem in such applications. Using XML as a data-exchange format makes your life easy. XML is an industry standard, so it has massive support, and almost all vendors support it in one way or another.

XML Can Be Used to Easily Share Data

The fact that XML is nothing but textual data ensures that it can be shared among heterogeneous systems. For example, how can a Visual Basic 6 (VB6) application running on a Windows machine talk with a Java application running on a Unix box? XML is the answer.

XML Can Be Used to Create Specialized Vocabularies

As you already know, XML is an extensible standard. By using XML as a base, you can create your own vocabularies. Wireless Application Protocol (WAP), Wireless Markup Language (WML), and Simple Object Access Protocol (SOAP) are some examples of specialized XML vocabularies.

XML-Driven Applications

Now that you know the features and benefits of XML, let's see what all these benefits mean to modern software systems.

Figure 1-1 shows a traditional web-based application. The application consists of Active Server Pages (ASP) scripts hosted on a web server. The client, in the form of a web browser, requests various web pages. On receiving the requests, the web server processes them and sends the response in the form of HTML content. This architecture sounds good at first glance, but suffers from several shortcomings:

- It considers only web browsers as clients.
- The response from the web server is always in HTML. That means a desktop-based application may not render this response at all.
- The data and presentation logic are tightly coupled. If we want to change the presentation of the same data, we need to make considerable changes.
- Tomorrow, if some other application wants to consume the same data, it cannot be shared easily.

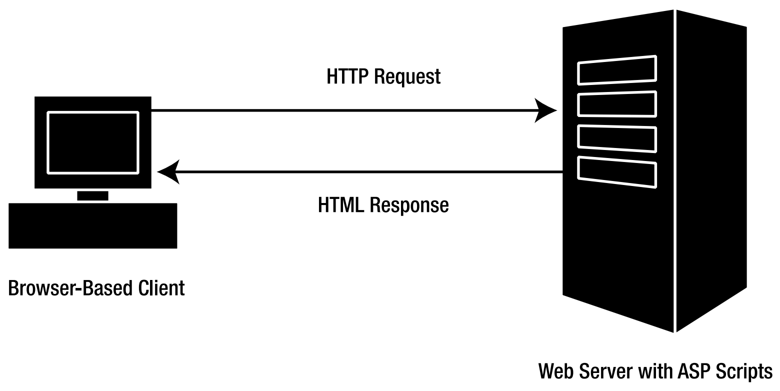


Figure 1-1. *Classic architecture for developing applications*

Now, let's see how XML can come to the rescue in such situations.

Have a look at Figure 1-2, where there are multiple types of clients. One is a web browser, and the other is a desktop application. Both send requests to the server in the form of XML data. The server processes the requests and sends the data in XML format. The web browser applies a style sheet (discussed later) to the XML data to render it as HTML content. The desktop application, on the other hand, parses the data by using an XML parser (discussed later) and displays it in a grid. Much more flexible than the previous architecture, isn't it? The advantages of the new architecture are as follows:

- The application has multiple types of clients. It is not tied only to web browsers.
- There is loose coupling between the client and the processing logic.
- New types of clients can be added at any time without changing the processing logic on the server.
- The data and the presentation logic are neatly separated from each other. Web clients have one set of presentation logic, whereas desktop applications have their own presentation logic.
- Data sharing becomes easy, because the outputted data is in XML format.

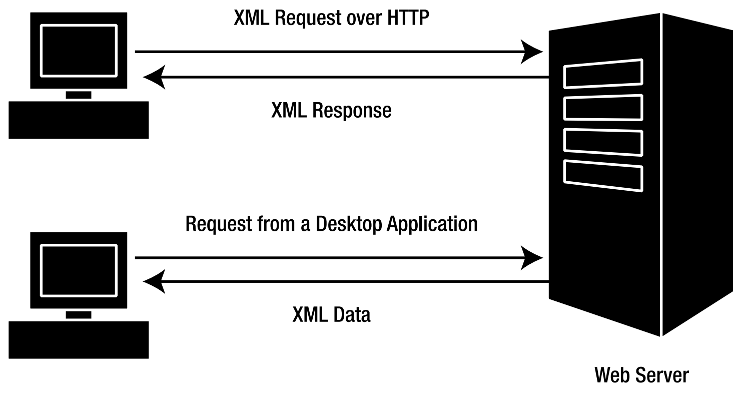
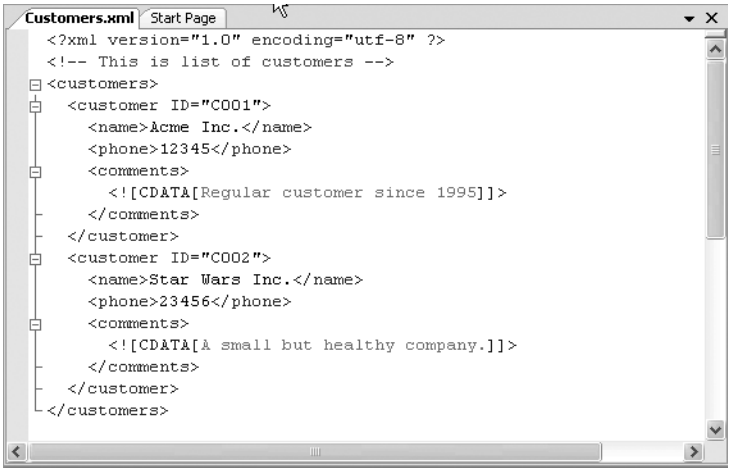


Figure 1-2. *XML-driven architecture*

Rules of XML Grammar

In the “What is XML?” section, you saw one example of an XML document. However, I didn’t talk about any of the rules that you need to follow while creating it. It’s time now to discuss those rules of XML grammar. If you have worked with HTML, you will find that the rules of XML grammar are more strict than the HTML ones. However, this strictness is not a bad thing, because these rules help ensure that there are no errors while we parse, render, or exchange data.

Before I present the rules in detail, you need to familiarize yourself with the various parts of an XML document. Observe Figure 1-3 carefully.



```
<?xml version="1.0" encoding="utf-8" ?>
<!-- This is list of customers -->
<customers>
  <customer ID="C001">
    <name>Acme Inc.</name>
    <phone>12345</phone>
    <comments>
      <![CDATA[Regular customer since 1995]]>
    </comments>
  </customer>
  <customer ID="C002">
    <name>Star Wars Inc.</name>
    <phone>23456</phone>
    <comments>
      <![CDATA[A small but healthy company.]]>
    </comments>
  </customer>
</customers>
```

Figure 1-3. *Parts of a typical XML document*

Line 1 is called a processing instruction. A *processing instruction* is intended to supply some information to the application that is processing the XML document. Processing instructions are enclosed in a pair of `<?` and `?>`. The `xml` processing instruction in Figure 1-3 has two attributes: `version` and `encoding`. The current W3C recommendations for XML hold version 1.0, hence the `version` attribute must be set to `1.0`.

Line 2 represents a comment. A *comment* can appear anywhere in an XML document after the `xml` processing instruction and can span multiple lines.

Line 3 contains the *document element* of the XML document. An XML document has one and only one document element. XML documents are like an inverted tree, and the document element is positioned at the root. Hence, the document element is also called a *root element*. Each element (whether or not it is the document element) consists of a start tag and end tag. The start tag is `<customers>`, and the end tag is `</customers>`.

It is worthwhile to point out the difference between three terms: element, node, and tag. When you say *element*, you are essentially talking about the start tag and the end tag of that element together. When you say *tag*, you are talking about either the start tag or end tag of the element, depending on the context. When you say *node*, you are referring to an element and all its inner content, including child elements and text.

Inside the `<customers>` element, you have two `<customer>` nodes. The `<customer>` element has one attribute called `ID`. The attribute value is enclosed in double quotes. The `<customer>`

element has three child elements: `<name>`, `<phone>`, and `<comments>`. The text values inside elements, such as `<name>` and `<phone>`, are often called *text nodes*. Sometimes, the text content that you want to put inside a node may contain special characters such as < and >. To represent such content, you use a character data (`CDATA`) section. Whatever you put inside the `CDATA` section is treated as a literal string. The `<comments>` tag shown in Figure 1-3 illustrates the use of a `CDATA` section.

Now that you have this background, you're ready to look at the basic rules of XML grammar. Any XML document that conforms to the rules mentioned next is called a *well-formed document*.

Markup Is Case Sensitive

Just like some programming languages, such as C#, XML markup is case sensitive. That means `<customer>`, `<Customer>`, and `<CUSTOMER>` all are treated as different tags.

A Document Must Have One and Only One Root Element

An XML document must have one and only one root element. In the preceding example, the `<customers>` element is the root element. Note that it is mandatory for XML documents to have a root element.

A Start Tag Must Have an End Tag

Every start tag must have a corresponding end tag. In HTML, this rule is not strictly followed—for example, tags such as `<br>` (line break), `<hr>` (horizontal rule), and `<img>` (image) are often used with no end tag at all. In XML, that would not be well formed. The end tag for elements that do not contain any child elements or text can be written by using shorter notation. For example, assuming that the `<customer>` tag doesn't contain any child elements, you could have written it as `<customer ID="C001"/>`.

Start and End Tags Must Be Properly Nested

In HTML, the rule about properly nesting tags is not followed strictly. For example, the following markup shows up in the browser correctly:

```
<B><I>Hello World</B></I>
```

This, however, is illegal in XML, where the nesting of start and end tags must be proper. The correct representation of the preceding markup in XML would be as follows:

```
<B><I>Hello World</I></B>
```

Attribute Values Must Be Enclosed in Quotes

In HTML, you may or may not enclose the attribute values. For example, the following is valid markup in HTML:

```
<IMG SRC=myphoto.jpg>
```

However, this is illegal in XML. All attribute values must be enclosed in quotes. Thus the accepted XML representation of the preceding markup would be as follows:

```
<IMG SRC="myphoto.jpg">
```

DTDs and XML Schemas

Creating well-formed XML documents is one part of the story. The other part is whether these documents adhere to an agreed structure, or *schema*. That is where Document Type Definitions (DTDs) and XML schemas come into the picture.

DTDs and XML schemas allow you to convey the structure of your XML document to others. For example, if I tell you to create an XML file, what structure will you follow? What is the guarantee that the structure that you create is the one that I have in mind? The problem is solved if I give you a DTD or schema for the document. Then, you have the exact idea as to how the document should look and what its elements, attributes, and nesting are.

The XML documents that conform to some DTD or XML schema are called *valid documents*. Note that an XML document can be well formed, but it may not be valid if it doesn't have an associated DTD or schema.

DTDs are an older way to validate XML documents. Nowadays, XML schemas are more commonly used to validate XML documents because of the advantages they offer. You will learn about the advantages of schemas over DTDs in Chapter 5. Throughout our discussion, when I talk about validating XML documents, I will be referring to XML schemas.

Parsing XML Documents

XML data by itself cannot do anything; you need to process that data to do something meaningful. As I have said, the software that processes XML documents is called a parser (or XML processor). XML parsers allow you read, write, and manipulate XML documents. XML parsers can be classified in two categories depending on how they process XML documents:

- DOM-based parsers (*DOM* stands for *Document Object Model*)
- SAX-based parsers (*SAX* stands for *Simple API for XML*)

DOM-based parsers are based on the W3C's DOM recommendations and are possibly the most common and popular. They look at your XML document as an inverted tree structure. Thus our XML document shown in Figure 1-3 will be looked at by a DOM parser as shown in Figure 1-4.

DOM-based parsers are read-write parsers, which means you can read as well as write to the XML document. They allow random access to any particular node of the XML document, and therefore, they need to load the entire XML document in memory. This also implies that the memory footprint of DOM-based parsers is large. DOM-based parsers are also called *tree-based parsers* for obvious reasons.

■ **Note** Microsoft's DOM-based parser implementation is nothing but a COM component popularly known as Microsoft XML Core Services (MSXML).

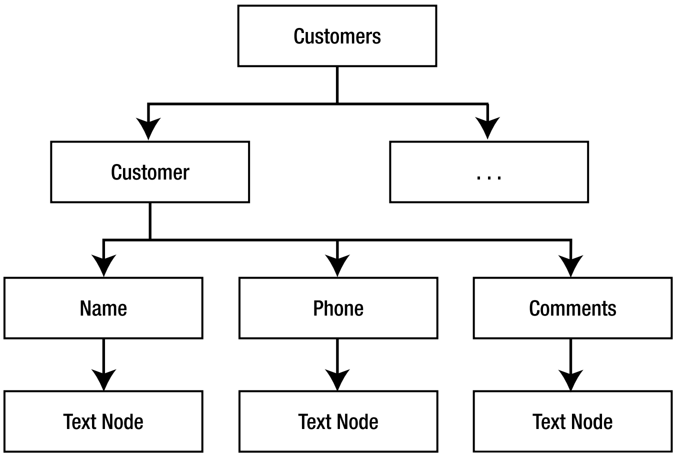


Figure 1-4. *The DOM representation of an XML document*

SAX-based parsers do not read the entire XML document into memory at once. They essentially scan the document sequentially from top to bottom. When they encounter various parts of the document, they raise events, and you can handle these events to read the document. SAX parsers are read-only parsers, which means you cannot use them to modify an XML document. They are useful when you want to read huge XML documents and loading such documents into memory is not advisable. These types of parsers are also called *event-based parsers*.

■ **Note** MSXML includes a component that provides a SAX implementation of the parser.

Parsers can also be classified as validating and nonvalidating. *Validating parsers* can validate an XML document against a DTD or schema as they parse the document. On the other hand, *nonvalidating parsers* lack this ability.

■ **Note** .NET 3.0 introduced Language Integrated Query (LINQ) extensions that offered a new way of reading and writing XML documents. .NET 3.5 and Visual Studio 2008 continue to harness these features. You will learn more about LINQ later in this chapter. Chapter 13 covers the LINQ features as applicable to XML data manipulation in fuller details.

XSLT

XML solves the problem of data representation and exchange. However, often we need to convert this XML data into a format understood by the target application. For example, if your target client application is a web browser, the XML data must be converted to HTML before display in the browser.

Another example is that of business-to-business (B2B) applications. Let's say that application A captures order data from the end user and represents it in some XML format. This data then needs to be sent to application B that belongs to some other business. It is quite possible that the XML format as generated by application A is different from that required by application B. In such cases, you need to convert the source XML data to a format acceptable to the target system. In short, in real-world scenarios you need to transform XML data from one form to another.

That is where XSLT comes in handy. *XSLT* stands for *Extensible Stylesheet Language Transformations* and allows you to transform XML documents from one form into another. Figure 1-5 shows how this transformation happens.

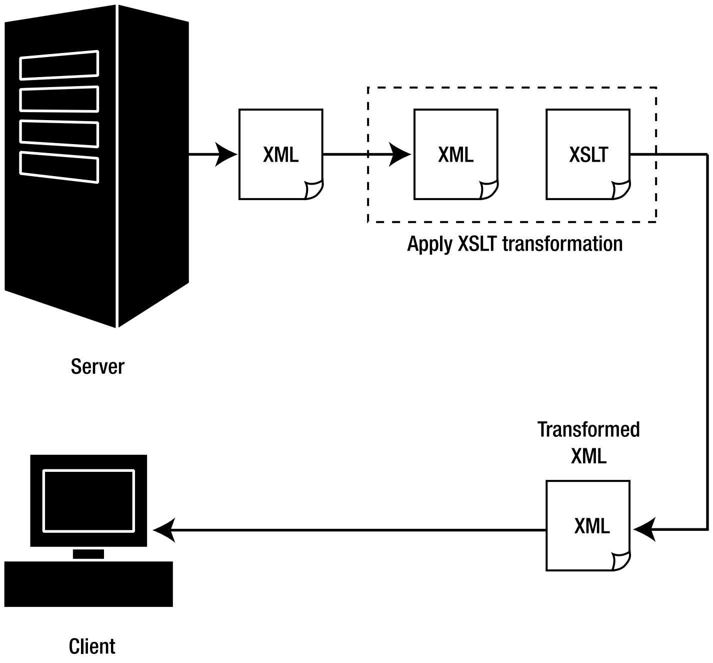


Figure 1-5. *XML transformation*

XPath

Searching for and locating certain elements within an XML document is a fairly common task. *XPath* is an expression language that allows you to navigate through elements and attributes in an XML document. XPath consists of various XPath expressions and functions that you can use to look for and select elements and attributes matching certain patterns. XPath is also a W3C recommendation. Figure 1-6 shows an example of how XPath works.

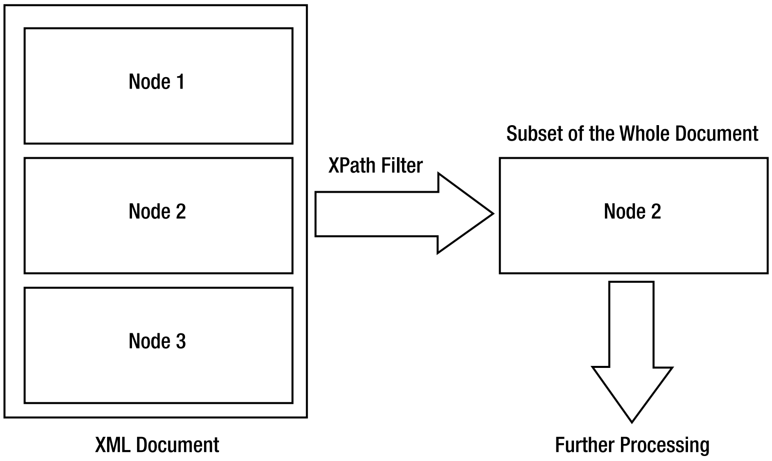


Figure 1-6. *Using XPath to select nodes*

The .NET Framework

Microsoft's *.NET Framework* is a platform for building Windows- and web-based applications, components, and services by using a variety of programming languages. Figure 1-7 shows the stack of the .NET Framework.

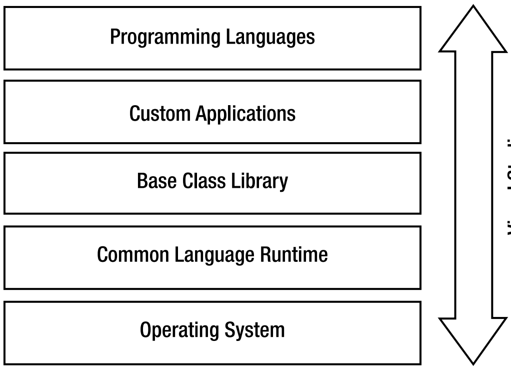


Figure 1-7. *Stack of the .NET Framework*

At the bottom level, you have the operating system. As far as commercial application development using the .NET Framework is concerned, your operating system will be one of the various flavors of Windows (including Windows 2000, Windows 2003, Windows XP, and Windows Vista).

On top of the operating system, you have the common language runtime (CLR) layer. The CLR is the heart of the .NET Framework. It provides the executing environment to all the .NET applications, so in order to run any .NET applications, you must have the CLR installed. The CLR does many things for your application, including memory management, thread management, and security checking.

On top of the CLR, a huge collection of classes called the Base Class Library gets installed. The Base Class Library provides classes to perform almost everything that you need in your application. It includes classes for file input/output (IO), database access, XML manipulation, web programming, socket programming, and many more things. If you are developing a useful application in .NET, chances are that you will use one or another of the classes in the Base Class Library, and hence, your applications are shown sitting on top of it.

These applications can be developed using a variety of programming languages. Out of the box, the .NET Framework provides five programming languages: Visual Basic .NET, Visual C#, Managed C++, JScript .NET, and Visual J#. There are many other third-party compilers that you can use to develop .NET applications.

As a matter of fact, you can develop any .NET application by using Notepad and command-line compilers. However, most of the real-world applications call for a short development time, so that is where an integrated development environment (IDE) such as Visual Studio 2008 can be very helpful. It makes you much more productive than the Notepad approach. Features such as drag and drop, powerful debugging, and IntelliSense make application development much simpler and faster.

.NET and XML

The .NET Framework Base Class Library provides a rich set of classes that allows you to work with XML data. The relationship between the .NET Framework and XML doesn't end here. There are a host of other features that make use of XML. These features include the following:

- .NET configuration files
- ADO.NET
- ASP.NET server controls
- XML serialization
- Remoting
- Web services
- XML documentation
- SQL Server XML features
- XML parsing
- XML transformation

In this section, you will take a brief look at each of these features.

Assemblies and Namespaces

The core XML-related classes from the Base Class Library are physically found in an assembly called `System.Xml.dll`. This assembly contains several namespaces that encapsulate various XML-related classes. The LINQ to XML classes are physically available in the `System.Xml.Linq.dll` assembly. In the following sections, you will take a brief look at some of the important namespaces from these assemblies.

System.Xml Namespace

The `System.Xml` namespace is one of the most important namespaces. It provides classes for reading and writing XML documents. Classes such as `XmlDocument` represent the .NET Framework's DOM-based parser, whereas classes such as `XmlTextReader` and `XmlTextWriter` allow you to quickly read and write XML documents. This namespace also contains classes that represent various parts of an XML document; these classes include `XmlNode`, `XmlElement`, `XmlAttribute`, and `XmlText`. We will be using many of these classes throughout this book.

System.Xml.Schema Namespace

The `System.Xml.Schema` namespace contains various classes that allow you to work with schemas. The entire Schema Object Model (SOM) of .NET is defined by the classes from this namespace. These classes include `XmlSchema`, `XmlSchemaElement`, `XmlSchemaComplexType`, and many others.

System.Xml.XPath Namespace

The `System.Xml.XPath` namespace provides classes and enumerations for finding and selecting a subset of the XML document. These classes provide a cursor-oriented model for navigating through and editing the selection. The classes include `XPathDocument`, `XPathExpression`, `XPathNavigator`, `XPathNodeIterator`, and more.

System.Xml.Xsl Namespace

The `System.Xml.Xsl` namespace provides support for XSLT transformations. By using the classes from this namespace, you can transform XML data from one form to another. The classes provided by this namespace include `XslCompiledTransform`, `XslTransform`, `XsltSettings`, and so on.

System.Xml.Serialization Namespace

The `System.Xml.Serialization` namespace provides classes and attributes that are used to serialize and deserialize objects to and from XML format. These classes are extensively used in web services infrastructures. The main class provided by this namespace is `XmlSerializer`. Some commonly used attributes classes such as `XmlAttributeAttribute`, `XmlRootAttribute`, `XmlTextAttribute`, and many others are also provided by this namespace.

System.Xml.Linq Namespace

The `System.Xml.Linq` namespace contains classes related to the LINQ to XML features. Using these classes you can manipulate XML documents and fragments efficiently and easily. Some of the important tasks that you can accomplish include loading XML from files or streams, creating XML trees via code, querying XML trees using LINQ operators, modifying XML trees, validating XML trees against schema, and transforming XML trees. Some of the frequently used classes from the `System.Xml.Linq` namespace are `XDocument`, `XElement`, `XNode`, `XAttribute`, and `XText`.

The Classic XML Parsing Model of the .NET Framework

The previous sections discussed two types of parsers: DOM- or tree-based parsers and SAX- or event-based parsers. It would be reasonable for you to expect that the .NET Framework supports parsing models for both types of parsers. Though you won't be disappointed at its offerings, there are some differences that you must know.

In the .NET Framework, you can categorize the XML parsers into two flavors:

- Parser based on the DOM
- Parsers based on the reader model

The first thing that may strike you is the lack of a SAX-based parser. But don't worry, the new reader-based parsers provide similar functionality in a more efficient way. You can think of reader-based parsers as an alternative to traditional SAX-based parsers.

The DOM-based parser of the .NET Framework is represented chiefly by a class called `XmlDocument`. By using this parser, you can load, read, and modify XML documents just as you would with any other DOM-based parser (such as MSXML, for example).

The reader-based parsers use a cursor-oriented approach to scan the XML document. The main classes that are at the heart of these parsers are `XmlReader` and `XmlWriter`. These two classes are abstract classes, and other classes (such as `XmlTextReader` and `XmlTextWriter`) inherit from them. You can also create your own readers and writers if you so wish.

Thus to summarize, the .NET Framework supports DOM parsing and provides an alternate and more efficient way to carry out SAX-based parsing. I will be discussing these parsers thoroughly in subsequent chapters.

The LINQ-Based Parsing Model of the .NET Framework

LINQ is a set of language and .NET Framework extensions that allows you to query in-memory collections, databases, and XML documents in a unified fashion. This implies that, irrespective of the underlying data source, your code will query the data in the same way. LINQ comes in three flavors:

- LINQ to objects
- LINQ to ADO.NET
- LINQ to XML

LINQ to objects provides a set of standard query operators for querying in-memory objects. The in-memory objects must implement the `IEnumerable<T>` interface. The most common objects for querying are generic `List` and `Dictionary` objects.

LINQ to ADO.NET provides a set of features for working with data from relational databases. LINQ to ADO.NET comes in two flavors: LINQ to `DataSet`, which allows you to query ADO.NET `DataSet` objects, and LINQ to SQL, which allows you to query relational databases such as SQL Server.

LINQ to XML is a new approach to programming with XML data. It provides the in-memory document modification capabilities of the DOM in addition to supporting LINQ query operators. LINQ to XML operations are more lightweight than traditional DOM operations. Classes such as `XDocument`, `XElement`, `XNode`, `XAttribute`, and `XText` provide functionality equivalent to `XmlDocument` and its family of classes, as you'll see later in this book.

.NET Configuration Files

Almost all real-world applications require configuration, which includes things such as database connection strings, file system paths, security schemes, and role-based security settings. Prior to the introduction of the .NET Framework, developers often used `.ini` files or the Windows registry to store such configuration settings. Unfortunately, the simple task of storing configuration settings used to be cumbersome in popular tools such as Visual Basic 6. For example, VB6 doesn't have a native mechanism to read and write to `.ini` files. Developers often used Windows application programming interfaces (APIs) to accomplish this. VB6 does have some features to work with the Windows registry, but they are too limited for most scenarios. Moreover, storing data in the Windows registry always comes with its own risks. In such cases, developers tend to rely on a custom solution. The impact is obvious: no standardization, more coding time, more effort, and repeated coding for the same task.

Thankfully, the .NET Framework takes a streamlined and standardized approach to configuring applications. It relies on XML-based files for storing configuration information. That means developers no longer need to write custom logic to read and write to `.ini` files or even the Windows registry. Some of the advantages of using XML files instead of the classic approaches are as follows:

- Because XML files are more readable, the configuration data can be stored in a neat and structured way.
- To read the configuration information, the .NET Framework provides built-in classes. That means you need not write any custom code to access the configuration data.
- Storing the configuration information in XML files makes it possible to deploy it easily along with the application. In the past, Windows-registry–based configuration posed various deployment issues.
- There are no dangers in manipulating the XML configuration files for your application. In the past, developer's needed to tamper with the Windows registry that is risky and often created unwanted results.
- .NET Framework configuration files are not limited to using the predefined XML tags. You can extend the configuration files to add custom sections.
- Sometimes, the configuration information includes some confidential data. .NET Framework configuration files can be encrypted easily, giving more security to your configuration data. The encryption feature is a built-in part of the framework needing no custom coding from the developer.

The overall configuration files of the .NET Framework are of three types:

- Application configuration files
- Machine configuration files
- Security configuration files

Application configuration files store configuration information applicable to a single application. For Windows Forms and console-based applications, the name of the configuration file takes the following form:

```
<exe name>.exe.config
```

That means that if you are developing a Windows application called `HelloWorld.exe`, its configuration file name must be `HelloWorld.exe.config`. The markup from Listing 1-2 shows sample configuration information for a Windows Forms–based application.

Listing 1-2. *XML Markup from an Application Configuration File*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="defaultemail"
         value="someone@somedomain.com"/>
  </appSettings>
</configuration>
```

On the other hand, a configuration file for a web application is called `web.config`. The markup in Listing 1-3 shows a sample `web.config` file.

Listing 1-3. *XML Markup from a web.config File*

```
<?xml version="1.0"?>
   <configuration>
       <connectionStrings>
         <add name="connstr"
                     connectionString="Data Source=.\SQLEXPRESS;
                     Integrated Security=True;
                    AttachDbFilename=|DataDirectory|AspNetDb.mdf"
                    providerName="System.Data.SqlClient"/>
      </connectionStrings>
   <system.web>
      <compilation debug="true"/>
      <authentication mode="Forms">
      <forms name="login" loginUrl="login.aspx">
      </forms>
      </authentication>
      <authorization>
         <deny users="?"/>
      </authorization>
      <membership defaultProvider="AspNetSqlProvider">
            <providers>
```

```
                  <add
                            name="AspNetSqlProvider"
                            type="System.Web.Security.SqlMembershipProvider"
                            connectionStringName="connstr"
                            passwordFormat="Clear"
                           enablePasswordRetrieval="true"
                          requiresQuestionAndAnswer="true"
                          maxInvalidPasswordAttempts="3">
                     </add>
              </providers>
       </membership>
   </system.web>
</configuration>
```

When you install the .NET Framework on a machine, a file named `machine.config` gets created in the installation folder of the .NET Framework. This file is the master configuration file and contains configuration settings that are applied to all the .NET applications running on that machine. The settings from `machine.config` can be overridden by using the application configuration file. Because the settings from `machine.config` are applied to all the .NET applications, it is recommended that you alter this file with caution. Generally, only server administrators and web-hosting providers modify this file.

The .NET Framework offers a secure environment for executing applications. It needs to check whether an assembly is trustworthy before any code in the assembly is invoked. To test the trustworthiness of an assembly, the framework checks the permission granted to it. Permissions granted to an assembly can be configured by using the security configuration files. This is called *Code Access Security*.

ADO.NET

For most business applications, data access is where the rubber meets the road. In .NET, *ADO.NET* is the technology for handling database access. Though ADO.NET sounds like it is the next version of classic ADO, it is, in fact, a complete rewrite for the .NET Framework.

ADO.NET gives a lot of emphasis to disconnected data access, though connected data access is also possible. A class called `DataSet` forms the cornerstone of the overall disconnected data architecture of ADO.NET. A `DataSet` class can be easily serialized as an XML document, and hence, it is ideal for data interchange, cross-system communication, and the like. A class called `XmlDataDocument` allows you to work with relational or XML data by using a DOM-based style. It can give a `DataSet` to you, which you can use further for data binding and related tasks. Another class called `SqlCommand` allows you to read data stored in Microsoft SQL Server and return it as an XML reader (`XmlReader`). I am going to cover XML-related features of ADO.NET in detail in Chapter 7.

ASP.NET Server Controls

You learned that the ASP.NET configuration file (`web.config`) is an XML file. The use of XML in ASP.NET doesn't end there. ASP.NET uses a special XML vocabulary to represent its server controls, which are programmable controls that can be accessed from server-side code. Consider the markup shown in bold in Listing 1-4.

Listing 1-4. *Server Control Markup*

```
<%@ Page Language="C#" %>
<script runat="server">
protected void Button1_Click(object sender, EventArgs e)
{
Label2.Text = TextBox1.Text;
}
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
<form id="form1" runat="server">
<asp:Label ID="Label1" runat="server" Text="Enter some text :"></asp:Label>
<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
<asp:Button ID="Button1" runat="server" Text="Submit" OnClick="Button1_Click" />
<asp:Label ID="Label2" runat="server"></asp:Label>
</form>
</body>
</html>
```

The preceding fragment shows the markup of a few ASP.NET server controls. As you can see, a `Label` control is represented by the `<asp:Label>` markup tag. Similarly, a `Button` control is represented by the `<asp:Button>` markup tag. This is a special vocabulary of XML and follows all the rules of XML grammar.

XML Serialization

Modern applications seldom run on a single machine. They are distributed and span two or more machines. Figure 1-8 shows a simple distributed application spanning three machines.

Here, the database and data-access components are located on a separate server. Similarly, business logic components are located on their own server, and the client applications access these components through a network. Imagine that the client wants some data from the database to display to the end user. The data is pulled out from the database from data-access components. But how will it reach the client? That is where serialization comes into the picture.

Serialization is a process in which data is written to some medium. In the preceding example, the medium is a network—but it can be a file or any other stream also. The data-access components will serialize the requested data so that it can reach the client application. The client application then deserializes it—that is, it reads from the medium and reconstructs the data in an object or any other data structure. In the case of XML serialization, this data is serialized in the XML format. XML serialization is used extensively by web services. The `XmlSerializer` class provides a programmatic way to serialize and deserialize your objects.

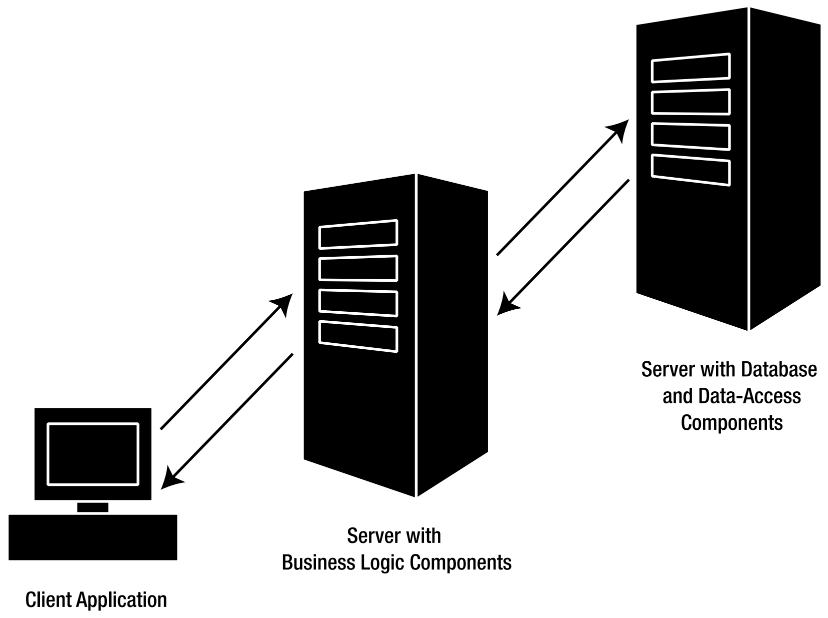


Figure 1-8. *A simple distributed application*

Remoting

In the previous example, we simply assumed that components residing on different machines talk with each other. But how? Remoting is the answer. *.NET remoting* provides an infrastructure for building distributed applications. Though remoting can be used over the Internet, more commonly it is used when the network involved is a local area network (LAN). For Internet-driven communication, web services are more appropriate (see the next section).

You can think of remoting as a replacement for Distributed Component Object Model (DCOM) under .NET. It is clear that remote components must serialize and deserialize data being requested by the client applications. This serialization can be in binary format or in XML format. Moreover, the remoting configuration can be carried by using XML-based configuration files.

Web Services

With the evolution of the Internet, distributed applications are spanning different geographical locations. You may have one server residing in the United States with clients talking to it from India. It is quite possible that the clients and server are running two entirely different platforms (Windows and Unix, for example). In such cases, it is necessary that a standard mode of communication be established between the server and clients so that communication can take place over the Internet. That is where web services come into the picture.

Formally speaking, *web services* are a programmable set of APIs that you can call over a network by using industry-standard protocols: HTTP, XML, and an XML-based protocol called SOAP (as noted earlier in this chapter, *SOAP* stands for *Simple Object Access Protocol*). You can think of a web service as a web-callable component.

Because a web service is supposed to serve cross-platform environments, it relies heavily on XML. HTTP, XML, and SOAP form the pillars of web services architecture. Web services are industry standards, and just like XML, they are standardized by the W3C and hence have massive industry support.

Have a look at Figure 1-8 again. Assume that the three machines involved are connected via the Internet and not a LAN. The components will now be replaced with web services, and they will perform the same jobs as the components did previously. In such cases, the client will call a web service residing on the business logic server, which in turn calls a web service residing on the database server. The requested data is sent back to the client in XML format. It doesn't matter whether the client is a VB6 application, a VB.NET application, or a Java application. Powerful, isn't it? You will explore web services thoroughly in later chapters.

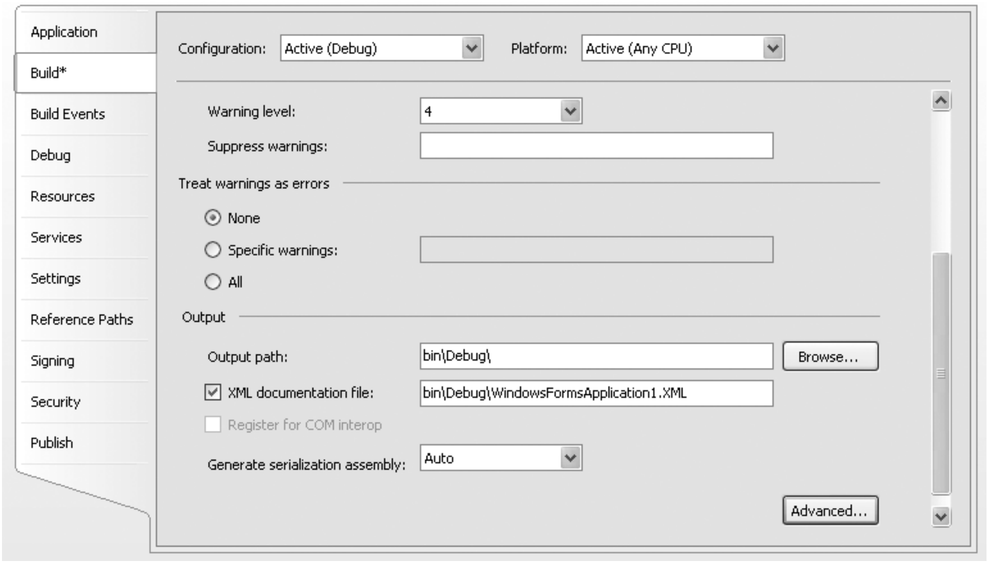
XML Documentation

Everybody knows the importance of well-documented code. However, this important task is often not given proper attention. One of the reasons is that comments left by the developer are not properly captured while creating program documentation or help files. C# as well as Visual Basic .NET support a special commenting syntax that is based on XML. These XML comments can be converted into HTML documentation later. Just to give you a feel for how it works, see the C# code shown in Listing 1-5.

Listing 1-5. *XML Commenting Syntax*

```
/// <summary>
/// This is the starting point.
/// </summary>
/// <param name="args">
/// This parameter receives command line arguments.
/// </param>
static void Main(string[] args)
{
}
```

As you can see, the XML commenting syntax uses three slashes (`///`). The tags such as `<summary>` and `<parameter>` are built-in tags, and I will cover them in detail in subsequent chapters. To generate XML documentation out of this code, you need to view the project settings, as shown in Figure 1-9.



Configuration: Active (Debug)
Platform: Active (Any CPU)
Warning level: 4
Suppress warnings:
Treat warnings as errors
(● None) Specific warnings: All
Output
Output path: bin\Debug\
[✓] XML documentation file: bin\Debug\WindowsFormsApplication1.XML
[ ] Register for COM interop
Generate serialization assembly: Auto

Figure 1-9. *Configuring a project for XML documentation*

Notice the check box titled XML Documentation File. After you select this check box and specify the output path, the compiler generates an XML file, as shown in Listing 1-6.

Listing 1-6. *Resultant XML Comments*

```
<?xml version="1.0"?>
<doc>
   <assembly>
      <name>Parts of XML</name>
   </assembly>
   <members>
      <member name="M:Parts_of_XML.Program.Main(System.String[])">
         <summary>
              This is the starting point.
         </summary>
         <param name="args">
             This parameter receives command line arguments.
         </param>
       </member>
   </members>
</doc>
```

You can now apply an Extensible Stylesheet Language (XSL) style sheet to the preceding XML file to get HTML documentation out of it.

SQL Server XML Features

SQL Server is one of the most powerful database engines used today. Moreover, it is from the creators of the .NET Framework. Naturally, you can expect good XML support in the product.

SQL Server provides some extensions to the `SELECT` statement, such as `FOR XML`, `AUTO`, `EXPLICIT`, `PATH`, and `RAW`, that return the requested data in XML form. The XML data returned by these queries can be retrieved by using the `ExecuteXmlReader()` method of the `SqlCommand` object. Further, Microsoft has released a set of managed classes called SQLXML that facilitate reading, processing, and updating data to and from SQL Server databases in XML format. Finally, SQL Server provides a new data type called `xml` to the standard data types. I will cover these features at length in Chapter 10.

Working with Visual Studio

Throughout the remainder of this book, you will be using Microsoft Visual Studio 2008 for developing various applications. Hence, it is worthwhile to quickly illustrate how Visual Studio can be used to develop Windows and web applications. Note that this section is not intended to give you a detailed understanding of Visual Studio. I will restrict our discussion to the features that you need later in this book.

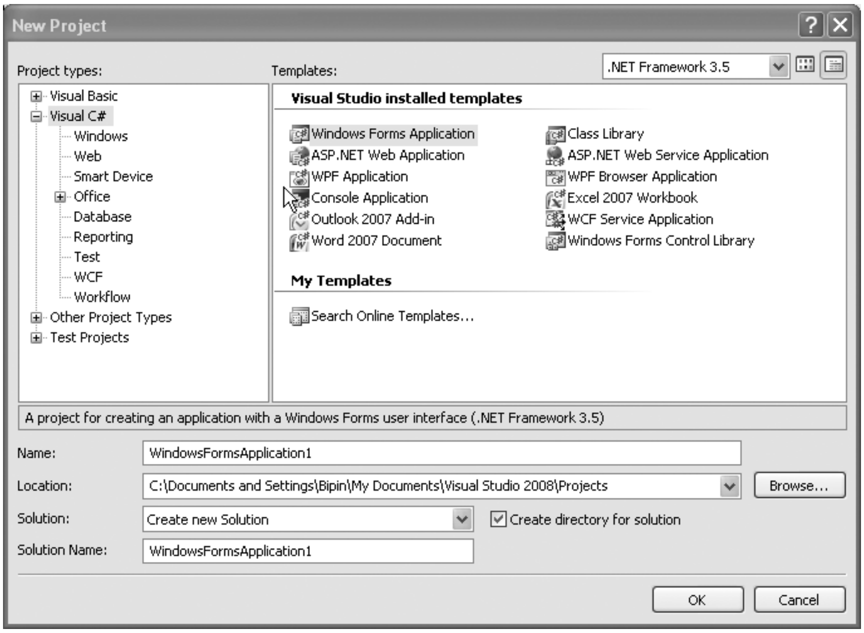
■**Note** Though the examples in this book are developed by using Visual Studio 2008, for most of the Windows Forms examples, you can also use Visual C# 2008 Express Edition. Similarly, for examples related to websites and web services, you can use Visual Web Developer (VWD). Visual C# 2008 Express Edition and Visual Web Developer can be downloaded from Microsoft's website free of charge.

Creating Windows Applications

In this section, you will learn how to create a Windows Forms–based application by using Visual Studio. To create a Windows Forms–based application, you need to create a project of type Windows Application. To begin creating such a project, click File ➤ New Project from the main menu. This opens the New Project dialog box, shown in Figure 1-10.

In the Project Types section, select Visual C#. This will display all the project templates applicable to the C# language. Now, choose Windows Forms Application from the templates. Type **HelloWindowsWorld** for the project name. Also, choose an appropriate location from your disk to store the project files. If you wish, you can also specify a solution name for the Visual Studio solution file. Finally, click the OK button to create the project.

Your Visual Studio IDE should resemble Figure 1-11.



A project for creating an application with a Windows Forms user interface (.NET Framework 3.5)

Name: WindowsFormsApplication1

Location: C:\Documents and Settings\Bipin\My Documents\Visual Studio 2008\Projects

Solution: Create new Solution

[✓] Create directory for solution

Solution Name: WindowsFormsApplication1

Figure 1-10. *Creating a Windows application in Visual Studio*

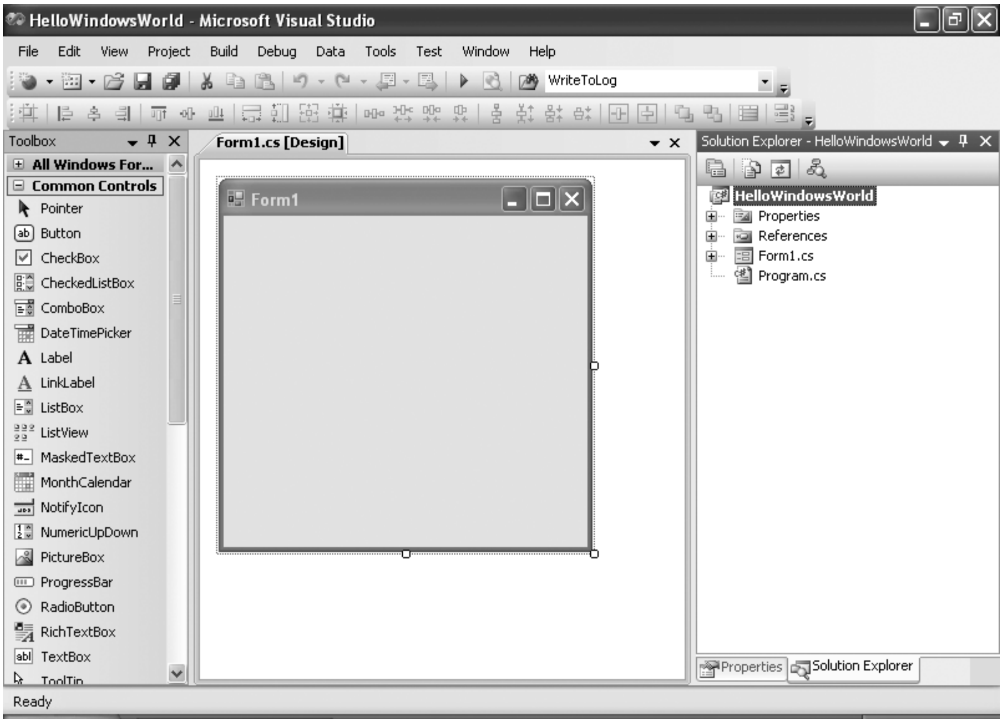


Figure 1-11. *A newly created Windows application in the Visual Studio IDE*

The project contains a single Windows form. You can drag and drop controls from the toolbox onto the form and handle their events. Just to illustrate how this is done, drag and drop a `Button` control onto the form. Open the properties windows by using the View menu and set its Text property to Click Me. Your form should now look similar to Figure 1-12.

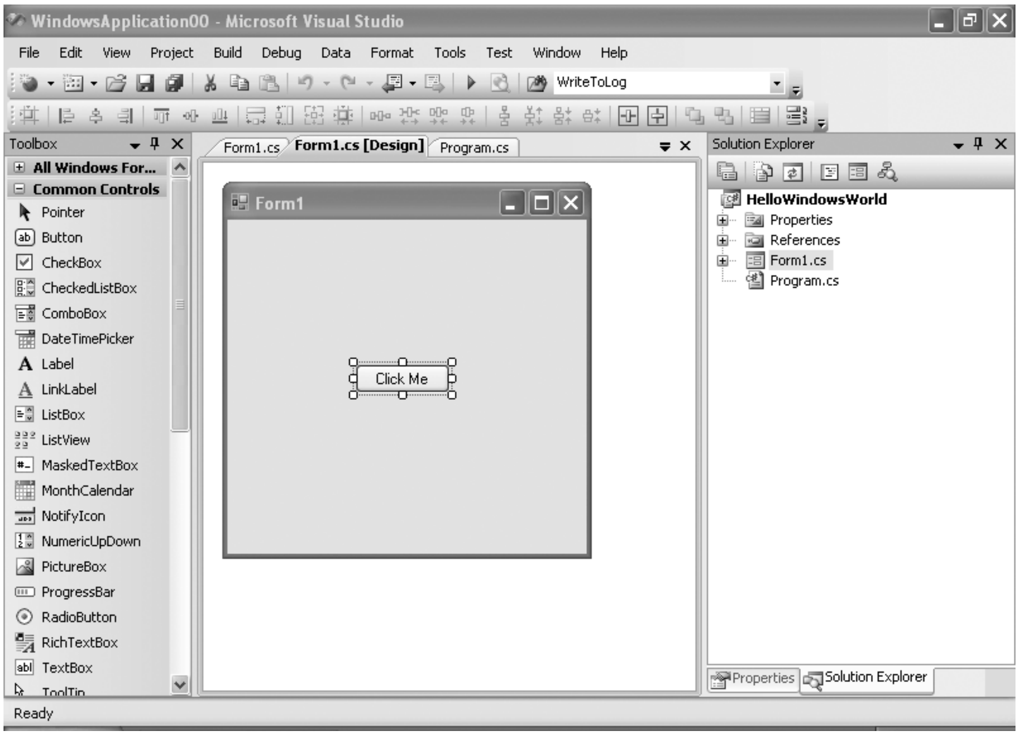


Figure 1-12. *Windows form with a Button control*

Double-click the Click Me button to go into its `Click` event handler. Type in the code shown in Listing 1-7.

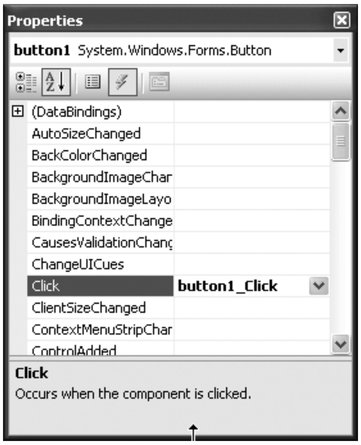
Listing 1-7. *Click Event Handler of the Button Control*

```
private void button1_Click(object sender, EventArgs e)
{
   MessageBox.Show("Hello from Windows Forms");
}
```

The code shows the `Click` event handler of the `Button` control. Notice the signature of the event handler carefully. Throughout the .NET Framework, Microsoft has maintained a uniform signature for event handlers. The first parameter of the event handler gives you the reference of the control (or object in general) that raised the event. The second parameter (often referred to as *event arguments*) supplies more information about the event, if any. The second parameter can be either an instance of the `EventArgs` class directly or of any other class inheriting from the `EventArgs` class.

Note You might be wondering why the `sender` parameter is needed. In the .NET Framework, one method can act as an event handler for multiple controls. For example, you can handle the `Click` event of two `Button` controls by writing just one event handler function. In such cases, the `sender` parameter can be used to identify the control that raised the event.

When you double-click a control, Visual Studio automatically takes you to its default event handler. However, you can wire various events and their handlers manually by using properties windows. Figure 1-13 shows how this is done.


Properties

button1 System.Windows.Forms.Button

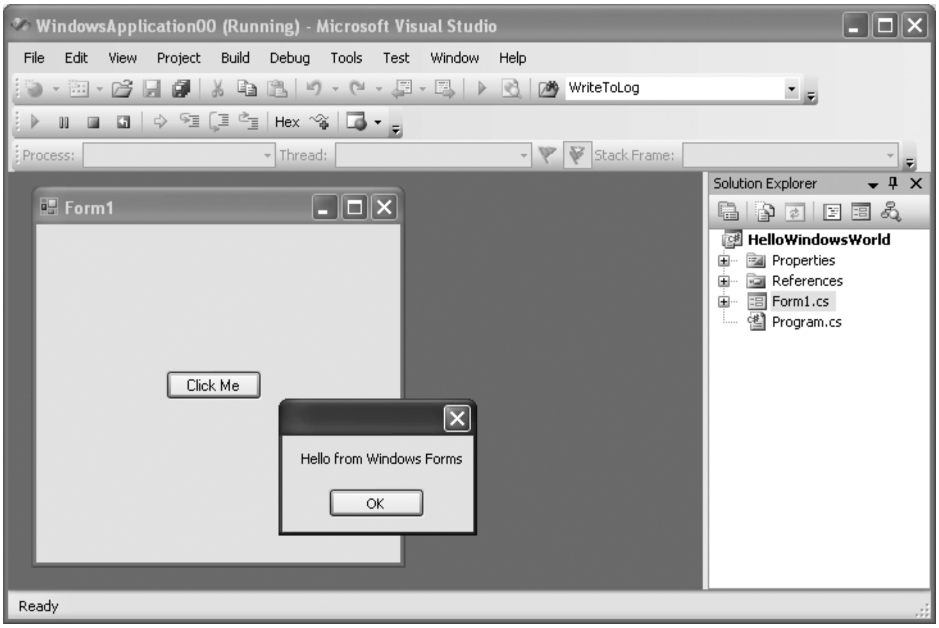
(DataBindings)	
AutoSizeChanged	
BackColorChanged	
BackgroundImageChar	
BackgroundImageLayo	
BindingContextChange	
CausesValidationChanç	
ChangeUICues	
Click	button1_Click
ClientSizeChanged	
ContextMenuStripChar	
ControlAdded	

Click
Occurs when the component is clicked.

Figure 1-13. *Wiring events and their handlers manually*

Inside the event handler, we have used the `MessageBox` class to display a message box. The `Show()` method of the `MessageBox` class has many overloads. We have used the one that accepts a message to be displayed to the user.

Now, use the Build menu to compile the application. Compiling the application will create an executable .NET assembly. Though you can run the `.exe` directly, you may prefer to run the application via the Visual Studio IDE so that you can debug it if required. To run the application, choose Debug ➤ Start Debugging from the menu. Figure 1-14 shows a sample run of the application.


WindowsApplication00 (Running) - Microsoft Visual Studio
File Edit View Project Build Debug Tools Test Window Help
WriteToLog
Hex
Process: Thread: Stack Frame:
Form1
Click Me
Solution Explorer
HelloWindowsWorld
Properties
References
Form1.cs
Program.cs
OK
Ready

Hello from Windows Forms

Figure 1-14. *Sample run of the application*

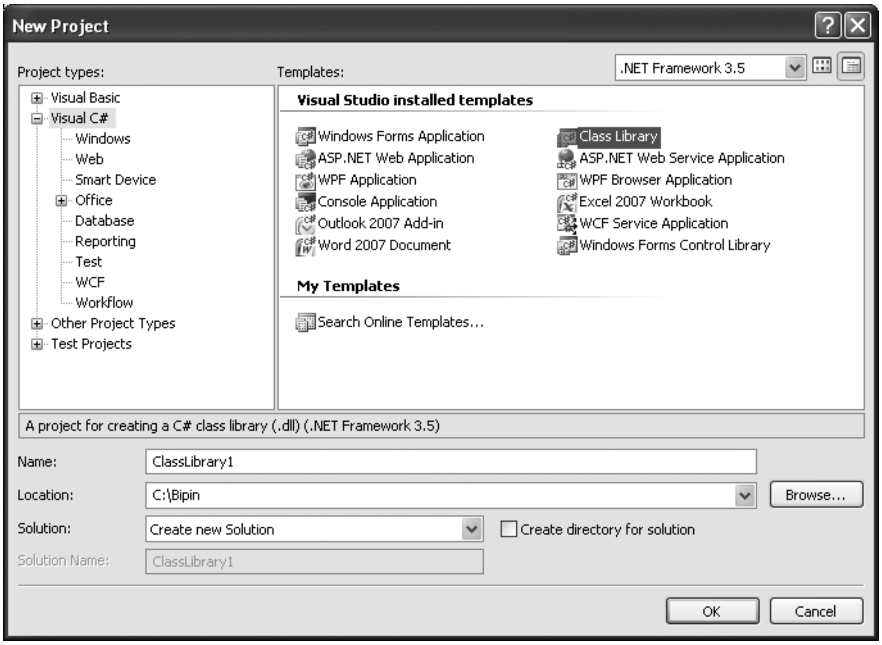
Creating Class Libraries

A project of type Windows Application outputs an `.exe` assembly. Generally, such applications present some type of user interface to the user. However, at times, you need to code functionality and create a component. Such components reside as dynamic link libraries (`.dll` files) and generally do not include any presentation logic. To create dynamic link libraries by using Visual Studio, you need to create a project of type Class Library.

To learn how to create and consume class libraries, you will create a Class Library project. The resultant assembly will be consumed by the Windows application that you developed in the preceding section.

Again, choose File ➤ New Project from the menu to open the New Project dialog box, shown in Figure 1-15.

This time select the Class Library project template and type **HelloWorldLib** for the name. At the bottom of the dialog box, there is a combo box that allows you to add the new solution to the existing solution. Ensure that you choose Add to Solution in this combo box. Finally, click the OK button. Your Visual Studio IDE should now resemble Figure 1-16.



A project for creating a C# class library (.dll) (.NET Framework 3.5)

Name: ClassLibrary1

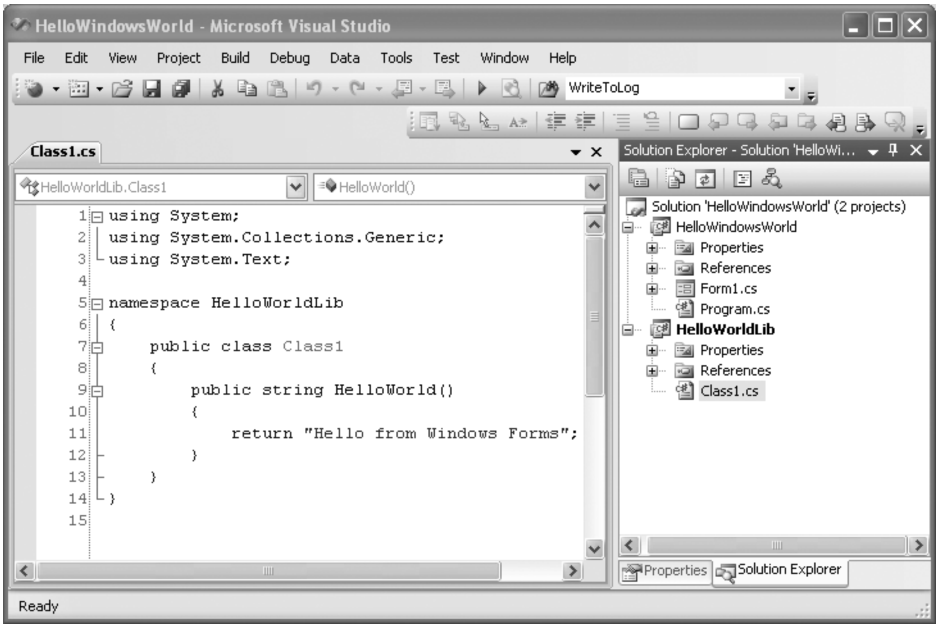
Location: C:\Bipin

Solution: Create new Solution

Create directory for solution

Solution Name: ClassLibrary1

Figure 1-15. *Adding a Class Library project*



```
using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorldLib
{
    public class Class1
    {
        public string HelloWorld()
        {
            return "Hello from Windows Forms";
        }
    }
}
```

Figure 1-16. *The Visual Studio IDE after adding the class library project*

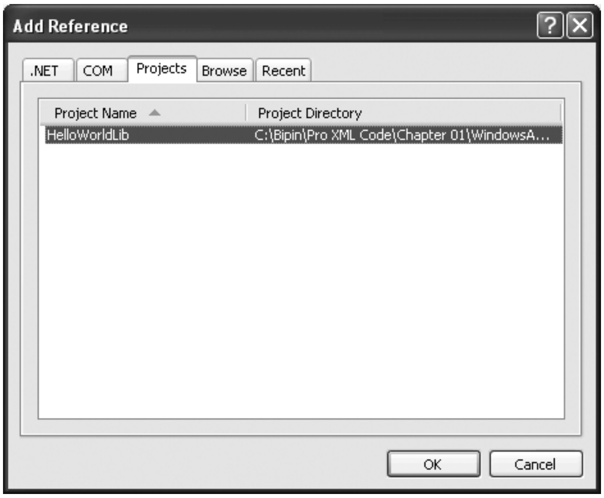
By default, the class library contains one class. You can, of course, add more classes at a later stage if required. Now, add a method named `HelloWorld()` in the class library. The method is shown in Listing 1-8.

Listing 1-8. *HelloWorld() Method*

```
public string HelloWorld()
{
            return "Hello from Windows Forms";
}
```

The method simply returns a string to the caller. Once we compile the class library as outlined before, our class library is ready to be consumed in another application.

Choose Projects ➤ Add Reference from the menu to open the Add Reference dialog box (see Figure 1-17).



Project Name	Project Directory
HelloWorldLib	C:\Bipin\Pro XML Code\Chapter 01\WindowsA...

Figure 1-17. *Adding a reference through the Add Reference dialog box*

This dialog box contains several tabs. The .NET and COM tabs are used to add a reference to built-in .NET Framework assemblies and COM components, respectively. The Projects tab is used to add a reference to another project from the same solution. Finally, the Browse tab can be used to add a reference to assemblies located somewhere on your machine. In our example, you need to add a reference to the `HelloWorldLib` assembly from the Projects tab.

Now, change the code of the Windows application as shown in Listing 1-9.

Listing 1-9. *Modified Code of the Windows Application*

```
using System;
using System.Windows.Forms;
using HelloWorldLib;
```

```
namespace HelloWindowsWorld
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Class1 obj = new Class1();
            MessageBox.Show(obj.HelloWorld());
        }
    }
}
```

Notice the code marked in bold. The code imports the `HelloWorldLib` namespace with the help of a `using` statement. In the `Click` event handler, an object of `Class1` from the `HelloWorldLib` project is created. `HelloWorld()` is then called on the instance and supplied to the `Show()` method of the `MessageBox` class.

If you run the application after modifying the code as shown in Listing 1-9, you should get the same result as before.

■ **Note** You will learn more about website and web service types of projects in Chapters 9 and 11.

Summary

XML is the de facto standard for representing, exchanging, and transporting data across heterogonous systems. All the members of the XML family of technologies (XML, XML Schema, XPath, XSL, and XSLT) are industry standards and hence enjoy massive support from all the leading vendors. Developing cross-platform applications becomes easy with the help of such standards.

Microsoft has harnessed the full potential of XML while developing the .NET Framework. The `System.Xml` and `System.Xml.Linq` namespaces along with several subnamespaces provide dozens of classes that allow you to read, write, and modify XML documents. The configuration files of .NET applications make use of exclusively XML markup. Distributed technologies such as remoting and web services also use XML heavily. The C# and VB.NET languages support XML commenting, which you can use to generate XML documentation for your applications. The .NET Framework also allows you to leverage XML-related features of SQL Server by exposing managed components such as SQLXML.