

Programming the .NET Framework with C#

Eko Suharto

Edited by

Lutfie Royan

1	THE MICROSOFT .NET PLATFORM.....	7
1.1	INTRODUCTION.....	7
1.1.1	Client/Server and Internet.....	7
1.1.2	Interoperability.....	7
1.2	THE ARCHITECTURE.....	7
1.2.1	Visual Studio.NET.....	9
1.2.2	.NET programming languages.....	9
1.2.3	Common Language Specifications (CLS).....	9
1.3	THE .NET FRAMEWORK.....	10
1.3.1	ASP.NET.....	11
1.3.2	Windows Forms.....	11
1.3.3	ADO.NET (Data and XML).....	12
1.3.4	.NET Base Class services.....	12
1.4	THE COMMON LANGUAGE RUNTIME (CLR).....	12
1.4.1	What is in the CLR for programmers?.....	14
1.4.2	The Programming Cycle.....	14
1.4.3	Portable Executables (PE's) explained.....	15
1.4.4	Metadata.....	18
1.4.5	Assemblies.....	20
1.4.6	Intermediate Language IL (Code).....	21
1.4.7	Common Type System (CTS).....	23
1.4.7.1	Value Type.....	23
1.4.7.2	Reference Type.....	23
1.4.7.3	Classes.....	24
1.4.7.4	Properties.....	24
1.4.7.5	Interfaces.....	25
1.4.7.6	Delegates.....	25
1.5	The program execution cycle.....	25
2	PROGRAMMING WITH C#.....	27
2.1	INTRODUCTION.....	27
2.1.1	The language.....	27
2.1.2	This book.....	28
2.2	SETTING UP THE ENVIRONMENT.....	28
2.2.1	The Affordable setup (Free): Microsoft .NET Framework SDK tools.....	28
2.2.2	The Less Affordable setup (costs \$\$\$): Visual Studio .NET 2003.....	33
3	WRITING AND COMPILING PROGRAMS.....	34
3.1.1	Using the .NET framework SDK (simply SDK from now on).....	34
3.1.1.1	The editor.....	34
3.1.1.2	The Compiler.....	36
3.1.1.3	Your IL code.....	37
3.2.1	Visual Studio .NET.....	40
3.2.1.1	Introduction.....	40
3.2.1.2	Launching Visual Studio .NET.....	41
3.2.1.3	Parts of the Visual Studio .NET IDE.....	42
3.2.1.4	Creating a project.....	46
3.2.1.5	Navigating the IDE.....	50
3.2.1.5.1	Window hopping:.....	50
3.2.1.5.2	Menus and toolbars.....	55
3.2.1.6	Using the Code Editor.....	58
3.2.1.7	Compiling and Running programs.....	59
3.2.1.8	Debugging programs.....	60
4	PROGRAM STRUCTURE AND GENERAL CONCEPTS.....	62
4.2	OVERVIEW:.....	62
4.3	PROGRAM STRUCTURE.....	62

4.3.1	Comments:	62
4.3.2	Statement blocks:	63
4.3.3	Namespaces:	63
4.3.4	Types:	64
4.3.5	Class:	64
4.3.6	Members:	64
4.3.7	Inside Main():	65
4.3.8	The <i>System.Console</i> class	65
4.4	NAMESPACES AND “USING” DIRECTIVE	67
4.4.1	Accessing namespaces:	67
4.4.2	Importing namespaces with the <i>using</i> keyword	67
4.4.3	Creating your own namespaces	68
5	DATA TYPES AND VARIABLES	69
5.2	BUILT-IN DATA TYPES	69
5.2.1	Reference Type	69
5.2.2	Value Type	69
5.2.3	Variables and intrinsic Data Types	70
5.2.3.1	Naming Variables	71
5.2.3.2	Categories of Variables	71
5.2.3.3	Declaring and Initializing Variables	72
5.2.3.4	Assigning values to Variables	73
5.3	OPERATORS	74
5.4	EXPRESSIONS AND OPERATOR PRECEDENCE	75
5.5	CREATING USER-DEFINED DATA TYPES	76
5.5.1	Enumeration Type	76
5.5.1.1	Defining an Enumeration Type	76
5.5.1.2	Using an Enumeration Type	76
5.5.1.3	Displaying an Enumeration Value	76
5.5.2	Structure Type	77
5.5.2.1	Defining a Structure Type	77
5.5.2.2	Using a Structure Type	77
5.6	CONVERTING DATA TYPES	78
5.6.1	Implicit data type conversion	78
5.6.2	Explicit data type conversion	78
5.6.3	Boxing	80
5.6.4	Un-Boxing	80
5.7	ARRAYS	82
5.7.1	Declaring Arrays	82
5.7.2	Creating Arrays	82
5.7.3	Initializing Arrays	83
5.7.4	Accessing Arrays	83
5.7.5	Array properties	84
5.7.5.1	Rank	84
5.7.5.2	Length	84
5.7.5.3	Array methods	84
5.7.5.4	GetLength	84
5.7.5.5	IndexOf	85
5.7.5.6	Clear	85
5.7.5.7	Sort	85
5.7.5.8	Reverse	86
6	STATEMENTS AND CONSTRUCTS	87
6.2	VARIABLE DECLARATIONS AND EXPRESSION STATEMENTS	87
6.3	UNCONDITIONAL BRANCHING (JUMP) STATEMENTS	87
6.3.1	The <i>goto</i> Statement	87
6.3.2	The <i>break</i> and <i>continue</i> Statements	87

6.3.3	<i>The return Statements</i>	88
6.4	CONDITIONAL BRANCHING (DECISION-MAKING OR SELECTION) STATEMENTS.....	88
6.4.1	<i>If-else statement</i>	88
6.4.2	<i>Switch statement</i>	89
6.5	LOOPS (ITERATION) STATEMENTS	90
6.5.1	<i>do statement</i>	90
6.5.2	<i>while statement</i>	91
6.5.3	<i>for statement</i>	91
6.5.4	<i>foreach statement</i>	92
6.6	ERROR HANDLING STATEMENTS	93
6.6.1	<i>checked and unchecked statements</i>	93
6.6.2	<i>try-catch statements</i>	94
7	CLASSES	96
7.2	DEFINING CLASSES	96
7.2.1	<i>Class Members</i>	96
7.2.2	<i>Access Modifiers</i>	97
7.3	USING CLASSES	98
8	METHODS	99
8.2	THE MAIN METHOD	100
8.3	GENERIC METHODS AND RETURN TYPES	101
8.3.1	<i>Passing parameters and returning values</i>	101
8.3.2	<i>Returning multiple values</i>	103
8.3.2.1	ref keyword	103
8.3.2.2	out keyword.....	104
8.4	CONSTRUCTORS.....	105
8.5	DESTRUCTORS	106
8.6	OVERLOADING METHODS	107
9	PROPERTIES	108
9.2	PRELUDE	109
9.3	DECLARING PROPERTIES.....	110
9.4	USING PROPERTIES	110
10	INTERFACES	111
10.2	DECLARING INTERFACES	111
10.3	IMPLEMENTING INTERFACES.....	111
10.4	IMPLEMENTING MULTIPLE INTERFACES	113
10.5	EXPLICIT IMPLEMENTATION OF MULTIPLE INTERFACES	114
11	DELEGATES AND EVENT HANDLERS	115
11.2	DECLARING DELEGATES.....	115
11.3	USING DELEGATES.....	115
11.4	EVENTS AND EVENT HANDLERS	119
12	DEVELOPING WINDOWS FORMS PROGRAMS IN C#	127
12.2	OVERVIEW.....	127
12.3	INTRODUCING WINDOWS FORMS.....	127
12.4	SIMPLE WINDOWS FORMS APPLICATION DEVELOPMENT.....	128
12.4.1	<i>The System.Windows.Forms Namespace</i>	128
12.4.2	<i>Application Class</i>	128
12.4.3	<i>Form Class</i>	129
12.4.4	<i>Event Handlers</i>	130
12.5	WINDOWS FORMS CONTROLS.....	131
12.6	USING VS.NET TO DEVELOP WINDOWS FORMS.....	134

12.6.1	Creating a project	134
12.6.2	Adding a Form	135
12.6.3	The code behind the form	135
12.6.4	Modifying properties of a Form	137
12.6.4.1	Modifying The Name property of a Form	138
12.6.4.2	Modifying The Text property of a Form	138
12.6.5	Adding event handler to the Form.....	138
12.6.5.1	Adding a default event handler to a Form	138
12.6.5.2	Adding an event handler to a Form	140
12.6.6	Adding controls to a Form	141
12.6.6.1	A simple application.....	142
12.6.6.2	A more useful application.....	143
12.6.6.3	Handling multiple events using a single handler	145
13	DEVELOPING DATA-AWARE PROGRAMS IN C# (ADO.NET)	147
13.1	OVERVIEW.....	147
13.2	ADO.NET ARCHITECTURE	147
13.2.1	.NET Data Providers.....	147
13.2.1.1	Connection Object.....	148
13.2.1.2	Command object.....	149
13.2.1.3	DataReader object	149
13.2.1.4	DataAdapter object.....	149
13.2.2	Dataset:	150
13.2.2.1	DataTable object.....	152
13.2.2.2	DataColumn object.....	153
13.2.2.3	DataRow object	153
13.2.2.4	DataRow object	153
13.2.2.5	DataRelation object	153
13.3	ACCESSING DATA IN ADO.NET	154
13.3.1	The Connection object.....	154
13.3.1.1	The SQL Server connection Object	155
13.3.1.2	The OLE DB connection Object.....	156
13.3.1.3	The ODBC .NET connection Object	157
13.3.1.4	The Oracle .NET connection Object.....	157
13.3.1.5	Using the connection Object.....	157
13.3.2	The Command Object.....	158
13.3.2.1	Creating a Command Object	158
13.3.2.2	Executing a Command Object	159
13.3.2.2.1	CommandObject.ExecuteNonQuery()	159
13.3.2.2.2	CommandObject.ExecuteScalar()	161
13.3.2.2.3	CommandObject.ExecuteReader()	161
13.3.3	The DataReader Object.....	162
13.3.3.1	Creating a DataReader Object	162
13.3.3.2	Using a DataReader Object	162
13.3.4	The DataAdapter Object	164
13.3.4.1	Creating a DataAdapter Object.....	164
13.3.4.2	Using a DataAdapter Object.....	166
13.3.5	The DataSet Object	166
13.3.5.1	Creating untyped DataSets	167
13.3.5.2	Using DataSets	167
13.3.6	What is next	171
14	DEVELOPING WEB FORMS PROGRAMS IN C# (ASP.NET).....	172
14.1	OVERVIEW.....	172
14.2	THE ENVIRONMENT	172
14.3	THE ASP.NET'S PROGRAMMING MODEL.....	173
14.3.1	A bit of history.....	173
14.3.2	The Infrastructure	173
14.3.3	The Components.....	173

14.3.4	<i>The Process</i>	174
14.4	THE ASPX PAGE	174
14.4.1	<i>A typical ASPX page</i>	174
14.4.1.1	Directives	176
14.4.1.2	Comments.....	176
14.4.1.3	HTML tags	176
14.4.1.4	Code Declaration Blocks	176
14.4.1.5	HTML Server controls	176
14.4.1.6	Render Blocks	176
14.4.1.7	Render Blocks	176
14.4.2	<i>The code behind</i>	176
14.4.2.1	Event handling.....	178
14.5	WEB FORMS AND SERVER CONTROLS	180
14.5.1	<i>Server Controls</i>	181
14.6	USING VS.NET TO DEVELOP WEB FORMS	186
14.6.1	<i>Creating a project</i>	186
14.6.2	<i>Adding a Web Form</i>	187
14.6.3	<i>Working with page properties</i>	187
14.6.3.1	Modifying the Title property of a page.....	188
14.6.3.2	Modifying the Page Layout.....	188
14.6.4	<i>The code behind the form</i>	188
14.6.5	<i>Running an application</i>	190
14.6.6	<i>Adding an event handler to a Form</i>	191
14.6.7	<i>Adding controls to a Form</i>	191
14.6.7.1	A simple application.....	192
14.6.7.2	A more useful application.....	194

1 The Microsoft .NET Platform

1.1 Introduction

The Microsoft's next big thing, the .NET was revealed at the Developers Conference in Orlando, FL, August 2000.

As an evolutionary architecture from the Distributed iNternet Architecture (DNA), developed in the 1990s, the .NET aims to address the following issues:

1.1.1 Client/Server and Internet

Client/Server environments have grown to cross boundaries, and networks, utilizing technologies such as the internet, and new standards such HTTP, SOAP and XML.

Microsoft also has a technology called Remoting, to facilitate the development of distributed Client/Server applications.

1.1.2 Interoperability

Integration and interoperability between various software components is increasingly becoming a major issue, in heterogeneous and highly distributed computing environments.

1.2 The architecture

The Microsoft .NET Platform consists of four main components, or product sets, most of which have been around for quite some time. The big picture is shown in figure 1.1

The well known Visual Studio has been re-written to support the new .NET platform, and is some times considered as the fifth major component in the .NET architecture. There are two releases of the product, 2002 or 7.0, and the latest version which is VS.NET 2003.

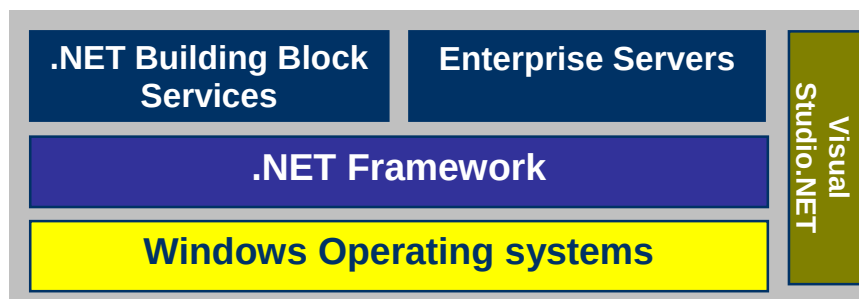


Figure 1.1 the .NET architecture components

The four major .NET product sets are:

- .NET Framework: Is the heart of the new architecture, comprising of
 - Common language runtime (Discussed later, at length)
 - Base Classes (covered later)
 - ADO.NET (for all Data and XML access)
 - ASP.NET (the web based development infrastructure)
 - Windows forms (For all windows based GUI programming)
- Building Block Services (Web Services)
- .NET Enterprise Servers: Although separate set of products, they represent an integral part of the Microsoft .NET vision, and form the robust infrastructure which the .NET builds upon.
 - Application Center 2000/2
 - BizTalk Server 2000/4
 - Commerce Server 2000/2
 - Exchange Server 2000/3
 - Host Integration Server 2000/3
 - Internet Security and Acceleration Server 2000
 - SQL Server 2000
- Windows operating system: Obviously the operating system of choice for the Microsoft .NET is no other that the Microsoft Windows:
 - Server 2000/2003
 - XP/XP Media Center
 - 2000 Pro
 - Me and 98

The figure 1.2 depicts a high level representation of the .NET platform, highlighting the .NET framework.

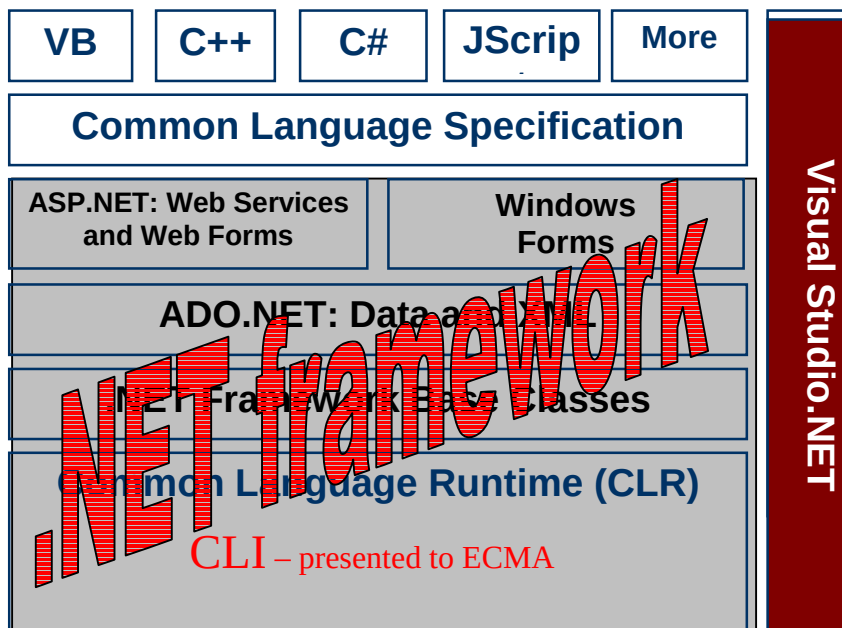


Figure 1.2 the .NET platform components

Let us take a close look at the figure above:

1.2.1 Visual Studio.NET

On the right side (vertically) we have Visual Studio (Developer Studio), although not part of the framework, is a sophisticated set of development tools, including compilers, a debugger, visual design tools, explorers and database connectivity tools.

With technologies from earlier releases of Visual studio, Interdev, the .NET framework SDK and several new innovations, VS.NET is the best development tool for the .NET platform.

BUT you don't really need VS.NET to be able to develop for the .NET platform, if you don't feel like spending the few hundred dollars that the product will cost you. .NET development tools are FREE.

The complete SDK containing samples, documentation, compilers and many more tools can be downloaded for free at:

<http://msdn.microsoft.com/netframework/technologyinfo/howtoget/default.aspx>

1.2.2 .NET programming languages

The top row of the figure 1.2 lists C++, VB.NET, C# and Jscript as the programming languages, that are, incidentally, supported by VS.NET. But there are over 20 languages that are supported by the .NET framework. (APL, COBOL, EIFFEL, HASKELL, ML, OBERON, PASCAL, PERL, PYTHON, SMALLTALK and several others).

Each programming language has its own compiler that conforms to the .NET standard as laid out and enforced by the Common Language Specification (CLS).

1.2.3 Common Language Specifications (CLS)

Since all languages targeting the .NET framework share the same infrastructure, then they have to conform to a strict standard. The least common denominator to which all .NET languages must adhere is published by Microsoft and is called the Common Language Specifications (CLS). To learn more about the CLS, refer to the ECMA documents, and other .NET SDK documents listed in section 1.4.

The CLS is presented in figure 1.2 right after the list of languages.

1.3 The .NET framework

The core component of the .NET platform is the .NET Framework, which we will discuss in more detail, as it is the most important component, as far as we programmers are concerned.

Figure 1.3 is a more elaborate listing of the .NET framework components, showing the same elements as Figure 1.2.

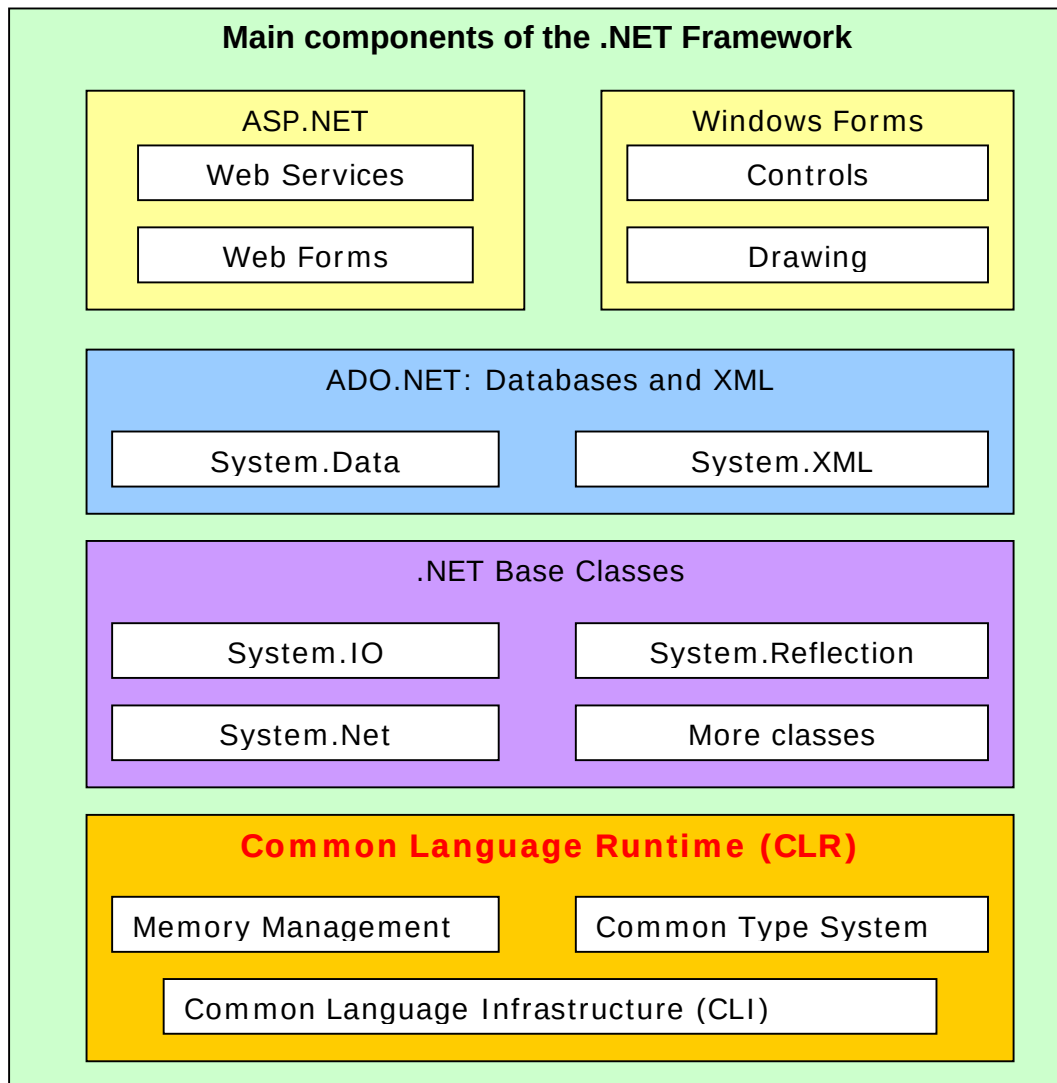


Figure 1.3 the .NET framework components (a closer look)

You may notice that the .NET Framework components are broken into five major blocks, for simplicity. Now let's examine each block, from the top down.

1.3.1 ASP.NET

ASP.NET is a new framework for web-based development, which evolved from Active Server Pages (ASP), into a complete server side programming platform using compiled languages such as C# and VB.NET.

ASP.NET introduces a number of new concepts, such as web forms, which is a programming library similar to the windows GUI development, enabling drag and drop building of web based applications.

ASP.NET also introduces the concept of code-behind, where user interface is just that, and the logic is hidden in logic files, or components (behind the UI) Databinding is another feature of the ASP.NET, where you can bind the UI controls to a data source, in a similar fashion to the windows based controls. But the most important innovation by far, in ASP.NET is the implementation of the XML Web services, which allows access to COM like components over the Internet, using standard HTTP protocol.

Debugging is added to ASP.NET to enable you debug your code in run time, in order of execution. No need to insert those annoying output lines to trace output. You would know what I mean if you ever did any web scripting. If not, then let's say that you have been spared some tedious task.

One last improvement in ASP.NET concerns the deployment of web applications, where you could deploy an ASP.NET application, without the need to reset the server. This is a real big deal if you were a system administrator for a web server running several other applications.

1.3.2 Windows Forms

Windows forms provide the functionality necessary to develop windows applications. Windows Forms was originally developed for the Windows Foundations Classes (WFC) that shipped with the Microsoft's own version of JAVA called J++ , which is no more supported after the SUN/Microsoft row and subsequent lawsuits.

Windows forms essentially wrap the win32 APIs, which C++ programmers are familiar with, and provide it to all CLR languages through a set of classes exposed by the Foundation Classes, we touched upon earlier. Unlike the Microsoft Foundation Class Library (MFC) and Active Template Library (ATL), which were the evolutionary steps, the .NET framework's windows forms provide a consistent VB-like integrated development environment to C#, Managed C++, and other languages.

Windows Forms fully support and integrate with, Web Services, ADO.NET, and the .NET classes. You can interface the components developed to access databases, and other backend servers using the same approach as in the ASP.NET model.

You can also have Windows Forms application acting as a user interface to a remote web service over the Web.

The Windows forms are contained in two main name spaces

The System.Windows.Forms Namespace, which provides most of the user interface functionality through a host of classes. The most important class in this family is the "Control" class, which would be written in your code as:

“System.Windows.Forms.Control”. If a control contains other controls, then it is a `ContainerControl`. We will discuss the details of the namespace and its classes in the Windows Forms chapter.

The `System.Drawing` Namespace provides access to the `Bitmap` and `Graphics` classes, also known as GDI+. They perform tasks such as painting an image file, manipulating pixels, and so on. We will explore the `System.Drawing` namespace and its classes in the Windows Forms chapter.

1.3.3 ADO.NET (Data and XML)

Active Data Objects (ADO) is a database technology that's built on top of Object Linking and Embedding for Databases (OLE-DB). Overall, ADO is a great technology that makes database development a lot easier. One of the most powerful features of ADO.NET is the idea of a disconnected Dataset, and the other is its support for both OLEDB (RDBMS) and XML, as a data sources.

1.3.4 .NET Base Class services

Base Class services, also known as the Base Class Library (BCL), is a set of classes that was previously contained in the class libraries, and provides a wide range of functionality. All classes in the .NET framework inherit from the base classes.

Base classes are grouped into namespaces, which makes the class names globally unique, by way of hierarchical name structure.

For example the “System” namespace contains the most fundamental of the classes, such as the data types, from which all other classes inherit.

The `System.Data` namespace consists of classes that constitute the ADO.NET object model, which contains the database services.

The `System.Net` namespace provides Transmission Control Protocol/Internet Protocol (TCP/IP) and sockets support. And the list goes on.

You can learn more about the BCL namespaces through the .NET framework SDK documentation.

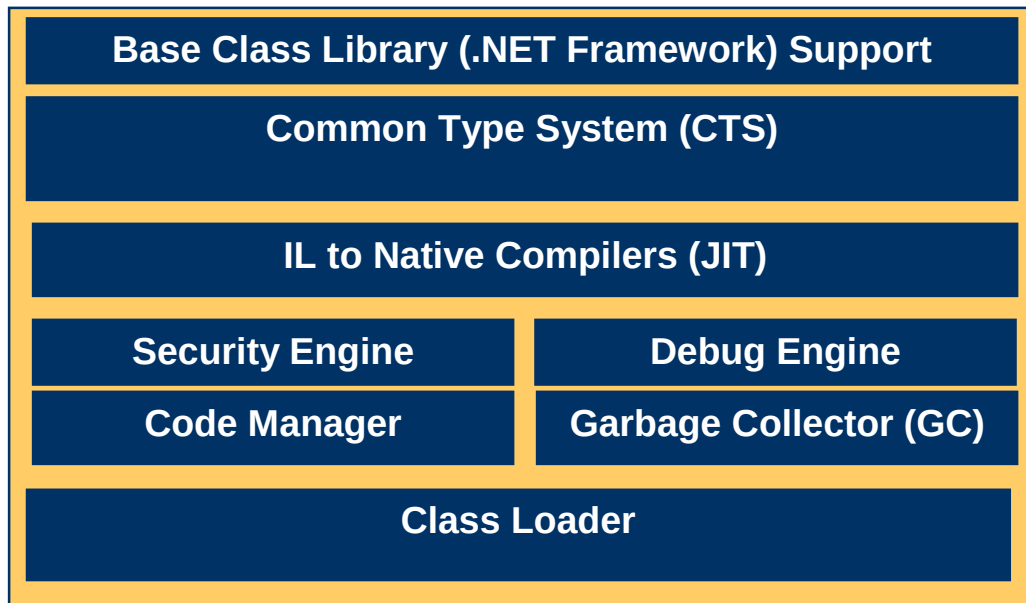
1.4 The common Language Runtime (CLR)

The most important component of the Framework is something called the CLR.

If you are familiar with Java, think of the CLR as the .NET equivalent of the Java Virtual Machine (JVM).

For the rest of us, the CLR is the core component of the .NET architecture.

The CLR essentially brings objects to life, performs security checks on them, assigns them memory, executes them, and garbage-collects them. The figure 1.3 depicts a high level snapshot of the various CLR components



The CLR and the JVM are similar in that they are both runtime infrastructures, which abstract the underlying platform. However, there are several differences between the two: The JVM currently supports only the Java language, while the CLR supports over 20 languages (as of the time we wrote this). And potentially many more languages, that can be compiled into the Common Intermediate Language (CIL) or MSIL.

The JVM executes bytecode, which is interpreted on each platform. Whereas the IL is compiled, making it faster and more efficient. Another conceptual difference between the two infrastructures was that Java code runs on multiple platforms with a JVM, whereas .NET code runs only on the Windows platforms with the CLR. But support for several operating systems has been created. Most notably, the Free BSD, and the MAC OS X versions, which were developed under Microsoft's Shared CLI source initiative. And can be downloaded from:
<http://www.microsoft.com/downloads/details.aspx?displaylang=en&familyid=3A1C93FA-7462-47D0-8E56-8DD34C6292F0>

You can learn more about the shared source initiative at:
<http://www.microsoft.com/resources/sharedsource/default.mspix>

The Common Language Infrastructure (CLI), which is a functional subset of the CLR, was submitted by Microsoft, to ECMA for adoption as an international standard, co-sponsored by several industry heavyweights, such as HP and Intel.

<http://msdn.microsoft.com/net/ecma/>

In this chapter, we discuss the CLR environment, Portable Executable files (PE), metadata, assemblies, manifests, the CTS, and the CLS.

1.4.1 What is in the CLR for programmers?

From a programmer's point of view, the CLR provides the basic class libraries (BCL), which are comparable -but far more superior to - C runtime library, the C++ standard template library (STL), the MFC library, the Active Template Library (ATL), the Visual Basic® runtime library, or the Java virtual machine.

The BCL contains hundreds of library classes, organized into groups, called namespaces, and accessed by the various programming languages.

All languages have equal access to the CLR features, and it is up-to the different languages to allow or disallow access to these features.

A typical example would be that C# gives programmers access to pointers and threads, which are supported by the CLR. On the other hand Visual Basic is designed for less involved developers, hence preventing access to things such as threads.

Another example is the C++ vs C#, where C++ supports both unmanaged and CLR managed code, which means that the code is run in the .NET environment. C# on the other hand runs only in a managed environment, since it was built specifically for the .NET framework.

There are several programming language compilers including compilers for APL, CAML, Cobol, Haskell, Mercury, ML, Oberon, Oz, Pascal, Perl, Python, Scheme, and Smalltalk, to mention a few. So whichever language you are most comfortable coding with, you should be able to port your skills to the .NET platform.

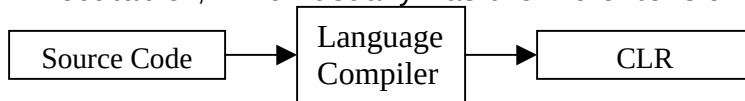
Some of the problems solved by the CLR include:

1. Runtime libraries: Take a scenario where a program written in VB 6.0, that when deployed on a machine that doesn't have the VB runtime library VBRUN60.DLL. You'd have to download a large DLL from somewhere to get the system to work. With the CLR, all runtime files are part of the environment.
2. Take the same scenario as in (1) above, but where the Runtime library that is on the system is different from the one you used to compile the program, and when you overwrite the existing one, other programs fail to operate (Ever heard of the DLL hell?!)

1.4.2 The Programming Cycle

The steps involved in developing for the .NET framework are similar to the traditional environments.

The obvious being, you write a program “Source” in the language of your choice, and then you would compile your “Source” code to produce an “Executable”, which usually has the file extension of .EXE or .DLL.



The “Executable” files are loaded into the system memory for execution by the CLR. We will examine the CLR in greater detail, at a later stage.

So far the steps involved are the same as what you would have done in the past, except, in this case the “Executable” - also known as “Portable Executable” or “PE”- has a few differences:

1. The code in the “PE” is not an x86 machine-code, but rather an Intermediate Language “IL” or MSIL code.
2. Included in the “PE” file is “Metadata”, which is used by the common language runtime to locate and load class types in the file, allocate memory to object instances, resolve method invocations, translate MSIL to native code, enforce security, and many other tasks.
3. A third difference is that the final product is not just EXEs or DLLs. Rather, the unit of reuse and deployment in .NET is an assembly. You may end up producing a single-file assembly or even a multi-file assembly. The single-file assembly carries an .exe or .dll extension, but your assembly may contain several reusable components, that are logically grouped to form your application. Examples of multi-file assembly may include binary DLLs, resource files, such as GIF/JPG image files, text files such as data and configuration, and so on.

We'll skip the source code and compiler parts, which we will cover in later chapters, and take a look at the PEs and assemblies.

1.4.3 Portable Executables (PE's) explained

PE's are a derivative of the Common Object File Format (COFF), which is a publicly available specification. You can learn more about standard PE at:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndebug/html/msdn_peeringpe.asp

You can learn more about COFF at:

<http://osr5doc.ca.caldera.com:457/topics/COFF.html>

Standard Windows PE files are divided into two major sections. The first section includes the PE/COFF headers that reference the contents within the PE file. In addition to the header section, the PE file holds a number of native image sections, including the .data, .rdata, .rsrc, and .text sections. These are the standard sections of a typical PE file, but Microsoft's C/C++ compiler allows you to add your own custom sections into the PE file.

For example, you can create your own data section to hold encrypted data that only you can read. Utilizing this feature, Microsoft has added new sections to the standard PE file to support the CLR's functionality. The CLR understands and manages the new sections, which contain – you guessed it - metadata and IL code which determine how the program will be executed.

PE File structure

PE/COFF Header		
CLR Header		
CLR Data		
Metadata	IL-Code	
Native language sections (DOS, Windows)		

To view the PE structure, you can use the *dumpbin.exe* utility, which dumps the content of a Windows executable in readable text. *dumpbin.exe* can be found in “\Program Files\Microsoft Visual Studio .NET 2003\Vc7\bin”.

Let us take the following C# source code, saved as test.cs (don't worry about the details of the source, as we'll explain them later):

```

Namespace test
{
    class test
    {
        static void Main()
        {
            System.Console.WriteLine(“Hello Universe”);
        }
    }
}

```

Compiling the above source code will produce test.exe, if you type:
> *csc test.cs*

Then, running the following command on the command prompt:
> *dumpbin.exe test.exe /all*

Generates the following data. (Partial dump is displayed)

Microsoft (R) COFF/PE Dumper Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file test.exe

PE signature found

File Type: EXECUTABLE IMAGE

```

FILE HEADER VALUES
    14C machine (x86)      ← MS-DOS/COFF HEADERS
    3 number of sections
...

OPTIONAL HEADER VALUES
    10B magic # (PE32)    ← WINDOWS PE HEADER
    7.10 linker version
...
SECTION HEADER #1        ← SECTION DATA
    .text name
...
Code
Execute Read

RAW DATA #1
00401000: C4 19 00 00 AE 19 00 00 9E 19 00 00 84 19 00 00  Ä...®.....
...
clr Header:              ← CLR Header area

    48 cb
    2.00 runtime version
...
Section contains the following imports:

    mscoree.dll
        402000 Import Address Table
        4022AC Import Name Table
        0 time date stamp
        0 Index of first forwarder reference

    0 _CorExeMain
...

```

As shown in the dump above, it is the section **SECTION HEADER #1** that stores the CLR header and data. Of particular interest is the line (0 _CorExeMain) in the dump. This is a function implemented in *mscoree.dll*, and is invoked by the windows loader, if the CLR header exists in PE file. Once loaded this jumps the CLR, which then calls your code's entry point – such as the Main () function – and takes over the execution and management of the code.

The MSIL instructions cannot be executed directly by the CPU. So, the CLR engine first compiles the managed IL code into native CPU instructions.

Now that you are familiar with what goes into the PE file, let's examine the metadata and IL sections of the PE file.

1.4.4 Metadata

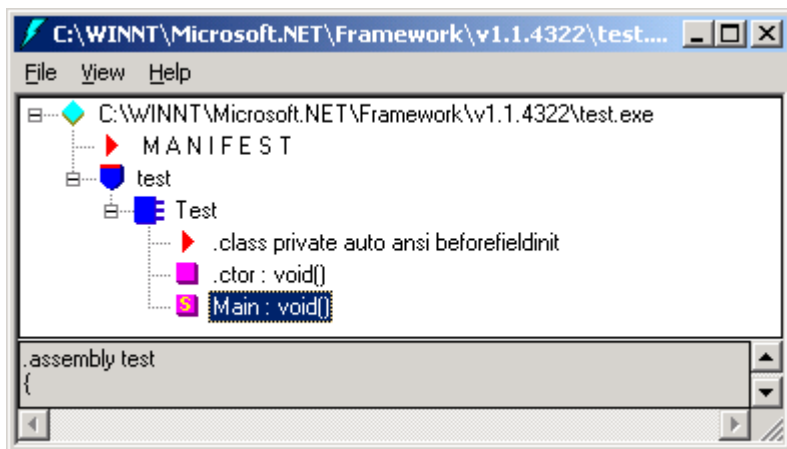
Dubbed as "data about data"; Metadata contains type definitions, version information, Security information and external assembly references.

If you have developed in C++, then you probably used IDL files, to define interfaces, and if you have a VB development background, then you probably used type libraries, which was a universalized interface, to allow COM enabled languages to access shared components in a uniform manner. Metadata is no more than an enhanced type library.

Much richer than a type library, metadata includes descriptions of an assembly and modules, classes, interfaces, methods, properties, fields, events, global methods, and much more.

At this point, we introduce an important .NET tool, the IL disassembler (ildasm.exe), which allows you to view both the metadata and IL code within a given .NET PE file.

For example, if you execute ildasm.exe and open the test.exe .NET PE file that you built earlier in this chapter, you will see something similar to the figure below:



The ildasm.exe tool displays the metadata for your .NET PE file in a tree view, so that you can easily drill down from the assembly, to the classes, to the methods, and so on. To get full details on the contents of a .NET PE file, you can select the Dump command from the File menu or press Ctrl-D to dump the contents out into a text file.

Here's an example of an ildasm.exe dump, showing partial contents:

```
// Microsoft (R) .NET Framework IL Disassembler. Version 1.1.4322.573
// Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
.assembly extern mscorlib           ← External assembly mscorlib is referenced by this assembly
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
}
.assembly test                     ← Our assembly, named test is described by this block
{
    .hash algorithm 0x00008004
}
.module test.exe                   ← Our PE file, named test.EXE
```

```

...
.namespace test                                ← Our namespace called test
{
    .class private auto ansi beforefieldinit Test extends [mscorlib]System.Object ← Our class name, extends System.Object (added automatically)
    {
        .method private hidebysig              ← The Main() entry point, "cil managed" added by the system.
            static void Main() cil managed
            {
                .entrypoint
                .maxstack 1
                IL_0000: ldstr    "Hello Universe"      ← The actual code revealed here
                IL_0005: call     void [mscorlib]System.Console::WriteLine(string)
                IL_000a: ret
            } // end of method Test::Main
        .method public hidebysig specialname rtspecialname ← ctor() is a generic constructor for Main()
            instance void .ctor() cil managed
            {
                // Code size 7 (0x7)
                .maxstack 1
                IL_0000: ldarg.0
                IL_0001: call     instance void [mscorlib]System.Object::.ctor()
                IL_0006: ret
            } // end of method Test::.ctor
        } // end of class Test
    } // end of namespace test
}

```

As you can see from the above example, we can use certain tools, such as ildasm.exe to inspect the PE's Metadata. This raises two interesting points:

1. The source code is readily accessible, by anyone with the right knowledge and tools. In desktop deployment scenarios, to protect your property, you have to apply some form of encryption. The best case for .NET based systems obviously, is a server based system, which is accessed via a thin client, such as a web browser.
2. You can develop some exciting products, by piggy-bagging on existing assemblies (discussed in the next section), but you need programmatic access to the Metadata of the assemblies. This is readily available through a set of system API called the *Reflection API*, and is accessible using the System.Reflection namespace.

The code sample below demonstrates the use of the Reflection APIs.

```

using System;
using System.IO;
using System.Reflection;
public class RefAPI
{
    public static int Main( )
    {
        // Load the assembly.
        Assembly assem = Assembly.LoadFrom("test.exe");
        // Get all the modules.
        Module[] mods = assem.GetModules( );
        // Get all the types in the first module.
        Type[] types = mods[0].GetTypes( );
        // Inspect the first type.
    }
}

```

```

        Type type = types[0];
        Console.WriteLine("Type [{0}] has these methods:", type.Name);
        // Inspect the methods supported by the first type.
        MethodInfo[] methods = type.GetMethods( );
        foreach ( MethodInfo mInfo in methods )
        {
            Console.WriteLine(" {0}", mInfo);
        }
        return 0;
    }
}

```

If you execute the above code, you will get the following output:

Type [RefAPI] has these methods:	
Int32 GetHashCode()	← Added by the system
Boolean Equals(System.Object)	← Added by the system
System.String ToString()	← Added by the system
Int32 Main()	← The only method we've declared Main()
System.Type GetType()	← Added by the system

Although we only have one method Main(), in the source above, the assembly denotes four more methods, because test inherits these methods from System.Object, which is the root of all classes in .NET.

One may argue that the above information could be obtained from the type libraries, with a little more effort! Yes. But you couldn't create COM objects dynamically then.

Reflection APIs introduce a uniquely .NET feature, that enables you to create .NET assembly dynamically at runtime. This feature is encapsulated in System.Reflection.Emit classes.

1.4.5 Assemblies

Assemblies are the means of packaging and deploying applications and components in .NET. Just like a compiled application or component today, assemblies can be made up of either single or multiple files. An assembly contains metadata information (covered in the previous section), which is used by the CLR for every-thing from type checking and security to invoking the methods.

An assembly is a set of files (also 1:1 with modules) containing Common Intermediate Language (CIL) code and metadata. It serves as the primary unit of a software component in the .NET environment.

Security, versioning and type resolution are defined per assembly.

In the JAVA world, the closest thing to an assembly would be a JAR file.

A manifest describes information about the assembly itself, such as its version, which files make up the assembly (if more than one), which types are exported from this assembly, and the optional security information such as a digital signature and public key of the manifest itself.

Consider the example in section 1.4.4 that displays the content of test.exe, which is a single file assembly. The sections starting with .assembly in the disassembled file are known collectively as the “Manifest” of the assembly. In this particular case, and similar cases where the application comprises of a single file, PE and assembly become the same. In case the application utilizes more than one file, then it becomes a multi-file assembly.

Any external files referenced by the application are also listed. Note the first line of code in the test.exe dump.

.assembly extern mscorlib

← External assembly mscorlib is referenced by this assembly

Where the external assembly mscorlib is listed, to tell the system what file(s) to look for.

An assembly that is shared between multiple applications also has a *shared name* (also known as a *strong name*). This is a key pair containing a globally unique name (Similar to GUID from COM) as well as an encrypted digital signature to prevent tampering.

The key pair information is optional and may not be in a component's manifest if it was not meant to be a shared component.

In summary: the assembly is the main unit of deployment, security and versioning. It could be the same as the PE in case of one file assembly. The Metadata part that describes the assemblies is known as the “**Manifest**”.

Now that we are comfortable with the Metadata, let's take a look at the other important component of the PEs, namely IL or MSIL.

1.4.6 Intermediate Language IL (Code)

The IL is an intermediate code, which is lower level than the source, but higher than machine code. It is somewhat comparable to the JAVA bytecode or the Pascal-P' s pcode. You could also think of it as comparable to the x86 or 6502 Assembly language instruction set, which is assembled, using an “Assembler”, and then executed by the respective processor.

To understand what IL is about, we need to be acquainted with the Common Language Infrastructure (CLI), which is the base upon which all languages build their code. You can think of CLI as the MEGA dictionary, which contains the definitions of the Intermediate Language IL, or MSIL. Hence we will use CLI and IL interchangeably for the purpose of this section, albeit there is some difference.

The CLI has about 220 instructions, and can be categorized into several major groups:

Constants, arguments, local variables, and pointers

Several instructions to manipulate many types of arguments, local variables, and pointers, push them onto the stack, and pop them off the stack.

Arithmetic

The usual arithmetic operations such as ADD, are provided by the CLI.

Simple control flow

The CLI supports the usual variety of (conditional) branch instructions. No unconditional JMP is provided.

Simple control flow

Can be broken down to Value Type and Reference Type

Value Type: includes structs, unions, and enums. Which are all similar to their respective types in the C and C++ languages.

Reference Type: Otherwise known as Objects, The CLI supports types such as classes, interfaces, arrays, delegates.

Method invocation

The CLI has three call instructions for invoking methods (functions) and interfaces, two for direct invocation and one for indirect call. They are call for static methods, callvirt for instances and calli for pointers to methods.

Intra-Type interaction (conversion)

Interaction (conversion); between value-type and reference-type is provided by CLI features, such as boxing and unboxing.

The IL specification (as part of the Common Language Infrastructure - CLI), was submitted by Microsoft; to the international standardization organization ECMA for adoption. The full set of documents is publicly available through: <http://msdn.microsoft.com/net/ecma/>

You could also learn more about IL through the .NET framework SDK. If you've downloaded and installed the SDK, then the documentation could be found at: \SDK\v1.1\Tool Developers Guide\docs. The drive and parent folder will vary, depending on whether you installed Developer Studio 200x or not.

Although it may be helpful to understand a bit of IL, especially if you intend to write tools for the .NET framework, but it is not necessary for the regular application development.

TIP: The fastest way to learn IL is by writing code in a high level language such as C# or VB.NET and examining the resulting PE file (DLL or EXE) using ILDASM.

The SDK also comes with an IL assembler tool ilasm.exe, which you can use to assemble an IL code into a PE.

1.4.7 Common Type System (CTS)

The other most crucial part of the CLR, for ensuring language interoperability is the Common Type System (CTS). A type is a template used to define data and its behavior. In other words, it specifies the values that a variable is allowed to hold, and the operations that can be performed on it.

Each type has the following elements: Properties, Fields and Methods.

Most notable of the CTS defined types are value types, reference types, classes, interfaces, and delegates. These types and many more are made available through many programming languages that target the .NET framework, including VB.NET, C# and C++ managed extensions.

It is very important to have a good understanding of the common types, to be able to write programs in the .NET framework environment, so let's take a closer look into these types.

Every type defined by the CTS is an object, derived from System.Object, and inherits the properties and methods of the System.Object class. Examples of such inheritance are the methods Equals() and GetHashCode(), which every type inherits.

The types defined by CTS can be categorized, broadly, as Value types and Reference types:

1.4.7.1 Value Type

As the name indicates, the value types variables contain the data (can't be null). Examples are character and integer values.

Look at this example:

```
int i = 0;  
int j = 1209;  
i=j;  
j=2705;
```

We define i and j as integers, assigned them values, then we set j=i, which assign a copy of the value of i into j; then we change the value of j to 2705. The value of i is not changed, after changing j's value, because each one of them had its own copy of the data.

Although a copy is created of each value, when assigned, or passed as a parameter, but it is a small price to pay for the performance gain, associated with Value-Types.

Another characteristic of Value Types is that they are allocated memory on the stack, and are removed once done working with.

1.4.7.2 Reference Type

Reference type contains a reference to its own data. It includes a location (Reference) created on the stack, data allocated on the managed heap.

Reference Types can be null.

They objects created on the heap are not removed, until the garbage collection process takes place, which is a highly non-deterministic operation.

When passed as parameter, the Reference-Types are passed by reference, which is faster and less resource hungry, making it ideal for dealing with large objects, such as arrays.

```
SomeObjectType myObj;  
SomeObjectType myObj2;  
myObj = someValue;  
myObj2 = myObj;           ← they are pointing to the same object on the heap
```

In the above example, we have two objects of the same type, which means they point to the same address.

Some examples of the reference types are object, array, class properties, interfaces and delegates. Some of which are discussed below:

1.4.7.3 Classes

The .NET framework has full support for the traditional object oriented programming features, such as encapsulation, inheritance and polymorphism. The primary unit of object orientation in the .NET framework is the class, which can be defined as a type in the CTS, with data and code (methods) that can manipulate the data.

Classes can inherit other classes, which is how the .NET Framework base classes used by programmers.

For example:

```
System.Console.WriteLine("Hello Universe");
```

Comprises of the namespace System, which contains a class Console.

Console in turn has methods, such as WriteLine(), which is code to perform some function.

Classes may contain the following elements:

Member variables, or "instance variables" or *Fields* as they are called in C#.

A *property*, which is a named value that is similar to a field, except it is implemented using a pair of **get** and **set** methods.

Methods are functions that perform some logic.

Other possible class elements are *delegates*, *events*, and *indexers*; which we discuss in the following sections.

1.4.7.4 Properties

Properties are a combination of fields and methods.

The consumer (or client) can assign values to them and retrieve values from them by specifying their name in an expression, in a similar fashion to fields.

The implementation inside the class, however, is quite different. Properties are implemented using **get** and **set** methods.

The **get** method returns a value of the property's type, while the **set** method receives the value from the caller and assigns it to the property.

To make a property read-only, just omit the **set** method.

You can relate to properties by recalling the properties you set when programming VB or similar language with windows development capabilities.

1.4.7.5 Interfaces

If you are familiar with the COM/+ development model, or C++ (IDL), then you have been exposed to interfaces. The .NET takes the same great concept of strictly enforceable contract between the developer of a system and its consumers.

Interfaces are purely abstract classes that contain no implementation. Which means that they cannot contain any fields or code. They can, however, contain events, properties, and abstract method declarations.

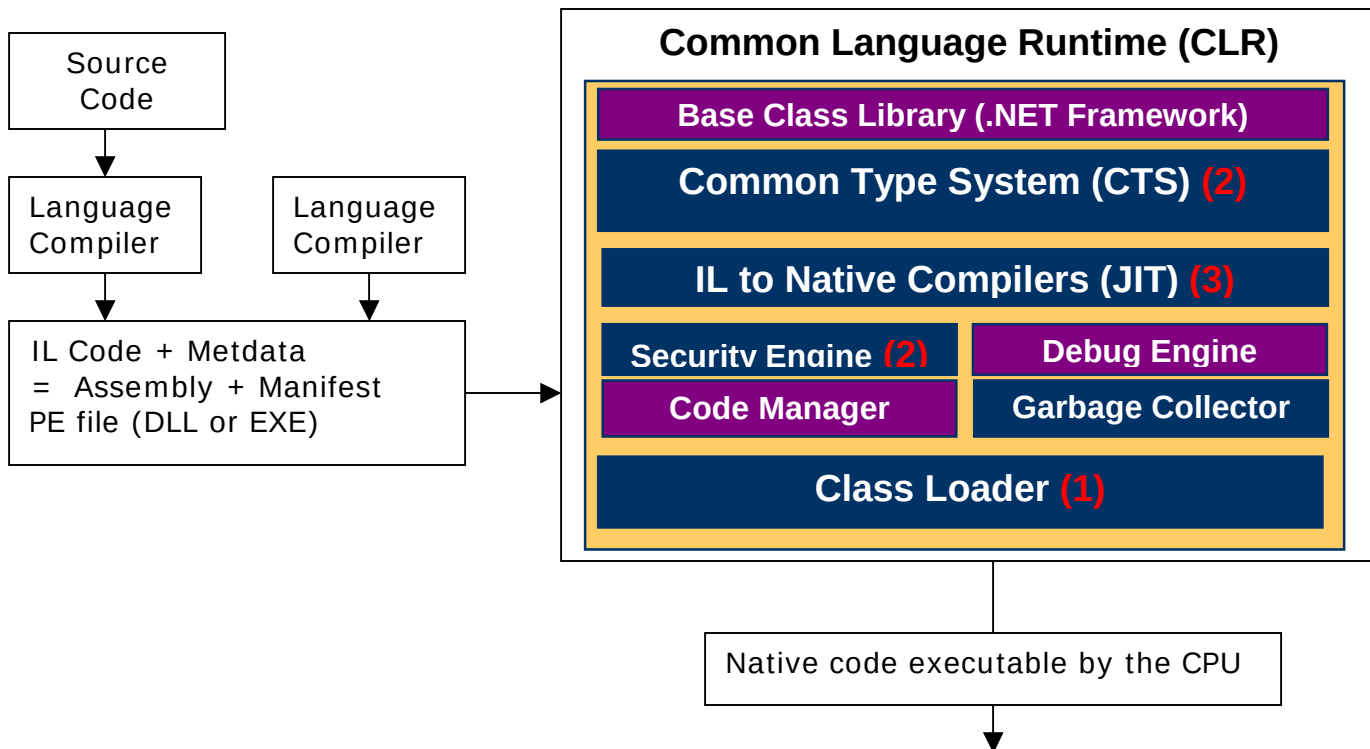
1.4.7.6 Delegates

If, you are familiar with C or C++ languages, then you probably know a thing or two about *“function pointers”* or *“pointers to functions”*. Delegates are the .NET framework version of that.

Delegates are the tools to implement callbacks or, as you will learn later, to implement event handlers. The whole event-driven programming is largely made possible thanks to Delegates.

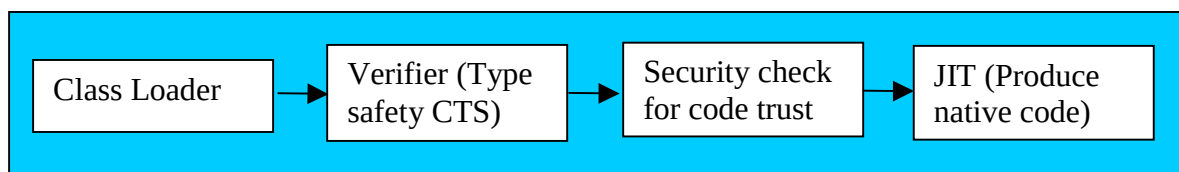
1.5 The program execution cycle

Let's revisit the diagram of “program development cycle” from 1.4.2 as follows:



What happens inside the CLR (the steps numbered in red) can be sequenced as follows:

1. The class loader loads the class, and caches it, along with any other classes it may be referencing.
2. Once the class has been loaded, the type verifier inspects for type safety against the CTS.
3. Once verified for type safety, or deemed safe, if trusted code, then the Just in time compiler, or JIT is kicked in to translate the IL into managed native code, which is executed by the CPU. You can alternatively pre-JIT the code, upon deployment, using a SDK utility called `ngen.exe`, which generates managed native code and caches the result, avoiding the JIT compilation, which can slow the execution a little bit. The CLR executes the managed native code in the Virtual Execution System or Engine (VES/VEE).
4. Once an object (including classes), is not needed anymore, the garbage collection process removes it, and returns the allocated heap to the OS resource manager.



2 Programming with C#

2.1 Introduction

C# is a simple, but powerful, type-safe, object-oriented language that allows programmers to build a wide range of applications.

The C# language is an evolution of C and C++. It also captures the ease of use of VB, while using many C++ features in the areas of statements, expressions, Object Oriented features, operators and constructs.

C# introduces considerable improvement and innovations in many areas

Developed from scratch, for the .NET platform, C# offers the best of the old programming languages, such as C/C++ and VB, and those from the new world of the internet such as JAVA.

As far as language Syntax goes, C# is so similar to Java and C++ languages that if you have experience with either of these languages, you can pick up C# and be productive almost immediately. The same can be said about a host of other C-family languages.

Microsoft has developed many tools using C#; in fact, most of the components in Visual Studio.NET were developed using C#.

Given the emphasis Microsoft is putting on C#, it looks as though C# will be the language of the future; at least as far as Microsoft technology based development is concerned.

2.1.1 The language

The C# language is extremely simple, with only about 80 keywords and a dozen built-in data types. However, C# includes all the support for structured, component-based, object-oriented programming, which is inherently built into the .NET framework, for which C# is designed. We've covered the .NET framework in Chapter 1.

It is very important that you have a solid understanding of the .NET framework before you get to learn C#.

The C# language was developed by a small Microsoft team led by Anders Hejlsberg and Scott Wiltamuth. Hejlsberg, a former Borland engineer, was responsible for the making of Turbo Pascal, one of my favorite languages. He also led the team that developed Delphi, one of the popular client/server programming languages, widely used for database programming.

If you have used either Turbo Pascal or Delphi, then you have also used - and I'm sure loved- one of the first (and certainly the best) integrated development environments IDE, for DOS, then Windows.

2.1.2 This book

The way we will approach programming in C# will be three fold. First I'll walk you through a typical C# program structure, with all the bells and whistles of the object oriented .NET programming framework. Second I'll list the most important features of the language, including the statements, constructs and so on. Third and most importantly we will build an application that uses, and in the process teaches you, the .NET framework technologies, including ADO.NET for database access, Windows Forms for basic Windows programming and ASP.NET for web development. Without any further due, let's get down to it.

2.2 *Setting up the environment*

In order to be able to try out the examples in this book, you need a setup with either one of the following:

2.2.1 The Affordable setup (Free): Microsoft .NET Framework SDK tools

Download the latest version of the Microsoft .NET framework SDK from:
<http://msdn.microsoft.com/netframework/technologyinfo/howtoget/default.aspx>

Microsoft .NET framework SDK comes in two packages; one is the redistributable version, whose size is 23MB, and is intended for redistribution with applications, written for the .NET, sort of like VBRUN60.DLL, or MFC42.DLL.

The second, which is what you should be getting, is about 110MB in size, and may take some time to download, but it is worth the wait, for it has all the goodies, any programmers would need to start developing hot applications for the .NET.

Once downloaded, execute the setup, and install the SDK to your system.

Warning:

The .NET framework comes with a copy of Microsoft Data Engine (MSDE) for Microsoft Visual Studio that is not patched for the slammer worm. So you have to download the latest service pack from:

<http://www.microsoft.com/sql/msde/default.asp>

The .NET Framework SDK includes everything you need to compile and test your applications.

The SDK comes with command-line compilers for each of the core .NET languages. If you intend to run the ASP.NET sample, you must install the .NET Framework SDK after installing IIS.

Once you have installed with the SDK. The runtime files and the Visual C# and Visual Basic compilers are located in `\WindowsDirectory\Microsoft.NET\Framework\v1.1.4322`. The C++ compiler is located in `\Program Files\Microsoft Visual Studio .NET\VC7\Bin`.

From our perspective, the most important files are the languages compilers, and the core system library, as listed below:

- `Mscorlib.dll` → MS Core Library, implicitly referenced by VBC and CSC
- `Vbc.exe` → VB compiler
- `Cl.exe` → C++ Sharp compiler
- `Csc.exe` → C Sharp compiler

All three languages' source is in plain text, hence, you can use any text editor — even Microsoft Notepad — for creating and modifying the program source files.

The .NET Framework SDK includes a variety of tools and samples, several of which are particularly useful when working with namespaces.

You can also explore the namespaces in files — either those that come with the common language runtime or those created by yourself or others — by using the MSIL Disassembler (`Ildasm.exe`).

The dis-assembler `ildasm.exe` is located in: `\Program Files\Microsoft Visual Studio .NET\SDK\v1.x\Bin`

You can debug any .NET assembly and/or executable file using the `DbgCLR.exe` which can be found in:

`\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\GuiDebug`

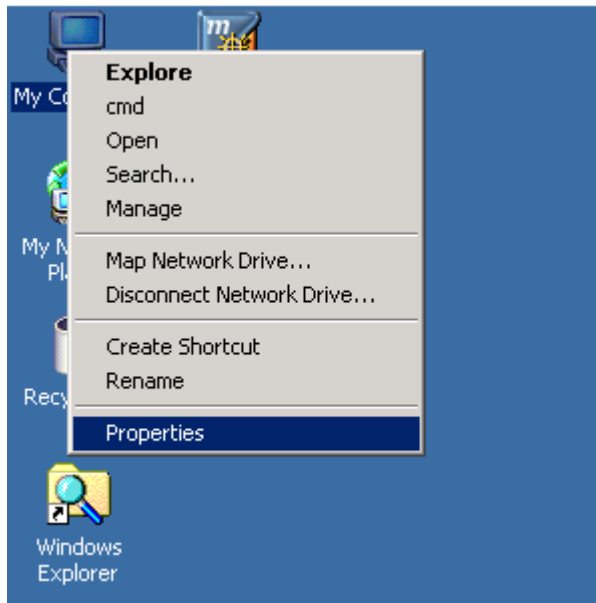
To compile and run the samples that accompany both the SDK and the tutorials in this book, you must have the path environment variable configured so that it includes the subdirectories where the compilers are located.

The environment should be configured using the setup process for the SDK.

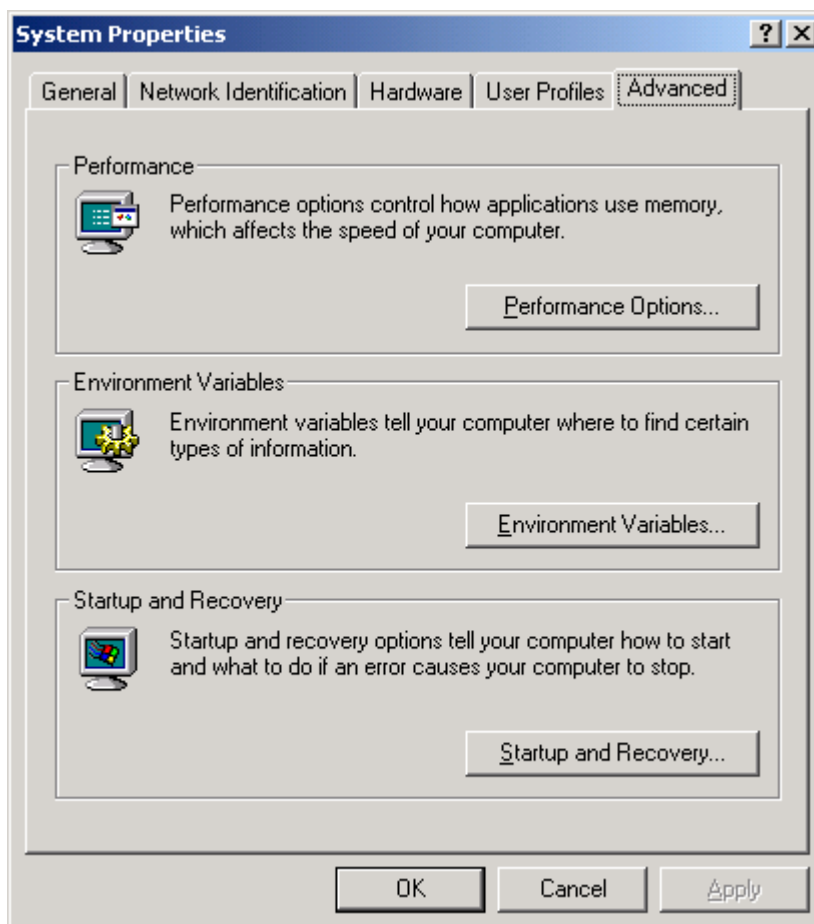
If not, then, running `sdvars.bat`, which is located in the `\Bin` subdirectory of the SDK, can set the appropriate environment variables.

If the above doesn't do the trick for you then follow the following steps:

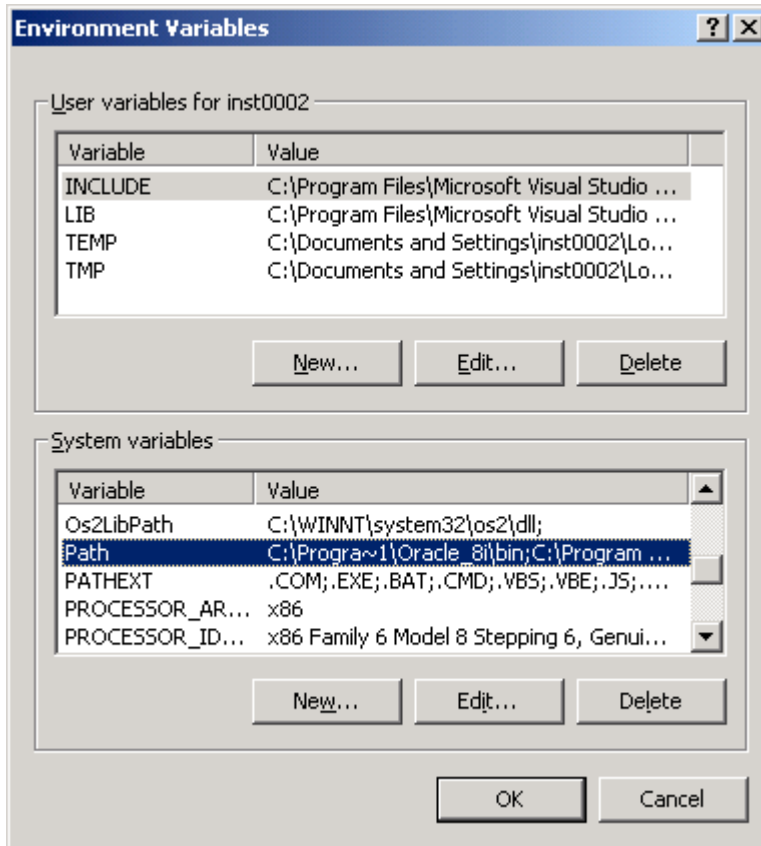
1. Find "My Computer" and right click on it. Select Properties



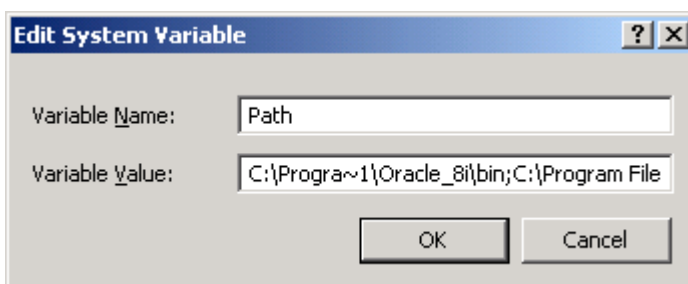
2. click the Advanced tab



3. Click Environment Variables button, and scroll down to “Path” in “System Variables” section, lower panel



4. Click “Edit”, and place the cursor at the beginning of the “Variable Value” text box.



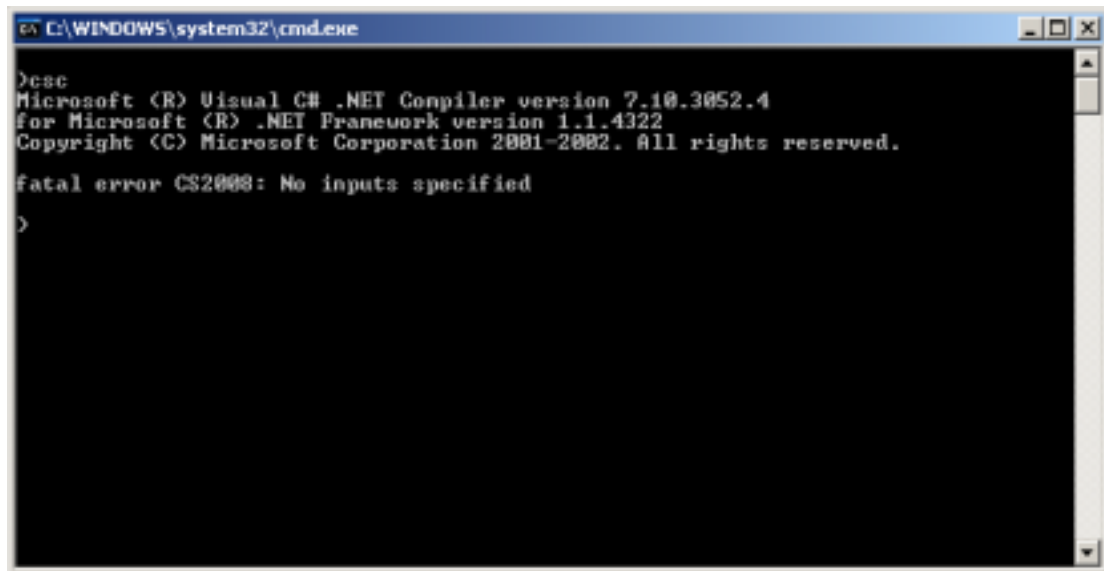
5. Insert (type) the path strings listed below at the cursor location. For accuracy, you may want to type them in a text editor such as notepad and then copy and paste them.

C:\Program Files\Microsoft Visual Studio
.NET\FrameworkSDK\Bin;c:\winnt\microsoft.net\framework\v1.1.4322;
C:\Program Files\Microsoft Visual Studio .NET\Vc7\bin;C:\Program Files\Common
Files\Microsoft Shared\VSA\7.0\VsaEnv; C:\Program Files\Microsoft Visual Studio .NET
2003\SDK\v1.1\GuiDebug;

Note: if you have not installed the .Net Framework SDK 1.1, then the number at the end of line two in the path would be something like "1.0.3705". However, I'd highly recommend you do upgrade to 1.1, since it is the latest version, which has all the security features, and many other enhancements over the 1.0 release.

Now that you have made sure your path is set, try it out, by doing the following:

1. Launch the command prompt, by clicking Start → Run. Then type "**cmd**" and press Enter
2. At the command prompt ">" type "**csc**" and press enter. You should get something like the figure below:



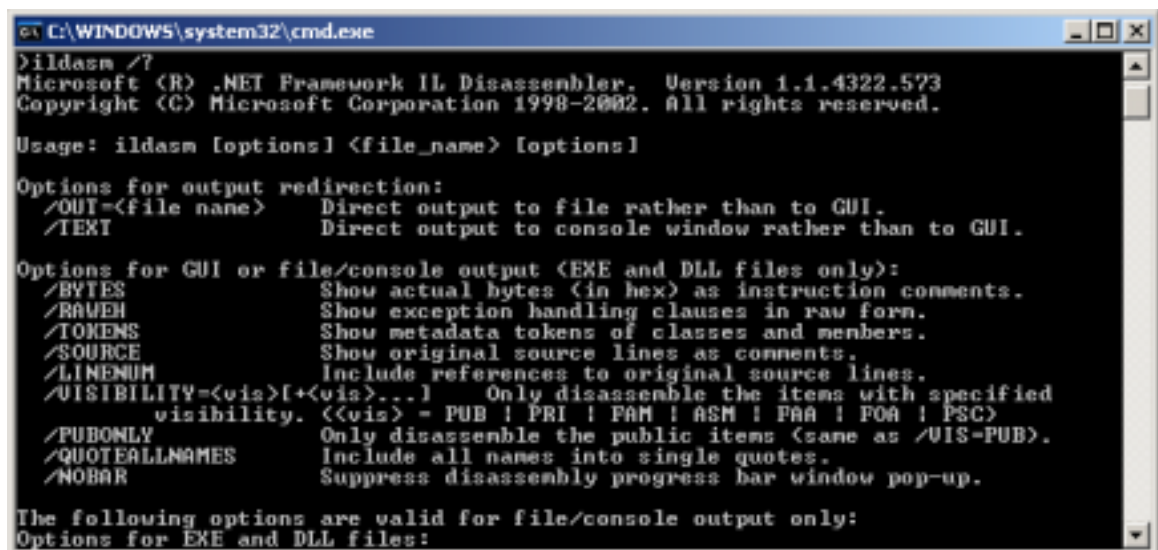
```
C:\WINDOWS\system32\cmd.exe

>csc
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

fatal error CS2008: No inputs specified

>
```

3. And while you are at it, try typing "**ildasm /?**" followed by Enter. You should receive something like this:



```
C:\WINDOWS\system32\cmd.exe

>ildasm /?
Microsoft (R) .NET Framework IL Disassembler. Version 1.1.4322.573
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.

Usage: ildasm [options] <file_name> [options]

Options for output redirection:
  /OUT=<file name>    Direct output to file rather than to GUI.
  /TEXT              Direct output to console window rather than to GUI.

Options for GUI or file/console output (EXE and DLL files only):
  /BYTES             Show actual bytes (in hex) as instruction comments.
  /RAVEH             Show exception handling clauses in raw form.
  /TOKENS            Show metadata tokens of classes and members.
  /SOURCE            Show original source lines as comments.
  /LINENUM           Include references to original source lines.
  /VISIBILITY=<vis>[+<vis>...] Only disassemble the items with specified
visibility. <<vis> - PUB ! PRI ! FAM ! ASM ! PAA ! POA ! PSC>
  /PUBONLY           Only disassemble the public items (same as /VIS-PUB).
  /QUOTEALLNAMES     Include all names into single quotes.
  /NOBAR             Suppress disassembly progress bar window pop-up.

The following options are valid for file/console output only:
Options for EXE and DLL files:
```

2.2.2 The Less Affordable setup (costs \$\$\$): Visual Studio .NET 2003

The most elegant and productive way for writing .NET programs is to use Visual Studio.NET. You can learn all about VS.NET and even try it online at:

<http://msdn.microsoft.com/vstudio/>

There are three main releases of the product, which are VS.NET professional, Enterprise developer and Enterprise Architect. The later two are identical except for the last one shipping with Visio 2002 for Enterprise Architects.

There is also one more release geared towards academic institutes, which is an affordable way for students to access the technology. So if you are a student or a faculty at a recognized institute then you could get a copy of the Visual Studio .NET at around \$100 US.

The various editions of VS.NET are listed at:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxconATourOfVisualStudio.asp>

Setting up VS.NET is a wizard based straight forward process that should not cause any hiccups. The only problem I encountered while setting up the software on a windows 2000 machine, was that I already had the system updates pre-installed, so I had to circumvent the issue by running SETUP.EXE that is contained in the “\setup” folder, instead of the one in the root, which is also the one Autostart loads, when you insert the media (CD/DVD) into the drive.

Warning:

The .NET framework comes with a copy of Microsoft Data Engine (MSDE) for Microsoft Visual Studio that is not patched for the slammer worm. So you have to download the latest service pack from:

<http://www.microsoft.com/sql/msde/default.asp>

Throughout the first part of the book, we will be explaining how to use both the SDK's command line compilers as well as the Visual Studio.NET, to compile and run the sample programs.

3 Writing and compiling programs

3.1.1 Using the .NET framework SDK (simply SDK from now on)

3.1.1.1 The editor

Use any text editor, such as EDIT.EXE, notepad or any other code editor you feel comfortable with.

#develop (short for SharpDevelop) is a free IDE for C# and VB.NET projects on Microsoft's .NET platform. It is open-source (GPL), and you can download both source code and executables from:

<http://www.icsharpcode.net/OpenSource/SD/Download/>

You can also buy some decent editor from a third party such as Antechinus C# Editor, which costs \$35 US. Check them out at:

<http://www.c-point.com/csharp.htm>

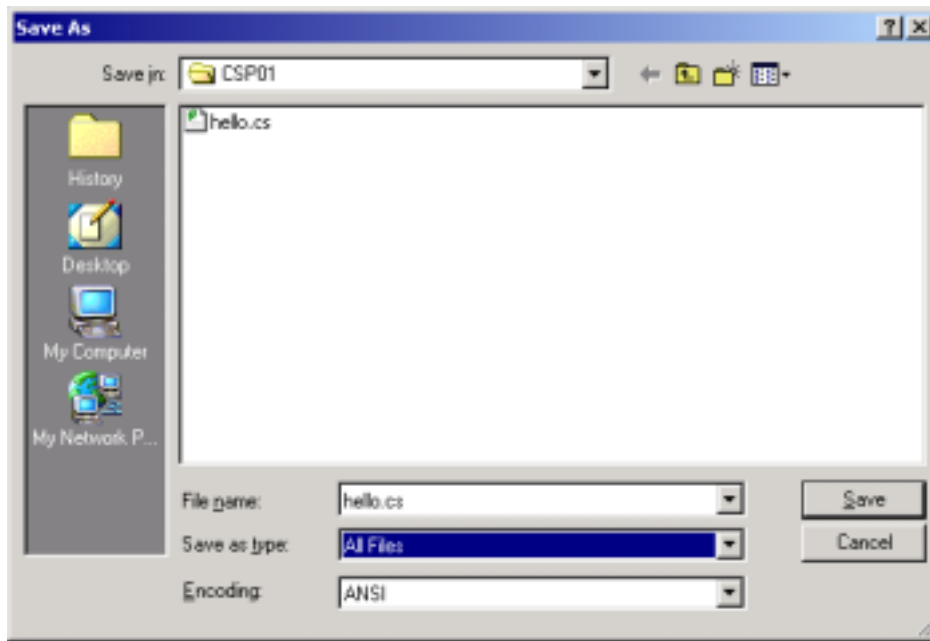
You can find a host of tools at:

<http://dmoz.org/Computers/Programming/Languages/C-sharp/Tools/>

Note: All of the above web links are active at the time of this writing, however, as we all know, web sites can easily disappear and product licensing and pricing schemes change without notice. Hence, we assume no responsibility for any of the sites, listed here within, for information purpose.

We'll assume you are using notepad.exe for the sake of the examples in this section. Complete the following steps:

- Create a new **folder** using either Windows Explorer, or the command prompt. Name the folder **CSP01** (C Sharp Project 01).
- Run **notepad.exe**. Either by clicking on it at: *Start → Programs → Accessories → notepad*, or alternatively by selecting *Start → Run* and then typing **notepad.exe** followed by Enter.
- Once opened a new document, select *File → Save* from the menu, and then select the **CSP01**, and name the file **hello.cs**. Make sure you select "All files" in the "Save as type" drop-down list after the file name, otherwise you will end up with a file that has .cs.txt extension.



- Type the following code (don't worry about what it means, we'll explain it all in a short while):

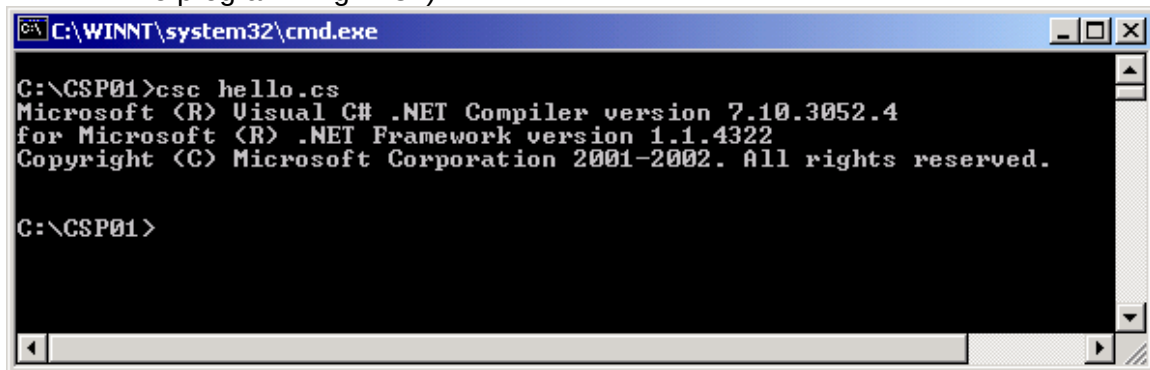
```
namespace myFirstProject
{
    class HelloUniverse
    {
        static void Main()
        {
            System.Console.WriteLine("Hello, Your excellence");
        }
    }
}
```
- Save your file once again and close Notepad.

3.1.1.2 The Compiler

Now that you wrote your first C# program – although you may not know what it says – let's see if we can compile and run it.

Launch the command prompt, whichever way you prefer. At the command prompt:

- Type **X:** and press Enter. Where **X** is the drive letter, where you **CSP01** folder was created. Say **C** or **D**.
- Then type **CD CSP01** and press enter
- Finally type **CSC hello.cs** and press enter
- If you did everything right, then you should get the command prompt back, similar to the figure below, with no error-like messages (no news is good news, while programming in C#).



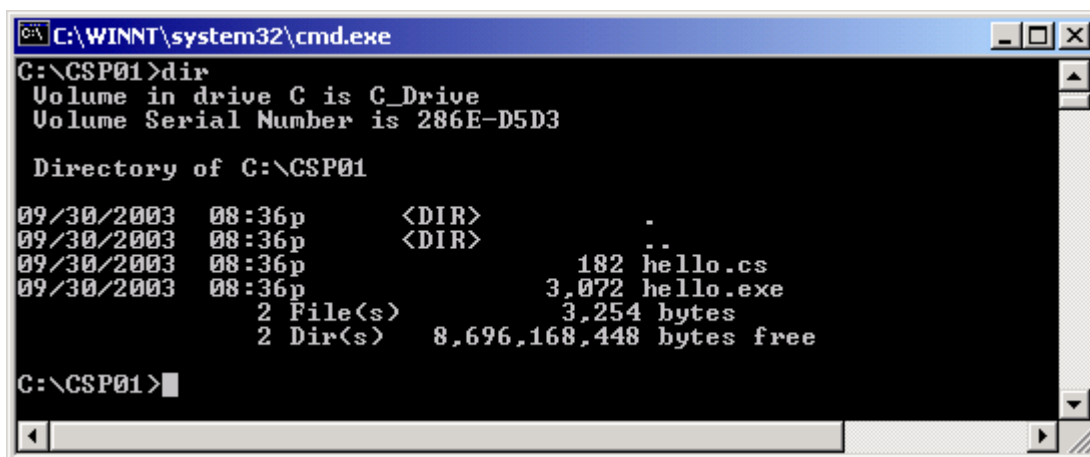
```
C:\WINNT\system32\cmd.exe

C:\CSP01>csc hello.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

C:\CSP01>
```

If you get any error messages back, don't worry, just take a closer look at the code, and make sure that you copied everything as it is, including the semicolon, parenthesis and braces. For the sake of this exercise nothing should be omitted or replaced.

Now if you type the command **dir** and hit **Enter**, you should see a screen similar to the one below, listing the files in your folder (**hello.cs** and **hello.exe**). You obviously recall that **hello.cs** is the source file that you created using Notepad, and **hello.exe** is the portable executable (PE) file created by the C# compiler, which you invoked in the preceding steps.



```
C:\WINNT\system32\cmd.exe

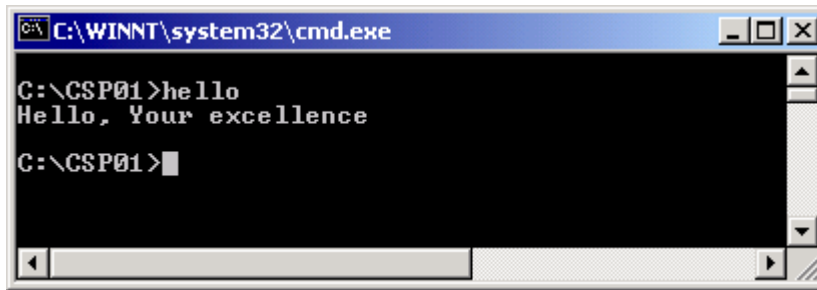
C:\CSP01>dir
Volume in drive C is C_Drive
Volume Serial Number is 286E-D5D3

Directory of C:\CSP01

09/30/2003  08:36p      <DIR>          .
09/30/2003  08:36p      <DIR>          ..
09/30/2003  08:36p                182 hello.cs
09/30/2003  08:36p            3,072 hello.exe
                2 File(s)              3,254 bytes
                2 Dir(s)  8,696,168,448 bytes free

C:\CSP01>
```

Finally, for the moment of glory, type **hello** and hit **Enter**. All going well; you should be greeted by your obedient computer, in a respectful manner, such as the one below. That's if you didn't change the wording in the code above!



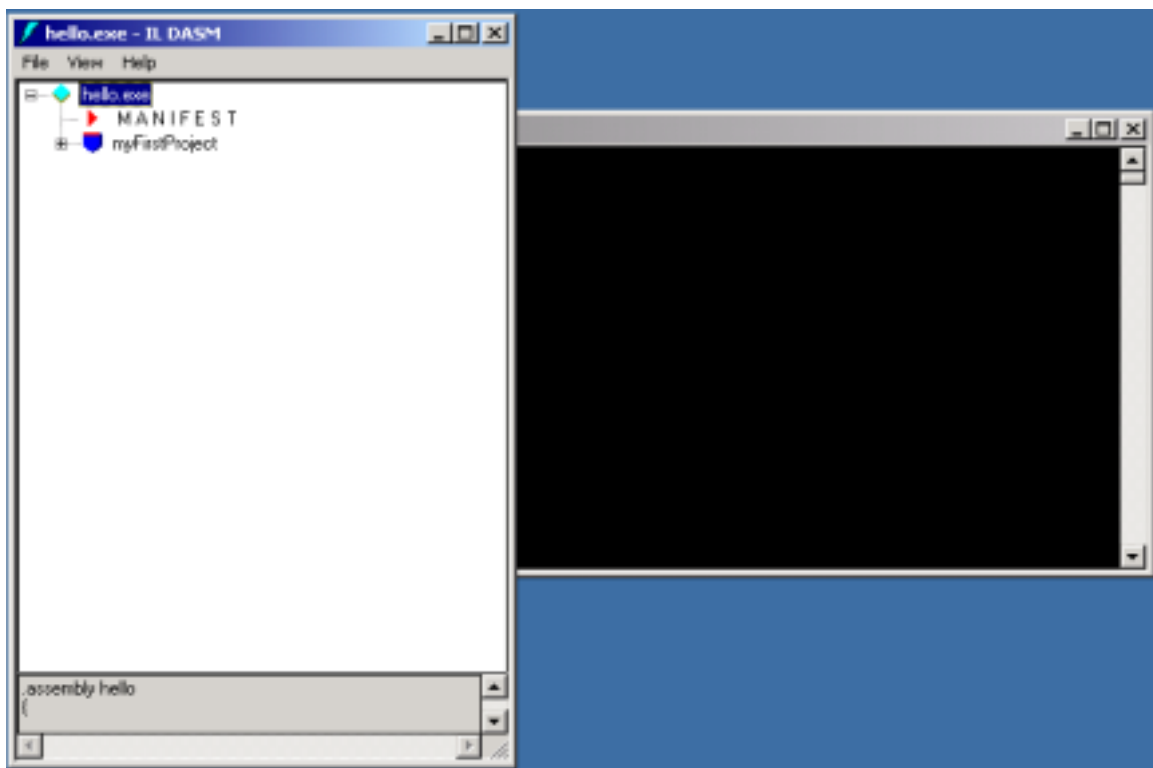
Savor the moment, for this could be the only time you will ever write code that compiles and runs from the first attempt!

3.1.1.3 Your IL code

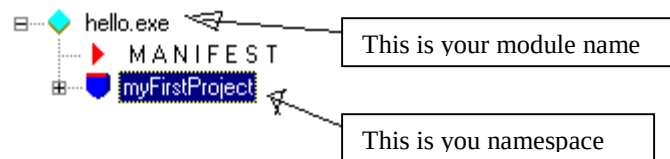
As we discussed in the first part of this book - the .NET introduction – the .EXE you just created is actually a Portable Executable (PE), which contains both the Metadata and an Intermediate Language Code (IL)

Let us see how it looks like:

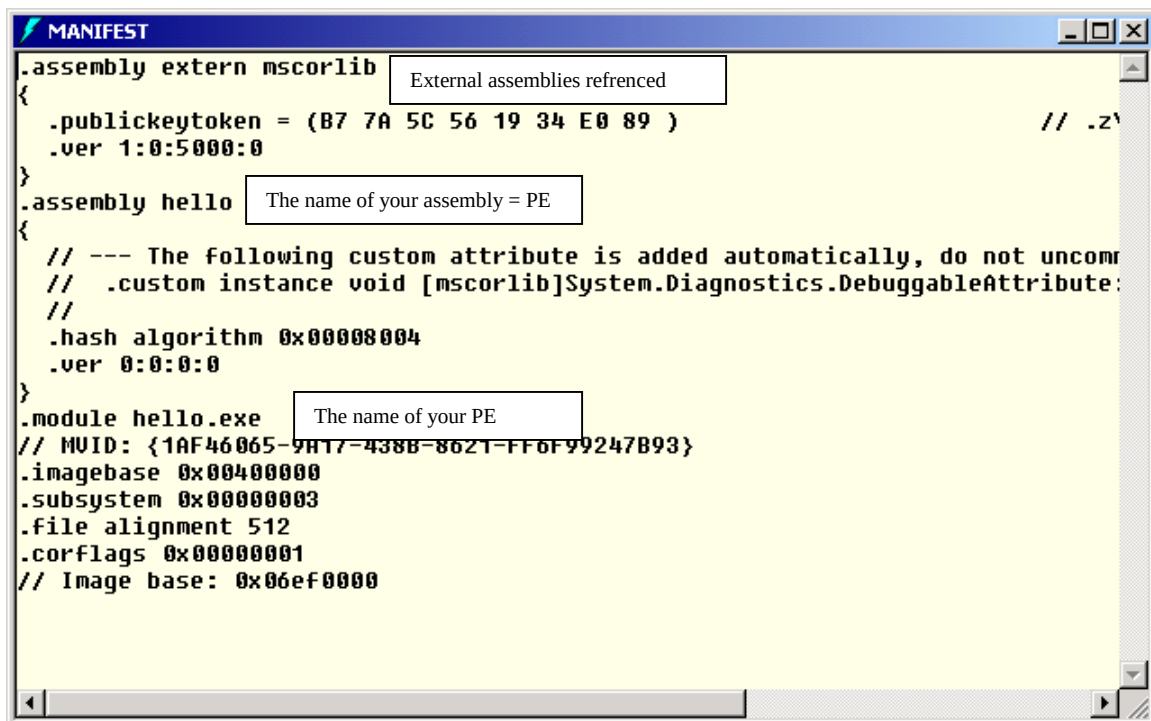
At the command prompt Type **ildasm.exe hello.exe** and hit Enter. This time you will get a GUI interface pop on top of the Command prompt's window. You should see something like the figure below:



Now switch back to your notepad and see if you can find any similarities to your code? You will notice “myFirstProject” which is on the first line of the source code



Now double click the **MANIFEST** to get a screen similar to the one below:
 We briefly discussed manifests, while explaining assemblies in the first part. The manifest is essentially your packing list, which tells the CLR how to process the application. Examine the listing you get in the manifest.
 Don't worry about the cryptic hexadecimal codes and abbreviations. Just get the big picture, and understand the major building blocks of the manifest, as pointed to by the inserted text boxes.



Notice the `.assembly` and `.module`? They are usually 1:1; except when you have multi-file assemblies, which may include several files.

Now click the **x** at the top-right corner, to close the window.

Go back to the ildasm screen and click on the plus sign of **myFirstProject**, or double-click the word "myFirstProject" to expand the list. You should see yet another familiar name, in this case "HelloUniverse", which is you Class name, from the source code you typed in the Notepad.



Now expand the  HelloUniverse in the same fashion, by either clicking the + sign, or double-clicking the name HelloUniverse.

At this point your screen should look something like the figure below:



You will notice that there are three items under “HelloUniverse”.

The first item with the little red arrow and starting with .class is information about the class name above it. You’ll learn about classes later in the book.

Next you have .ctor: void() and Main: void() which are methods. Don’t worry, you will see enough methods till you get used to them, even if you don’t understand what they are all about. But you should, really, to be able to write any useful C# code.

Main () is from your code, the little s in the symbol denotes “static”, which is in your code that reads “static void Main()”, if you remember. Look it up if you don’t.

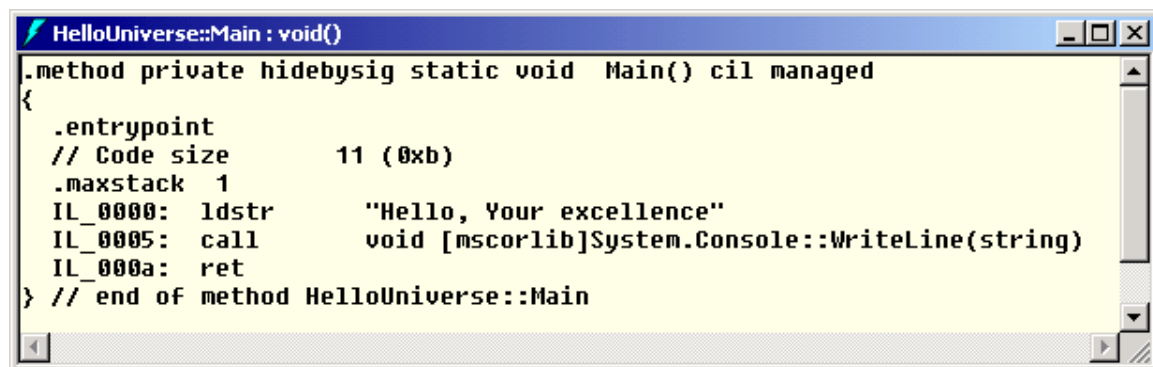
.ctor is a constructor method, which was added for you by the system. You’ll learn a bit more about constructors and destructors later in the book.

Now if you double click on Main  Main : void() you should be able to see the rest of your source code as it looks after having been compiled into IL code. See any resemblance? Check out the first line: blah blah ... **Main() cli managed** which is your method, with a few more items. By the way, Methods and Functions are the same. Method is the OOPS! (Object Oriented Programming System) Way of calling code that you call and may or may not return a value.

ldstr “Hello, Your Excellence” loads the value to the stack

call invokes the method WriteLine() from the assembly [mscorlib] which was referenced in the Manifest (Recall?). Its parameter is the value on the stack.

ret simply returns control to the caller (Command prompt in this case)



Enough of the low level stuff. If IL caught your fancy then you can have all the IL that you want in the SDK documentation.

You can close the **ildasm** window as well as the command prompt window.

If you have no intention of using Visual Studio .NET, then you can skip the next section.

3.2.1 Visual Studio .NET

Visual Studio.NET (VS.NET) comes loaded with features, which can be broadly categorized as:

1. **An Integrated Development Environment (IDE)**
that's shared among the various programming languages supported by VS.NET, such as C#, VB.NET and C++. Being the most visible feature of VS.NET, we'll spend some time explaining it later in this chapter.
2. **Windows Management features**
To facilitate easier management, and better control of the screen's real estate; the windows management features include things such as
 - a. Dockable windows, which means that windows can be attached to other windows and appear as a TAB.
 - b. Internet Explorer-like navigation of windows, with the Back-Forward buttons in the toolbar area.
 - c. Multiple screen support, giving the developer even more real estate
 - d. Integration with the Internet Explorer favourites, allowing access to, and manipulation of the favourites list.
3. **Visual Designers**
VS.NET brings the What-You-See-Is-What-You-Get (WYSIWYG) design into every aspect of programming. From the now-traditional windows forms, to the cutting edge, not-so-traditional mobile web development. This is accomplished through a set of Visual designers that include:
 - a. Mobile (ASP.NET)
With the advent of mobile internet, people are using cell phones, pagers, Wi-Fi and GPRS enabled PDAs, to browse the NET anytime anywhere. Using the VS.NET Mobile Designer you can create mobile Web applications, and then build and run the application, all from within Visual Studio.
 - b. Web forms (ASP.NET)
Helps developers design HTML, ASP and ASP.NET forms graphically, using drag and drop, in a fashion similar to the windows programming in VB 6.0
 - c. Windows Forms
An enhanced version of the good old windows forms designer, that I've come to know since VB 3.0
 - d. Components
Even components that have no real drag-droppable UI element such server components can be designed in VS.NET, using the Component designer
 - e. XML

Full featured XML designer that is actually three in one. One to edit XML source code, another to create and edit XML schema definition (XSD) files and a third one for editing XML data files.

4. Database Tools

VS.NET provides visual access to features such as creation and manipulation of database schemas, stored procedures and other database elements. It also has SQL script editor and debugger, which enables you to debug stored procedures. The database could be of any type, and located anywhere as long as it can be accessed over the network.

5. Smart device development

The VS.NET 2003 integrates the tools necessary to develop applications for smart devices, such as the hand held Pocket PC, the mobile Smartphone and Tablet PC. Which target the .NET Compact Framework, a subset of the .NET Framework,

6. .Net Framework

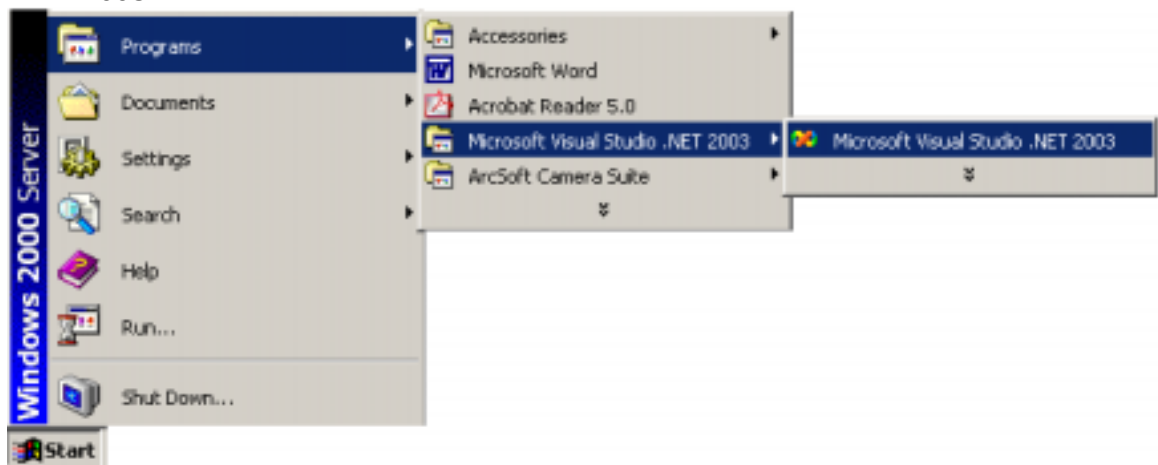
The latest version of the .NET Framework 1.1 ships with the VS.NET package. We have covered the .NET Framework in the first part of this book. If you have not read that chapter, then we strongly recommend that you do so, before proceeding any further, unless of course you have prior knowledge of what the .NET framework is all about.

7. Debugging

VS.NET delivers a truly integrated, Multilanguage debugger, which allows the developer to step into source lines of multiple languages in real time.

3.2.1.2 Launching Visual Studio .NET

1. You can launch Visual Studio .NET from the Start menu by clicking **Start → Programs → Microsoft Visual Studio .NET 2003 → Microsoft Visual Studio .NET 2003**.



2. You can also launch Visual Studio .NET from the command line, to do so: select **Start→Run** then type **DevEnv** and press **Enter**.

3. You can also create a short cut to the Visual Studio .NET, at the place of your liking in as follows:
 - a. Using the mouse Left-click and hold down “**Start → Programs → Microsoft Visual Studio .NET 2003 → Microsoft Visual Studio .NET 2003**”
 - b. Drag it to the desktop, or to the Quick Launch bar, or wherever else you may want a short cut to the IDE.

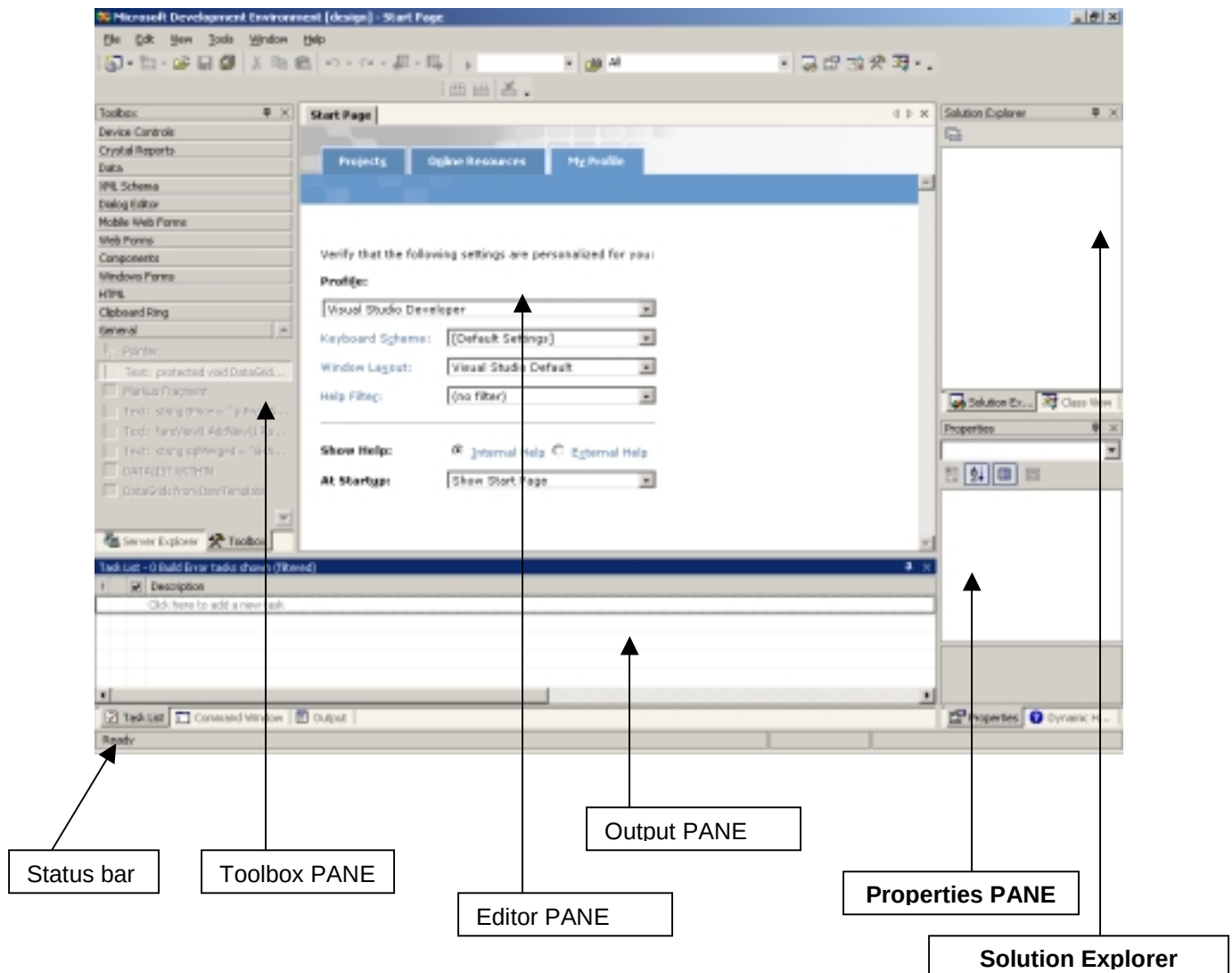
3.2.1.3 Parts of the Visual Studio .NET IDE

Alternatively, you can do it the hard way, by Right-clicking on the desktop, and selecting **New → Short Cut**. This will launch a wizard that will walk you through the process.

And in case you are wondering, the IDE's executable is located at:

"\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE\devenv.exe"

The first time you launch the Visual Studio .NET IDE, you should see a window similar to the figure below.



The IDE has a main frame (window) which hosts the menus, tool bars and **The Status bar** in addition to 5 major PANES. Each pane can host one or more child windows. In case the pane hosts more than one window, the windows will be represented by TABS, which you can use to switch between the different windows. Refer to the Docking features in the Windows management features, of the previous section. We have illustrated the different panes of the IDE in the figure above. Let's explore the various parts of the IDE screen:

1. **The Toolbox pane:** hosts the Toolbox window as well as the "Server Explorer" window which was introduced with the Visual Studio .NET. You can use the TABS at the bottom of the pane to switch between the two windows.
 - a. **The Toolbox** Changes depending on the designer or editor you use
 - b. **Server Explorer:** Provides access to servers and components running on them, through a hierarchical tree structure. It also enables the developer to access and work on databases such as Access, SQL Server and Oracle. Server Explorer also maintains all the data connections created for use by the developer's programs.
You can display the server explorer by selecting the menu item:
View → Server Explorer.
Or by pressing **CTRL+ALT+S** simultaneously.
2. **The Editor pane** hosts the visual designer and the code editor windows. It also hosts the **Start Page** (Home Page), which is a multi-TAB window that opens when you launch VS.NET for the first time.
 - a. **The Designer and Code editor windows**
 - i. **The designer windows:**
As briefly discussed in the introduction, there are various WYSIWYG designers, which you can use to develop UI applications and component, rapidly and with ease. They all fit in the Editor pane. In a tabbed formation, if you have more than one window open at time. You'll learn more about the features of the code editor through the various exercises.
 - ii. **The code editor window**
Double clicking a UI control on the designer window, including the designer itself (which is the container control) invokes a code editor window, otherwise known as a code-behind editor. With complete intellisense features, and auto-formatting depending on the language used, and access to both dynamic as well as context sensitive help, the code editor is the best tool to write .NET base programs. You'll learn more about the features of the code editor through the various exercises.
 - b. **The Start Page** contains three tabs:
 - i. **Projects:** keeps shortcuts to your most recent projects, giving you easy access to the files.
 - ii. **Online Resources:** links you to the development community and MSDN help online, giving you access to online help, code samples and more, provided that you are connected to the Internet

- iii. **My Profile:** allows you to set preferences, such as whether or not you want the start page to be your first page, set the help filters for your preferred language, set the IDE layout in your favourite style; for example VB6 or VC++ 6. In addition

You can show the start page by selecting the menu item: **Help → Show Start Page.**

- 3. **The Output pane** hosts the output window, the Task List window and The Command window.
 - a. **Output window:** The output window displays messages of the compiler.
 - b. **Task List:** A neat feature that was available for the Interdev users, and now in VS.NET. It enables the developer to keep track of tasks created using comments in the code, and then tracking the progress on carrying those tasks out.

You can display the task list window by selecting the menu item:
View → Show Tasks → All
 - c. **Command Window:** Extended DOS Command window, merged with Visual Basic's Immediate Window. It has two modes: Command and Immediate. In Command mode, you can enter IDE command names after the angle bracket (>). Where you get access to the full set of VS.NET commands, such as menus access, execute search, navigate windows and items within a solution, browse the web and run external commands. In Immediate mode, you can execute code statements, operate on variables and expressions. It also has full intellisense support, where you get help as you type partial command.

You invoke the immediate mode by typing **immed** and press **Enter**. To switch back to the command mode by type **>cmd** and press **Enter**.
You can show the command window by pressing **CTRL+ALT+A** simultaneously
Or by selecting the menu item
View → Other Windows → Command Window
- 4. **The Properties pane** hosts the properties window, and the Dynamic Help (MSDN) windows.
 - a. **Properties window:** displays the property sheet for the selected component. We'll be discussing this window at various points, throughout the text. You can invoke the properties window by pressing  **F4** or by clicking the Toolbar item
or by selecting the menu item
View → Properties Window
 - b. **Dynamic Help (MSDN),** monitors what the developer is doing and provides help based on the task on hand. The help outline is displayed in the help window. Selecting a help topic by clicking on the hyperlink opens an editor window with the details of the selected help item.

You can open the Dynamic Help window by pressing **CTRL+F1** or by clicking the
Or by selecting the menu item
Help → Dynamic Help

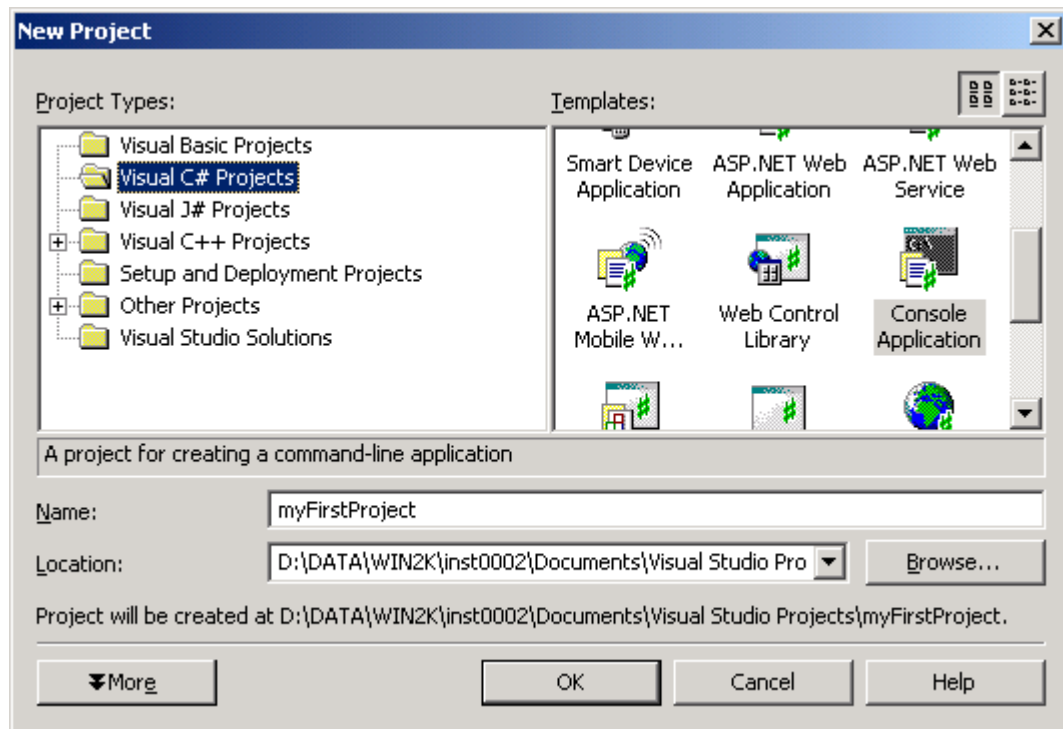
5. **The Solution Explorer pane** hosts the Solution Explorer window and the Class View window.
- a. **Solution Explorer:** Shows the list of projects and files and folders associated with each project. Also known as File View, which may ring a bell if you come from VC++ background. The solution explorer supports creation of a single solution from multiple projects, which could also be written in multiple languages.
You can invoke the Solution explorer window by pressing **CTRL+ALT+L** or by clicking the Tool  item or by selecting the menu item **View → Solution Explorer**
 - b. **Class View:**
An alternative to file view, Allows you to view and navigate to symbols in your solution. The symbols, which are organized by project, display in a hierarchical tree view, indicating the containment relationships between them. This is again similar, albeit much superior, to the VC++ 6.0 version.

I'd like to note here that the above arrangement is highly customizable. All you have to do is just drag a window and drop it elsewhere to get a different layout.

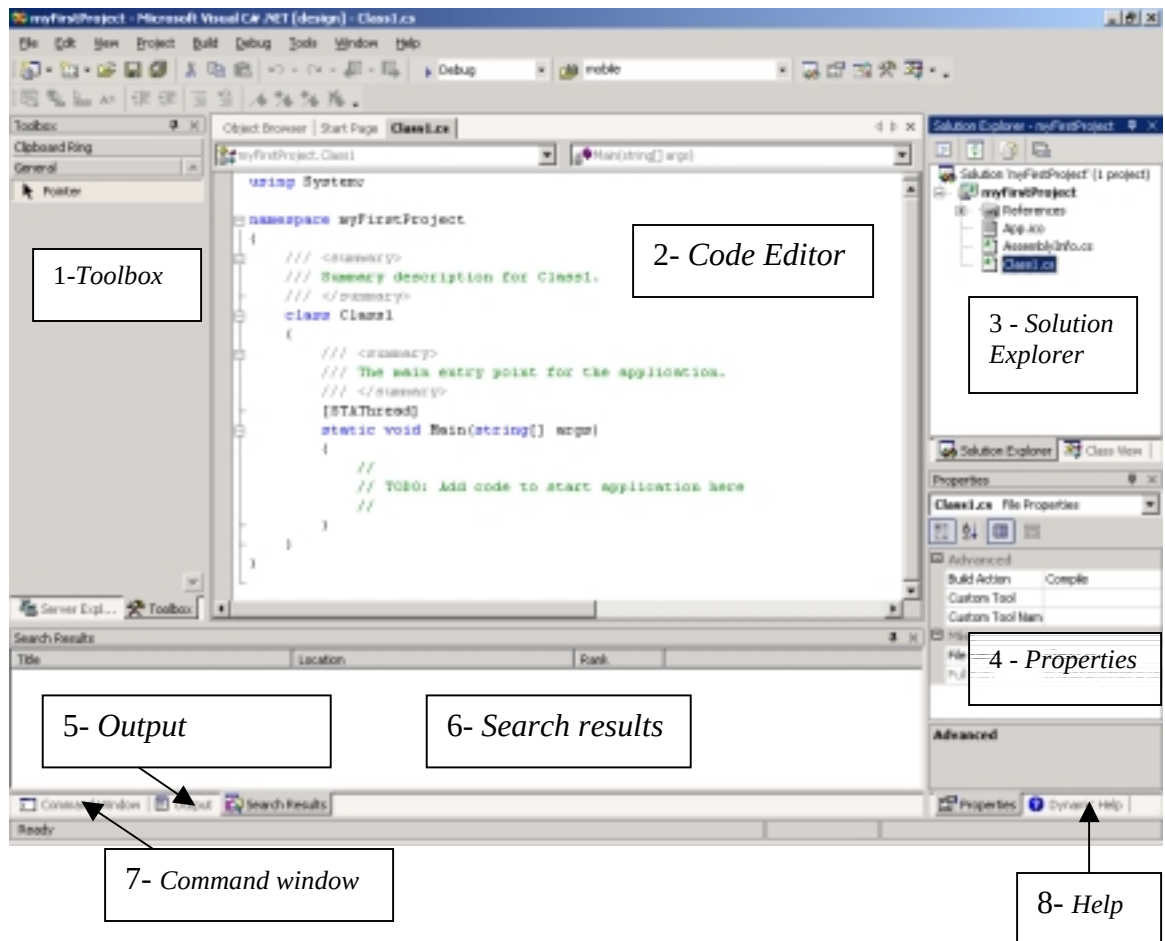
3.2.1.4 Creating a project

To create a new project,

1. Start VS.NET.
2. Make sure you have the “Start Page” open (use menu item: **Help → Show Start Page**)
3. Click “New Project” hyperlink at the bottom of the screen
4. You will get a screen similar to the figure below

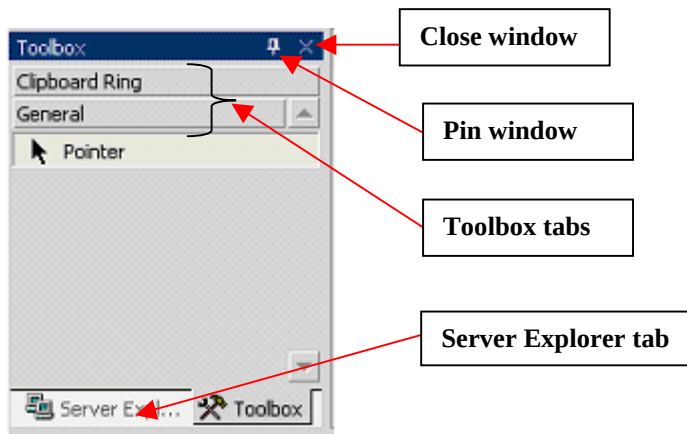


5. Click on “*Visual C# Projects*” on the upper-left explorer-style folder. This will ensure that we select the right language.
6. Select “*Console Application*” In the right-side panel. This will ensure we are creating a console application, which works in the text mode, command window.
7. In the “*Name:*” text box type “**myFirstProject**”.
8. The “*Location:*” drop-down list (Combo box if you are a VB programmer), should contain the correct path for your files. You can click the “*Browse...*” button to launch a windows explorer, and then pick the right folder.
9. Press OK.
10. You should see a screen similar to the figure below:

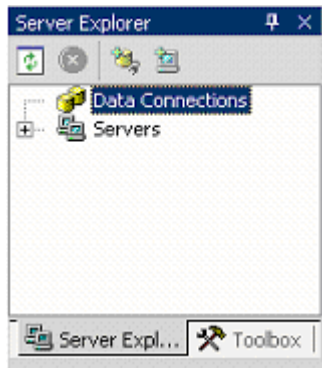


Now let's revisit the parts of the IDE in light of our newly created project. You will notice that all parts of the IDE came to life.

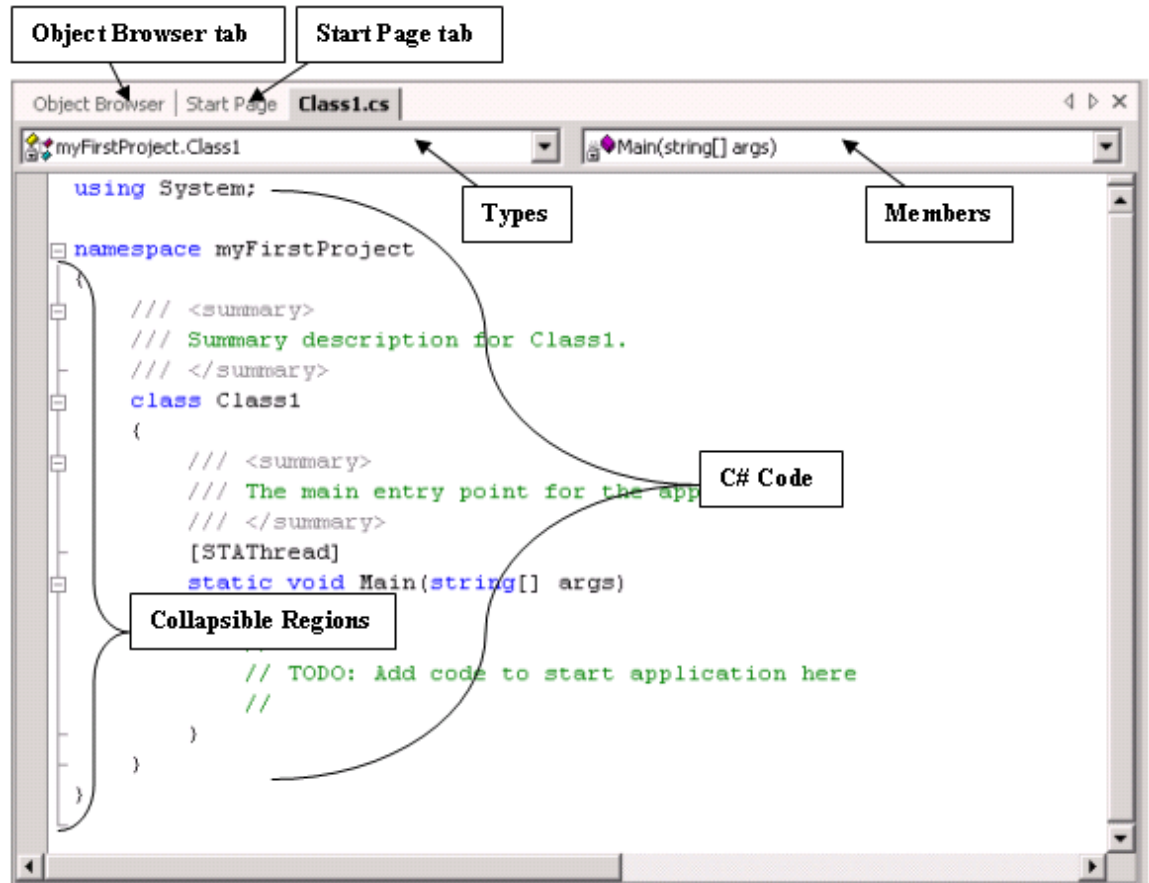
1. Toolbox – Not much of tools there yet, but you can see that it has two tabs, featuring “General” which is selected, and “Clipboard Ring”, which when you click slides down closing the current one, and replacing it. This behaviour is common among all the different toolboxes.



If you click the tab at the bottom of the Toolbox, which reads “Server Explorer” you will get a different window, similar to the figure below. I will let you explore it on your own.



2. The Editor pane has a familiar-looking text editor, although more colourful than notepad, and certainly more graphical.
The editor itself is a standard word-like editor, with the usual commands and shortcuts, as in VS 6.0. There are however several innovations, unique to VS.NET, which you will learn about as you progress through the book. Worth noting at this point are the following major parts of the editor:
 - a. The tabs at the top are to switch between the different windows.
 - b. The two drop down lists beneath the tabs are to display Types (CTS types were discussed in chapter 1, and will be discussed some more) and Members (Anything contained within a type, such as methods, variables ...etc.)
 - c. The code area features several characteristics, such as the collapsible regions, which is very helpful, particularly when the number of lines in your code starts to grow. You can also define your own regions using `//REGION //END REGION` comments. The actual program lines are color coded, where keyword, comments and other types, each have different colors. The last thing to mention here is the intellisense feature, which auto-completes known words as you type. This feature virtually eliminates typing errors, and reduces the amount of keystrokes by a great deal.



3. The properties window has not changed much from the earlier releases of visual studio. We will be covered in several sections throughout the book.
4. The output window is just that. However, you may want to learn more about the command window, which is a very powerful tool, as mentioned in the introduction to VS.NET.

3.2.1.5 Navigating the IDE

3.2.1.5.1 Window hopping:

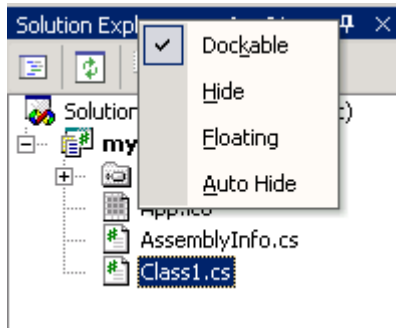
Let's look at how to navigate the IDE, starting by looking at the windows.

You can activate a window by clicking anywhere within the window, or on the title bar (the top bar, with X at its top-right corner).

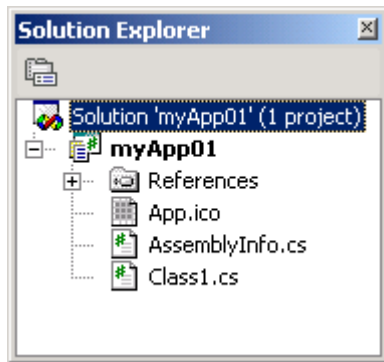
Windows states:

Each window in the IDE panes can have one of four states:

Dockable, Hidden, Floating, and Auto Hide mode. You can see and select the window state by right-clicking on the title bar (the top bar, which is blue)



Docking and Floating are the default behaviours for all windows, depending where you release the window, after moving it around. The window above is docked (pinned down). I can tell because it says so, and because of the pin (next to the X at the top right corner). The window below, on the other hand, is floating. No pin!



Moving a window around:

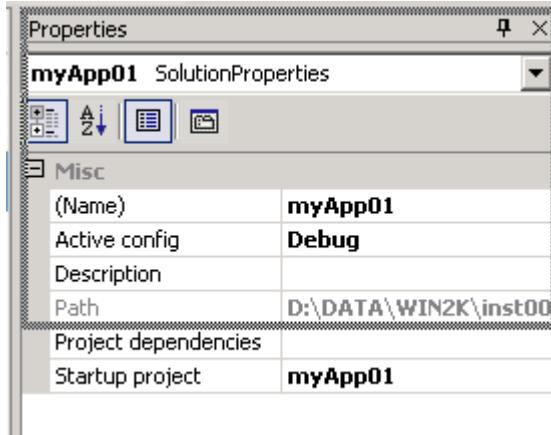
You can move a window by pointing at either the title bar or the tab (in case of tabbed windows), then pressing and holding down the left mouse button, and dragging it while moving the mouse.

Docking and Floating:

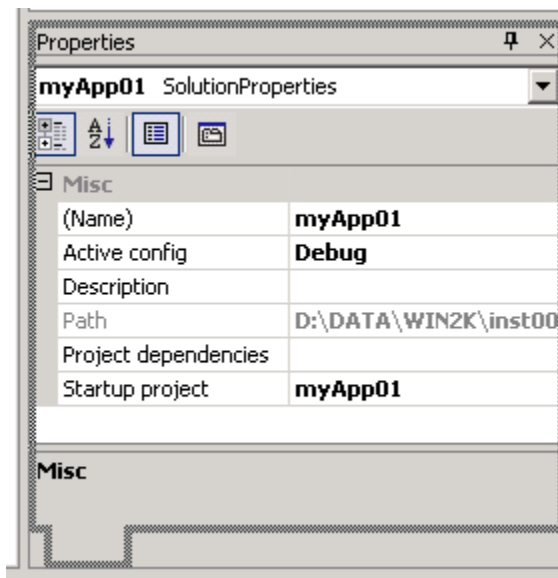
While moving a window around, keep an eye on the outline, which will indicate what it would look like when you release the mouse. Examples are, when you move a window on top of another window, near the title bar, it will take the shape of the underlying one, and a tab is added - for as long as you are in the same position - to the tabs at the top (editor and output panes) or the bottom for all the

other panes). Releasing the window in this case would dock it on top of the existing one.

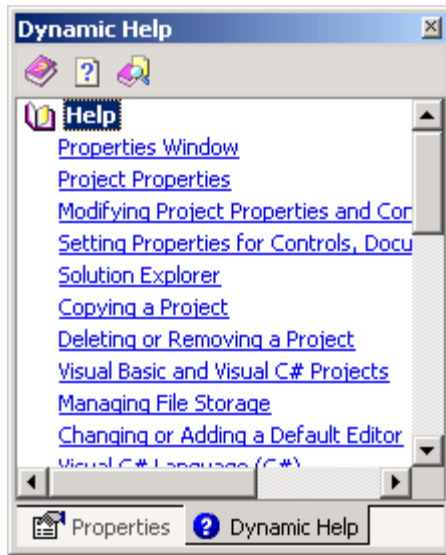
The figure below displays a thick grey outline or a window frame, showing where the docked window will end up, if I released the mouse.



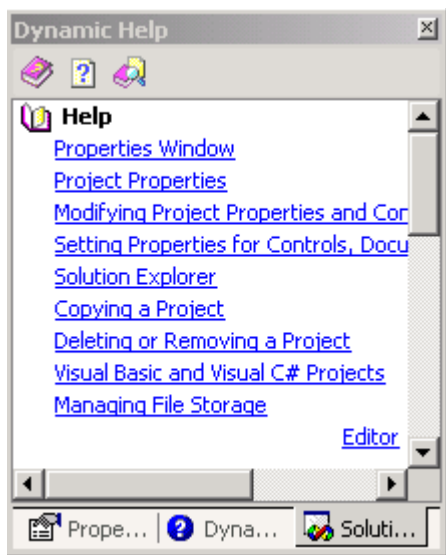
Now examine the figure below, you can see that the window floating above the properties window is about to dock on top of it, creating a tab.



If the underlying window (properties in this case) already had a tab, then you wouldn't see the grey outline, but rather a new tab displaying the name of the floating window. This can be a little confusing, if not intimidating for those new to the product. If the result of the above operation was the figure below; the dynamic help window is added as a tab, and on top of the properties window.



If I were to drag the Solution Explorer window on top of the above, I'd get something like the figure below:



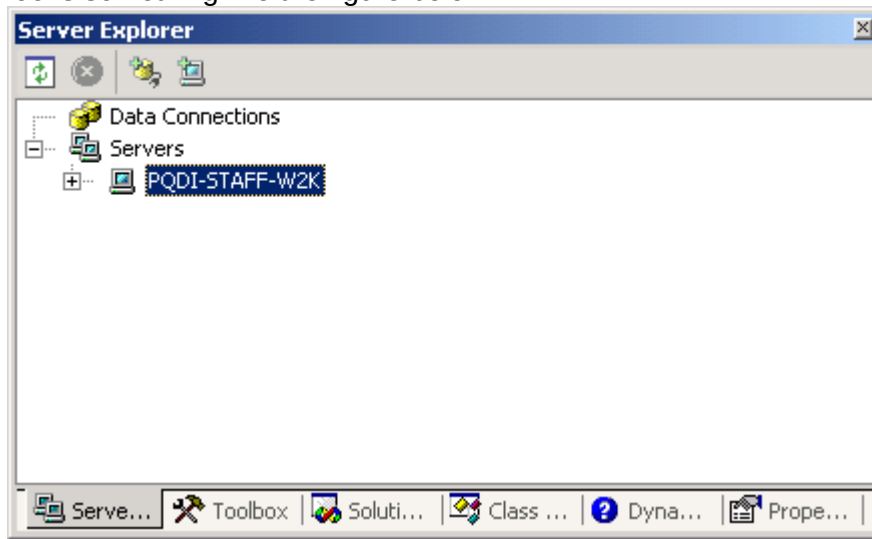
Notice that while floating on top of the window, the underlying window's title bar always remains grey, reminding you that the active window is actually the one you are dragging.

If you change your mind about docking the window at this location, all you have to do is keep holding the left mouse button down and use your other hand, to press ESC!

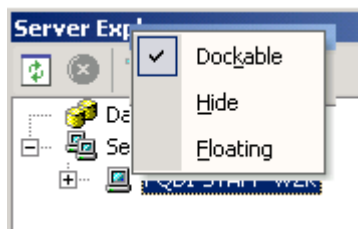
Auto Hide:

Auto Hide is simply a state of a docked window (only available if the window is set to dockable and actually docked into one of the 5 panes), where the PIN is set horizontal (not pinned), and the window would collapse to a vertical TAB, displaying the name of the top window, and the icons of all the other underlying windows.

Take the following case, where I've docked all windows on top of each other. It looks something like the figure below:

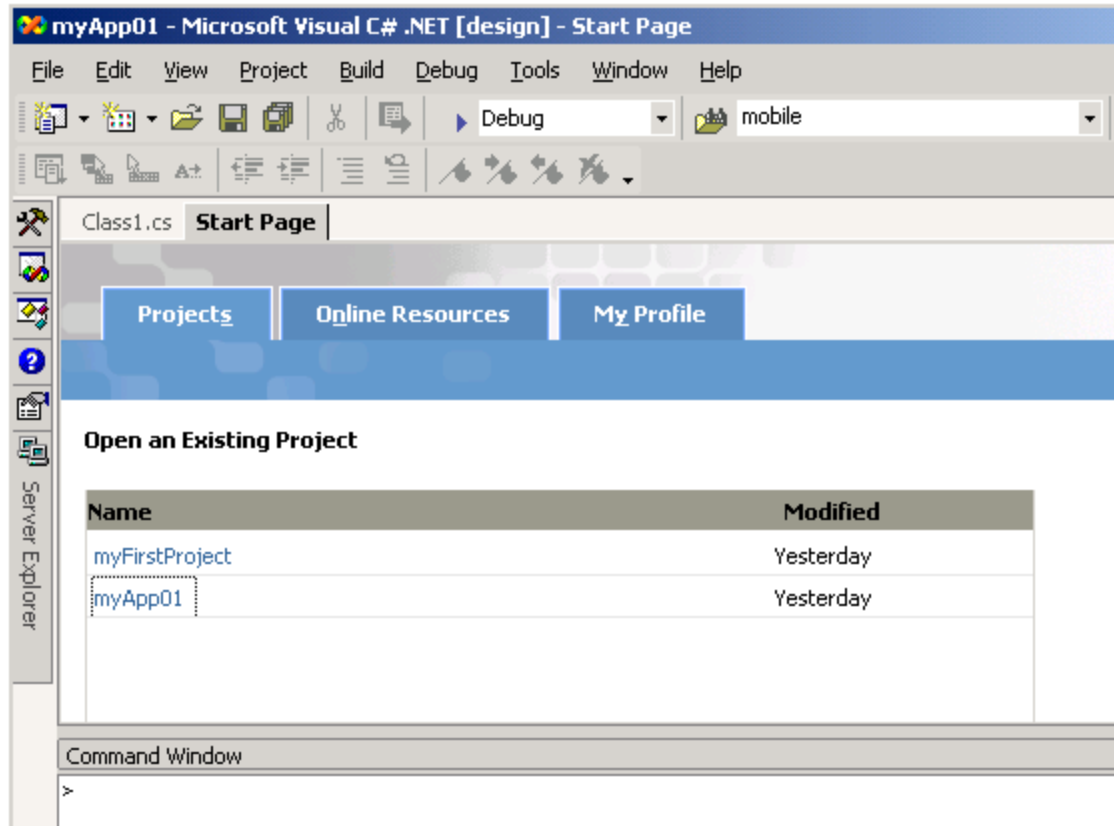


You can tell that this window is floating, because there is no PIN. This means if I right click on the title bar, I get only three options. No Auto Hide option.



So I first have to dock the window by dropping it into a pane, I chose the Toolbox pane.

Then either by simply clicking on the PIN or by right-clicking the title bar and selecting Auto Hide, both actions turn the PIN horizontal and hide the window. Resulting in a figure similar to the one below:



The reason it is dubbed Auto Hide is, because the system hides the window the moment the mouse pointer leaves it. Inversely the moment you point at the icon of the window you want, it will slide open. While the window is open, you can PIN it down, by clicking on the pin, or, you know how.

Not Dockable:

You can select the window to be **not dockable**. This means it will be part of the Editor pane., either as a tabbed window or in a split pane.

You can only deselect the dockable option if the window is docked and pinned.

Closing (Hiding) a window:

You can always close the window by press the X at the top-right corner. Or right click and select Hide from the context menu.

Showing (un-hiding) a window:

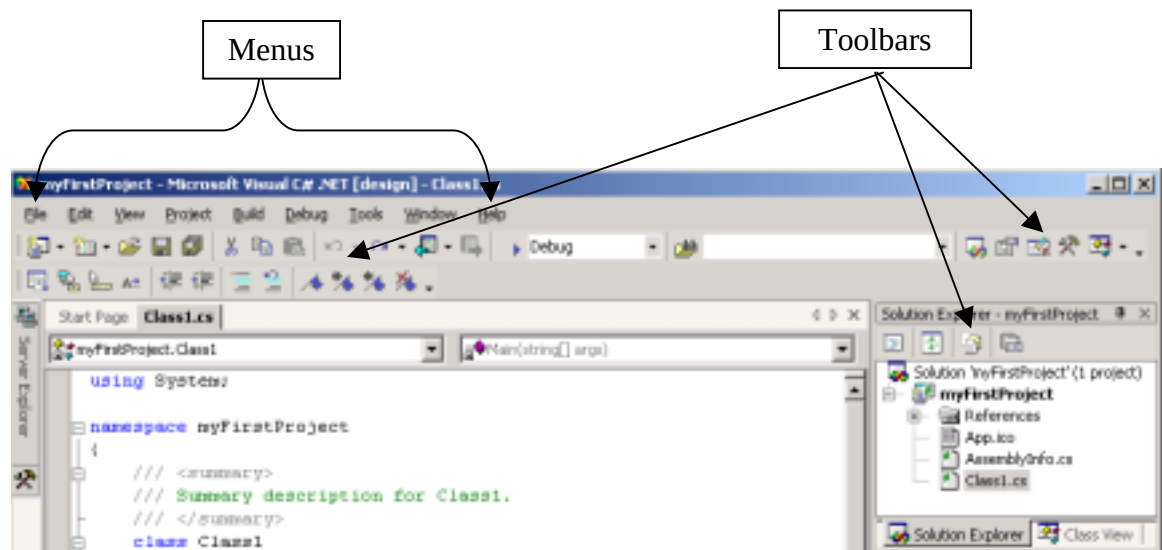
Once you hide or close a window, it will disappear from your IDE. You can recall the window(s) using Menus, Toolbars and keyboard short cuts, described in section “3.2.1.6 Parts of the Visual Studio .NET IDE”.

3.2.1.5.2 Menus and toolbars

We will not discuss each menu option and toolbar item, in this section, but we'll introduce the relevant menu commands and tool bars as we need them.

The IDE menus are standard Windows-style drop-down menus with one or more levels of depth, which can be accessed using the mouse, keyboard short cuts (ALT+) or both.

You can also execute menu commands from the command window in the > (command) mode.

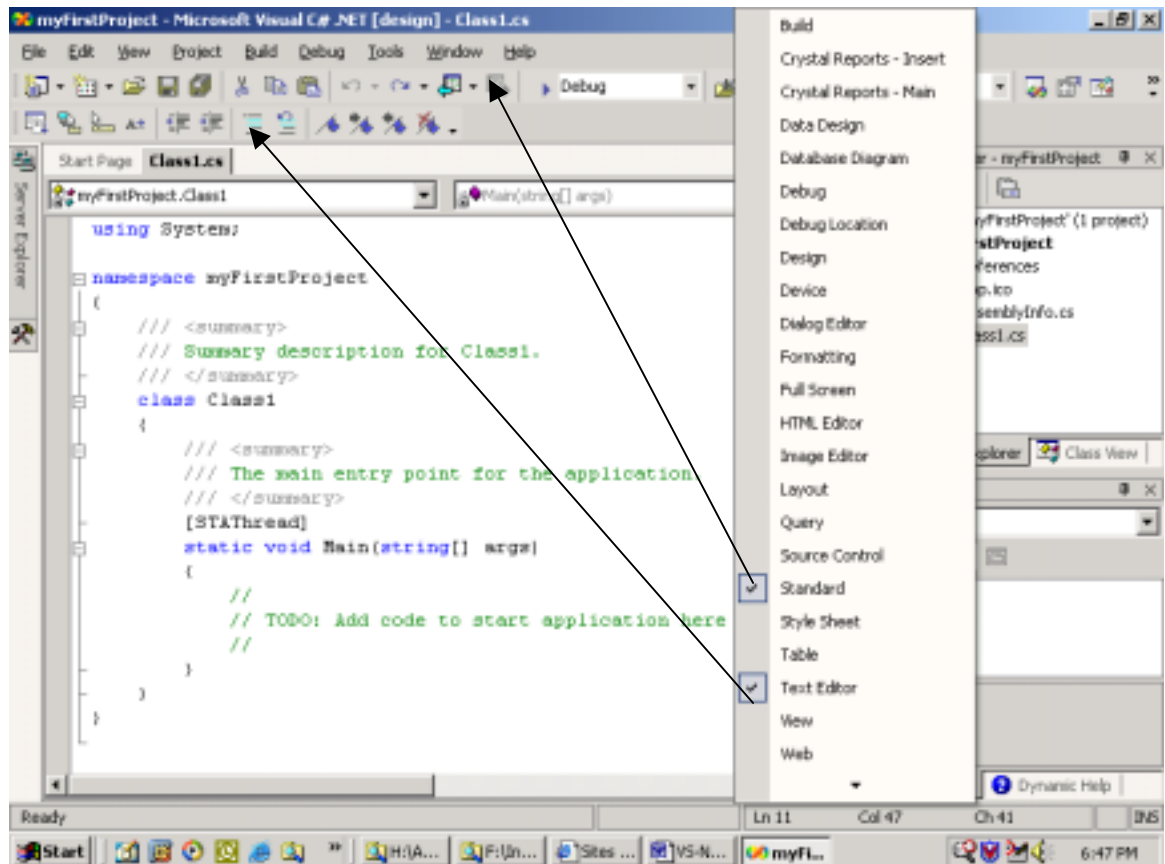


The toolbars provide a convenient, single mouse-click access to various common commands.

They are laid out in small icons across the screen, right under the menus. The number of toolbars displayed at one point depends on your settings and the state of the IDE. You could choose to display or hide toolbars and re-arrange their layout as follows:

To display or hide a toolbar:

Right click anywhere, in the toolbars area. You will get a rather long pop-up menu, which displays all available toolbars. You can click the one you would like to toggle on/off. The ones currently displayed will have a tick mark in front of them. The figure below demonstrates a sample.



Tool tip:

If you hover the mouse, over a toolbar icon for about 2-3 seconds you get a tool tip (little yellow box, explaining the task carried out by that particular toolbar item.)

Accessing toolbars:

Accessing toolbars generally means clicking a button that represents an action, which is carried out immediately. However, there are three main categories of toolbars:

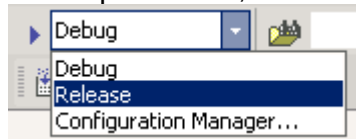
- Simple toolbars with one-button = one action type. Such as:



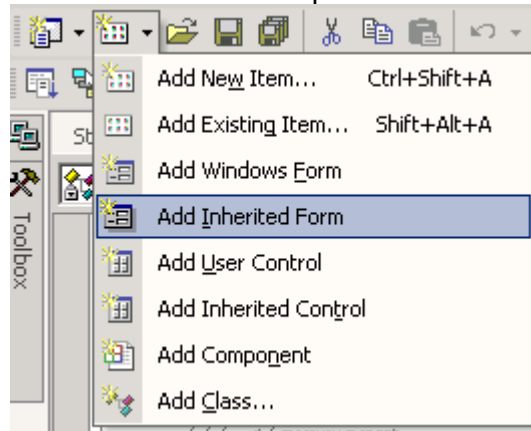
for Save, Save All, Cut, Copy and Paste respectively, from left to right.

- Simple with settings, Same as the previous one, but with the button producing different results depending on the selection from the

accompanying drop-down list. The RUN button  would produce different output depending on whether you select Debug or Release, from the drop down list, access by clicking the down arrow .



- c. Drop-down toolbars with multiple buttons, grouped under one category for relevance. An example of this is the add-new toolbars:



Moving toolbars:

At the left edge of each tool bar, there is a vertical bar made up of dashes, as

highlighted by the oval shape, in the figure.

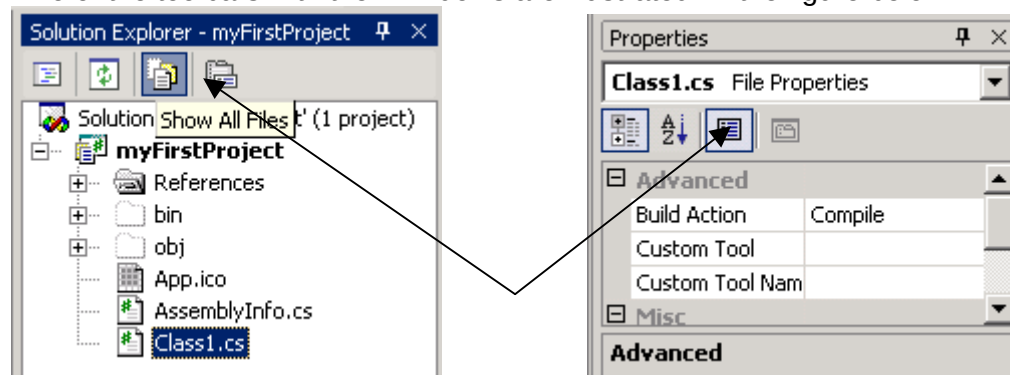


If you point at that bar, the mouse pointer will turn into a cross with four pointed arrows

Press and hold down the left mouse button, then drag it around. The toolbar should start moving around, following your mouse movements. Drop it when you have it where you want it, or press ESC while still holding the mouse, to cancel the action.

The various windows in the IDE panes, such as the Solution Explorer and Properties also have their own toolbars. Although you can not manipulate these toolbars, as you could, the main ones.

Two of the toolbars with their windows are illustrated in the figure below.



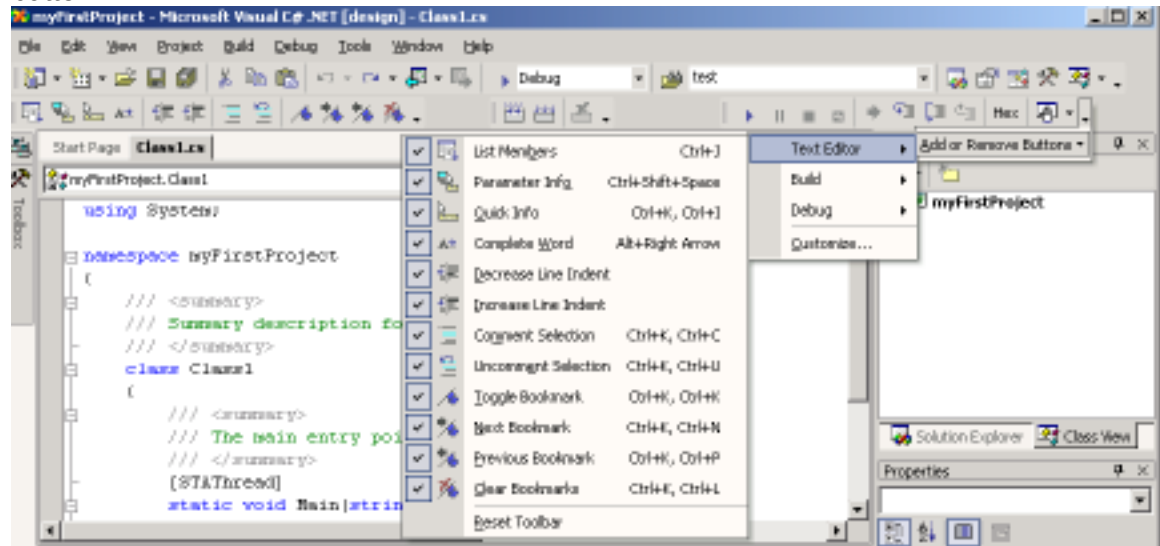
Customizing toolbars:

You can control the number of buttons displayed on a particular toolbar, depending on frequency of use and available screen real-estate.

You can accomplish this by clicking on the down-arrow at the end of the row, where toolbars are displayed. You will get access to a drop-down menu allowing

you add or remove buttons to/from the toolbars displayed on that row (to the left of the ).

The figure below demonstrates the result of accessing the option on the 2nd row of toolbars, which includes “Text Area”, “Build” and “Debug”. Clicking on the desired group opens yet another menu listing the currently displayed buttons on that toolbar. You can toggle the display status of each button by clicking on each button.



The most commonly used (by me at least) are the following toolbars: Standard (Add project/item, Save/all, Navigate back/forward), Text editor (Comment, Bookmark), Build (Build and Build all), Debug (Start, Stop, Step into/out).

It takes a while to get used to the toolbars, but they are great boost to productivity.

3.2.1.6 Using the Code Editor

The code editor is pretty standard editor with all the keyboard short cuts behaving in a manner consistent with expectations.

Left and Right Arrows, moves the cursor one letter in the direction of the arrows. CTRL + Left/Right arrows jump the cursor by one word in the direction of the arrow.

Page Up and Page Down do just that.

Home and End keys move the cursor either to the end or the beginning of the current line.

CTRL + Home moves to the top of the file, CTRL + End to the bottom of the current file.

CTRL+A selects all text

CTRL+C, CTRL+X and CTRL+V do what they always did, copy, cut and paste.

CTRL+Z and CTRL+Y reverse your actions with Undo and Redo respectively.

CTRL+S and CTRL+SHIFT+S do save and save all respectively.

CTRL+F and CTRL+H would trigger find and find-and-replace dialog boxes respectively.

And many more features that you can access from the menus, toolbars, keyboard shortcuts or a combination of all of the above.

There is also a new set of keyboard short cuts, known as chords, like the piano chords! Simply put, they are CTRL+M followed by CTRL + some letter. My favourite is CTRL+M followed by CTRL+M and CTRL+M followed by CTRL+L. This collapses/expands the code outline regions. We discussed the collapsible regions earlier in section 3.2.1.4

To learn more about Outlining and Hiding Code

<http://msdn.microsoft.com/library/en-us/vsintro7/html/vxtskOutliningCode.asp>

You can learn all that you need to learn about the editors at:

<http://msdn.microsoft.com/library/en-us/vsintro7/html/vxtskEditingCodeText.asp?frame=true>

3.2.1.7 Compiling and Running programs

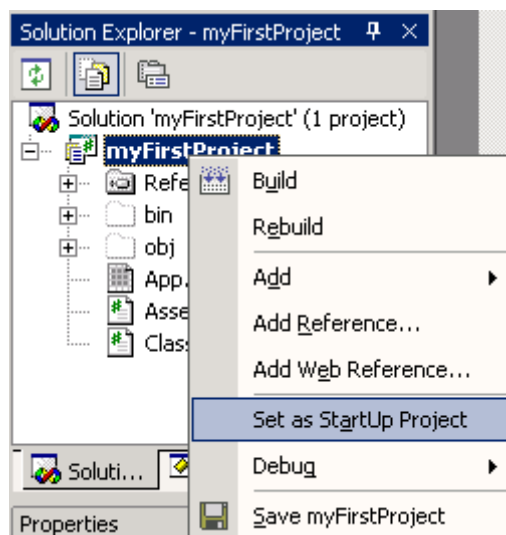
Compiling a program could never be easier. It is as simple as clicking one of these

toolbar buttons: . You can also access the same command(s), from the **Build** menu. And finally **CTRL+Shift+B**, is the short cut for the Pros!

Clicking the Start button on either the Debug  or the Standard  toolbar will also compile the code implicitly.

Running the successfully compiled code can be achieved either by pressing CTRL+F5 or by selecting the appropriate command from the Debug menu. This will run the default (star tup project)

Start up project: If you have more than one project in your solution, you must make sure that the project intended for debugging is set as the start-up project. You can do this by right-clicking on the project name (in the Solution explorer) and selecting **Set as StartUp Project**.



All going well, you should see the results of the program.

Note:

The above mentioned Start buttons will also invoke the RUN function, but should you have any breakpoints, set in your code (which will be discussed later), then you will step through the code rather than see the final result.

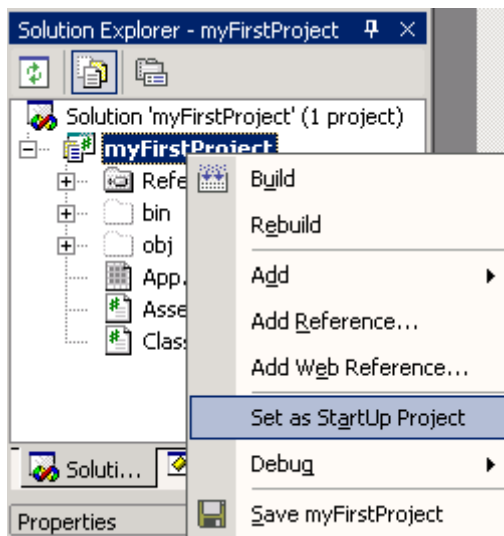
3.2.1.8 Debugging programs

Debugging is something you will have to do at one point in your programming life time. Whether you use a special tool, a debugger or write program statements to trap errors you always need some mechanism to help find bugs and fix them.

VS.NET comes with a powerful debugger that is integrated in the IDE environment. You can invoke the debugger by Selecting **Debug → Start** from the menus or by pressing **F5** or by using either the Debug or the Standard tool bar.

Before you do that however, you need to understand a few concepts:

1. **Break points:** You can insert breakpoints in your code, at any points of interest, where you intend to start taking a closer look at the code. The effect of inserting a break point is that the code will execute until that point and stop, entering a Break-mode. From that point onward you can step through the code using F10/F11 (into/over).
2. **Monitoring values:** VS.NET provides a set of tools to compliment the debugger, such as up to 4 Watch-windows, up to 4 Memory-window, and nine other windows to help you accomplish the task of finding that problem and fixing it. You can display all the different windows from the **Debug → Window** menu option.
3. **Start up project:** If you have more than one project in your solution, you must make sure that the project intended for debugging is set as the start-up project. You can do this by right-clicking on the project name (in the Solution explorer) and selecting **Set as StartUp Project**.



You can learn everything there is to learn about the VS.NET debugger at:

<http://msdn.microsoft.com/library/en-us/vsdebug/html/vcoriDebuggingUIReference.asp>

You can also get this information from the VS.NET documentation accompanying the product.

Exercise:

Create a new C# project

Type the following code in the code editor

Save all by selecting the appropriate toolbar, menu item or keyboard short cut

Compile the code using the appropriate toolbar, menu item or keyboard short cut

Correct any problems that may reported by the compiler

Run the compiled program the appropriate toolbar, menu item or keyboard short cut

4 Program structure and general concepts

4.2 Overview:

C# belongs to the C/C++ family of languages, from which C#'s syntax is derived, and hence shares this family's characteristics such as:

1. Each statement must be terminated using a semicolon (;)
2. Statement could be a single command terminated with a semicolon or a block of commands, enclosed between a pair of curly braces {}. Each command must also be terminated with a semicolon.
3. The application must have an entry point, a method (function) named Main(), followed by a block of zero or more statements.
4. Most of the statement syntax, and the language constructs' format is very similar to C/C++ as you will learn shortly.

C# also shares in many features of the .NET infrastructure, which is common with other languages such as VB.NET and the managed C

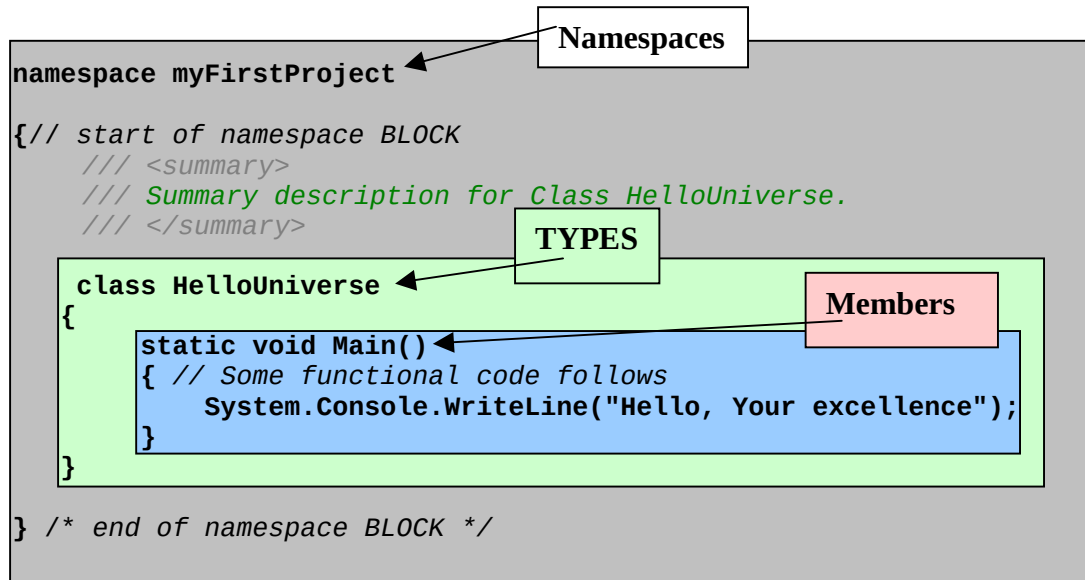
4.3 Program structure

Let's examine the general program structure (layout).

4.3.1 Comments:

Let's us start with one of the most underrepresented, yet very important element of programming, the comments. I can't emphasize enough, the importance of documenting your code, and C# has more than one way to help us do just that.

1. The C++ style double forward slashes (//) preceding a single line of comment(s).
2. New to C# is the XML documentation which requires some level of XML tags knowledge, and starts with triple-forward slashes (///), followed by an XML tag pair `///<someTag> ///</someTag>`. Sandwiched between the pair of tags is one more lines of the actual comment, or description, also preceded with `///`. You can generate an XML documentation of your code using the compiler tag `/doc`. So typing:
>csc hello.cs /doc:myXMLdoc.xml will generate myXMLdoc.xml file that you can view using any XML viewer, such as internet explorer.
3. Just for the sake of those with C programming experience, the block comment, which starts with a single forward slash followed by an asterisk (`/*`), and ends with an asterisk followed by a single forward slash (`*/`), with one or more lines of comments enclosed between the `/*` and `*/`.



4.3.2 Statement blocks:

Just like in C/C++, and even more so in C#, all code must be enclosed in some block. To mark the start and end of a code blocks we use a pair of curly braces. We use `{` to mark the start of a block, and `}` to mark the end of a block.

4.3.3 Namespaces:

As you will see throughout the programs in this book, and as we mentioned earlier in the .NET framework section, namespaces are a very important part of programming for the .NET.

C# allows you define your own namespaces, and access any available ones, including the namespaces provided by the base classes in the .NET framework.

Namespaces are just a bundling mechanism, where you enclose all components of a certain system between the starting and ending curly braces `{ }`, following a namespace declaration.

To declare a namespace, use the keyword `namespace` followed by a name of your choosing. We will cover naming conventions later in this chapter.

Using namespaces yields two important results:

1. It enables you to define boundaries of systems, or subsystems, by properly naming them. In the above example: **myFirstProject**
2. It helps avoid name conflicts, in case any component such as a class has a similar name to another component in the environment. This is accomplished using the dot operator (`.`) to reference the names, starting with the `namespace.TypeName.MemberName`. To execute the above listed code the Class loader would invoke something like this: **myFirstProject. HelloUniverse.Main()**.

4.3.4 Types:

Namespaces can only contain types or another namespace. Specifically, you can define one or more of the following inside a namespace block:

- another namespace
- class
- interface
- struct
- enum
- delegate

As we discussed earlier in the Common Type System (CTS) section, of the .NET Framework chapter, .NET provides two types, Value type and Reference type. Class, Interface and Delegate are Reference types, whereas enum and struct are value types, although generally user defined.

4.3.5 Class:

In the above example we have a single Reference type (**Class**) defined as **class HelloUniverse**

Followed by a block of code enclosed between the curly braces { }.

It important to note here, that everything in C# must belong to a *class*, or a *struct*, which is a lightweight form of a class. We'll discuss both in detail, later in this book.

There is no notion of global functions in C#. Also the order of definition is irrelevant, unlike C/C++, where you had to prototype, and/or use .h files. The code of a class is all inline, and no multiple files are allowed for splitting a single class.

4.3.6 Members:

We call them members, because they are members of a "Class".

As mentioned earlier, everything in C# MUST be contained within a class or struct.

Members can be member classes, structs, enums and basic value types, such as int and float. Basic value type members are known as **fields** of a class.

Members can also be functions, known as **Methods**.

In the example on hand, we have defined a method **Main()**. If you have written any programs in a C-like language, then you probably know that a function is always followed by parenthesis () and a block of code immediately after the parentheses, which is represented in this case by a pair of curly braces { }.

What goes inside the block is presumably the functional code that produces the intended result.

Class members are referenced using the dot (.) notation, as mentioned in the namespace section. So to access the **Main()** member of the **HelloUniverse** class, which belongs to the **myFirstProject** namespace, one would write: **myFirstProject. HelloUniverse.Main()**

You can write as many Methods (functions), within a class, as your problem requires. Then you would access them from other classes in the manner described above.

4.3.7 Inside Main():

Main() method is also the entry point for the whole application. We will discuss the Main() method itself in section 3.5.

For the moment, let us examine what is enclosed within the Main() method: **System.Console.WriteLine("Hello, Your excellence");**

The first word **System** is the namespace that is defined in the .NET framework. The second word **Console** represents the class name, which is comparable to our own **HelloUniverse**. Finally we have the method **WriteLine** that is the real target for our task. In simple terms **WriteLine** is the function that will take the value between parentheses "Hello, Your excellence" and display it on the screen.

So the whole thing could be read as Method **WriteLine** that is a member of **Console** class and belongs to the **System** namespace.

4.3.8 The System.Console class

In a console application, which targets the text-based console, the most important component is the Console class, which is part of the System namespace, defined by the .NET framework.

The console class exposes several methods, the most important of which is the **WriteLine()** methods that send output to the current output device, which is generally the screen, and **ReadLine()** that gets input from the standard input device, which is generally the keyboard.

WriteLine() method appends a line feed/carriage return to the end of the output. The general format for **WriteLine** method is:

```
Console.WriteLine ();           // displays a blank line
Console.WriteLine ("{0}, {1}", x, y); //displays values of first variable
                                     //{0}=x and second variable {1}=y
```

```
Console.WriteLine ("{0:g}, {1:x}", a, b); //displays values of first variable
                                     //in general format {0:x}=b and
                                     //second variable {1:x}=y in
                                     //hexadecimal format
```

ReadLine() method gets input until carriage return is detected. The general format for **ReadLine()** method is:

strVar = Console.ReadLine (); // Return a line of inputs from the keyboard

Summary:

To recap, a C# application is generally written in the following format:

```
namespace myNamespace
{
    class MyClass1
    {
        // Members
    }

    class MyClass2
    {
        public static void Main()
        {
        }
        // other Members
    }
}
```

4.4 Namespaces and “using” directive

Namespace is a hierarchical system to organize C# code into manageable and clearly understandable structure.

Namespaces are the only means by which a C# compilation unit (source file) can reference and import other code.

You can think of it as equivalent to the `#include` directive of C++, except far better structured and much more elegant.

A good example of namespace; is the system namespaces, provided by the .NET base class services, which give you access to the various functional components that, the .NET has to offer. The system namespaces includes *System*, *System.IO*, *System.Windows.Forms*, *System.XML*, *System.Web* to name a few.

4.4.1 Accessing namespaces:

You can access name spaces by directly referencing them in the code, as we have done so far. We are said to be using a “Fully Qualified Name” access method.

The fully qualified name of a class is the class name prefixed with the namespace. The dot “.” character is used to separate namespaces nested within other namespaces. It is also used to separate the class name from the innermost namespace. For example, within the *System* namespace is a class named *Console*. The format to access this class is:

System.Console

The dot separator is once more used to separate the members of the class, such as the method *WriteLine* in the following example:

System.Console.WriteLine(“Somestring”)

4.4.2 Importing namespaces with the using keyword

Accessing namespaces using the fully qualified name can be a little tedious, and error-prone, when you have to write several lines of code (hundreds?). Instead, C# provides a keyword that imports the namespace into your file, sparing you any further typing of the name in the subsequent lines of your program. This keyword is the **using** directive.

You can use any number of **using** directives, but you must place them all at the beginning of the source file, prior any of your own code.

Let us rewrite our *helloUniverse* class to show you how the using keyword, displayed on the first line of code (just as in the C/C++ **#include** directive).

```
Using System;
namespace myFirstProject
{
    class HelloUniverse
    {
        static void Main()
        {
            // Using the class directly
            Console.WriteLine("Hello, Your excellence");
            // Using the class directly
        }
    }
}
```

```

        Console.WriteLine("Hello again, Your excellence");
        Console.WriteLine("Hello 3rd time, Your excellence");
    }
}

} /* end of namespace BLOCK */

```

4.4.3 Creating your own namespaces

Namespaces can be very critical in the success of any programming project, utilizing C#, and should be used as much as possible.

Although it is entirely up to the system designer, to structure the hierarchy of the namespace structure, there some best practices documented for such hierarchy. One possible naming convention for your projects may be:

CompanyName.Division.Project.Subproject

It worth noting here that, you can have any number of levels, in a nested namespace structure. But be judicious with the length of names.

The example we have been looking at so far, demonstrates the syntax for creating a namespace **myFirstProject**:

```

Using System;
namespace myFirstProject
{
    class HelloUniverse
    {
        static void Main()
        {
        }
    }
}

```

5 Data Types and variables

As we briefly discussed in the chapter about .NET framework, all C# types are provided by the .NET framework's Common Type System (CTS). There are two major types, basic data types intrinsic (predefined) by the CTS, and user defined types that you can also define yourself.

5.2 Built-In Data Types

There are two major types, predefined by the CTS, which are value type and reference type.

5.2.1 Reference Type

Reference types are type safe pointers, similar to the C++ references. Being type safe means they not only point to an object, but to an object of specific type.

Instances of classes are reference types (Remember instantiation?).

Reference types are allocated on the heap.

Unlike the value type, Reference types can be null.

The predefined reference types are object and string. The type object is the ultimate base type of all other types.

Object	System.Object	Object type	object i = 9; object i = 9;
string	System.String	Unicode chars	String s = "hello";

In addition to Object and String types, there are several other Reference types defined in the .NET and made available through C#, including:

- Classes
- Arrays
- Interfaces
- Delegates

Classes and Arrays will be covered later in this chapter, while Interfaces and Delegates will be discussed in later chapters.

5.2.2 Value Type

Value types contain actual data, hence cannot be null. Value types are allocated memory on the stack. They are assigned values as soon as they are created. They are all primitive, or basic data types, that exist in most programming languages.

When passing value types as parameters to methods, you actually pass a copy of the value.

The predefined value types include signed and unsigned integer types, floating point types, char type and the types bool, and decimal, which is special floating-point type of very high precision.

The bool type is used to represent values that are either true or false.

The introduction of `bool` in C#, helps eliminate the C++ coding error caused by the uses of “=” instead of “==”.

The table below lists the different pre-defined value types, and shows you an example of their use in C# code.

Type	Class	Description
Signed Integer types:		
sbyte	System.SByte	8-bit Signed (values -128 to 127).
short	System.Int16	16-bit Signed (short) (values -32,768 to 32,767).
int	System.Int32	32-bit Signed integer values between -2,147,483,647 and 2,147,483,647.
long	System.Int64	64-bit Signed integers ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
Unsigned Integer types:		
Byte	System.Byte	8-bit Unsigned (values 0-255).
Ushort	System.UInt16	16-bit Unsigned (short) (values 0 to 65,535).
UInt	System.UInt32	32-bit Unsigned integer values between 0 and 4,294,967,295.
Ulong	System.UInt64	64-bit Unsigned integers ranging from 0 to 0xffffffffffffff.
Floating point types:		
float	System.Single	Floating point number. Holds the values from approximately $\pm 1.5 \times 10^{-45}$ to approximate $\pm 3.4 \times 10^{38}$ with 7 significant figures.
double	System.Double	Double-precision floating point; holds the values from approximately $\pm 5.0 \times 10^{-324}$ to approximate $\pm 1.7 \times 10^{308}$ with 15-16 significant figures.
Boolean and Character types:		
Bool	System.Boolean	Boolean type; holds a true or false.
char	System.Char	Unicode char
High precision float types:		
decimal	System.Decimal	Fixed-precision up to 28 digits and the position of the decimal point. This is typically used in financial calculations. Requires the suffix "m" or "M."

All the basic built-in types are included in the `System` namespace for C#. The predefined value types are derived from **`System.ValueTypes`** namespace, which itself is derived from the **`System.Object`** namespace.

You can inquire about the base type of an existing variable by the code:

`var.GetType().BaseType`

Where **`var`** is the variable, whose type you want to determine.

5.2.3 Variables and intrinsic Data Types

In C# we operate on data using variables, to which we assign values, and then manipulate those values. CTS ensures the type safety of the variables we declare, using the built-in as well as user defined types.

A variable is a storage location with a type. Variables can have values assigned to them, and that value can be changed programmatically. There are some guidelines that should be followed when creating variables in C#.

5.2.3.1 Naming Variables

You create a variable by declaring its type and then giving it a name. You can initialize the variable when you declare it, and you can assign a new value to that variable at any time, changing the value held in the variable. This is illustrated in Example below.

There rules and guidelines to follow in naming variables for use in C# :

Rules:

- Variables must start with a letter that can be followed by any combination of alphanumeric characters and underscores
- C# variable names are case sensitive
- C# has 79 reserved keywords. Variable names can not be a reserved C# keyword (listed in the table below). Unless preceded with the @ sign (then they are not the same anymore, are they?)

Recommended guidelines:

- Avoid using abbreviations
- Use nouns and noun phrases for classes
- Use Verbs and verb phrases for Method names
- Use PascalCasingNotation for classes and Methods
- Use camelCasingNotation for variables and parameters
- Avoid using all upper case letters

Keywords in C#: List of keywords that are reserved in C#.

abstract	const	extern	int	out	set	try	void
as	continue	false	interface	override	short	typeof	while
base	decimal	finally	internal	params	sizeof	uint	
bool	default	fixed	is	private	stackalloc	ulong	
break	delegate	float	lock	protected	static	unchecked	
byte	do	for	long	public	string	unsafe	
case	double	foreach	namespace	readonly	struct	ushort	
catch	else	goto	new	ref	switch	using	
char	enum	if	null	return	this	value	
checked	event	implicit	object	sbyte	throw	virtual	
class	explicit	in	operator	sealed	true	volatile	

5.2.3.2 Categories of Variables

There are seven categories of variables that can be defined in C#. These are, **Static variables**,

If declared with a **static** modifier, a variable created upon loading of its type, and lives throughout the program lifetime

Instance variables,

If declared without the **static** modifier, then, its lifetime (scope) is the same as the class or struct that it belongs to. Everything must be a member of a class or struct.

Array elements,

Like instance variables, array elements are in existence for as long as the array is referenced.

Value parameters,

Any parameter declared without any modifiers, except for the type is a value parameter. Its lifetime is from invocation of the method, to the return from it.

Reference parameters,

Reference parameter is a variable declared with a **ref** modifier. It points to the same location as the value of the variable in the calling function.

Output parameters

A variable declared with the **out** modifier uses the same location as the variable supplied by the calling method.

Local variables

Any variables declared within a method, property or an indexer.

5.2.3.3 Declaring and Initializing Variables

You create a variable by declaring its type and then giving it a name.

You can initialize the variable when you declare it using the “=” operator.

The table below lists all the different value types, and the syntax used to declare variables of these types.

Examples of how to declare variables of the most common data types

Description	Example
8-bit Signed (values -128 to 127).	<i>sbyte var = 12;</i> <i>sbyte var1, var2;</i>
16-bit Signed (short)	<i>short var = 09;</i>
32-bit Signed integer values	<i>int var = 2001;</i>
64-bit Signed integers	<i>long var1 = 1309;</i> <i>long var2 = 64L;</i>
8-bit Unsigned (values 0-255).	<i>byte var1 = 12;</i>
16-bit Unsigned (short) (values 0 to 65,535).	<i>ushort var1 = 09;</i>
32-bit Unsigned integer values	<i>uint var1 = 2001;</i> <i>uint var2 = 1964U;</i>
64-bit Unsigned	<i>ulong var1 = 2705;</i> <i>ulong var2 = 1978U;</i>

	<i>ulong var3 = 1958L;</i>
Floating point number with significant figures.	<i>float var = 1.23F;</i>
Double-precision floating point; holds the 15-16 significant figures.	<i>double var1 = 9.01;</i> <i>double var2 = 12.9D;</i>
Boolean type; holds a true or false.	<i>bool var1 = true;</i> <i>bool var2 = false;</i>
Unicode char	<i>char var = 'A';</i>
Fixed-precision up to 28 digits and the position of the decimal point. This is typically used in financial calculations. Requires the suffix "m" or "M."	<i>decimal var = 1.23M;</i>

5.2.3.4 Assigning values to Variables

You can assign a new value to that variable, changing the value of the variable, at any time.

All variables in C# must be assigned a value before they can be used. We use the assignment operator "=" to assign values to variables. You can also assign values to variables upon declaration, in one statement, as mentioned above.

This is illustrated in Example below.

```
class DecalreVars
{
    static void Main( )
    {
        int myInt = 12;
        System.Console.WriteLine("Initial Value, myInt: {0}", myInt);
        myInt = 13;
        System.Console.WriteLine("New Value, myInt: {0}", myInt);
    }
}
```

C# provides compound assignment operator, which updates the current value of the variable rather assigning a new value. The example below demonstrates the concept.

```
class DecalreVars
{
    static void Main( )
    {
        int myInt = 12;
        System.Console.WriteLine("Initial Value, myInt: {0}", myInt);

        //Compound assignment updates the value
        myInt = myInt + 13;      // myInt = 25
        System.Console.WriteLine("New Value, myInt: {0}", myInt);
        myInt = myInt - 13;      // myInt = 12
        System.Console.WriteLine("New Value, myInt: {0}", myInt);
        myInt = myInt * 2;       // myInt = 24
        System.Console.WriteLine("New Value, myInt: {0}", myInt);
        myInt = myInt / 2;       // myInt = 12
        System.Console.WriteLine("New Value, myInt: {0}", myInt);
    }
}
```

```

    }
}

```

C# also provides a shorthand form of the compound assignment operator, which updates the current value of the variable rather assigning a new value. This concept is illustrated in Example below.

```

class DecalreVars
{
    static void Main( )
    {
        int myInt = 12;
        System.Console.WriteLine("Initial Value, myInt: {0}", myInt);

        //Compound assignment updates the value
        myInt += 13;    // myInt = 25
        System.Console.WriteLine("New Value, myInt: {0}", myInt);
        myInt -= 13;    // myInt = 12
        System.Console.WriteLine("New Value, myInt: {0}", myInt);
        myInt *= 2;     // myInt = 24
        System.Console.WriteLine("New Value, myInt: {0}", myInt);
        myInt /= 2;     // myInt = 12
        System.Console.WriteLine("New Value, myInt: {0}", myInt);
    }
}

```

5.3 Operators

Operators are C# elements designed to produce a value different than the values operated on. There values operated on are known as operands. There are three categories of operators, based on the number of operands they expect. These are Unary, binary and ternary operators. The table below lists the most commonly used operators.

Operator category	Operators	Explanation
Assignment	<u>=</u> <u>+=</u> <u>=</u> <u>-=</u> <u>*=</u> <u>/=</u> <u>%=</u> <u>&=</u> <u> =</u> <u>^=</u> <u><<=</u> <u>>>=</u>	Assigns values to variables. Operands on both sides of the op must be the same type.
Arithmetic	<u>+</u> <u>-</u> <u>*</u> <u>/</u> <u>%</u>	The usual arithmetic operations
String concatenation	<u>+</u>	Appends a string to the end of another
Increment, decrement	<u>++</u> <u>--</u>	Unary operators to inc and dec
Relational	<u>==</u> <u>!=</u> <u><</u> <u>></u> <u><=</u> <u>>=</u>	Used to compare two values
Member access	<u>.</u>	Used to access members of a class also used to form fully qualified names
Indexing	<u>[]</u>	Used with arrays and indexers
Cast	<u>()</u>	Used to explicitly change one type to another

Conditional	?:	Selects between 2 expressions depending on Boolean value
Object creation	new	Used to create objects on the heap

Let us use some of the most common operators in an example:

```
int x=0
```

```
x = x+9;    can be written as x+=9;
```

```
x = x+1;    could be written as x++;
```

```
x = x-1;    could be written as x--;
```

5.4 Expressions and operator precedence

Expressions are any valid statement that results in a value. Examples of valid expressions are:

```
int x=9;
```

```
int x=y=9;
```

```
int z=x+y;
```

```
float q = z/4;
```

When an expression contains more than one operator, the precedence of the operators controls the order in which the individual operators are evaluated

An example of such an expression would be:

```
float z = 2+3*4;
```

The above expression will produce the result of 14, given that the multiplication has higher precedence than the addition.

The following table lists the most common operators in order of precedence from highest to lowest

Category	Operators
Primary	x.y f(x) a[x] x++ x-- new typeof checked unchecked
Unary	+ - ! ~ ++x --x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational and type testing	< > <= >= is as
Equality	== !=
Conditional AND	&&
Conditional OR	
Conditional	?:
Assignment	= *= /= %= += -= <<= >>= &= ^= =

5.5 Creating User-Defined Data Types

5.5.1 Enumeration Type

Enumeration type is a value type in the .NET system, which enables you to define constants with integral value type, and access them using their names.

An example of such use is the color palettes in HTML, where you have to remember the Hexadecimal codes for colors, such as Red=FF0000.

Another example is weekdays, where you have Monday-Sunday, which could be represented using numbers 0-6.

5.5.1.1 Defining an Enumeration Type

To declare an enumeration, use the enum keyword followed by a variable name and initial values. For example, the following enumeration defines seven integer constants. The integer constants are called enumerator values.

```
enum WeekDays { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}
```

By default, enumerator values start from 0. In the preceding example, Monday has a value of 0, Tuesday has a value of 1, and so on.

You can also initialize an enumeration by specifying integer literals.

```
enum color {Red = 0xFF0000, Blue=0x00FF00, Green=0x0000FF}
```

In the preceding example, we forced Red represent its hex value of 0xFF0000, Blue to have 0x00FF00 and Green was assigned 0x0000FF. Essentially what we have done in the above scenarios is we have created new value types with the names Weekdays and Color, which become part of our system, just like the other basic data types defined by the CTS. Hence, we can use the in exactly the same manner.

5.5.1.2 Using an Enumeration Type

You can declare a variable myWeekDay of WeekDays type by using the following syntax:

```
WeekDays myWeekDay; // Declare the variable  
myWeekDay = WeekDays.Friday; // Set value
```

You can declare a variable myColor of Color type by using the following syntax:

```
Color myColor; // Declare the variable  
myColor = Color.Blue; // Set value
```

5.5.1.3 Displaying an Enumeration Value

Displaying an enumeration value is fairly simple. The Console.WriteLine() method automatically displays the value in its readable format. So the following statement would produce Friday:

```
Console.WriteLine("{0}", myWeekDay);
```

And the following statement would produce Blue:

```
Console.WriteLine("{0}", MyColor);
```

We could also write the above two in one statement, as follows:

```
Console.WriteLine("My color is: {0} and My weekday is {1}", MyColor, myWeekDay);
```

The result of the above would be:

My color is: Blue and My weekday is Friday

The following is a complete working class to demonstrate the different uses of enum.

```
using System;

namespace mySecondApp
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        enum color {Red = 0xFF0000, Blue=0x00FF00, Green=0x0000FF}
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            color c = color.Blue;
            Console.WriteLine(c.ToString("g"));

            Console.WriteLine("{0:g}, \t{1:d}, \t{2:x}, \t{3}", c, c, c, c.GetType().BaseType);
            Console.WriteLine(c.ToString("d"));
            Console.WriteLine(c.ToString("x"));
            Console.WriteLine(c.GetType().BaseType);
            Console.ReadLine();
        }
    }
}
```

5.5.2 Structure Type

5.5.2.1 Defining a Structure Type

You can use a structure to group together several arbitrary types, as shown in the following example:

```
public struct Pilot
{
    string firstName;
    string lastName;
    string airline;
    int age;
    float salary;
}
```

This code defines a new type called **Pilot** that consists of five elements: first name, last name, airline, age and salary.

5.5.2.2 Using a Structure Type

To access elements inside the **struct**, use the following syntax:

```
Pilot ourPilot; // Declare variable
ourPilot.firstName = "Adam"; // Set value
ourPilot.lastName = "Salah";
ourPilot.age = 39;
ourPilot.salary = 200;
```

You can use structures to create objects that behave like built-in value types.

Because structs are stored inline and are not heap allocated, they are destroyed as soon as not in use, unlike classes, which have to be garbage collected. Hence the term *struct* is a lightweight class.

In the .NET Framework, simple data types such as *int*, *float*, and *double* are all built-in structures.

5.6 Converting Data Types

In C#, there are two types of conversion, implicit and explicit:

5.6.1 Implicit data type conversion

You can convert data from one numeric type to another implicitly when you have an upwardly conversion. Which means if you are converting a smaller type such as byte (8 bits) to a larger type such int (32 bits). This can occur in many situations, such as assignment statements, arithmetic expressions and method invocations. For example:

```
byte b=129;
```

```
int i = 2001;
```

```
i = i + b; // b will get converted implicitly to int.
```

The compiler handles all such implicit conversions when they can be performed safely. There is a set of predefined implicit conversions between the various numeric types, here a subset of the common one:

byte/sbyte → *short, int, long, float, double or decimal*

int → *long, float, double, or decimal*

long → *float, double or decimal*

float → *double*

5.6.2 Explicit data type conversion

You will see examples of how to perform both implicit and explicit data conversion in this section.

You can convert variable types explicitly by using a cast expression, either upon assignment or during a method invocation. This simply means preceding the value being converted (casted) by the destination type between parentheses. For example, the following expression converts the floating-point value 139.64 to an integer. Notice the F suffix, with the number? It's required, because in C# all literal float numbers are internally represented as double, hence the F casting! which is an example of explicit conversion:

```
float floatNum = 1309.64F;
```

```
int i = (int) floatNum;
```

A word of caution here that the value being converted should be of a magnitude that is suitable for the variable type it is being converted to, otherwise you risk a loss of value.

So the above conversion would produce an erroneous result if the integer were replaced with a byte as in:

```
float floatNum = 1309.64F;
```

```
sbyte b = (sbyte) floatNum;
```

The following example shows how to convert the variable longValue from a long data type to an int data type by using a cast expression:

```
using System;
class ConversionTest
{
    static void Main( )
    {
        float floatNum = 1309.64F;
        int i = (int) floatNum;
        sbyte b = (sbyte) floatNum;

        Console.WriteLine("(int) {0} = {1}", floatNum, i);
        Console.WriteLine("(sbyte) {0} = {1}", floatNum, b);
    }
}
```

The output of this example is as follows; notice the result of the (sbyte) casting?

(int) 1309.64 = 1309

(sbyte) 1309.64 = 29

This is one of the hard logical errors to trace, so avoid using this type of conversion, unless you absolutely can't.

A good practice in C# to avoid such a situation from passing uncaught is to use the **checked** statement to raise an exception when a conversion fails, as follows:

```
using System;
class ConversionTest
{
    static void Main( )
    {
        checked
        {
            float f = 1309.64F;
            int i = (int) f;
            sbyte b = (sbyte) f;

            Console.WriteLine("(int) {0} = {1}", f, i);
            Console.WriteLine("(sbyte) {0} = {1}", f, b);
        }
    }
}
```

This version will invoke the debugger, rather than quietly return an erroneous value as in the previous version.

All of the above discussion was about converting data among the various Value types. But since all CTS types are derived from the Reference type System.Object, then conversion between Value types and Reference types is possible. Conversion from Value type to a Reference type (an object) is known as boxing and the inverse, which is converting from a Reference type (an object) to a Value type is called unboxing.

5.6.3 Boxing

Boxing is a form of implicit casting, and it involves converting a Value type to a Reference type (to an object). It is called boxing, because an envelope or a box is created to encapsulate the Value type, and present it as a Reference type (an object).

Let us look at an example, and then explain what actually takes place:

```
int iVar = 39;           // Value type.
object oBox = iVar;      // iVar is boxed into oBox.
```

What happens in the above case can be summarized as follows:

- Memory is allocated on the heap, because iVar will be presented as an object.
- The iVar's value is then copied to the newly allocated heap memory.
- The address of the newly allocated object is placed on the stack pointing to a reference type.

5.6.4 Un-Boxing

Un-Boxing is a form of explicit casting, and it involves converting a Reference type (to an object) to a Value type. It is called unboxing, because an envelope or a box is created to encapsulate the Value type, and present it as a Reference type (an object).

Let us look at an example, and then explain what actually takes place:

```
int iVar = 39;           // Value type.
object oBox = iVar;      // iVar is boxed into oBox.
int uVar = (int) oBox;    // Unboxed (Casted) back to int.
```

Notice the explicit casting, using the (int), which is necessary, because theoretically speaking, you can cast the object into any Value type, including **int**.

The steps taken by the system when unboxing are:

- The system checks for two things: existence of the object, and validity of the conversion that is about to take place, by checking the value type.
- Once validation is a pass, a pointer to the value within the object is returned.

The following example demonstrates both concepts of boxing and unboxing:

```
using System;
class ConversionTest
{
    static void Main( )
    {
        int iVar = 42;    // Simple unboxed value type
        object o = iVar;  // Will cause a boxing operation of iVar
        // We display the object's type and unboxed value
        // Will output System.Int32, 42
        Console.WriteLine(o.GetType() + ", " + (int)o);
    }
}
```

The output of the above code will be:
`System.Int32, 42`

Although we boxed the value into an object, but the base type is `System.Int32` which is the internal representation of `int`. Remember that at the very basis of every type, including all value types, is the type `Object`.

5.7 Arrays

Another member of the Reference types is Arrays, which are also known as collection classes in the .NET Framework. The .NET Framework provides a suite of collection classes, including Array, ArrayList.

C# provides built-in support only for the Array type, which can be single dimensional, multidimensional, and jagged arrays.

Arrays can hold a list of values of the same type, allowing access to the individual elements by means of indexes that start from 0.

Array bounds in C# are checked, and this feature cannot be disabled.

5.7.1 Declaring Arrays

Declaring Arrays is similar to declaring a simple Value type variable. You simply state the type followed by the variable name and a semicolon. Except for an Array you add square brackets between the type and the name as follows:

```
int [] myIntArray;           // Single dimensional array (Rank = 1)
int [ , ] my2DIntArray;      // Two dimensional array (Rank = 2)
int [ , , ] my3DIntArray;    // Three dimensional array (Rank = 3)
```

At first glance there are many similarities between the syntax above and that of C++. But beware of the following:

You can't put the square brackets in front of the variable name, nor can you specify the size of the array inside the brackets. You can however, specify the rank of the array, also known as the number of dimensions. You can define the rank (dimension) by way of commas inside the brackets. Each comma will increment the rank by 1. So no comma means rank=1, one comma means 2 dimensional array, 2 commas means 3 dimensions and so on.

5.7.2 Creating Arrays

Before you can use an array, you must create an instance of the Array variable. The declaration of the array creates a reference type, which must be instantiated to be useable.

You can accomplish in two ways:

1. Using the key word new as follows:

```
int [] myIntArray = new int[3];
int [ , ] my2DIntArray = new int[3, 2];
```

or

```
int x=3, y=2;
int [] myIntArray = new int[x];
int [ , ] my2DIntArray = new int[x, y];
```

Using the above form you can also defer the assignment of values to x and y, and then read them from some source, such as user input or a file, allowing you to define variable sized arrays, at runtime.

2. Using values assignment (array initialization)

```
//initializes the array to a size of 6
int [] intArray={13, 09, 12, 09, 2001, 1964};

//initializes the array to a size of 2 * 3
int [ , ] my2DIntArray = {{00,01, 02}, {10, 11, 12}};
```

5.7.3 Initializing Arrays

All arrays are automatically initialized, upon creation. You can also initialize arrays explicitly, either at creation time, or later in the code.

At the time of array instance creation, you can either one of the format we demonstrated in the previous section, and as shown below:

1. Explicitly define size and initialize values:

```
int [] myIntArray = new int[3]{1, 2, 3};
int [ , ] my2DIntArray = new int[3, 2]{{00,01,02},{10,11, 12}};
```

2. Explicitly initialize values and implicitly define size:

```
//initializes the array to a size of 6
int [] intArray={13, 09, 12, 09, 2001, 1964};

//initializes the array to a size of 2 * 3
int [ , ] my2DIntArray = {{00,01, 02}, {10, 11, 12}};
```

5.7.4 Accessing Arrays

You can access array elements once declared, created and initialized, or assigned, using their indexes, to read and/or write their values.

Example 1, one-dimensional array:

```
//initializes the array to a size of 6
int [] intArray={13, 09, 12, 09, 2001, 1964};
intArray[1] = 27;
// Display the second element in the array (0th+1)st
Console.WriteLine(intArray[1]);
// Display the fifth element in the array (0th+4)st
Console.WriteLine(intArray[4]);
```

The output of the above code is:

```
27
2001
```

Example 2, two-dimensional array:

```
//initializes the array to a size of 2 * 3
int [ , ] my2DIntArray = {{00,01, 02}, {10, 11, 12}};
my2DIntArray[0, 1] = 27;//change current value (Row=1st, Col=2nd)
my2DIntArray[1, 2] = 57;//change current value (Row=2nd, Col=3rd)
//Display changed values
Console.WriteLine(my2DIntArray[0, 1]);
Console.WriteLine(my2DIntArray[1, 2]);
```

The output of the above code is:

```
27
57
```

5.7.5 Array properties

Arrays as objects have properties that you can access, get and set. We discussed array's Rank, which is the number of dimensions of the array.

5.7.5.1 Rank

The array object defines a **Rank** property that returns the number of dimensions the array has.

```
//initializes the array to a size of 2 * 3
int [ , ] my2DIntArray = {{00,01, 02}, {10, 11, 12}};
//Display array Rank
Console.WriteLine(my2DIntArray.Rank);
```

The preceding example displays the Rank of the array, which is 2.

5.7.5.2 Length

The array object also defines a **Length** property that returns the total number of elements the array can hold, this is also known the size of the array. The length of the array can be calculated as the multiplication of all dimensions: # of Rows * # of Columns. Which in this case is $2 * 3 = 6$

```
//initializes the array to a size of 2 * 3
int [ , ] my2DIntArray = {{00,01, 02}, {10, 11, 12}};
//Display array Rank
Console.WriteLine(my2DIntArray.Length);
```

The preceding example displays the Length of the array, which is 6.

5.7.5.3 Array methods

Arrays in C# have several methods associated with them, most important of which are GetLength, IndexOf, Clear, Sort and Reverse.

The last three methods are accessed through the Array object, rather than its instance that you create, which is instead passed as a parameter.

Let us learn about each of method:

5.7.5.4 GetLength

Returns the length of a single dimension of a given array. You can also use this to calculate the Rank of the array.

This method operates on a single dimension of arrays of any Rank

An example of this is:

```
//initializes the array to a size of 2 * 3
int [ , ] my2DIntArray = {{00,01, 02}, {10, 11, 12}};
//Display array Rank
int iRows = my2DIntArray.GetLength(0);
int iCols = my2DIntArray.GetLength(1);
int iRank = iRows * iCols;
Console.WriteLine("Rows = {0} , Cols = {1}",iRows, iCols);
Console.WriteLine("Rank = {0}", iRank);
```

The output of the above program would be:

Rows = 2 , Cols = 3

Rank = 6

5.7.5.5 IndexOf

This is an effective way to search for an element within an array. You pass the array name and the item you are searching for, to get back the index (position) of the first occurrence of the element, a -1 will be returned if the item is not found in the array.

This method operates only on one-dimensional arrays.

```
int [] intArray={13, 09, 12, 09, 2001, 1964};  
// IndexOf method example  
int iPos = System.Array.IndexOf(intArray, 2001);  
Console.WriteLine("Position = {0}", iPos);
```

The preceding example displays **4**, which is the index of **2001**, starting at 0. Pay attention to the format of accessing the method, using the System.Array object, instead of intArray, which is our instance, as was the case of my2DIntArray.GetLength(1); in the example before this one.

5.7.5.6 Clear

The Clear method, sets the values of the array elements to 0 of numeric, or null otherwise. It accepts three parameters, which are array name, starting index, and number of elements to clear.

This method operates on arrays of any Rank

This example set the values in entire array to zeros. It is also a good way to initialize arrays.

```
int [ , ] my2DIntArray = {{00,01, 02}, {10, 11, 12}};  
System.Array.Clear(my2DIntArray, 0, 6);
```

The following example sets the contents of the first 2 positions to null:

```
string [] strArray={"Taxi", "Accelerate", "TakeOff"};  
System.Array.Clear(strArray, 0, 2);  
Console.WriteLine(strArray[0]+" , "+strArray[1]+" , "+strArray[2]);
```

5.7.5.7 Sort

Simply sorts the elements of an array in ascending order using the QuickSort algorithm. This method operates only on one-dimensional arrays.

```
//initializes the array to a size of 6  
string [] strArray={"Taxi", "Accelerate", "TakeOff"};  
Console.WriteLine(strArray[0]+" , "+ strArray[1]+" , "+ strArray[2]);  
  
System.Array.Sort(strArray);  
Console.WriteLine(strArray[0]+" , "+ strArray[1]+" , "+ strArray[2]);
```

The output of this code is:

```
Taxi , Accelerate , TakeOff  
Accelerate , TakeOff , Taxi
```

5.7.5.8 Reverse

Reverses the order of the items in a single-dimensional array.

```
//initializes the array to a size of 6
string [] strArray={"Taxi", "Accelerate", "TakeOff"};
Console.WriteLine(strArray[0]+" , "+ strArray[1]+" , "+ strArray[2]);

System.Array.Reverse(strArray);
Console.WriteLine(strArray[0]+" , "+ strArray[1]+" , "+ strArray[2]);
```

The output of the above code is:

```
Taxi , Accelerate , TakeOff
TakeOff , Accelerate , Taxi
```

6 Statements and constructs

C# inherits most of its statement syntax from C and C++, with some significant additions and changes. Statements in C# can be broadly categorized into five major groups:

6.2 Variable declarations and Expression statements

Variable declaration statements enable the C# programmer to do just that. The Expression statements are simply any statements that evaluate an expression, resulting in a value of a CTS type. It usually involves operators and operands.

Variable declaration and Expression statements were discussed extensively in the data and variables section, however, it worth mentioning here that, the keywords used for variables declaration are merely aliases for the internal CTS-provided .NET classes.

The table below lists the various data types and their aliases

6.3 Unconditional Branching (Jump) statements

The *goto*, *return*, *break*, and *continue* statements are used to unconditionally transfer control to another statement.

6.3.1 The goto Statement

goto is one of the oldest mechanisms to jump to a labeled line of code. The label could be any letter(s) followed by a colon (:).

Example:

```
using System;
namespace cSharpStatements
{
    class branchingClass
    {
        static void Main()
        {
            goto Label1;
            Label2: Console.WriteLine("Universe");
            return;
            Label1: Console.Write("Hello, ");
            goto Label2;
        } // end of Main()
    } // End of Class
} // End of namespace
```

6.3.2 The break and continue Statements

A *break* statement exits the enclosing *switch*, *while*, *do*, *for*, or *foreach* statement; a *continue* starts a new iteration of the enclosing *while*, *do*, *for*, or *foreach* statement. Examples of *break* statement will be given in the course of discussing the *switch* and *foreach* statements.

6.3.3 The return Statements

A return statement returns control to the caller of a member. If a member's return type is declared as void (as in ***static void Main()***) A return statement with no expression is used. When a member is declared with a return type, other than void, then the return statement with an expression that evaluates to the declared return type is used.

Examples of valid return statements are:

```
return;  
return 0;  
return x+y;  
return "Hello there..";
```

6.4 Conditional branching (Decision-making or Selection) statements

A conditional branch is created by a selection statement, which can be an if-else, or switch statement, which make choices based on the value of a condition expression. A conditional branch occurs when the condition expression evaluates to true.

6.4.1 If-else statement

The ***if*** statement is the primary decision-making statement. It can be coupled with an optional ***else*** clause, as shown:

```
if ( Boolean-expression )  
statement or statement block  
else  
statement or statement block
```

The ***if*** statement evaluates a Boolean expression to make a decision. If the Boolean expression evaluates to true, the control is transferred to the first embedded statement. If the Boolean expression evaluates to false, and there is an ***else*** clause, the control is transferred to the second embedded statement. Otherwise the control will flow through to the next line of code. The Boolean expression can be a simple comparison of a complex expression formed with logical operators such as logical OR (||) and logical AND (&&).

Examples

Expressions can come in any valid Boolean form, such as:

```
if (salary != 0)  
if (salary != 0 && age < 61)  
if (salary != 0 || contracted = true)
```

You can use a simple if statement such as the following:

```
if (salary != 0)  
Console.WriteLine("employed");
```

Although braces are not required in the statements, unless it is a block of more than one line, it is highly recommended using them because they make

your code easier to read and maintain. We can rewrite the previous example with braces as follows:

```
if (salary != 0)
{
    Console.WriteLine("employed");
}
```

You can also use an if statement block, with more than one statement, as in the following:

```
if (age >= 61)
{
    tax = 0;
    senior = true;
}
```

Unlike VB, C# does not support a cascading else-if statement. Although you can nest if-else constructs to any level, the order of execution is different than that of VB-like if-else-if construct.

The statements controlled by a cascading *if* statement are mutually exclusive, so that only one statement from the set of *else if* constructs is executed.

Nesting one *if* statement within another *if* statement can also create a potential ambiguity called a *dangling else*, and could be difficult to debug.

6.4.2 Switch statement

The *switch* statement provides an elegant mechanism for handling complex conditions that would otherwise require nested *if* statements. It consists of a Switch followed by the left hand side operand of the Boolean expression, and multiple *case* blocks, each of which specifies a single constant and an associated *case* label, representing the right hand side operand. Each case statement block must be terminated with a *break*; statement, to prevent falls through to the next line.

An optional Default: statement block can be written at the end of the switch statement, to execute when all cases fail.

You can only compare for equality in a switch statement. You are also restricted to the following types: any *int* type, a *char*, an *enum*, or a *string*

Examples

The following example illustrates all the various aspects of a typical switch statement:

```
string birthDay = Console.ReadLine();
switch (birthDay)
{
    case "Sat": { Console.WriteLine("Born on a Saturday!"); }
    break;      // break has to be outside the statement block

    case "Sun": {
        Console.WriteLine("Born on a Sunday!");
    }
    break;      // break has to be outside the statement block
```

```

        default :
        {
            Console.WriteLine("Born on a weekday!");
        }
        break;          // break has to be outside the statement block
    } //end switch

```

6.5 Loops (Iteration) statements

C# provides four different iteration constructs. These are *for*, *do..while* which are from C and C++, in addition to *while* and *foreach* constructs which are available to the VB programmers.

The *for*, *do*, *while*, and *foreach* statements execute repeatedly as long as a specific condition is true. They are also known as looping statements. Each of these statements is appropriate for a certain style of iteration.

6.5.1 do statement

A *do* statement conditionally executes a statement or a block of statements one or more times. Hence you would use the *do* statement when you know that the condition will hold true, at least once, or when you'd like the statement(s) executed at least one time, before the Boolean expression is checked.

The following example implements an array search, similar to the `IndexOf` method provided by the system.

We first declare and initialize an array of integers, an integer to search for `ivalue`, and a loop counter `i`, which we initialize to -1. The *do* loop starts by incrementing the index variable `i`, bringing its value to 0, then it performs a test by comparing the array element of index `i`, `arr[i]` against the `ivalue`, and at the same time make sure that we didn't reach the end of the array, by checking the `Length` property of the array. Note the `<` rather than `<=`, because arrays in C# start indexing from 0.

Finally we check the result, and display an appropriate value. Either the index of the value we found or a message indicating that none was found.

```

using System;
namespace cSharpStatements
{
    class WhileDemoClass
    {
        static void Main()
        {
            // Example of a do statement
            int ivalue = 13;
            int[] arr = {10, 21, 11, 13, 15, 17};
            int i = -1;
            do
            {
                i++;
            }

```

```

        while ((i < arr.Length) && (arr[i] != ivalue));

        // Display the index of the value or an error message
        if (i < arr.Length)
        {
            Console.WriteLine(i);
        }
        else
        {
            Console.WriteLine("The value was not found");
        }
    } // end of Main()
} // End of Class
} // End of namespace

```

6.5.2 while statement

A **while** statement conditionally executes a statement or a block of statements zero or more times – as long as the result of a Boolean expression is true. Let us rewrite the example we used for the **do** statement with a **while** loop instead of **do**.

```

using System;
namespace cSharpStatements
{
    class WhileDemoClass
    {
        static void Main()
        {
            int ivalue = 13;
            int[] arr = {10, 21, 11, 13, 15, 17};
            int i = 0;
            while ((i < arr.Length) && (arr[i] != ivalue))
            {
                i++;
            }
            // Display the index of the value or an error message
            if (i < arr.Length)
            {
                Console.WriteLine(i);
            }
            else
            {
                Console.WriteLine("The value was not found");
            }
        } // end of Main()
    } // End of Class
} // End of namespace

```

6.5.3 for statement

The **for** statement has the following format:

for (initialization; Boolean-expression; increment/decrement) Statement;

Where **initialization** is a list of 0 or more variable initialization(s). The **Boolean-expression** is the stopping rule, which for as long as it evaluates to true, the **for**

loop will continue. Finally the **increment/decrement** is where the variables initialized in the first part are incremented or decremented, as needed. Let us rewrite the previous example using a **for** loop. Notice how the initialization, checking and increment of *i* become part of the **for** construct?

```
using System;
namespace cSharpStatements
{
    class WhileDemoClass
    {
        static void Main()
        {
            // Example of a for statement
            int i;
            int ivalue = 13;
            int[] arr = {10, 21, 11, 13, 15, 17};
            for (int i=0; (i < arr.Length) && (arr[i]!=ivalue);
i++)
            {
            }
            // Display the index of the value or an error message
            if (i < arr.Length)
            {
                Console.WriteLine(i);
            }
            else
            {
                Console.WriteLine("The value was not found");
            }
        } // end of Main()
    } // End of Class
} // End of namespace
```

6.5.4 foreach statement

Intended for use with collections such as arrays, the **foreach** statement repeats a statement or block of statements for each element in the collection.

The general format of **foreach** statement is:

foreach (type variable in collection) Statement;

Where **type**, is the same as the **collection** element type. The **variable** will represent each value in the **collection** until the last element in the collection has been reached.

If we rewrite the example we used in the three preceding sections, then we will get the following:

```
using System;
namespace cSharpStatements
{
    class WhileDemoClass
    {
        static void Main()
```

```

{
    // Example of a foreach statement
    int i = 0;
    int iValue = 13;
    int[] arr = {10, 21, 11, 13, 15, 17};
    foreach (int iTmp in arr) // assign arr[i] to iTmp
    {
        if (iValue == iTmp)
        {
            break;        // exit the loop
        }
        else
        {
            i++;          // increment index
        }
    }
    // Display the index of the value or an error message
    if (i < arr.Length)
    {
        Console.WriteLine(i);
    }
    else
    {
        Console.WriteLine("The value was not found");
    }
} // end of Main()
} // End of Class
} // End of namespace

```

You may have noticed the use of **break** to exit the loop once we get a match.

6.6 Error handling statements

Comprising of try-catch, checked and unchecked statements, Error handling statements help C# programs handle the exceptions in a clean manner.

6.6.1 checked and unchecked statements

The **checked** statement is used to raise an exception at runtime, in response to type safety issues. C# normally recovers from such problems gracefully, and doesn't throw an exception. In other words, the default state is the **unchecked** state. You can force the checked state as discussed in the section about data conversion, and listed again, below for your convenience.

You can use the **checked** statement to raise an exception when a conversion fails, as follows:

```

using System;
namespace cSharpStatements
{
    class ConversionTest
    {
        static void Main( )
        {
            checked

```

```

        {

            float f = 1309.64F;
            int i = (int) f;
            sbyte b = (sbyte) f;

            Console.WriteLine("(int) {0} = {1}", f, i);
            Console.WriteLine("(sbyte) {0} = {1}", f, b);

        }
    } // end of Main()
} // End of Class
} // End of namespace

```

This version will invoke the debugger, rather than quietly return an erroneous value as in the previous version.

6.6.2 try-catch statements

Rather than invoking a debugger, or displaying some JIT error message that the end user is less like to understand, you can catch the exception raised in the above example, and display a more appropriate error message, whatever it might be. C# provides a mechanism to do just that.

Simply enclose your code in a **try** statement block and enclose your exception handling logic in a **catch** block as follows:

```

try
{
    // your entire functional code
}
catch (Exception e)
{
    // process errors using the (stacktrace, message and source) properties of e
    // you can also use the Console.WriteLine(e), which will display all properties of e
    // return goes here
}

```

So if we rewrite the code from the preceding section to display an error message instead of invoking the runtime debugger, we get the following:

```

using System;
namespace cSharpStatements
{
    class ConversionTest
    {
        static void Main( )
        {
            // Example of a try-catch statement
            try
            {
                checked
                {
                    float f = 1309.64F;
                    int i = (int) f;
                    sbyte b = (sbyte) f;

```

```

        Console.WriteLine("(int) {0} = {1}", f, i);
        Console.WriteLine("(sbyte) {0} = {1}", f, b);
    }
}
catch(Exception e)
{
    Console.WriteLine("The following Exception
occured:");
    Console.WriteLine(e);
}

    } // end of Main()
} // End of Class
} // End of namespace

```

We simply place the entire checked block in the **try** block, and a not so sophisticated error reporting code in the **catch** block.

7 Classes

Simply put, classes enable you to define new Reference types. In object oriented programming terms, a class is the blueprint of an object. An object is the instance of a class, created using the **new** keyword.

Classes can contain member classes, structs, enums and basic value types, such as `int` and `float`. Basic value type members are known as **fields** of a class. Classes can also define functions, known as **Methods**.

A class that contains the **Main()** method is known as an application class, and can not be instantiated.

7.2 Defining classes

Defining classes abides by a general format which is:

```
class ClassName
{
    // Members
}
```

Classes can contain member classes, structs, enums and basic value types, such as `int` and `float`. Basic value type members are known as **fields** of a class. Classes can also define functions, known as **Methods**.

Let's look at an example of a class declaration:

```
class Pilot
{
    private string pilotId;
}
```

As you can see, we have a class named *Pilot* that contains a single member named *pilotId*. Notice `private`, which precedes our member. This is called an access modifier, which we will discuss shortly.

7.2.1 Class Members

Classes consist of members. Everything you define within a class is considered to be a member of that class. You already know that all the types available to C# programmers originate from the common type system (CTS). Here's a list of the various types that you can define as members of a C# class:

- **Fields** A field is a member variable used to hold a value. You can apply several access modifiers to a field, depending on how you want the field used. These modifiers include *static*, *readonly*, and *const*. We will discuss these modifiers and how to use them shortly.
- **Methods** A method is the actual code that implements the behavior of the class, by performing specific operations on the object's data. We will discuss methods in the next chapter, and in this chapter, we'll focus on class data.

- **Properties** Properties in C# are a combination of fields and methods. For any client, accessing the class members, properties look like fields in the way they are accessed. That is by just specifying their name in an expression. Internally however, their implementation involves using methods to **set** and **get** their values. And because of this fact, Properties are also known as smart fields.
- **Constants** Just as in C++, a constant is a field that whose value can't be changed.
- **Indexers** Indexers are similar to properties, except they help build indexed containers such as arrays, rather simple type fields. C#'s own arrays and all other collections that allow indexed "square-brackets" access use indexers internally. Just as a property is a *smart field*, an indexer is sometimes referred to as a *smart array*
- **Delegates** A delegate declares a reference type (similar to class) that has a method with a particular set of arguments and a return type. A delegate does not know or care about the class of the object that it references. It can access any member function as long as the method's signature matches that of the delegate's, which makes delegates ideal for anonymous invocation. The method's signature is just the return type and the parameter list.
- **Events** An *event* in C# is a way for a class to notify its consumers (clients) of something that happens to an object. A typical use for events is in Microsoft Windows programming, where an event is fired when a user moves or clicks the mouse, or resizes a window. An invocation of a method (function) occurs in response to the event. The methods (functions) are given to the class by its clients using delegates.
Simply put, events enable class users to define delegates to methods which the class can use when something happens. You can think of this situation as a call that one makes to the class, and leaving a call back number, where one could be reached in case of an event occurrence.
- **Operators** C# gives you the ability, to redefine the functionality or behaviour of an operator, by means of operator overloading.

7.2.2 Access Modifiers

The different types that can be defined as members of a C# class, defined in the preceding section, can be declared in a class to provide varying levels of visibility (accessibility) to the clients of the class.

Clients of a class are simply code outside class, which attempt access the class members.

The member visibility is specified by means of modifiers, called access modifiers, listed in the table below:

Class and Class Member Visibility Access Modifiers

Access Modifier	Visibility
<i>Public</i>	Accessible from anywhere
<i>protected</i>	Accessible from this class or any class derived from this class
<i>Internal</i>	Accessible within current program source unit (assembly) only
<i>protected internal</i>	Accessible within current program (assembly) or any class derived from this class
<i>Private (default)</i>	Accessible only within current class

The default access modifier of private, which will be assigned to all members not proceeded with an access modifier.

For example, in the following code, the members `iPublicVar` and `sPublicVar` are defined with public visibility, and the members `iProtectedVar` is defined as a protected member. The member `iPrivateVar` is declared as private because no access modifier was specified

```
class MyAccessModifiers
{
    public int iPublicVar;
    public string sPublicVar;
    protected int iProtectedVar;
    int iPrivateVar;
}
```

Now that you have all the ingredients for a class definition let us get down to it and have some fun creating different classes with various types of members.

7.3 Using classes

Creating an instance of the class (instantiation or creating an object), using the **new** keyword, allows the use of a Class. Once the class has been instantiated, its members are accessed using the dot operator.

```
class myClass1
{
    public type1 member1;    // example is int x=1209;
    type2 member2;          // example string myName="Adam"
}
class myClass2
{
    type MthodName(param List)
    {
        myClass1 myInstance = new myClass1();    // Create an object
    }
}
```

```

        type1 myInstanceVar = myInstance.member1; // Create instance variable
        myInstance.member1 = 2001;
    }
}

```

Example:

```

using System;

namespace cSharpClasses
{
    class FirstClass
    {
        public int member1; //public field initialized to 0 by system
        int member2=1964; //local member, implicitly declared private
    } // end of FirstClass

    class SecondClass
    {
        static void Main(string[] args)
        {
            //Create a FirstClass type object (Instantiate the class)
            FirstClass myInstance = new FirstClass();

            // Create instance variable
            int myInstanceVar = myInstance.member1;

            Console.WriteLine(myInstanceVar); // prints 0

            //Replace the value of the public field member1
            myInstance.member1 = 2001;

            //copy the value of field into instance variable
            myInstanceVar = myInstance.member1;

            Console.WriteLine(myInstanceVar); // prints 2001
        } // end of Main()
    } // end of SecondClass
} // end of namespace

```

8 Methods

Methods are defined as members of a class, and can be thought of as the actions performed by an object.

In an analogy to the English language, one could think of methods as the verbs. For example an airplane could perform actions such as Start engine, Taxi, Take off, Land and Crash.

In programming language terms, the Method is where all functional code resides. The general format for defining a method is:

```

Accessmodifier type MethodName([modifier param1, modifier param2 ...])
{
    // some code
    [return [value];]
}

```

```
}
```

In C# there is one particular method that has to be present, for our program to start. This method is the `Main()` method. So we could say, there `Main()` and everything else.

8.2 The Main method

The `Main()` method is a special method that is required by C#, to act as an entry point for the application. And it conforms to a specific format and follow certain rules.

It has to be written with a capitalized M and can take one of four different formats:

- a. `static void Main()`
- b. `static int Main()`
- c. `static void Main(string[] args)`
- d. `static int Main(string[] args)`

The word “**Static**” before “**Main()**” simply means that the function can be invokes by calling its name, without the need for creating an instance of the containing class.

void means that `Main()` will not return any values. On the other hand, where we have `int`, it means **`Main()`** will return an integer value, when done.

The (**`string[] args`**) is the optional argument list, which can be received by the main method, if the user passed any values on the command line.

Say for example, assuming our code above was compiled into `test.exe` typing:

>**`test.exe how are you?`**

At the command prompt would pass the three strings “how” “are” “you” as arguments to the `Main()` method’s `args` array.

Note: Although C# allows you to create any number of `Main()` methods, in different classes, only one of them can represent the entry point of the application. Which means, in case of multiple `Main()` methods are present in the code, you have to tell the C# compiler, explicitly, which one of them is the entry point. So unless you have a good reason, then there should only be one `Main()` method in the entire application.

8.3 Generic methods and return types

All the theory aside, Methods are just functions, which comprise of a function name, preceded by optional return type and access modifier, followed by a pair of parenthesis which may or may not contain parameters.

Example:

```
using System;

namespace cSharpMethods
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class myFirstClass
    {
        private int iValue = 1209;
        public int iField = 2001;

        public void displayValues()
        {
            //Use a Fully Qualified Name (FQN) to access methods
            System.Console.WriteLine(iField);
            System.Console.WriteLine(iValue);
        }
    }

    class mySecondClass
    {
        static void Main()
        {
            // Create an instance of class myFirstClass
            // named myInst = object
            myFirstClass myInst = new myFirstClass();

            // display the original values
            myInst.displayValues ();           // invoke a method

            // display updated values
            myInst.iField = 2003;              // modify a field value
            myInst.displayValues ();           // invoke a method
        }
    }
}
```

In the preceding example, we see a simple method, displayValues(), that accepts no parameters and returns no values.

8.3.1 Passing parameters and returning values

If we were to rewrite the above example to demonstrate returning values and accepting parameters, we end up with the following:

```

using System;

namespace cSharpMethods
{
    /// <summary>
    /// Summary description for myFirstClass.
    /// </summary>
    class myFirstClass
    {
        private int iValue = 1209;
        public int iField = 2001;

        public void displayValues()
        {
            //Use a Fully Qualified Name (FQN) to access methods
            System.Console.WriteLine(iField);
            System.Console.WriteLine(iValue);

            displayAddedValues();
        }

        private void displayAddedValues()
        {
            //Use a Fully Qualified Name (FQN) to access methods
            int addedValues = AddValues(iValue, iField);
            System.Console.WriteLine("The result of addition = {0}",
addedValues);
        }

        private int AddValues(int x, int y)
        {
            //Use a Fully Qualified Name (FQN) to access methods
            return (x + y);
        }
    }
}

class mySecondClass
{
    static void Main()
    {
        // Create an instance of class myFirstClass
        // named myInst = object
        myFirstClass myInst = new myFirstClass();

        // display the original values
        myInst.displayValues ();           // invoke a method

        // display updated values
        myInst.iField = 2003;              // modify a field value
        myInst.displayValues ();           // invoke a method

        //myInst.displayAddedValues ();    // invoke a method
    }
}

```

```
}
```

In the above example, you will see two new methods, `private void displayAddedValues()` And `private int AddValues(int x, int y)`, both with a `private` access modifier, deeming them inaccessible from outside the class, or hiding them, from the outside world, also said to be **encapsulating** them! The second method has two more features: A return type `int` and two parameters of type `int` between parentheses. This means that `AddValues()` expects two integer values as parameters when invoked, and will return an `int` value upon exit. An important fact to mention here, is that only class members have access to private fields, so it would not be possible to carry out operations such as the one in `AddValues()` outside the class, because `iValue` is a private field.

8.3.2 Returning multiple values

8.3.2.1 ref keyword

Methods can return only a single value, but there are several ways to return multiple values from a method. The obvious is to use a collection, such as an array. But arrays will only contain elements of the same type. The other way to return multiple values is to pass parameters by reference and have the invoked method modify their values, giving the caller access to the new values upon return of control.

All parameters in C# can be passed by reference, even if they are of value type, using the key word `ref` in both the declaration as well as the calling.

```
using System;
namespace cSharpMethods
{
    /// <summary>
    /// Summary description for myFirstClass.
    /// </summary>
    class myFirstClass
    {
        private int iValue = 12;
        private int iField = 13;

        public void ReturnValues(ref int i, ref int j)
        {
            //Use a Fully Qualified Name (FQN) to access methods
            i = iField;
            j = iValue;
        }
        public void AddValues(int x, int y, ref int z)
        {
            //Use a Fully Qualified Name (FQN) to access methods
            z = x + y;
        }
    }
}

class mySecondClass
{
    static void Main()
```

```

    {
        // Create an instance of class myFirstClass
        // named myInst = object
        myFirstClass myInst = new myFirstClass();

        int a=0, b=0, c=0;
        // display the original values
        myInst.ReturnValues (ref a, ref b); // invoke a method

        myInst.AddValues (a, b, ref c);      // invoke a method
        Console.WriteLine("{0} + {1} = {2}", a, b, c);
    }
}

```

The preceding code defined two methods,
`public void ReturnValues(ref int i, ref int j)`
and
`public void AddValues(int x, int y, ref int z)`

Each parameter preceded with `ref` will operate on the original variable, replacing its value with whatever the new one is, which are 12 and 13 in the first method, and 25 in the second one.

Invoking the method we have to do the following:

1. Declare variables of the same type as defined in the method header
2. Initialize the variables to some value (ref requirement).
3. Invoke the method and pass the parameters preceded with `ref`
4. Use the variables, assuming that new values has been received

8.3.2.2 out keyword

The use of `ref` makes sense if you are setting variables to some value, which would be used in the invoked method, before replacing them. In the preceding example, however, the values of the variables were immediately replaced by new ones, so initializing them in the calling method is a waste. C# provides a mechanism to create variables meant just for receiving values, and whose initial value is irrelevant. Using `out` in place of `ref` deems the variables output (place holders) only. So if we rewrite the above example to use `out` instead of `ref`, we don't have to initialize `a`, `b` and `c`.

```

using System;
namespace cSharpMethods
{
    /// <summary>
    /// Summary description for myFirstClass.
    /// </summary>
    class myFirstClass
    {
        private int iValue = 12;
        private int iField = 13;

        public void ReturnValues(out int i, out int j)
        {

```

```

        //Use a Fully Qualified Name (FQN) to access methods
        i = iField;
        j = iValue;
    }
    public void AddValues(int x, int y, out int z)
    {
        //Use a Fully Qualified Name (FQN) to access methods
        z = x + y;
    }
}

class mySecondClass
{
    static void Main()
    {
        // Create an instance of class myFirstClass
        // named myInst = object
        myFirstClass myInst = new myFirstClass();

        int a, b, c;
        // display the original values
        myInst.ReturnValues (out a, out b); // invoke a method

        myInst.AddValues (a, b, out c);      // invoke a method
        Console.WriteLine("{0} + {1} = {2}", a, b, c);
    }
}

```

8.4 Constructors

Notice how when we create an object (instance of a class), we add the () pair of parentheses at the end of the class name, as if we were calling a function?

```
myFirstClass myInst = new myFirstClass();
```

We actually are invoking a method (calling a function), every time we instantiate a class. This method is known as a constructor! A constructor is a mandatory method, which either you define, or the CLR does.

Declaring and using Constructors follows the exact syntax of a generic method (discussed in the preceding section), only the name has to be the same as the class name.

Constructors have no return type and are usually declared public.

If the constructor happened to have parameters, then the class instance could be passed those parameters upon creation.

It is a good practice to have a constructor for each class, and implement any initialization tasks that are required for the object to perform its intended job, in the constructor method.

```

using System;

namespace cSharpClasses
{

```

```

public class ClassWithConstructor
{
    private int iValue = 1209;
    private int iField = 2001;

    //Constructor method
    public ClassWithConstructor(int i, int j)
    {
        iValue = i;
        iField = j;
    }

    public void DisplayValues()
    {
        Console.WriteLine(iValue);
        Console.WriteLine(iField);
    }
}

public class MainClass
{
    static void Main(string[] args)
    {
        // Create an object type variable
        ClassWithConstructor myInstance;

        // Create an object      (instance of class)
        myInstance = new ClassWithConstructor(9, 13);

        // Displays 9 and 13
        myInstance.DisplayValues();
    }
}

```

8.5 Destructors

The opposite of the constructors are destructors. C# builds upon the CLR which provides garbage collection and thus does not need an explicit destructor.

However, if you are using an unmanaged resource, such as a file handle, you will have to explicitly free that resource when not using it anymore.

This is accomplished using a `Finalize()` method, which will be called by the garbage collector when your object is destroyed.

The `Finalize()` method need only release resources that are used by your object, and should not reference any other objects.

You can implement a `Finalize()` method directly or indirectly using the C++ destructor syntax, where you declare a C# destructor with a tilde character preceding the class name as follows:

```

~ClassWithConstructor()
{
    // Release some resources
}

```

Which is simply a shortcut for declaring a `Finalize()` method that chains up to its base class. And is identical to writing:

```
ClassWithConstructor.Finalize()  
{  
    // Release some resources  
    base.Finalize();  
}
```

8.6 Overloading methods

The opposite of the constructors are destructors. C# builds upon the CLR. A Some times it is convenient to have more than one function with the same name. An example of this is having more than one constructor.

A class can have any number of methods, as long as each method's signature is unique, where the signature of a method is defined by its name and its parameter list. Two methods have different signatures if they have different names or different parameter lists. Parameter lists can differ in number of parameters or types of parameters.

For example, the following methods have different signatures:

```
void someMethodName(int i);  
int someMethodName (int i, int j);  
double someMethodName (int i, double d);
```

You overload a method, by changing the signature. Most common way to do so is changing the number of parameters, while keeping the same name for all of the overloaded methods.

You can also change the return type, but changing only the return type does not overload the method.

Example:

If we were to take the example from the preceding section and create an overloaded constructor for it, we'd end up with the following code:

```
using System;  
namespace cSharpClasses  
{  
    public class ClassWithConstructor  
    {  
        private int iValue;  
        private int iField;  
  
        //Overloaded Constructor method  
        public ClassWithConstructor()  
        {  
            iValue = 10;  
            iField = 20;  
        }  
        // Overloaded Constructor method  
        public ClassWithConstructor(int i)
```

```

    {
        iValue = i;
        iField = 20;
    }
    //Overloaded Constructor method
    public ClassWithConstructor(int i, int j)
    {
        iValue = i;
        iField = j;
    }

    public void DisplayValues()
    {
        Console.WriteLine(iValue);
        Console.WriteLine(iField);
    }
}
public class MainClass
{
    static void Main(string[] args)
    {
        // Create an object type variable
        ClassWithConstructor myInstance;

        // Create an object no parameters
        myInstance = new ClassWithConstructor();

        // Displays 10 and 20
        myInstance.DisplayValues();

        // Create an object      (instance of class)
        myInstance = new ClassWithConstructor(9);

        // Displays 9 and 20
        myInstance.DisplayValues();

        // Create an object      (instance of class)
        myInstance = new ClassWithConstructor(9, 13);

        // Displays 9 and 13
        myInstance.DisplayValues();
    }
}

```

9 Properties

Properties are a combination of fields and methods that look just like public fields to the accessing object yet have total control on how values are set and returned.

9.2 Prelude

Now let us look at a sample code, where we create a class, instantiate it, and retrieve a member field's value in the main method and display it there:

```
using System;

namespace cSharpMethods
{
    class myFirstClass
    {
        private int iValue = 1209;
        public int iField = 2001;

        public int getValue1()
        {
            //Return the value of the private field iValue
            return iValue;
        }
        public int getValue2()
        {
            //Return the value of the field iField
            return iField;
        }
    }
}

class mySecondClass
{
    static void Main()
    {
        // Create an instance of class myFirstClass
        // named myInst = object
        myFirstClass myInst = new myFirstClass();

        int value1 = myInst.getValue1 ();    // invoke a method

        // display updated values
        myInst.iField = 2003;                // modify a field value
        value2 = myInst.getValue2 (); // invoke a method
        System.Console.WriteLine("{0}\n{1}",value1, value2);
    }
}
```

In the preceding code the statement `myInst.iField = 2003;` sets the value of the public field that is declared and initialized inside the `myFirstClass` class. Should the value be invalid, then the class would have no control over it, neither could it do anything about it. The private field, on the other hand, is assigned a value locally, and can not be altered outside. And its value is accessed by means of the `getValue1()` method. C# introduces a seamless way of dealing with the situation above, using Properties. One of the many reasons Visual Basic was made so popular.

9.3 Declaring properties

The general format of a property is:

```
type ClassName
{
    type memberx;           // the internal private field
    type propertyName
    {
        get
        {
            // Some logic
            return memberx;
        }
        set
        {
            // Some logic
            memberx = value;
        }
    }

    // Some more class members
}
```

9.4 Using properties

We can set the initial value of the property (just as with the field) to 0; or whatever fits. Then we declare two methods to get and set the value of the property. If you omit the set method, then the property becomes read-only.

Accessing the property is just like accessing regular public fields. Except in the case of properties, the internal handling is different.

```
using System;

namespace cSharpProperties
{
    class PropertiesClass
    {
        int iValue = 0;    //the internal private field set to 0
        public int pValue
        {
            get
            {
                // Some logic
                return iValue;
            }
            set
            {
                // Some logic
                iValue = value;
            }
        }

        // Some more class members
    }
}
```

```

/// <summary>
/// Summary description for MainClass.
/// </summary>
class MainClass
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    static void Main(string[] args)
    {
        //
        // TODO: Add code to start application here
        //
        PropertiesClass pClass = new PropertiesClass();
        Console.WriteLine(pClass.pValue);
        pClass.pValue = 1978;
        Console.WriteLine(pClass.pValue);
    }
}
}

```

10 Interfaces

An interface is a pseudo definition of class members, which could contain methods, properties, events, and indexers.

An abstract signature list that the class implementing this interface promises to deliver to any object created as an instance of itself. The interface itself does not provide implementations for the members that it defines.

10.2 Declaring interfaces

Interfaces are defined using the interface keyword, and take the general form of:

Access modifiers `interface` `interfaceName` { `interface-body` }

```

interface IPlane
{
    String name {get; set; }
    Boolean TakeOff( );
    Boolean Land(object);
}

```

10.3 Implementing interfaces

A class can implement one or more interfaces, by adding the interface name to its declaration, following a colon. As in: `public class Boing : IPlane`

Where `Boing` is the class name and `IPlane` is the interface it is implementing. All members of an interface are public, by default, and they should be declared public in the implementing class.

The following example lists a working code:

```

using System;

namespace cSharpInterfaces
{
    interface IPPlane
    {
        String model {get;}
        Boolean TakeOff( );
        Boolean Land();
    }
    public class Boing : IPPlane
    {
        public string model
        {
            get
            {
                return "747-400";
            }
        }
        public Boolean TakeOff( )
        {
            return true;
        }
        public Boolean Land()
        {
            return false;
        }
    }
}
/// <summary>
/// Summary description for MainClass.
/// </summary>
class MainClass
{
    static void Main(string[] args)
    {
        Boing myBoing = new Boing();
        Console.WriteLine(myBoing.model);
        Console.WriteLine("Take off? {0}",myBoing.TakeOff());
        Console.WriteLine("Landed safely? {0}",myBoing.Land());
    }
}

```

The preceding example declares an interface with one Read-Only property and two Boolean methods.

The property is said to be read-only because it doesn't have a set method!

Interfaces are a good mechanism to create a system blueprint, on large projects where multiple programmers work on implementing class internals.

Interfaces also serve another purpose, since a class can implement multiple interfaces, and C# only allows a class to inherit from a single base class, it is an ideal mechanism to inherit from multiple bases.

10.4 Implementing multiple interfaces

A class can implement more than one interface, by adding the interface names to its declaration, following a colon and separated by commas. As in:

```
public class Boing : IPlane, IFloat
```

The following example lists a working code:

```
using System;

namespace cSharpInterfaces
{
    interface IPlane
    {
        String model {get;}
        Boolean TakeOff( );
        Boolean Land();
    }
    interface IFloat
    {
        int draft {get;}
        Boolean Float( );
    }
    public class Boing : IPlane, IFloat
    {
        public int draft
        {
            get
            {
                return 13;
            }
        }
        public Boolean Float( )
        {
            return true;
        }

        public string model
        {
            get
            {
                return "747-400";
            }
        }
        public Boolean TakeOff( )
        {
            return true;
        }
        public Boolean Land()
        {
            return false;
        }
    }
    /// <summary>
    /// Summary description for MainClass.
    /// </summary>
    class MainClass
    {
```

```

static void Main(string[] args)
{
    Boing myBoing = new Boing();
    Console.WriteLine(myBoing.model);
    Console.WriteLine("Took off? {0}",myBoing.TakeOff());
    Console.WriteLine("Landed safely? {0}",myBoing.Land());
    Console.WriteLine("Floating? {0}",myBoing.Float());
    Console.WriteLine("What Draft? {0} feet",myBoing.draft);
}
}
}

```

10.5 Explicit implementation of multiple interfaces

In the case of the previous example we have a class implementing two interfaces. If both were to declare a method with identical signature then the implementing class would create an ambiguous implementation of the method.

A way around this is provided by the hierarchical name structure using the dot operator, where the method name, would be preceded by the Interface name, followed by a dot. So if we had both interfaces declaring a method Boolean TakeOff(); As in the listing below

```

interface IPlane
{
    String model {get;}
    Boolean TakeOff( );
    Boolean Land();
}
interface IFloat
{
    int draft {get;}
    Boolean Float( );
}

```

The implementing class would have to use the explicit notation with at least one of the method implementations. I would personally use it on both:

```

public Boolean IPlane.TakeOff( )
{
    return true;
}

```

and/or

```

public Boolean IFloat.TakeOff( )
{
    return true;
}

```

There is some more stuff:

If I were to make a final note here, then I would point you to more study on what you could do with interfaces. You can for instance extend interfaces, to add new methods and properties. You can also combine multiple interfaces, test for

interface type using the `is` operator, and cast interfaces using the `as` operator. You can learn more about these topics from the extensive documentation provided by MSDN.

11 Delegates and event handlers

If, you are familiar with C or C++ languages, then you probably know a thing or two about delegates.

A delegate declares a reference type (similar to class) that has a method with a particular set of arguments and a return type. A delegate does not know or care about the class of the object that it references. It can access any member function as long as the method's signature matches that of the delegate's, which makes delegates ideal for anonymous invocation. The method's signature is just the return type and the parameter list.

11.2 Declaring Delegates

Delegates are defined using the delegate keyword, and take the general form of:

```
[modifiers] delegate result-type identifier ([formal-parameters]);
```

Where *[modifiers]* are any of the access modifiers discussed previously.

delegate is a C# keyword to declare a delegate

result-type identifier ([formal-parameters]); defines the method being delegated

Example:

The following code declares a public delegate of a method that returns nothing, and accepts three parameters

```
public delegate void AddPilot(string FullName, int FLYingHours, int age
);
```

11.3 Using delegates

Delegates are used just like classes, by first creating a variable of the delegate type, then assigning the variable an instance of the delegate, as follows:

```
AddPilot MyAddPilot;
MyAddPilot = new AddPilot( ClassInstance.AddPilotMethod );
```

ClassInstance is the object that has the implementation of the method AddPilotMethod with the signature defined in the delegate declaration.

```
public delegate void AddPilot(string FullName, int FLYingHours, int age
);
```

ClassInstance is defined as follows:

```
PilotClass ClassInstance = New PilotClass();
```

The complete listing follows:

```

using System;
namespace cSharpDelegates
{
    // Create a delegate
    public delegate void AddPilot(string FullName, int FLYingHours, int age
    );

    class MainClass
    {
        // The main entry point for the application.

        static void Main(string[] args)
        {
            // Create an instance of the PilotClass class
            PilotClass ClassInstance = new PilotClass();

            // Create an instance variable of the delegate type
            AddPilot MyAddPilot;

            // Create an instance of the delegate, with the method name
            MyAddPilot = new AddPilot( ClassInstance.AddPilotMethod );

            // Invoke the method with the delegate name
            MyAddPilot("Adam Salah", 750, 39);
        }
    }
    public class PilotClass
    {
        public void AddPilotMethod(string FullName, int FLYingHours, int age )
        {
            Console.WriteLine("Pilot {0} added", FullName);
        }
    }
}

```

This is similar to the notion of pointers to functions, or function pointers in C++. However, you will notice from the syntax in the preceding examples, that the instance creation follows the rules of the CTS, which also makes sure that the object type-safe and is initialized to null if no value is assigned to it.

The previous example shows a single cast delegate. A single delegate invokes only one callback method. Defined using the code in the line:

```

// Create an instance of the delegate, with the method name
AddPilot MyAddPilot;
MyAddPilot = new AddPilot( ClassInstance.AddPilotMethod );

```

This could also be written in a single statement as:

```

// Create an instance of the delegate, with the method name
AddPilot MyAddPilot = new AddPilot( ClassInstance.AddPilotMethod );

```

The delegate method is invoked using the code in the line:

```

// Invoke the method with the delegate name

```

```
MyAddPilot("Adam Salah", 750, 39);
```

Now let us rearrange the code, by moving all processing out of the Main() method into the PilotClass class. We first create a class constructor method to create the delegate's instance variable. Then we move the rest of the functional code into a new method body that we name AddThePilot(), just to allow us access using the object we create in Main().

Main() contains just two lines of code, the first creates an instance of the PilotClass class. The second invokes its method ClassInstance.AddThePilot().

The full code follows:

```
using System;

namespace cSharpDelegates
{
    // Create a delegate
    public delegate void AddPilot(string FullName, int FLyingHours, int age
    );

    class MainClass
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            // Create an instance of the PilotClass class
            PilotClass ClassInstance = new PilotClass();

            // Invoke the class methods using the Delegate
            ClassInstance.AddThePilot("Adam Salah", 750, 39);

        }
    }
    public class PilotClass
    {

        // Create an instance variable of the delegate type
        AddPilot MyAddPilot;

        public PilotClass() // Constructor method
        {
            // Create an instance of the delegate, with the method name
            MyAddPilot = new AddPilot( this.AddPilotMethod );
        }

        public void AddThePilot(string FullName, int FLyingHours, int age )
        {
            // Invoke the method with the delegate name
            MyAddPilot(FullName, FLyingHours, age);
        }

        public void AddPilotMethod(string FullName, int FLyingHours, int age )
        {

```

```

        Console.WriteLine("Pilot {0} added", FullName);
    }
}
}

```

The above may seem like a lot of work, and it is. Because the real power of delegates is not in its single cast use, but in the multicast delegation, which is when a delegate is used to invoke a sequence of methods, using a single call.

Taking the previous example of the PilotClass class, we could easily extend the delegate to invoke more methods, to carry out extra tasks.

Let us create two more methods. The first method to notify the tax-Authority, and the second method to notify the union, of our newly hired pilot.

To create a multicast delegate, we simply create multiple instances of the delegate, with the methods in sequence, added use the addition operator (+) on the methods.

```

// Create an instance of the delegate, with the method name
MyAddPilot = new AddPilot( ClassInstance.AddPilotMethod );
MyAddPilot = MyAddPilot + new AddPilot(ClassInstance.NotifyCCRAMethod);
MyAddPilot      = MyAddPilot + new AddPilot(ClassInstance.NotifyUnionMethod);

```

Or using the combined += short form:

```

// Create an instance of the delegate, with the method name
MyAddPilot = new AddPilot( ClassInstance.AddPilotMethod );
MyAddPilot += new AddPilot( ClassInstance.NotifyCCRAMethod );
MyAddPilot += new AddPilot( ClassInstance.NotifyUnionMethod );

```

So if we were to rewrite the previous example with the two new methods, and delegate instances modified accordingly, we end up with the following source:

```

using System;

namespace cSharpDelegates
{
    // Create a delegate
    public delegate void AddPilot(string FullName, int FLyingHours, int age
    );

    class MainClass
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            // Create an instance of the PilotClass class
            PilotClass ClassInstance = new PilotClass();

            // Invoke the class methods using the Delegate

```

```

        ClassInstance.AddThePilot("Adam Salah", 750, 39);
    }
}
public class PilotClass
{
    // Create an instance variable of the delegate type
    AddPilot MyAddPilot;

    public PilotClass()
    {
        // Create an instance of the delegate, with the method name
        MyAddPilot = new AddPilot( this.AddPilotMethod );
        MyAddPilot += new AddPilot( this.NotifyCCRAMethod );
        MyAddPilot += new AddPilot( this.NotifyUnionMethod );
    }

    public void AddThePilot(string FullName, int FLYingHours, int age )
    {
        // Invoke the method with the delegate name
        MyAddPilot(FullName, FLYingHours, age);
    }

    public void AddPilotMethod(string FullName, int FLYingHours, int age )
    {
        Console.WriteLine("Pilot {0} added", FullName);
    }

    public void NotifyCCRAMethod(string FullName, int FLYingHours, int age
    )
    {
        Console.WriteLine("CCRA notified of {0}'s hiring", FullName);
    }
    public void NotifyUnionMethod(string FullName, int FLYingHours, int age
    )
    {
        Console.WriteLine("Union notified of {0}'s hiring", FullName);
    }
}
}

```

11.4 Events and event handlers

An *event* in C# is a way for a class to notify its consumers (clients) of something that happens to an object. A typical use for events is in Microsoft Windows programming, where an event is fired when a user moves or clicks the mouse, or resizes a window.

An invocation of a method (function) occurs in response to the event. The methods (functions) are given to the class by its clients using delegates. Simply put, events enable class users to define delegates to methods which the class can use when something happens. You can think of this situation as a call that one makes to the class, and leaving a call back number, where one could be reached in case of an event occurrence.

Let us rewrite the previous example to introduce the event programming in C#.

The .NET Framework provides a built in delegate of type *System.Event*. This provides a single signature for the delegate regardless of the data that is passed to the method. This delegate signature comprises of two arguments. One is an object derived from the .NET Framework class *System.EventArgs* and contains the data the callback method requires (the function parameters). The other is the object requesting a call back also said to be the subscriber.

You declare a class derived from *System.EventArgs* with the data your callback needs. When the event takes place, you instantiate your derived *EventArgs* object and invoke the event.

Callback functions subscribed to the event are called passing the object derived from *EventArgs*. Changing the code from the previous sample to use events instead of delegates would involve the following changes:

1. Create a class derived from *System.EventArgs* to handle the data, which means the members of the class will be the parameters we used to pass in the case of delegates (delegate signature). The class derives from *System.EventArgs* as in the code:
`public class AddPilotEventArgs:EventArgs`
2. Change the delegate signature to match that of Events using the format: `AddPilotEvent( object sender, AddPilotEventArgs e )`
3. Then we change the delegate instance variable into an event using:
`public event AddPilotEvent OnAddPilot;`
4. The class constructor remains almost the same.
5. The action method `AddThePilot()` is changed to create an instance of the *EventArgs* class, and raising the Event with two parameters. `this` (which represents the current object) and the newly created data object *Args*, which contains the data.

Here is the full source, which looks much like the previous one:

```
using System;

namespace cSharpEvents
{
    // Event signature for AddPilotEvent
    public delegate void
    AddPilotEvent( object sender, AddPilotEventArgs e );

    // Create an EventArgs derived class
    public class AddPilotEventArgs : EventArgs
    {
        string privFullName;
        int privFLyingHours;
    }
}
```

```

    int privAge;

    // Indirect "Set" of properties implemented using constructor
    public AddPilotEventArgs(
        string FullName, int FLYingHours, int age)
    {
        privFullName = FullName;
        privFLYingHours = FLYingHours;
        privAge = age;
    }
    // Event argument properties contain the data to pass to the
    // callback methods subscribed to the event.
    public string FullName { get { return privFullName; } }
    public int FLYingHours { get { return privFLYingHours; } }
    public int age { get { return privAge; } }
} // end of EventArgs derived class

class MainClass
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    static void Main(string[] args)
    {
        // Create an instance of the PilotClass class
        PilotClass ClassInstance = new PilotClass();

        // Invoke the class methods using the Delegate
        ClassInstance.AddThePilot("Adam Salah", 750, 39);
    }
}

public class PilotClass
{
    // Instance of the AddPilotEvent.
    // Has to be defined at class level
    public event AddPilotEvent OnAddPilot;

    public PilotClass()
    {
        // Wire the event to the call back methods
        OnAddPilot += new AddPilotEvent( this.AddPilotMethod1 );
        OnAddPilot += new AddPilotEvent( this.NotifyCCRAMethod1 );
        OnAddPilot += new AddPilotEvent( this.NotifyUnionMethod1 );
    }

    public void AddThePilot(
        string FullName, int FLYingHours, int age )
    {
        // Create the EventArgs object (record)
        AddPilotEventArgs Args = new AddPilotEventArgs(
            FullName, FLYingHours, age );

        //Invoke the method, by raising an event
        OnAddPilot( this, Args );
    }
}

```

```

        public void AddPilotMethod1(object sender, AddPilotEventArgs e )
        {
            Console.WriteLine("Pilot {0} added", e.FullName);
        }
        public void NotifyCCRAMethod1(object sender, AddPilotEventArgs e
    )
        {
            Console.WriteLine("CCRA notified of {0}'s hiring", e.FullName);
        }
        public void NotifyUnionMethod1(
            object sender, AddPilotEventArgs e )
        {
            Console.WriteLine("Union notified of {0}'s hiring",
e.FullName);
        }
    }
}

```

All methods invoked by the event (wired to the event) are called event handlers.

Notice how all the methods wired to the event in this case are members of the same class? They don't have to be. They could easily be members of any other class. Furthermore, they could be wired to the event anywhere in the program. Let us rewrite the above example to demonstrate just that:

```

using System;

namespace cSharpEvents
{
    // Create a delegate
    public delegate void AddPilot(
        string FullName, int FLYingHours, int age );

    // Event signature for AddPilotEvent
    public delegate void AddPilotEvent(
        object sender, AddPilotEventArgs e );

    // Create an EventArgs derived class
    public class AddPilotEventArgs : EventArgs
    {
        string privFullName;
        int privFLYingHours;
        int privAge;

        // Indirect "Set" of properties implemented using constructor
        public AddPilotEventArgs(string FullName, int FLYingHours, int age)
        {
            privFullName = FullName;
            privFLYingHours = FLYingHours;
            privAge = age;
        }

        // Event argument properties contain the data to pass to the
        // callback methods subscribed to the event.
        public string FullName { get { return privFullName; } }
        public int FLYingHours { get { return privFLYingHours; } }
        public int age {get { return privAge; } }
    }
}

```

```

} // end of EventArgs derived class

class MainClass
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    static void Main(string[] args)
    {
        // Create an instance of the PilotClass class
        PilotClass ClassInstance = new PilotClass();

        // Create an instance of the BigBrotherClass class
        BigBrotherClass x = new BigBrotherClass();

        // Wire the event with call back methods
        // i.e. Subscribe to PilotClass class
        ClassInstance.OnAddPilot += new AddPilotEvent(x.Notifer);

        // Just to demonstrate that we can create the
        // data object anywhere
        // Create the EventArgs object (record)
        AddPilotEventArgs Args = new
            AddPilotEventArgs( "Adam Salah", 750, 39 );

        //Invoke the method, which will raise the event
        ClassInstance.AddThePilot(Args);
    } //end of Main()
} //End of MainClass

public class BigBrotherClass // always watching.. Newly added
{
    public void Notifer(object sender, AddPilotEventArgs e )
    {
        Console.WriteLine("We are watching the new Pilot {0}", e.FullName);
    }
}

public class PilotClass
{
    // Instance of the AddPilotEvent.
    // Has to be defined at class level
    public event AddPilotEvent OnAddPilot;

    public PilotClass()
    {
        PilotMethods pm = new PilotMethods();
        // Wire the event with call back methods
        OnAddPilot += new AddPilotEvent( pm.AddPilotMethod );
        OnAddPilot += new AddPilotEvent( pm.NotifyCCRAMethod );
        OnAddPilot += new AddPilotEvent( pm.NotifyUnionMethod );
    }

    public void AddThePilot(AddPilotEventArgs Args ) //string FullName, int
    FLyingHours, int age )
    {
        //Invoke the method, by raising an event
    }
}

```

```

        OnAddPilot( this, Args );
    }
}
public class PilotMethods
{
    public void AddPilotMethod(object sender, AddPilotEventArgs e )
    {
        Console.WriteLine("Pilot {0} added", e.FullName);
    }
    public void NotifyCCRAMethod(object sender, AddPilotEventArgs e )
    {
        Console.WriteLine("CCRA notified of {0}'s hiring", e.FullName);
    }
    public void NotifyUnionMethod(object sender, AddPilotEventArgs e )
    {
        Console.WriteLine("Union notified of {0}'s hiring", e.FullName);
    }
}
}
}

```

We made the following changes:

- a. Moved the logic to create the EventArgs object to Main().
- b. Created two new classes, BigBrotherClass and PilotMethods.
- c. The PilotMethods class contains the methods that were formerly members of the PilotClass class. Hence the addition of the line of code to create the PilotMethods-type class instance:
PilotMethods pm = new PilotMethods();
- d. Wired the event to Notifer method of the BigBrotherClass class, using: OnAddPilot += new AddPilotEvent(x.Notifer); in the Main() method.

The last point is really what we intended to highlight here, were we wire an event of the OnAddPilot of the PilotClass class, to a method of BigBrotherClass class, in the Main() method which in turn, is a member of the MainClass class.

The chain of events would be as follows:

1. Main() is invoked by the CLR which will do the following:
 - a. Create an instances of PilotClass and BigBrotherClass
 - b. As a result of (a), the PilotClass class's constructor will be invoked, creating an instance of PilotMethods class, and then wiring its methods to the event.
 - c. Add the BigBrotherClass event handler to the OnAddPilot event. Effectively subscribing to the PilotClass class
 - d. Create the Args object of AddPilotEventArgs type, with the data to be supplied to the subscribing methods.
 - e. Invoke the AddThePilot(Args)method of the PilotClass class
2. The invocation of the AddThePilot() method of the PilotClass class simply fires the event OnAddPilot(), effectively starting the delegates'

execution. Or in other words, start the notification of the subscribing objects.

The event driven programming model is based on the concepts discussed in the preceding sections. The event driven model, also known as publish/subscribe model is very well suited for the windows Graphical User Interface (GUI) programming. The concept of an object responding to an event, by invoking one or more handlers, corresponds to the GUI objects, almost one to one.

A button on a windows form is a good example of an object, waiting for a user initiated event, such as a mouse move-over, click, or a double click. The button object could have event handlers wired to these events. The event handlers would then respond to the events with an appropriate action. We will discuss the various components of the windows based programming objects in the next chapter.

12 Developing Windows Forms programs in C#

12.2 Overview

Microsoft .NET was developed with a lot of emphasis on the Web (interNET), but this doesn't mean they abandoned Windows development (god forbid!). The .NET Framework actually benefits from the development of web applications, and improves the way standard Windows applications are built. So much has been done to bridge the gap between the web-based and windows-based application development models, that the line is quite blurred. My guess is, in a few releases of the .NET framework, we will end up with a single programming model, with code generators taking care of the actual nitty-gritty details of the implementation.

In this chapter, we introduce Windows Forms, Using .NET classes to create Windows Forms-based applications, using the C# language and Visual Studio .NET 2003. And of course all the concepts we discuss apply perfectly to the SDK based development.

The components provided in the .NET library for writing Windows applications can be divided into two categories: Windows Forms (providing classes for windows and controls) and the graphics device interface known as GDI+ (the classes provide graphics and printing functions). This chapter introduces you to Windows Forms, by taking you step by step through the process of creating a simple windows application.

GDI+ is beyond the scope of this text. However, you can learn about it through the MSDN and/or the SDK documentation.

12.3 Introducing Windows Forms

If you have developed Windows applications in the early days of Win32 SDK, then you probably remember some of the Windows APIs by heart! Even if you were a VB programmer, then you had to use DLL calls to accomplish certain tasks.

Then Microsoft Foundation Classes (MFC) came to the rescue of C++ programmers, while VB programmers had the Visual Basic object library, and VC++ programmers could use the Active Template Libraries (ATL), which I found to be quite restrictive, myself.

One major challenge in the above was that each one dealt with the matter from a language specific perspective. No common library was available to provide an interface to the Windows APIs. Furthermore, you still needed to make low-level calls to carry out some tasks, which may not be facilitated by these libraries.

There has also been a parallel development of third party libraries, such as such as the Object Windows Library (OWL) from Borland, which build on top of the core Win32 APIs.

Windows Forms thrives to solve of the above issues and more, by providing a unified programming model that is much richer and more powerful than any of the existing libraries.

Windows Forms provides a hierarchy of classes, organized in namespaces (in the .NET fashion). Which means that any .NET has access to a uniform set of classes, making it easier on programmers with expertise in different languages to work together, and not having to learn new languages, just to be able to write some subroutine.

Windows Forms makes it possible for VS.NET to bring a VB-like integrated development environment (IDE) to C#, Managed C++, and many other languages.

12.4 Simple Windows Forms Application development

Writing Windows Forms applications is not much different from writing a console application, in that you still have to develop the code that handles your business logic. The only new element here is the Graphical User Interface (GUI), instead of the text based console UI. Which requires the inclusion of the new assembly: SYSTEM.WINDOWS.FORMS.DLL that must be referenced in your project. This assembly contains the classes of the namespace, which we discuss in the next section.

12.4.1 The System.Windows.Forms Namespace

The classes that make up the Windows Forms are contained in the *System.Windows.Forms* namespace. Presenting a clean and simple architecture, which comprises of three major components:

Containers: Defines a class representing a window to which you can add controls (hosting other controls, hence the containers name)

Controls: Provides the core functionality for a visual control that responds to mouse clicks and keyboard strokes, accepts focus, and can participate in drag and drop operations.

Components: Represents non-visual elements, such as timer control.

All of the three components inherit from a *Control* class, which is the mother of all user interface objects, which in turn, inherits from *Object*.

12.4.2 Application Class

Windows Forms applications also have an entry point which is the *Main()* function. In windows Forms, you use the *System.Windows.Forms.Application* object to open the main window and enter the message loop.

The *Application* class offers the static *Run* method, which gets a reference to the main form as parameter. The *Run* method starts or stops the application message loop. After calling *Run*, your application is ready to receive events and interact with the user.

The static *application.Run* method is used in programs as follows:

```
System.Windows.Forms.Application.Run(new MyWinForm( ));
```

, And it takes an instance of the Form's class as a parameter (notice the *new* keyword?).

12.4.3 Form Class

The core element in Windows Forms programming is the Form (window and Form are used interchangeably to mean the same thing), which gets its name from VB!

Form is a container type of control, because it contains other controls.

The first step in creating a Windows Forms application is to derive your class from `System.Windows.Forms.Form` as follows:

```
public class MyFormsClass : Form
```

Once the form is created, you can set its properties to control its appearance, size, and color, invoke its methods, and add other controls to it.

Let take a look at a simple Windows Forms application, which displays an empty form:

```
using System;
using System.Windows.Forms;
namespace MyFirstWindowsForm
{
    public class MyWinFrom : Form
    {
        public MyWinFrom ( )
        {
            this.Location = new System.Drawing.Point(100, 100);
            this.Text = "Hello, Universe";
        }
        public static void Main( )
        {
            Form MyFormX = new MyWinFrom( );
            Application.Run(MyFormX);
        }
    } //class
} // namespace
```

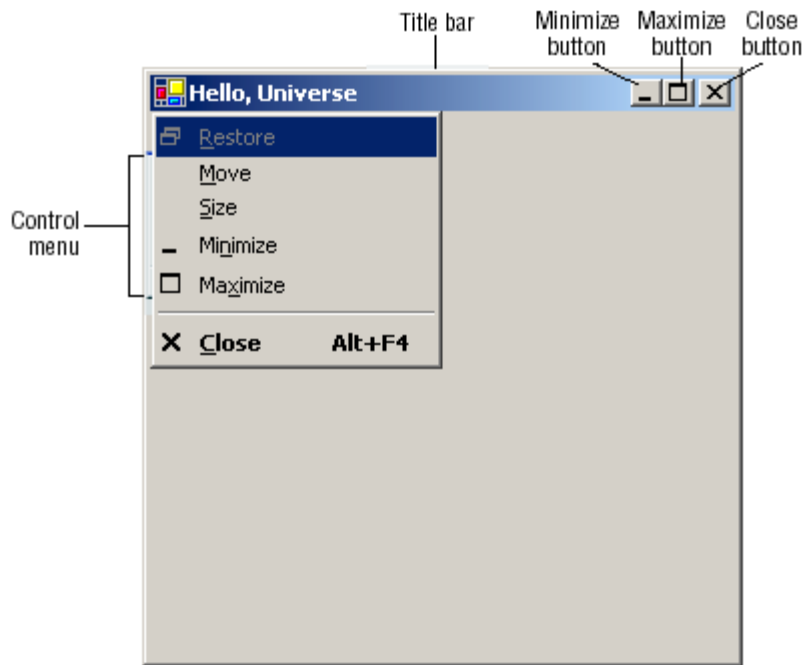
The preceding code includes a reference to the `System.Windows.Forms` namespace, then creates a new class `MyWinFrom`, which derives from `System.Windows.Forms.Form` (written `Form`, because the `using` statement imports the `System.Windows.Forms` namespace). In the class constructor, the form's `Location` is set to (100,100) and the `Text` property is set to "Hello, Universe", this will appear at the title bar.

Finally in the `Main( )` method, we invoke `System.Windows.Application.Run` with the class instance as a parameter. The result of compiling and running the above code displays a typical standard form (window).

A standard form usually contains a title bar, with an icon, title text, and control box (Minimize, Maximize, and Close buttons). Forms can also have a menu, and may contain child controls inside its frames.

The form boundaries (window frame) can be resized and also may contain scroll bars.

The form resulting from running the preceding code is depicted in the figure below. The control menu is automatically created by the `Form` class and can be accessed by clicking on the Icon at the top left corner of the form, or by pressing ALT + Spacebar:



Summary:

To create a form we:

1. Include the namespace
2. Create a new class deriving from Form
3. Set forms properties in the class constructor
4. Write a Main() method (entry point)
5. Create an instance of the class
6. Call Application.Run() method with the instance of the class

12.4.4 Event Handlers

In the Preceding chapter we discussed Delegates and events.

Events are a critical concept in windows programming, as all messages exchanged between the different components are communicated by way of events and event handlers.

Almost every form and control raises a number of events to which we can wire a handler method.

Let us add an event handler to the form created in the example of the preceding section.

The Form class defines a Load event, to which we can wire a handler that will be called by the object, in response to the Load event. Remember how we wire event handlers to object's events? += HandlerName.

Here is the full code:

```
using System;
using System.Windows.Forms;
namespace MyFirstWindowsForm
{
```

```

public class MyWinFrom : Form
{
    public MyWinFrom ( )
    {
        this.Location = new System.Drawing.Point(100, 100);
        this.Text = "Hello, Universe";
        // Define an event handler
        this.Load += new System.EventHandler(this. MyWinFrom_Load);
    }
    public static void Main( )
    {
        Form MyFormX = new MyWinFrom( );
        Application.Run(MyFormX);
    }
    private void MyWinFrom_Load(object sender, System.EventArgs e)
    {
        Text = Text + " my new name!";
    }
} //class

} // namespace

```

12.5 Windows Forms Controls

The core element in Windows Forms programming is the Form (window) and a Windows application uses one or more Forms. Each form may contain child controls for user or system interaction. The windows controls have different uses and features. Most commonly used controls include buttons, List controls, Data bound controls, and user controls. Adding controls to an application usually involves creating a container form, as in the previous section, then following a four-step process as follows:

1. Create the control instance
2. Set the properties of the control such as location, dimensions and Text
3. Create and wire event handlers for the control
4. Add the control to the controls collection of the form

Let us go ahead and add a button control to the previous example:

```

using System;
using System.Windows.Forms;
namespace MyFirstWindowsForm
{
    public class MyWinFrom : Form
    {
        public MyWinFrom ( )
        {
            // step
            //Set form's location
            this.Location = new System.Drawing.Point(100, 100);
            this.Text = "Hello, Universe";
            // Define an event handler
            this.Load += new System.EventHandler(this. MyWinFrom_Load);

```

```

        //Create a Button control
        Button myButton;
        // step 1: create an instance
        myButton = new Button();
        // step 2: Set properties
        myButton.Text = "Say Hello -:)";
        myButton.Location = new System.Drawing.Point(60,40);
        myButton.Size = new System.Drawing.Size(100, 40);
        // step 3: create event handler
        myButton.Click += new EventHandler( this.myButton_Click );
        //step 4: add control to the Controls list
        this.Controls.Add( myButton );
    }
    public static void Main( )
    {
        Form MyFormX = new MyWinFrom( );
        Application.Run(MyFormX);
    }
    private void MyWinFrom_Load(object sender, System.EventArgs e)
    {
        Text = Text + " my new name!";
    }

    private void myButton_Click(object sender, System.EventArgs e)
    {
        MessageBox.Show("Hello! Hello.. -:)");
    }
} //class

} // namespace

```

Let us add yet another control to the above example, this time a TextBox. When you run this program, you will get a text box at the top, and a button underneath it. Type your name in the textbox, and then click the button, to receive a personalized greeting.

```

using System;
using System.Windows.Forms;
namespace MyFirstWindowsForm
{
    public class MyWinFrom : Form
    {
        //Create a TextBox control
        TextBox myTextBox;

        public MyWinFrom ( )
        {
            // step
            //Set form's location
            this.Location = new System.Drawing.Point(100, 100);
            this.Text = "Hello, Universe";
            // Define an event handler
            this.Load += new System.EventHandler(this. MyWinFrom_Load);
        }
    }
}

```

```

        //Create a Button control
        Button myButton;
        // step 1: create an instance
        myButton = new Button();
        // step 2: Set properties
        myButton.Text = "Say Hello -:)";
        myButton.Location = new System.Drawing.Point(60,40);
        myButton.Size = new System.Drawing.Size(100, 40);
        // step 3: create event handler
        myButton.Click += new EventHandler( this.myButton_Click );
        //step 4: add control to the Controls list
        this.Controls.Add( myButton );

        // step 1: create an instance
        myTextBox = new TextBox();
        // step 2: Set properties
        myTextBox.Text = "";
        myTextBox.Location = new System.Drawing.Point(0,0);
        myTextBox.Size = new System.Drawing.Size(300, 30);
        // step 3: create event handler
        // skipped for the TextBox control..

        //step 4: add control to the Controls list
        this.Controls.Add( myTextBox );
    }
    public static void Main( )
    {
        Form MyFormX = new MyWinFrom( );
        Application.Run(MyFormX);
    }
    private void MyWinFrom_Load(object sender, System.EventArgs e)
    {
        Text = Text + " my new name!";
    }

    private void myButton_Click(object sender, System.EventArgs e)
    {
        MessageBox.Show("Hello! Hello.. -:) "+ myTextBox.Text);
    }
} //class
} // namespace

```

In this case we added a TextBox control. I want you to take note of the following:

1. We added the TextBox declaration at the class level, outside the constructor, because we needed to access it from another place (the Button's Click event handler)
2. We elected not to have an event handler for the TextBox control

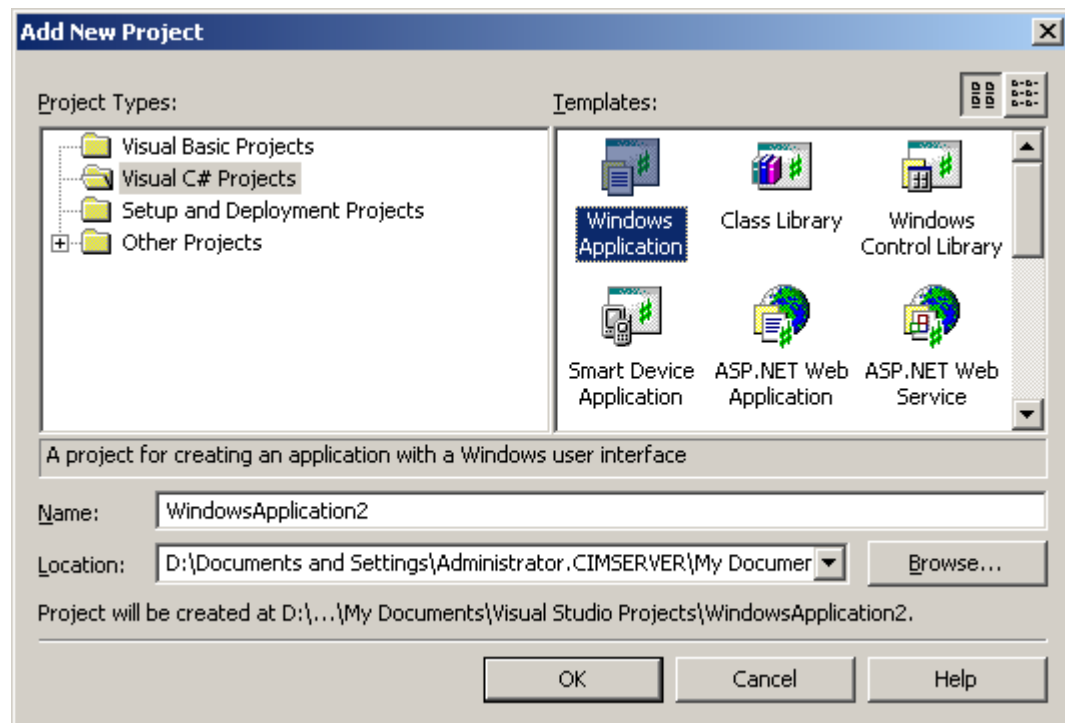
The above is obviously the hard way of doing things. The easier way being the use of VS.NET, which automates most of the tedious visual layout tasks, as well as creating standard templates for things such as event handler methods.

12.6 Using VS.NET to develop Windows Forms

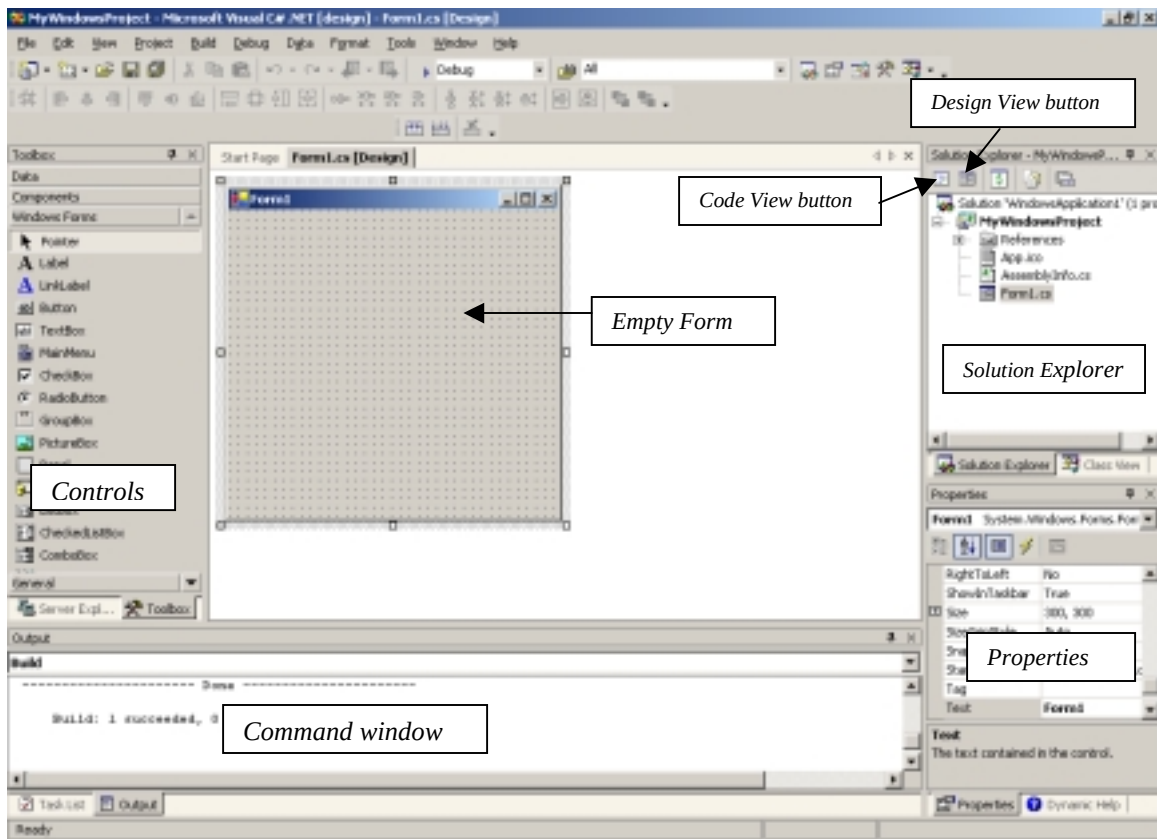
12.6.1 Creating a project

To create a new project,

11. Start VS.NET.
12. Make sure you have the “Start Page” open (use menu item: **Help → Show Start Page**)
13. Click “New Project” hyperlink at the bottom of the screen
14. You will get a screen similar to the figure below



15. Click on “Visual C# Projects” on the upper-left explorer-style folder. This will ensure that we select the right language.
16. Select “Window Application” in the right-side panel. This will ensure we are creating a Windows Forms Application.
17. In the “Name:” text box type “myWindowsProject”.
18. The “Location:” drop-down list (Combo box if you are a VB programmer), should contain the correct path for your files. You can click the “Browse...” button to launch a windows explorer, and then pick the right folder.
19. Press OK.
20. You should see a screen similar to the figure below:



12.6.2 Adding a Form

When you create a new Windows Application project, VS.NET automatically creates a Form for you. You can also add a new form to an existing project, using menus and tool bars.

12.6.3 The code behind the form

Now let us examine the code that was generated for us by the VS.NET. To access the corresponding code, right click the form's name in the solution explorer and select "View Code". You can also switch between the designer and code views using the corresponding buttons at the top of the solution explorer, as indicated on the figure above.

The code generated by VS.NET, which is listed below, features a template that you will see over and over when using the IDE to develop Windows Forms applications, so let us take a closer look at it:

1. We have the usual referencing of namespaces by virtue of the `using` keyword. Notice that there are some namespaces, added by VS.NET, which we have not discussed so far, such as `Collections`, `ComponentModel` and `Data`.
2. Create the project namespace, which is by default the same as the project name that we typed in the project creation dialog.

3. A class Form1 which inherits (:) from the System.Windows.Forms.Form base class is created for us. The Form1 name is the name of the visual form, and the source file Form1.cs. But they don't have to be the same.
4. An object of type System.ComponentModel.Container named components is created for internal designer use.
5. The Form1 class constructor contains a single line of code that invokes a InitializeComponent() method. which is the method used by VS.NET designer to paint the design view. It crucial that you don't tamper with the code in this method, unless you intend to take control of the creation and management of forms and controls! The InitializeComponent() method is normally encapsulated in a collapsed region. The contents of this method are very highly dynamic, as its contents represent the code persistent version of the visual elements
6. Then we have the implementation of the overridden dispose method is invoked internally when the form is closed. You override a method that has been declared a base class, by redefining its action, while maintaining its signature, and us the keyword `override`.
7. Then we have the Collapsible region we mentioned in point 4. Where the InitializeComponent() method is implemented. As we mentioned earlier, the contents of this method will keep changing, as you work on the designer. The first line will only be there after you first create the project. Let us think of it, as the declaration of default container. Immediately after that we have the code that set the Form properties. Using "`this`" operator refers to the current class, which is Form1!
8. Finally the Main() method, where the Application.Run is invoked to kick start the windows message loop.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace MyWindowsProject
{
    /// <summary>
    /// Summary description for Form1.
    /// </summary>
    public class Form1 : System.Windows.Forms.Form
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;

        public Form1()
        {
            //
            // Required for Windows Form Designer support

```

```

        //
        InitializeComponent();
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
        if( disposing )
        {
            if (components != null)
            {
                components.Dispose();
            }
        }
        base.Dispose( disposing );
    }

    #region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        this.components = new System.ComponentModel.Container();
        this.Size = new System.Drawing.Size(300,300);
        this.Text = "Form1";
    }
    #endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
        Application.Run(new Form1());
    }
}

```

12.6.4 Modifying properties of a Form

You can modify the properties of a form once created, using the visual designer, by accessing the properties form. Press F4, select View→Properties window from the main menu, or simply press the  icon from the toolbars.

The properties window located at the lower right hand side of the VS.NET IDE is similar to the familiar VB or Access property sheet. Comprising of a name-value pairs, you select the property that you want to modify and change its value in the text field next to its name.

12.6.4.1 Modifying The Name property of a Form

Say we want to give our Form a more meaningful name, and then we select the Name property, erase the current “Form1”, and type the new name. Say something original such as MyForm01.

The immediate effect of this can be observed in the code, where MyForm01 will replace all references to Form1.

You may have to change the form name in the Main() method manually. So the line `Application.Run(new Form1());` should be replaced with `Application.Run(new MyForm01());`.

12.6.4.2 Modifying The Text property of a Form

Now let us change the text in the title bar to something more humanely readable, such as “My first form”. The property is called **Text**. This will affect the code in the `InitializeComponent()` method.

12.6.5 Adding event handler to the Form

Using VS.NET, you have two types of event handlers that you can add to a form or a control: The default-event handlers, or custom-event handlers. Each form and control has a default event that is associated with it, and is created automatically along with a default handler named `controlName_HandlerName()`. For example `MyForm01_Load()` is the default handler for MyForm01, and the event is Load.

12.6.5.1 Adding a default event handler to a Form

To add a default event handler you simply double-click the form or control and two things will happen:

1. The event handler name will be wired, automatically to the event in the `InitializeComponent()` method.
2. You will be taken into the body of a method (an event handler) that conforms to the naming convention discussed in the preceding paragraph.

If we double-click on MyForm01 and add an event handler code to simply change the form’s text property, we will end up in the code view with the following:

1. The cursor will be within the body of the `private void MyForm01_Load(object sender, System.EventArgs e)` method. Which is the event handler for the Load event. Complete with the sender object and EventArgs type parameters. Sender in this case will be MyForm01.
2. Looking at the `InitializeComponent()` method reveals the following:
 - a. The form’s name and Text properties changes reflected in:

```
this.Name = "MyForm01";  
this.Text = "My First Form";
```

- b. The event handler has been wired to the form's Load event
 `this.Load += new System.EventHandler(this.MyForm01_Load);`
3. Everything else remains the same

The complete source is listed below.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace MyWindowsProject
{
    /// <summary>
    /// Summary description for Form1.
    /// </summary>
    public class MyForm01 : System.Windows.Forms.Form
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;

        public MyForm01()
        {
            //
            // Required for Windows Form Designer support
            //
            InitializeComponent();

        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }

        #region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {

```

```

        //
        // MyForm01
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(292, 273);
        this.Name = "MyForm01";
        this.Text = "My First Form";
        this.Load += new System.EventHandler(this.MyForm01_Load);
    }
#endregion

/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main()
{
    Application.Run(new MyForm01());
}

private void MyForm01_Load(object sender, System.EventArgs e)
{
    this.Text = this.Text + " Handling events";
}
}
}

```

12.6.5.2 Adding an event handler to a Form

To add an event handler for events other than the default, go through the following steps.

1. Switch to designer view by clicking () in the Solution explorer window
2. Click the form to make it active
3. Click the Events button () , located at the top of the Properties window,
4. Just as with the properties, click the event that you want to create an event handler for
5. In the text box to the right of the event name, type the name of the handler and press ENTER.

The name could be any valid C# method name, but using the naming convention adopted by VS.NET is a good practice. If you were handling the Move event, for instance, then you may want to call the handler `MyForm01_Move`. If you accept this naming convention, then you actually don't have to type the name. Just double click the text box and VS.NET will type it for you.

6. The preceding step will have a similar effect to double-clicking the form, except now you will be taken into a handler for an event that you chose, with a name that you typed.

The events in the event list can be broadly categorized into two groups: Events that raised before the occurrence of the actions, usually ending with an “ing”, and events which are raised after the occurrence of the action, usually ending with “ed” or neither “ing” nor “ed”. It is important to know what event to handle at what point.

Let us handle the closing event of the form to do some pre-closing work and double check with the user that closing the form is what they really meant. By selecting the closing event and double-clicking the text box, then typing the two lines of code enclosed within method, we handle the closing event, by changing the title bar text and displaying a message box, for the user to respond to.

```
private void MyForm01_Closing(object sender,
                             System.ComponentModel.CancelEventArgs e)
{
    this.Text = this.Text + " I am being closed";
    MessageBox.Show("Are you leaving?");
}
```

Now let us examine the InitializeComponent() method, where the event handler is wired to the event:

```
this.Closing += new
System.ComponentModel.CancelEventHandler(this.MyForm01_Closing);

this.Load += new System.EventHandler(this.MyForm01_Load);
```

Notice the difference between the two events? The reason is that some of the events that are raised before the occurrence of the action are CANCELLABLE! Which means we can roll back the entire action, and resume normal execution. In this particular case, we could ask the user whether they mean to exit, and give them a chance to retreat.

These events use the **CancelEventHandler** and **CancelEventArgs** classes. The **CancelEventArgs** class contains a property called **Cancel**. If this property is set to **true**, when the event-handling method returns, the event is cancelled.

12.6.6 Adding controls to a Form

Double-clicking a control in the toolbox adds it to the active form. The code generated by the designer will typically carry out the following tasks:

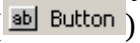
1. Create an instance of the control
2. Call the **SuspendLayout** method and then set the Name, Size, Location and other properties of the control

3. Add the control to Controls list
4. Call the **ResumeLayout** method to allow the changes to take effect.

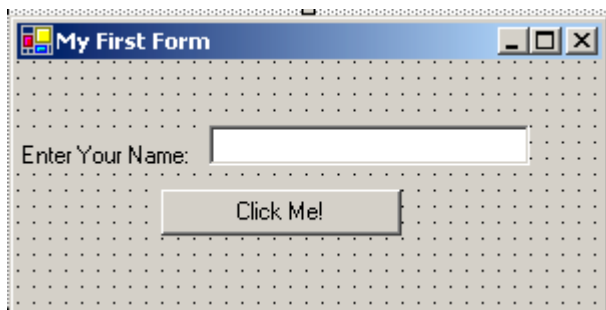
Once the control has been laid out on the form, you can move it around using the mouse, change its properties, and create event handlers for it, using the same techniques discussed for Forms.

Let us create a small application using some of the common controls in the toolbox.

12.6.6.1 A simple application

1. Create a new project, and call it MyFirstApplication
2. Add the controls, and set their properties as follows:
 1. Add a Label control by double clicking on ( Label)
 2. Select the newly added control, and press F4
 3. Change the Text property, in the properties sheet to "Enter your Name: "
 4. Change the AutoSize property to "True", which means the Label control will always fit the size of the text.
 5. Add a TextBox control by double clicking on ( TextBox)
 6. Empty its Text property, by deleting its content in the properties
 7. Resize it by dragging its rightmost corner horizontally
 8. Add a Button control by double clicking on ( Button)
 9. Change the Text property, in the properties sheet to "Click Me!"
 10. Re-arrange the controls by moving them around to take the following shape:

Hint: you can press the ALT key while moving the mouse to override snapping to grid, resulting in a finer move/resize.



3. Create an Event handler for the button control as follows:
 1. Double click the button control
 2. Type the following code in the event handler

```

if (this.textBox1.Text != "")
{
    MessageBox.Show("Hello there, " + this.textBox1.Text);
}
else
{

```

```

        MessageBox.Show("Type your name PLEASE!");
    }

```

3. Press F5 to run the program and try it out

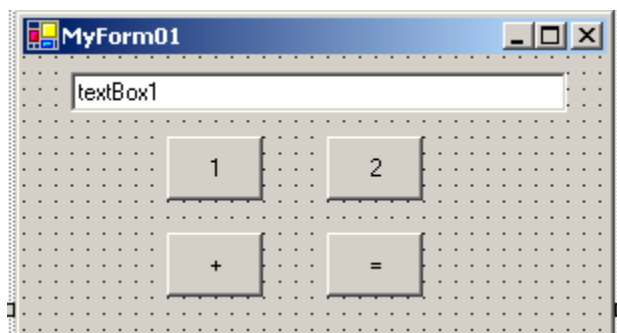
Further enhancements to the program would include, clearing out the contents of the textbox after the message box, checking for a valid name (say make sure it is only alphabets).

12.6.6.2 A more useful application

Let us add a few more buttons to the previous program and transform it into a simple calculator.

1. Open the project from the previous example: MyFirstApplication
2. Remove the Label control, since we wont need it
3. Add some more controls, and set their properties as follows:
 1. Select the "Click Me!" Button control, and press F4
 2. Change the Text property, in the properties sheet to "1"
 3. Add a Button control by double clicking on (ab| Button)
 4. Change the Text property, in the properties sheet to "2"
 5. Add a Button control by double clicking on (ab| Button)
 6. Change the Text property, in the properties sheet to "+"
 7. Add a Button control by double clicking on (ab| Button)
 8. Change the Text property, in the properties sheet to "="
 9. Re-arrange the controls by moving them around to take the following shape:

Hint: you can press the ALT key while moving the mouse to override snapping to grid, resulting in a finer move/resize.



4. Add the following code to the Form's class constructor

```

private int operand1, operand2;
private int result;
private bool clearDisplay = false;

```

Including the designer generated code; the result should look something like the following snapshot:

```

public class MyForm01 : System.Windows.Forms.Form
{

```

```

        private int operand1, operand2;
        private int result;
        private bool clearDisplay = false;
        private System.Windows.Forms.TextBox textBox1;
        ...
    }

```

5. Create an Event handler for the each button control as follows:

1. Double click the first button control “1”

2. Create the following event handler:

```

private void button1_Click(object sender, System.EventArgs e)
{
    if (clearDisplay)
    {
        clearDisplay = false;
        this.textBox1.Text = "";
    }
    this.textBox1.Text += button1.Text;
}

```

3. Switch back to the Designer view

4. Double click the first button control “2”

5. Create the following event handler:

```

private void button2_Click(object sender, System.EventArgs e)
{
    if (clearDisplay)
    {
        clearDisplay = false;
        this.textBox1.Text = "";
    }
    this.textBox1.Text += button2.Text;
}

```

6. Switch back to the Designer view

7. Double click the first button control “+”

8. Create the following event handler:

```

private void button3_Click(object sender, System.EventArgs e)
{
    this.operand1 = int.Parse(this.textBox1.Text);
    this.clearDisplay = true;
}

```

9. Switch back to the Designer view

10. Double click the first button control “=”

11. Create the following event handler:

```

private void button4_Click(object sender, System.EventArgs e)
{
    this.operand2 = int.Parse(this.textBox1.Text);
    this.result = this.operand1 + this.operand2;
    this.textBox1.Text = this.result.ToString();
    this.clearDisplay = true;
}

```

12. Press F5 to run the program and try it out

The above calculator obviously leaves a lot to be desired! Some of the enhancements that one could think of include, handling more digits (3-9 and 0), more arithmetic operations (-, *, /), handling of fractions (.). Also handling the backspace and Enter keys would be a plus.

12.6.6.3 Handling multiple events using a single handler

Notice the similarities in the code of the Button event handlers? We could reduce the number of event handlers by using a single handler for the first two buttons, or even for the whole bunch!

The way to accomplish this is by defining a method and wiring all of the events to this method. Inside the method, you use the sender object to identify who raised the event.

Let us rewrite the example in the previous section to use a single event handler instead of the four, which we have.

1. First we create a generic handler called ButtonX_Click() as follows:

```
private void buttonX_Click(object sender, System.EventArgs e)
{
    if (sender == button1 || sender == button2)
    {
        if (clearDisplay)
        {
            clearDisplay = false;
            this.textBox1.Text = "";
        }
        this.textBox1.Text += ((Button) sender).Text;
    }

    if (sender == button3)
    {
        this.operand1 = int.Parse(this.textBox1.Text);
        this.clearDisplay = true;
    }
    if (sender == button4)
    {
        this.operand2 = int.Parse(this.textBox1.Text);
        this.result = this.operand1 + this.operand2;
        this.textBox1.Text = this.result.ToString();
        this.clearDisplay = true;
    }
}
```

Buttons 1 and 2 have the same functionality, which adds the button's Text contents to Text value of the TextBox. We use the sender object's Text property, after casting it to button, from Object, using (Button) notation. We have to do this, because Object is a generic type, which has no Text property.

Buttons 3 and 4 have different functionalities so we keep them separate.

2. Second we wire the new event handler to the click event of each button, by simply modifying the code inside the InitializeComponent() method, as follows:

```
this.button1.Click += new System.EventHandler(this.buttonX_Click);  
...  
this.button2.Click += new System.EventHandler(this.buttonX_Click);  
...  
this.button3.Click += new System.EventHandler(this.buttonX_Click);  
...  
this.button4.Click += new System.EventHandler(this.buttonX_Click);  
...
```

3. Finally we remove the old event handlers, which is practically a dead code that not accessed by any part of our application.

13 Developing Data-aware programs in C# (ADO.NET)

13.1 Overview

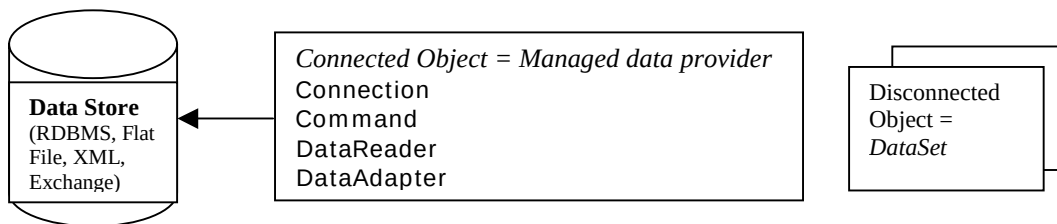
ADO.NET, is right in the core of the .NET framework, as you may recall from the architecture diagram in chapter 1.

Which makes it available to all programming languages, and of course, making sure that it conforms to, and inherits all the .NET goodies.

13.2 ADO.NET Architecture

At a very high level, ADO.NET can be thought as a PIPE that provides data from a storing source (data store) to a consuming application (consumer).

This PIPE can be grouped into two logical levels, the **connected** layer which comprises of the *managed providers* and **disconnected** layer which is primarily represented by a *DataSet* object. The following diagram attempts to visualize the idea:



13.2.1 .NET Data Providers

The *.NET Managed Provider* also known as *Data Provider*, handles all data-related activities by providing access to the data store, which could be an RDBMS, A flat file, OLEDB source; such as Excel, Access or Exchange. The actions carried out by a .NET data providers include **connecting** to a database, executing **commands**, and retrieving the **results** of the commands' execution.

The **resulting** data can be stored in memory, in its entirety, for access by application, and is known as *DataSet*. Any modifications carried out on the data in memory (*DataSet*) can be written back to the data store, using the .NET data providers.

The .NET data provider uses four components known as the *.NET data provider objects*. The .NET data provider objects are:

- The **Connection** object: Used to connect a command object to a data store
- The **Command** object: Executes commands against a data store, usually SQL.
- The **DataReader** object: Returns a forward-only, read-only stream from a data store
- The **DataAdapter** object: Used to populate a DataSet with data, stores 4 command objects: Select, Update, Insert and Delete.

As a consumer of data you receive a standard stream of data, regardless of the source, which could be on the same machine or across planet over the internet. The data store could also be of any type, including SQL Server, Oracle, Access or even a good old

ODBC. To facilitate this, and to accomplish the data access in the most efficient way, the .NET Framework includes different types of .NET data providers. There are four types that ship with the .NET framework 1.1, and Visual Studio .NET 2003. The four types of .NET data providers are:

- **SQL Server Managed data provider.** The SQL Server .NET data provider to specifically written to access an SQL Server 7.0 or later database, using proprietary protocols.
- **OLE DB Managed data provider.** The OLE DB .NET data provider to access data from any OLE DB-compliant data source, such as Oracle, Access and SQL Server too!
- **ODBC .NET Managed data provider.** The ODBC .NET data provider to access good old ODBC-compliant data sources.
- **The .NET Managed data provider for Oracle.** The .NET data provider for Oracle, to access Oracle RDBMS data sources, written specifically using proprietary protocols. The .NET Data Provider for Oracle supports Oracle client software version 8.1.7 or later.

Each of the four managed providers implements the core objects of: **Connection**, **Command**, **DataReader** and **DataAdapater**, in addition to a host of other objects.

SQL data provider comes with **SqlConnection**, **SqlCommand**, **SqlDataReader** and **SqlDataAdapter** objects. The **OleDb** managed provider comes with a similar set of objects, whose names start with **OleDb** instead of **Sql**. The same goes for ODBC and Oracle with the prefix changing to **Odbc** and **Oracle** respectively. These objects are created from the *System.Data.OleDb*, *System.Data.SqlClient*, the *System.Data.Odbc* and *System.Data.OracleClient* namespaces.

Now let us examine each of the four .NET data provider objects:

13.2.1.1 Connection Object

The connection object provides just that; a connection to the data store. Unlike ADO, where Connection object provided the Execute() method, to run queries, the ADO.NET connection object provides Open() and Close() methods to establish and terminate a connection to the data store. Think of the connection object as the pipeline between the consumer and the data store.

Each Managed Provider implements a connection object, which is specific to the data sources it will connect to. The *OleDbConnection* connection object is specifically written to connect to a data source that understands the OLE-DB protocols. The same holds true for the *SqlConnection*, *OdbcConnection*, and *OracleConnection* connection objects. Making a database connection in ADO.NET is straight forward; with the most difficult part of creating the connection is the Connection string. This is a semicolon-delimited string of name-value pairs, mapping closely to the ODBC, OLE-DB, and Oracle client strings respectively. The *SqlConnection* object however, has a slightly different format. Because the only data source that the *SqlConnection* object can connect to is Microsoft SQL Server, you do not need to specify a provider as part of the connection string. We will demonstrate the various strings later in this chapter.

13.2.1.2 Command object

The *Command* object is the vehicle for executing all commands relating to data access, retrieval and updates. Each Managed Provider implements a *Command* object, which is specific to the data sources it will connect to. The *OleDbCommand* *Command* object is specifically written to connect to a data source that understands the OLE-DB protocols. The same holds true for the *SqlCommand*, *OdbcCommand*, and *OracleCommand* *Command* objects.

Unlike classic ADO, *Command* objects are the only means available in ADO.NET to execute commands against a data source. The *Command* objects are particularly suited for calling stored procedures, which are the preferred method for relational data access. They are also used to return a *DataReader* type, which we will discuss shortly.

The *Command* object exposes several *Execute* methods such as *ExecuteReader* to return a *DataReader* object, *ExecuteScalar* to return a single value and *ExecuteNonQuery* to execute commands that do not return rows; as in the INSERT, Update and DELETE actions.

The *Command* object is associated with the *Connection* object through the *Command* object's *Connection* property.

13.2.1.3 DataReader object

The *DataReader* is a read-only, forward-only data object that allows you to access rows of data. It is used in lightweight query only situations, or when data is so large that it doesn't fit in a dataset. However, *DataReader* does not allow for disconnected data access, and the connection to the data store must stay open throughout the data access process, which can put a strain on the resources, it also means that you should use it sparingly, and make sure that the connection closed as soon as it is not needed anymore.

The *DataReader* is provider specific, so as in the *Connection* and *Command* Objects, we have four different variations of the *DataReader*, namely *SqlDataReader*, *OleDbDataReader*, *OdbcDataReader* and *OracleDataReader*.

The *DataReader* object is associated with the *Command* object through the *Command* object's *ExecuteReader* method.

13.2.1.4 DataAdapter object

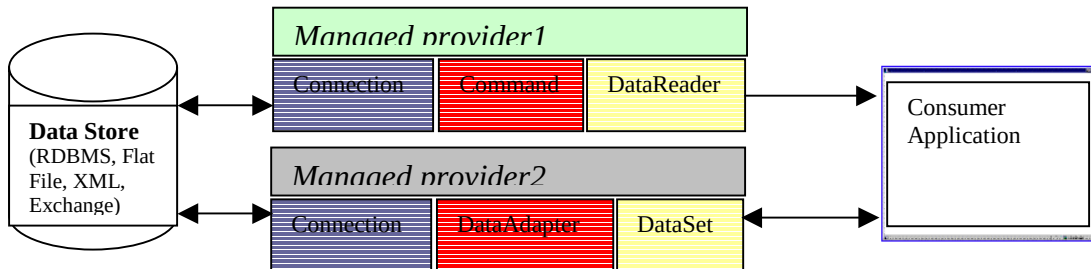
The *DataAdapter* object, acts as the main carrier between the data source and the disconnected *DataSet*. It contains a connection and four commands for retrieving the data from the data store into one *DataTable* in the *DataSet* and updating the data store with changes from the data in the *DataSet*.

You can have multiple adapters to fill the *DataSet* object with multiple *DataTables*.

The *DataAdapter* does its magic through the two methods, *Fill()* and *Update()*.

Just like the *Connection*, *Command* and *DataReader* Objects, the *DataAdapter* is provider specific, with have four different incarnations named *SqlDataAdapter*, *OleDbDataAdapter*, *OdbcDataAdapter* and *OracleDataAdapter*.

Using ADO.NET, a consumer application interacts with the data store in two ways, by way of A DataReader, which is a forward-only, read-only mechanism, or by way of a DataSet, which utilizes a DataAdapter to create an in-memory cached copy of the database or part of it, size not permitting.



In the preceding sections we discussed the connected part of the ADO.NET architecture, which represent the actual providers of data, acting a link, or bridge between the data source and the consumer, which either reads a stream of data from the DataReader or accesses an offline version of the data cached in memory by way of the disconnected component of the ADO.NET. The latter is what we discuss in the sections to come.

13.2.2 Dataset:

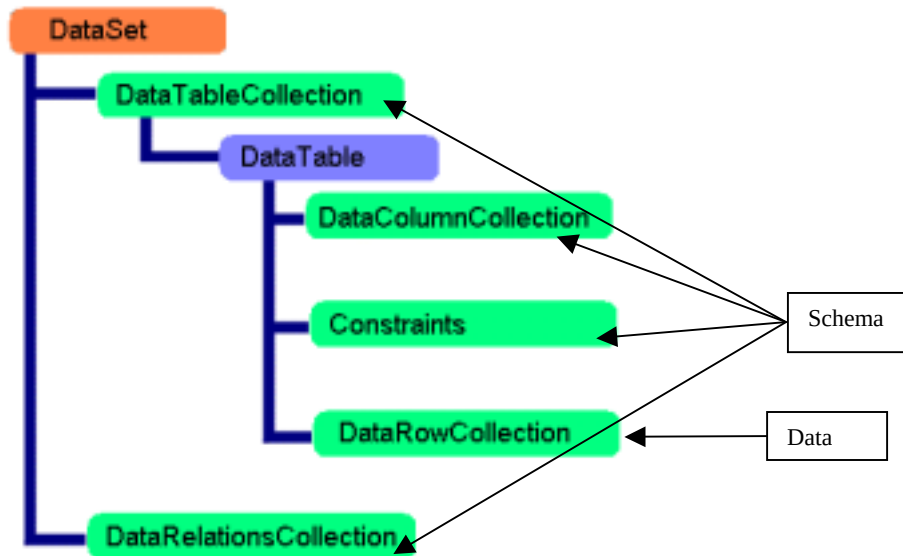
The DataSet is a very important addition to the Universal Data Access technologies of Microsoft especially geared towards the disconnected, Internet focused world of today's software.

The DataSet is a disconnected copy of the database, or part of it. It comprises of a set of tables that contain the data, the schema (table structure), and the relations between tables.

The DataSet is organized into five *Collections*, four of which are the collections of Tables, Columns and Relations; they make up the Relational schema of the database. The fifth collection, which is the Rows collection, contains the actual data.

In the .NET hierarchical model, the collections are exposed as properties of their respective object. In this case you notice that there two objects. The DataSet object that exposes DataTables and DataRelations. And The DataTable Object, itself a member of DataTableCollection exposes the rest of the five

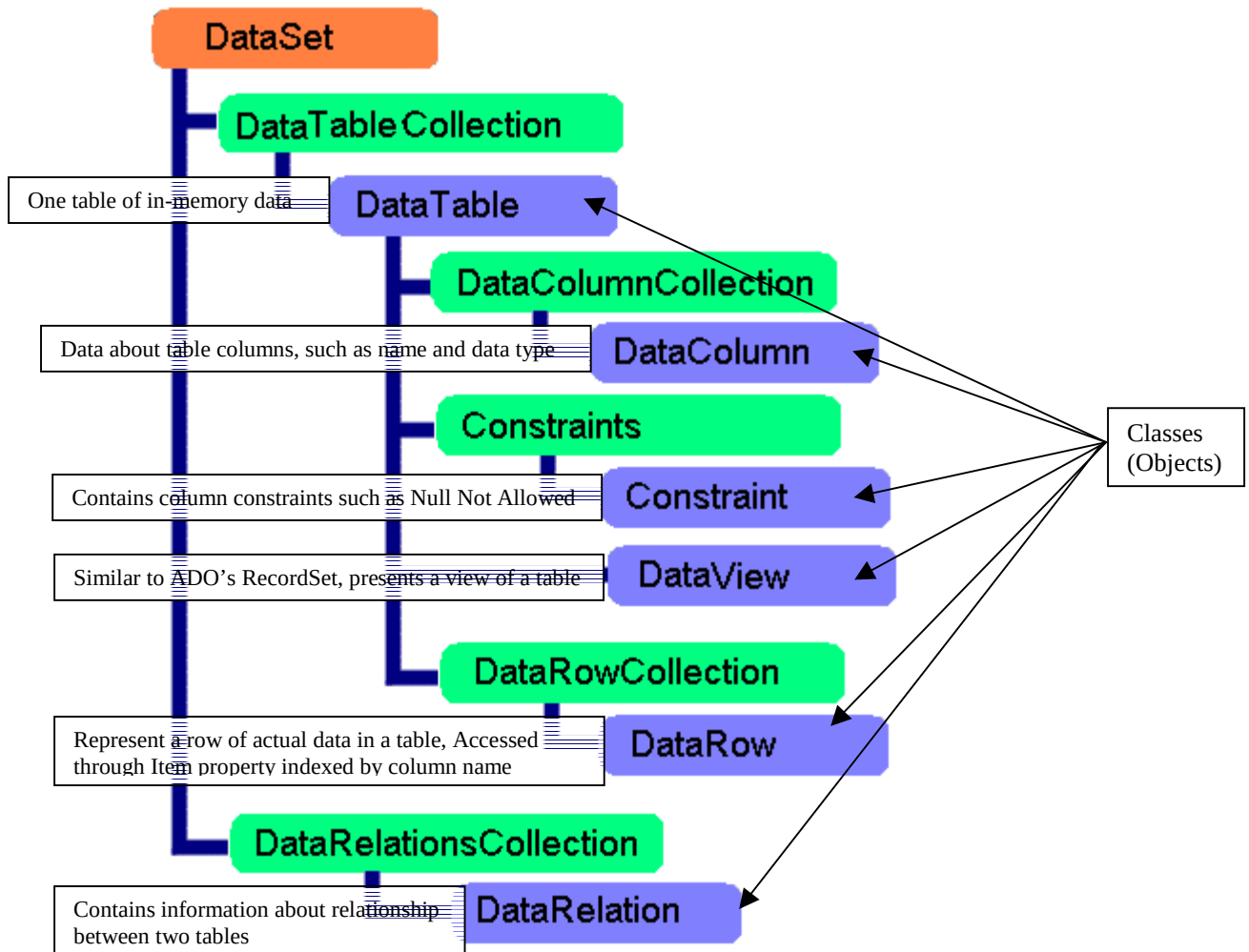
The DataSet uses one or more DataAdapter objects to get its tables filled with rows from a single or multiple data stores.



DataSet is implemented in the *System.Data* namespace. This is provider neutral, which means that it doesn't care what connected component is supplying the data, as long as it is providing it in a compatible format.

Unlike the Connected objects, which are managed provider specific, and so were contained in the respective namespaces of *System.Data.OleDb*, *System.Data.SqlClient*, the *System.Data.Odbc* and *System.Data.OracleClient*.

DataSet implements its functionality through a set of objects, properties and methods. The most important classes (objects) are depicted in the figure below, and description of each one is to follow at once.



13.2.2.1 DataTable object

Data in the RDBMS world is usually thought of in terms of tables, that contain rows and columns. So is the case in the ADO.NET's DataSet world too! The central component in a DataSet object is a DataTables collection, which is exposed as Tables property, and contains *DataTable* Objects, which are accessed using an index of ordinals or names. The *DataTable* represents one table of in-memory data. This could be a one to one representation of a physical table that is in the data source, or a totally new compilation of records from various sources.

DataTable also could be related to another *DataTable* in the *DataSet*, using the definitions in the *DataRelation* object, discussed later in this chapter.

DataTable contains a collection of *DataColumns*, which represents the definition of data and is exposed as a *Columns* property. It also contains a collection of *DataRows*, which contains the actual data, and is exposed as a *Rows* property.

Fields in the *DataTable* are represented as *DataColumn* objects, and table records are represented as *DataRow* objects. Which we discuss in the following sections

13.2.2.2 DataColumn object

The *DataColumn* is a fundamental building block of a *DataTable*, contained in *Columns* collection, it doesn't contain any data, but defines the schema, which determines what type of data may be contained in the *DataTable*.

13.2.2.3 DataRow object

The *DataAdapter* object, acts as the main carrier between the data source and the *DataSet*. Represents the actual data in a *DataTable*, contained in *Rows* collection, and conforms to the schema defined by *DataColumns*, which we covered in the preceding section.

A *DataRow* also represents the state of a *DataTable* by storing information about row versions, such as new, changed, and deleted.

When you work with data you are manipulating the *DataRowCollection* of a *DataTable*.

13.2.2.4 DataView object

The *DataView* object is similar to a view in RDBMS programming. You can also find a similarity with the old ADO recordset, if you have any ADO programming experience. You can create different views of a *DataTable*, each with different sorting and filtering conditions. You can then access these views, either by binding them to some visual data-bound control, or by simply traversing records one a time. It is important to note here that you must have a *DataTable* to create a *DataView* object.

13.2.2.5 DataRelation object

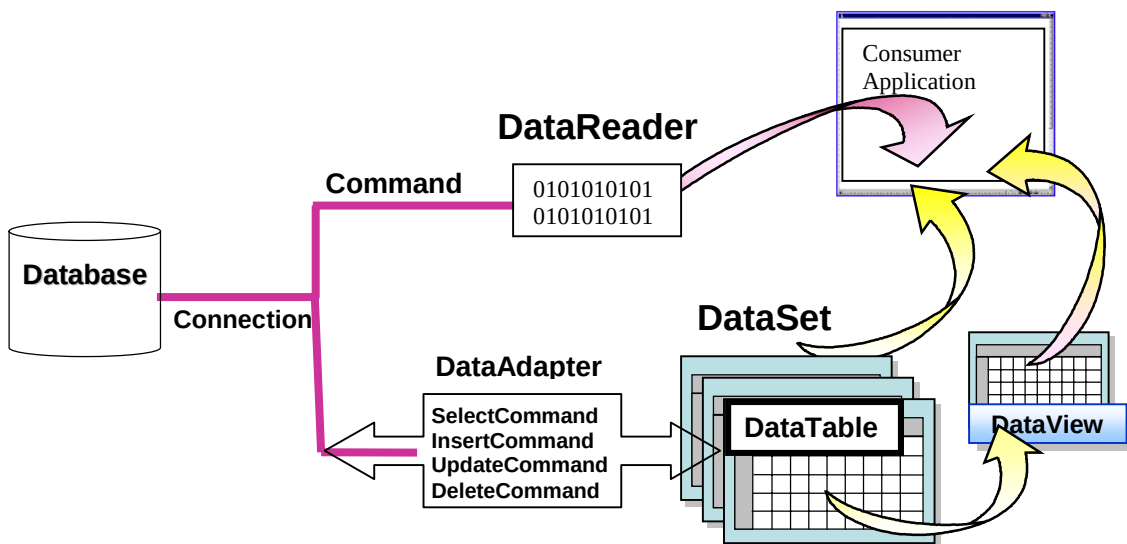
The *DataAdapter* object, acts as the main carrier between the data source and the *DataSet*. Represents a parent/child relationship between two *DataTable* objects, similar to the RDBMS primary key/Foreign Key relation. Once created, related tables can be traversed up and down the relation path. Related columns (Keys) must be of the same type and size.

13.3 Accessing Data in ADO.NET

So now that we know a great deal about the different components of the ADO.NET architecture, let us put it all in perspective, and see how we make use of it, to access real life data.

As we aforementioned, there are two distinct mechanisms to access data from our applications, namely the DataReader way, which is sort of low level, and the DataSet mechanisms, which is higher level. The figure below depicts the two approaches and highlights the main differences. We will discuss both schemes in the coming sections.

You will notice that there is one object that is common to both techniques that is the Connection object. Whether you are using DataReader or DataSet you have to establish a connection to the data source, using one of the Managed providers.



13.3.1 The Connection object

Before you can do anything with ADO.NET you have to create a connection to the data source. This is accomplished by creating a connection object in the following manner:

```
SqlConnection sCon = new SqlConnection();  
con.ConnectionString = "...";
```

Alternative you can use the *Connection* object's constructor that accepts a value for the *ConnectionString* property as a parameter. The following code is similar to the previous example.

```
SqlConnection sCon = new SqlConnection("...");
```

Then you open a connection by executing the *Open()* method. Once you are done with the connection, then you close it by invoking the *Close()* method.

The Connection string itself is highly provider dependant, as it must conform to the parameter requirements of its provider. ODBC provider for example requires parameters that match the standard ODBC string closely, and so on. Generally speaking however, it is made up of pairs of **variable=value** terminated by a semicolon (;), except for the last pair, which doesn't require a semicolon.

Let us look at the connection string for each type of the four managed providers:

13.3.1.1 The SQL Server connection Object

For *SqlConnection* you supply three items in the string, which are the data source, the database name and Authentication information, which is made up of a user ID and a password. This user should have permissions for the database objects seeking access to:

```
SqlConnection sCon = new SqlConnection("Data Source=localhost;" +  
    "Initial Catalog=Northwind;user id=userid;password=password");
```

```
sCon.Open();
```

```
...
```

```
// do something with the connection
```

```
sCon.Close();
```

If you setup SQL Server for integrated Windows authentication, or for the Mixed mode, you can use the trusted connection for user authentication. In the connection string you would supply "*Integrated Security=SSPI*" instead of the user ID/Password pairs. The authentication will be delegated to the Windows operating system which supplies the user account token for the currently account. This is the recommended approach, for security reasons, since you don't have to hard code or store passwords anywhere in your application, neither would you have to seek such information from the user, which would open a host of other issues to worry about. The trusted connection format is:

```
SqlConnection sCon = new SqlConnection("Data Source=localhost;" +  
    "Initial Catalog=Northwind;Integrated Security=SSPI");
```

```
sCon.Open();
```

```
...
```

```
// do something with the connection  
sCon.Close();
```

If you were connecting to a SQL server instance other than the default one, such as the NetSDK instance installed with the .NET framework SDK, or VSdotNET which is installed with the VS.NET samples, then you should specify the "MachineName\SQLInstanceName" for the *Data Source* instead of the *localhost*. An example would be: "CIMSERVER\NETSDK" where CIMSERVER is the computer name and NETSDK is the instance name.

13.3.1.2 The OLE DB connection Object

The OLE DB connection string is similar to that of the SQL Server connection string, except that OLE DB provider can connect a variety of data sources, such as Oracle database, Access database, or even SQL Server database, though the latter is not recommended for SQL server release 7.0 or later for performance reasons. Because of this versatility, OLE DB connection string requires one more parameter pair, which "Provider" that specifies the particular provider.

The following example demonstrates a connection string that connects to a SQL Server database through the OLE DB provider. SQL provider does not versions of SQL Server earlier than 7.0 (e.g. 6.5):

```
OleDbConnection oCon = new OleDbConnection("Data Source=localhost;" +  
    "Initial Catalog=Northwind;user id=sa;password=****;" +  
    "Provider=SQLOLEDB");  
oCon.Open();  
...  
// do something with the connection  
oCon.Close();
```

The following example demonstrates a connection string that connects to an Access database using the Microsoft Jet engine OLEDB provider.

```
OleDbConnection oCon = new OleDbConnection("Data Source=localhost;" +  
    "Initial Catalog=c:\Nortwdind.mdb;" +  
    "Provider=Microsoft.Jet.OLEDB.4.0");  
oCon.Open();  
...  
// do something with the connection  
oCon.Close();
```

13.3.1.3 The ODBC .NET connection Object

The ODBC connection string is similar to that of the SQL Server connection string, but the ODBC provider also requires a Driver parameter or a DSN (Data Source Name), created on the local machine using the ODBC32 applet. The Driver or DSN must be enclosed between a pair of curly braces.

Here is an example that uses *OdbcConnection* to connect to a SQL Server database. Notice the “*Trusted_Connection=yes*” for integrated security? It is an ODBC format.

```
OdbcConnection xCon = new OdbcConnection(
    "Driver={SQL Server};Server=localhost;" +
    "Trusted_Connection=yes;Database=northwind");
xCon.Open();
...
// do something with the connection
xCon.Close();
```

How would you like to access an Excel file using ADO.NET? Here is how:

```
OdbcConnection xCon = new OdbcConnection(
    "Driver={Microsoft Excel Driver (*.xls)};" +
    "DBQ=c:\book1.xls");
xCon.Open();
...
// do something with the connection
xCon.Close();
```

13.3.1.4 The Oracle .NET connection Object

The Microsoft Oracle provider is designed to connect to the MSDAORA OLEDB and has the following format:

```
OdbcConnection cCon = new OracleConnection("Data
    Source=MyOracleServer;Integrated Security=yes;");

cCon.Open();
...
// do something with the connection
cCon.Close();
```

13.3.1.5 Using the connection Object

The following example puts the Connection object to use. Observe the error trapping code.

```
using System;
```

```

using System.Data.SqlClient;

namespace cSDataAccess01
{
    public class ConnectionTest
    {
        public static void Main()
        {
            SqlConnection sCon = new
                SqlConnection("Data Source=CIMSERVER\\NETSDK;" +
                    "Initial Catalog=Northwind;Integrated Security=SSPI");

            try
            {
                sCon.Open();
                Console.WriteLine("Connection is " +
                    sCon.State.ToString());
                sCon.Close();
                Console.WriteLine("Connection is " +
                    sCon.State.ToString());
            }
            catch (Exception e)
            {
                Console.WriteLine(e);
            }
        }
    }
}

```

13.3.2 The Command Object

Once connected to the data source, the next step is to execute some commands against that data source. The commands could be data definition language (DDL) such as create tables and drop tables, Data Manipulation Language (DML) such retrieval of data and updates to records.

ADO.NET provides the *Command* object, which is a managed provider dependant as in *SqlCommand*, *OleDbCommand*, *OdbcCommand* and *OracleCommand*.

13.3.2.1 Creating a Command Object

The respective command object requires two parameters, a connection and a command text. Command objects can be created using several formats the most common being:

```
SqlCommand cmd = new SqlCommand(commandText, con);
```

Where *commandText* is a string that can be executed by the data store, normally a SQL query or a stored procedure call. Where *con* is a connection object or a connection string such as the ones discussed in the preceding sections about connection objects.

Here's an example of the *SqlCommand* class:

```
SqlConnection con = new SqlConnection(connectionString);
```

```
SqlCommand cmd = new SqlCommand(commandText, con);
```

You can also create the object with the empty constructor, and assign the Connection and CommandText properties of the object after its creation:

```
SqlConnection sCon = new SqlConnection("Data Source=localhost;" +  
    "Initial Catalog=Northwind;user id=userid;password=password");
```

```
SqlCommand sQlCmd = new SqlCommand();  
sQlCmd.Connection = sCon;  
sQlCmd.CommandText = "SELECT * FROM Customers";
```

13.3.2.2 Executing a Command Object

For each type, there are three ways to execute a command object, depending on the required result, which are:

```
CommandObject.ExecuteNonQuery( )
```

```
CommandObject.ExecuteScalar( )
```

```
CommandObject.ExecuteReader( )
```

13.3.2.2.1 CommandObject.ExecuteNonQuery()

To execute a command that doesn't return any results, such as DDL, Insert, Delete and Update statements.

The following example demonstrates the Insert, Update and Delete SQL statement:

```
using System;  
using System.Data.SqlClient;  
  
namespace cSDataAccess02  
{  
  
    public class CommandTest  
    {  
        public static void Main()  
        {  
            SqlConnection sCon = new  
                SqlConnection("Data Source=CIMSERVER\\NETSDK;" +  
                    "Initial Catalog=Northwind;Integrated Security=SSPI");  
  
            string SqlInsert = "INSERT INTO Customers (CustomerID,  
                CompanyName, ContactName, City, Region) " +  
                "VALUES ('CIM', 'CIM High Tech Inc.', 'Adam', 'Vancouver', 'B.C')";  
  
            string SqlUpdate = "UPDATE Customers SET Region='BC' " +  
                "WHERE Region='B.C'";  
  
            string SqlDelete = "DELETE FROM Customers WHERE  
                CustomerID='CIM'";  
  
            // Create the Command objects.  
            SqlCommand sQlCmdU = new SqlCommand(SqlUpdate, sCon);
```

```

SqlCommand sqlCmdI = new SqlCommand(SqlInsert, sCon);
SqlCommand sqlCmdD = new SqlCommand(SqlDelete, sCon);

try
{
    //Open the Connection
    sCon.Open();

    Console.WriteLine("Connection is " +
        sCon.State.ToString());

    // Execute the Insert Command object.
    int rowsInserted = sqlCmdI.ExecuteNonQuery();
    // Display the result of the operation.
    Console.WriteLine(rowsInserted.ToString() +
        " row(s) Inserted");

    sCon.Close();

    //Open the Connection
    sCon.Open();

    // Execute the Update Command object.
    int rowsUpdated = sqlCmdU.ExecuteNonQuery();
    // Display the result of the operation.
    Console.WriteLine(rowsUpdated.ToString() +
        " row(s) updated");

    sCon.Close();

    //Open the Connection
    sCon.Open();

    // Execute the Delete Command object.
    int rowsDeleted = sqlCmdD.ExecuteNonQuery();
    // Display the result of the operation.
    Console.WriteLine(rowsDeleted.ToString() +
        " row(s) deleted");

    sCon.Close();
    Console.WriteLine("Connection is " +
        sCon.State.ToString());
}
catch (Exception e)
{
    Console.WriteLine(e);
}

}
}
}

```

13.3.2.2.2 CommandObject.ExecuteScalar()

To execute a command that returns a single value, example of which is aggregate functions such as Sum, Count and Average.

The following example demonstrates the Insert SQL statement:

```
using System;
using System.Data.SqlClient;

namespace cSDataAccess02
{
    public class CommandTest2
    {
        public static void Main()
        {
            SqlConnection sCon = new
                SqlConnection("Data Source=CIMSERVER\\NETSDK;" +
                    "Initial Catalog=Northwind;Integrated Security=SSPI");

            string SqlAgregate = "SELECT COUNT(*) FROM Customers WHERE " +
                "Region = 'BC'";

            // Create the Command objects.
            SqlCommand sQLCmdA = new SqlCommand(SqlAgregate, sCon);

            try
            {
                //Open the Connection
                sCon.Open();

                // Execute the Aggreagate Query Command object.
                int iResult = (int)sQLCmdA.ExecuteScalar();

                // Display the result of the operation.
                Console.WriteLine(iResult.ToString() + " Customers in BC");

                sCon.Close();
                Console.WriteLine("Connection is " +
                    sCon.State.ToString());
            }
            catch (Exception e)
            {
                Console.WriteLine(e);
            }
        }
    }
}
```

13.3.2.2.3 CommandObject.ExecuteReader()

There is a whole section dedicated to this type of command that returns results of DataReader type, which is a streamed data, read from the source, one row at a time.

13.3.3 The DataReader Object

The *DataReader* provides a read-only; forward only (Fire hose) data access, used in lightweight query only situations, or when data is so large that it doesn't fit in a dataset.

Returning a single row at a time, keeps the connection to the data source open, which means that you have to be judicious in using it. Make sure to retrieve the data of interest and close the connection immediately afterwards.

Like the *Connection* and *Command* objects, *DataReader* is Managed provider dependant; hence we have a different *DataReader* object for each provider.

For the SQL server provider the *DataReader* is *SqlDataReader*, *OleDbDataReader* for the OLEDB provider, *OdbcDataReader* for ODBC and *OracleDataReader* for the Oracle provider. And is included in the respective provider namespace, which must be referenced in your code.

13.3.3.1 Creating a DataReader Object

The *DataReader* has the following characteristics:

1. Has no *Open()* method, however it has a *Close()* method.
2. Is not initialized through a constructor, but by assigning it the output of a *Command* object's *ExecuteReader()* method.

13.3.3.2 Using a DataReader Object

Unlike the previously discussed *Connection* and *Command* objects, *DataReader* is Using a *DataReader* object involves the following steps:

1. Create a connection

```
OleDbConnection oCon = new OleDbConnection("Data Source=localhost;" +  
    "Initial Catalog=Northwind;Integrated Security=SSPI;" +  
    "Provider=SQLOLEDB");
```
2. Create a command object with the SELECT query and the connection object.

```
OleDbCommand oleCmd = new  
    OleDbCommand("SELECT * FROM Customers", oCon);
```
3. Open the connection

```
oCon.Open();
```
4. Create the *DataReader* and assign it the live object

```
OleDbDataReader myDataReader = null;  
myDataReader = oleCmd.ExecuteReader();
```
5. Use the *DataReader*'s *Read()* method to iterate through the data rows, until you reach the end of the data, which is indicated by the *Read()* method returning *false* value. You can access the individual field values using their column names.

```
// Iterate through the records.
```

```

while(myDataReader.Read()) {
    Console.WriteLine(myDataReader["CustomerID"]);
}

```

Alternatively you can access the fields using one of the *DataReader* object's methods, which take an ordinal index, starting with 0 for first column, 1 for second and so on. The following syntax, which can be read as "Return a string value from the first column", produces the same result as the statement listed above:

```

Console.WriteLine(myDataReader.GetString(0));

```

In addition to *GetString()* method, there are several useful methods provided by the *DataReader* object; such as:

<i>GetValue(x)</i>	- Returns the data in its native format.
<i>GetDataTypeName(x)</i>	- Returns the data type of column x.
<i>GetName(x)</i>	- Returns the name of the column
<i>GetOrdinal(name)</i>	- Returns the index of the column of column <i>name</i> .

6. Close the *DataReader* and the *Connection* objects using their respective *Close()* methods.

```

// Close the DataReader and Connection objects.
myDataReader.Close();
oCon.Close();

```

Putting the above code snippets together, and adding the *System.Data.OleDb* namespace reference at the beginning, we get a working code, which follows:

```

using System;
using System.Data.OleDb;

namespace cSDataAccess03
{
    public class DataReaderTest
    {
        public static void Main()
        {
            OleDbConnection oCon = new OleDbConnection("Data
Source=CIMSERVER\\NETSDK;" +
                "Initial Catalog=Northwind;Integrated Security=SSPI;" +
                "Provider=SQLOLEDB");

            OleDbCommand oleCmd = new
                OleDbCommand("SELECT * FROM Customers " +
                    "Where Region='BC'", oCon);

            // Create the DataReader object.
            OleDbDataReader myDataReader = null;

            try
            {
                // Open the Connection object.

```

```

oCon.Open();

// Fill the DataReader
myDataReader = oleCmd.ExecuteReader( );

// Iterate through the records.
while(myDataReader.Read())
{
    // Display value of the first two columns on each line
    Console.Write(myDataReader["CustomerID"] + " - ");
    Console.Write(myDataReader.GetValue(1) + " - ");
}

// Close the DataReader
myDataReader.Close();

// Close the connection
oCon.Close();

Console.WriteLine("Connection is " +
    oCon.State.ToString());
}
catch (Exception e)
{
    Console.WriteLine(e);
}
}
}
}
}

```

13.3.4 The DataAdapter Object

The *DataAdapter* object acts as a bridge between a *DataSet* object, which is a disconnected in-memory representation of a database and a data source. The data is retrieved into the *DataSet* from a data source using the *DataAdapter's Fill()* method. Any changes made to the *DataSet* are sent back to the data source using the *Update()* method of the *DataAdapter*.

The *DataAdapter* object is also a provider dependant, so we have an adapter per Managed provider, just as in the case of the other connected provider. Internally, the *DataAdapter* object contains four command objects, and requires a valid connection to the data source, with which it would interact.

13.3.4.1 Creating a DataAdapter Object

The syntax for creating the *DataAdapter* object is very similar to creating a *Command* object, where you create an object of *DataAdapter* type, with a constructor that accepts an SQL string and a connection string, for example:

```

OleDbConnection oCon = new OleDbConnection("Data
Source=CIMSERVER\NETSDK;" +
    "Initial Catalog=Northwind;Integrated Security=SSPI;" +
    "Provider=SQLOLEDB");
string SqlSelect = "SELECT * FROM Customers Where Region='BC'"

```

```
// Create the DataAdapter object.
OleDbDataAdapter oleDa = new
    OleDbDataAdapter(SqlSelect, oCon);
```

Here is another variation of how you can directly create and assign the specific *SelectCommand* object. Remember there are four command objects.

```
OleDbConnection oCon = new OleDbConnection("Data
Source=CIMSERVER\\NETSDK;" +
    "Initial Catalog=Northwind;Integrated Security=SSPI;" +
    "Provider=SQLOLEDB");
string SqlSelect = "SELECT * FROM Customers Where Region='BC'"

OleDbCommand selectCMD = new OleDbCommand (SqlSelect, oCon);

OleDbDataAdapter MyDA = new OleDbDataAdapter();
MyDA.SelectCommand = selectCMD;
```

We mentioned that the *DataAdapter* object contains four *Command* objects, which it uses internally, which are: *SelectCommand* (as in the preceding example), *DeleteCommand*, *InsertCommand* and *UpdateCommand*.

We could easily modify the second variation of the *DataAdapter* creation example to add the three commands as follows:

```
OleDbConnection oCon = new OleDbConnection("Data
Source=CIMSERVER\\NETSDK;" +
    "Initial Catalog=Northwind;Integrated Security=SSPI;" +
    "Provider=SQLOLEDB");
string SqlSelect = "SELECT * FROM Customers Where Region='BC'"

OleDbCommand selectCMD = new OleDbCommand (SqlSelect, oCon);

OleDbDataAdapter MyDA = new OleDbDataAdapter();
MyDA.SelectCommand = selectCMD;

string SqlDelete = "DELETE FROM Customers WHERE
    CustomerID='CIM'";
OleDbCommand DeleteCMD = new OleDbCommand (SqlDelete, oCon);
MyDA.DeleteCommand = DeleteCMD;

string SqlInsert = "INSERT INTO Customers (CustomerID, CompanyName,
    ContactName, City, Region) " +
    "VALUES ('CIM', 'CIM High Tech Inc.', 'Adam', 'Vancouver', 'B.C')";

OleDbCommand InsertCMD = new OleDbCommand (SqlInsert, oCon);
MyDA.InsertCommand = InsertCMD;

string SqlUpdate = "UPDATE Customers SET Region='BC'" +
    "WHERE Region='B.C'";
OleDbCommand UpdateCMD = new OleDbCommand (SqlUpdate, oCon);
MyDA.UpdateCommand = UpdateCMD;
```

ADO.NET also provides you with a very simple mechanism to create the latter three command objects. This mechanism is the *CommandBuilder*, which handles this task for you as long as you create the select command.

The *CommandBuilder* creates the Delete, Insert and Update commands from the Select statement. For example the following statement creates the three other command objects, from the Select command.

```
OleDbConnection oCon = new OleDbConnection("Data
Source=CIMSERVER\\NETSDK;" +
    "Initial Catalog=Northwind;Integrated Security=SSPI;" +
    "Provider=SQLOLEDB");
string SqlSelect = "SELECT * FROM Customers Where Region='BC'"

// Create the DataAdapter object.
OleDbDataAdapter oleDa = new
    OleDbDataAdapter(SqlSelect, oCon);

// Create Update, Delete and Insert command using CommandBuilder
OleDbCommandBuilder oleCb = new OleDbCommandBuilder(oleDa);
```

One note of caution though, *CommandBuilder* creates the logic to operate on a single table only. So if you have more than one table and complex relations, you may want to consider alternatives, such as using command objects directly, or writing custom update SQL statements. This is beyond the scope of this book.

13.3.4.2 Using a DataAdapter Object

The *DataAdapter* object won't be of much use until the time that you understand what *DataSets* are, and learn how to use them, when you will need the *DataAdapter* object to fill them with data.

Suffice to know, at this point that you use the *DataAdapter* object in conjunction with a *DataSet* object. You retrieve data from a data source using the following format:

```
da.Fill(DataSetObject, "SomeTableName");
```

And assuming that you have used the *CommandBuilder* to create the rest of the command objects, or created the individual commands manually, you can update the data source with changes that take place in the disconnected version as follows:

```
da.Update(DataSetObject, "SomeTableName");
```

Let us see how we can put this to real use, by understanding how *DataSets* work.

13.3.5 The DataSet Object

As you could see from the sections on understanding *DataSet* objects and its classes, we could author several chapters detailing the different components provided by this complex structure. But we won't! Let us leave the reference

where it belongs, in the Microsoft .NET Framework SDK documentation and get to the practical use of *DataSets*.

To help understand creating and using *DataSets*, it's worth mentioning that there are two types of *DataSets*, Typed and Untyped. Where the typed *DataSet* has an XML-file representing its schema information (column structure), usually created at design time. The untyped *DataSet*, on the other hand, does not have an XML-persisted schema. Although it can be created at design time, untyped *DataSets* are mostly created at run time using code, either implicitly, or explicitly.

Each one has uses, advantages and disadvantages. The main advantage of a typed *DataSet*, as far as I'm concerned, is that it is well integrated with VS.NET providing you with *Intellisense* features, auto-completing names for you as you write code in the IDE, it also means that the compiler will detect any typing errors, in case you decided to ignore the auto-completion.

13.3.5.1 Creating untyped DataSets

The easiest way to create a *DataSet* is by using the VS.NET IDE's Component Designer or XML-Designer. It is not hard at all to create a *DataSet* in code either. Leaving the visual exercise for another time, let us show how you can create a *DataSet* in your code.

1. *DataSet* are provider neutral, and they belong to the `System.Data` namespace. So the first step is obviously to incorporate the aforementioned namespace.
2. We create an object of type *DataSet* instantiated with the keyword `new`.
3. A *DataSet* can have an optional name, supplied as a parameter to its constructor. If the name were omitted, the system would assign it the default name of "NewDataSet".

Here is an example

```
DataSet DsObject = new DataSet("MyDataSet");
```

At this point, I'll mention that, once you create a *DataSet* object, there are a thousand and one things you can do with it. Some of the things you can do is create structures in memory, by adding tables, columns and relations. You can create rows of data manipulate these rows (records). You can also interact with data sources, using *DataAdapter* objects in numerous ways.

We, however, will focus on populating the *DataSet* with data from a source, and making use of that data.

13.3.5.2 Using DataSets

The easiest way to create a *DataSet* is by using the VS.NET IDE's Component

Using ADO.NET's *DataSet* involves the following steps:

1. Make sure to include the proper namespaces of `System.Data` which contains the `DataSet` classes and `System.Data.OleDb` which contains the OLEDB provider, or whichever provider you choose to work with.

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

2. Setup the query string for the internal `SelectCommand` object
`string` `SqlSelect` = "SELECT * FROM Customers Where Region='BC'"
3. Setup the connection string for the `SelectCommand`'s `Connection` object as follows:

```
OleDbConnection oCon = new OleDbConnection("Data
    Source=CIMSERVER\\NETSDK;" +
    "Initial Catalog=Northwind;Integrated Security=SSPI;" +
    "Provider=SQLOLEDB");
```

4. Create a data-adapter object (for example an `OleDbDataAdapter`, `SqlDataAdapter` or `OdbcDataAdapter`) as in the following:

```
// Create the DataAdapter object.
OleDbDataAdapter oleDa = new OleDbDataAdapter(SqlSelect, oCon);
```

5. Optionally, Setup the `InsertCommand`, `UpdateCommand`, or `DeleteCommand` query strings and connections, either directly or using a *CommandBuilder* object. As in:

```
// Create Update, Delete and Insert command using CommandBuilder
OleDbCommandBuilder oleCb = new OleDbCommandBuilder(oleDa);
```

6. Create a *DataSet* object as follows:

```
DataSet DsObject = new DataSet("MyDataSet");
```

7. Invoke the *DataAdapter*'s `Fill()` method to retrieve data resulting from the `Select` query, use the *DataSet* object just created as a parameter to the method, along with an optional table name. For example:

```
oleDa.Fill(DsObject, "Customers");
```

8. At this point the *DataSet* contains Table `Customers`, which is accessible, through the `Tables` collection that is indexed either by the table name or its position, starting from 0 for the first table, and incrementing to the `(Count-1)`. We can access the `Customers` table created in the preceding point using either one of the following two formats:

```
DataTable MyTable = DsObject.Tables["Customers"];
Or
DataTable MyTable = DsObject.Tables[0];
```

9. The *DataTable* object, which is a member of the *Tables* collection, has collections of its own, such as the *Rows* collection. The *Rows* collection comprises of *DataRow*

objects, which is where the data stored. To access individual rows we create a *DataRow* object that accesses each row by its index within the *Rows* collection, starting at 0. Here is an example that accesses the first row of the Customers table:

```
DataRow MyRow = MyTable.Rows[0];
```

In real programs you would iterate through the *Rows* collection until exhausted.

```
foreach (DataRow MyRow in MyTable.Rows)
{
    // Display the number of column in this MyRow
    Console.WriteLine(MyRow.ItemArray.Length);
}
```

We could also utilize the *Count* property of the *Rows* collection, if you prefer the old *for* syntax, instead of the *foreach* construct! I prefer the preceding style.

```
for (int i=0; i<MyTable.Rows.Count; i++)
{
    // Display the number of column in this MyRow
    Console.WriteLine(MyTable.Rows[i].ItemArray.Length);
}
```

10. Whichever method we use to retrieve the a row of data, we can manipulate its contents using yet another index, this time the column name, or as always using a number starting at 0 for the first field or column of data. Assuming the we have a single row returned by the statement:

```
DataRow MyRow = MyTable.Rows[0];
```

We can access its individual data fields in either one of following ways:

```
string MyId = MyRow["CustomerID"];
string MyName = MyRow["CompanyName"];

string MyId = MyRow[0];
string MyName = MyRow[1];
```

We can also iterate through the full list of columns using a loop as follows:

```
//Iterate through the colums of the row
for (int i=0; i< MyRow.ItemArray.Length; i++)
{
    //Display each field (column) value
    Console.WriteLine(MyRow[i]);
}
```

11. Optionally, Make changes to the data in the *DataSet* and call the adapter's *Update()* method to reflect the changes in the data source. Let us say we change a single field's value in the table, we'd update the data source as follows:

```
DataRow MyRow0 = MyTable.Rows[0];
//Change the value of "Bottom-Dollar Markets";
```

```

MyRow0["CompanyName"] = "CIM High Tech Inc";
//effect the updates
oleDa.Update(DsObject, "Customers");

```

Let us put the code fragments from the preceding points to get a working program:

```

using System;
using System.Data;
using System.Data.OleDb;

namespace cSDataAccess04
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class DataSetAndDataAdapter
    {
        static void Main(string[] args)
        {

            string SqlSelect = "SELECT * FROM Customers Where Region='BC'";

            OleDbConnection oCon = new
                OleDbConnection("Data Source=CIMSERVER\\NETSDK;" +
                "Initial Catalog=Northwind;Integrated Security=SSPI;" +
                "Provider=SQLOLEDB");

            // Create the DataAdapter object.
            OleDbDataAdapter oleDa = new
                OleDbDataAdapter(SqlSelect, oCon);

            // Create Update, Delete and Insert command using CommandBuilder
            OleDbCommandBuilder OleCb = new OleDbCommandBuilder(oleDa);

            //Create a DataSet
            DataSet DsObject = new DataSet("MyDataSet");

            //Invoke the DataAdapters Fill to get data from source
            oleDa.Fill(DsObject, "Customers");

            //Create a table object containing a single table
            DataTable MyTable = DsObject.Tables["Customers"];

            //Iterate through the Rows collection in the table
            foreach (DataRow MyRow in MyTable.Rows)
            {
                //Iterate through the columns of the row
                for (int i=0; i< MyRow.ItemArray.Length; i++)
                {
                    //Display each field (column) value
                    Console.WriteLine(MyRow[i]);
                }
            }
            DataRow MyRow0 = MyTable.Rows[0];
            //Change the value of "Bottom-Dollar Markets";
            MyRow0["CompanyName"] = "CIM High Tech Inc";

```

```
//effect the updates
oleDa.Update(DsObject, "Customers");

} //end Main()
} //end class
} //end namespace
```

Uses a DataAdapter Object to connect to a data store, retrieve data, and update it the data store with changes.

User's data request is sent to a data store using DataAdapter, which connects to the data store, retrieves the data, stores it in the DataSet table(s), and disconnect from the data store.

13.3.6 What is next

Now that you know a thing or two about ADO.NET, I will point you to a few more things you can learn on your own.

Creating, deleting and working with Table objects, Adding, deleting and manipulating Row objects, Adding removing and dealing with relations are some of the subjects you may want to spend time learning and practicing with.

The SDK documentation is a great source of information and samples. So is the ADO.NET's MSDN site, where the entire documentation is available online at:

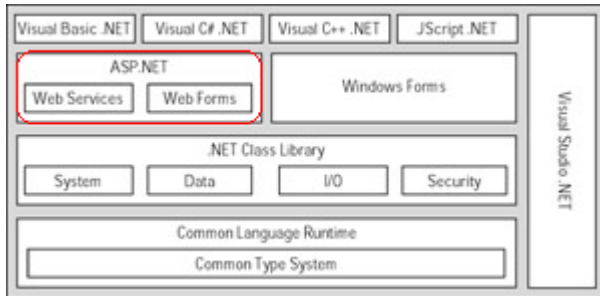
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconAccessingDataWithADONET.asp>

Data binding to windows forms' list controls is also an interesting subject you need to learn more about.

14 Developing Web Forms programs in C# (ASP.NET)

14.1 Overview

Microsoft .NET was developed with the NET at the core of its focus, as you can tell from the name. You may remember from the Framework architecture (included below for your convenience) that there are two user interface layers, Windows Forms (discussed in Chapter 12) and ASP.NET.



In this chapter, we will introduce ASP.NET, Using .NET classes to create Web Forms-based applications, using the C# language and Visual Studio .NET 2003. And of course all the concepts we discuss apply perfectly to the SDK based development.

The components provided in the .NET library for writing Web applications can be divided into two categories: Web Forms (providing classes for forms and controls) and the XML web services (which provides remote function access over the web). This chapter introduces you to Web Forms, by taking you step by step through the process of creating a simple windows application.

Web Services, also known as XML Web Services are beyond the scope of this text. However, you can learn about it through the MSDN and/or the SDK documentation.

14.2 The environment

You can develop ASP.NET applications just like Console and Windows applications that are supported by the .NET framework, using the SDK tools and a text editor, or the easier way using VS.NET. There is a third option, which is also free of charge for developers, namely WebMatrix! It is a nice WYSIWYG tools developed to help you create ASP.NET applications visually. BUT, to use VS.NET and to be able to test the application in all cases, you require access to a web server running HTTP 1.1, which means that it has the .NET framework installed. The .NET Framework version installed on the web server machine could be the minimal redistributable version of the .NET framework.

14.3 The ASP.NET's programming model

14.3.1 A bit of history

ASP.NET is an evolution from Active Server Pages (ASP) 3.0, which is a popular web development platform, and is syntax compatible with it. ASP introduced an extension to the web server, which provided the script interpreters and a set of COM objects that facilitated interaction between the server-side and the browser.

The VS.NET environment actually inherits most of its ASP.NET development components from another popular product called Microsoft InterDev, which is the ultimate ASP IDE.

14.3.2 The Infrastructure

The ASP.NET infrastructure consists of two main parts:

- A set of classes that enables the Web browser interaction with the Web server. These classes are contained in the *System.Web* namespace.
- The ASP.NET worker process or `aspnet_wp.exe`, which handles the ASP.NET tasks for the web server. This is an enhanced version of ASP, in many ways, the most clear of which is that it is an independent process.

These two components are installed when you deploy the .NET framework, VS.NET or enable the "Application server" role on Window 2003

Important note: For these components to work though, you need some form of web server to front the requests. ASP.NET is but a back-end process served by Web servers.

The web server of choice is obviously Microsoft's own IIS, which comes free of charge with all Windows releases 2000 and above, including the XP family.

14.3.3 The Components

The ASP.NET programs are modeled around the web-based application development, which comprises of a web server hosting one or more files, accessed with a web browser, via a Uniform Resource Locator (URL).

From the programmer's point of view, the web application normally comprises of two components:

1. Front-end or user interface page, which is usually created using HTML tags, with or without client side scripts.
2. Server side logic providing back-end functionality such as access to data sources, performing complex proprietary calculations and many other tasks, usually written in some programming language. Typically ASP, Perl, Python and other Common Gateway Interface (CGI) or ISAPI (Internet Server Application Programming Interface) components are used to develop this code.

In ASP 3.0, or classic ASP as it is known now, both of the passive HTML tags and the active server-side were contained in a single file with an .ASP extension, also known as page, hence the **Active** in the Active Server Pages

name. This is still valid in ASP.NET, although not highly recommended, and the file extension is .ASPX.

14.3.4 The Process

So how does an ASP.NET work?

1. A page is requested from the web server, by a browser, using a URL
2. The web server analyzes the request, and based on the .ASPX extension it determines that the request is for an ASP.NET file, and invokes the ASP.NET worker process (aspnet_wp.exe)
3. ASP.NET checks the local cache to see if this file has been requested before, if yes, then it jumps to step 4, if not then it goes through the following:
 - a. It parses the content, translating the tags it recognized into methods and fields of a class and inserting the markup tags as they are.
 - b. Once created a page class, ASP.NET invokes an appropriate compiler for the code in the file, to produce an Intermediate Language (IL) code
 - c. At this point an assembly containing a class derived from Page (containing our code, from the .ASPX file) is created, and put in the cache for further use
4. The assembly hosting the code of the requested page is loaded into the CLR, which will execute the managed code. You remember this part, don't you?
5. The code of the object carries out what you designed it to do, and produces the output that it is supposed to, hopefully in HTML.
6. The CLR passes the output back to ASP.NET, which passes it in turn to the web server.
7. The web server sends the output to the browser.

14.4 The ASPX page

The ASPX files are text files, with an ASPX extension, but you should understand some inherent features of the pages.

1. During compilation, each ASPX page is converted into a class. That derives most of its functionality from the .NET framework base class `System.Web.UI.Page`.
2. The Page class itself derives from `System.Web.UI.TemplateControl`, `System.Web.UI.Control`, and `System.Object`.

The above implies that you inherit all methods and properties from the parent classes. So you should learn as much as you can about those classes to be able to make the most out of ASP.NET.

14.4.1 A typical ASPX page

Let us take a look at the contents of a typical ASPX page

```

    Diectives
    <%@ Page language="c#" %>

    <HTML>

    Comments {
        <!--
            Here begins the HTML document body
            We could document the page using multi-line
            Comments such as this one.
            Also used while debugging to temporarily disable code
        -->
    }

    <body>

    HTML tag {
        <h1>
            Welcome to ASP.NET    ← Inline text
        </h1>
    }

    <!--This will be compiled to a method of the class-->

    Code Declaration block {
        <script runat="server" language="C#">
            public void vTest()
            {
                // This is a C# comment
                Response.Output.Write(this.GetType()+ "<BR>");
                Response.Write(this.GetType().BaseType.BaseType + "<BR>");
            }
            public int iTest()
            {
                return 13;
            }
        </script>
    }
    C# Code

    <form id="Form1" method="post" runat="server">

        <%=iTest()%>    ← Render code (Evaluation)
        <br>             ← HTML tag
        <%=vTest();%>    ← Render code (Method call)

    </form>

    <!-- inline code that will be part of __Render_contro1 method -->

    Render block {
        <%
            Response.Write("To be written to __Render_contro1(<BR>");
            Response.Write("Don't use this type of code often");
        %>
    }

    </body>

    </HTML>

```

One could argue that there are some more things to be added, but let us just accept the above template for now and study its parts:

14.4.1.1 Directives

Directives are keywords that start with a @ and are placed set at the page top. They are similar to the #include pre-processor tags, and are generally used for things such as specifying the code language, including external files, setting up page properties and many more.

14.4.1.2 Comments

Scattered all over the code, comments are a very important element of the ASPX page. Use them all the time.

14.4.1.3 HTML tags

These are standard HTML markup tags to be rendered by the browser.

14.4.1.4 Code Declaration Blocks

Lines of code enclosed in <script> tags, and contain the **runat=server** attribute, the code block itself consists of the definition of member variables and methods. Everything in Declaration blocks abides by the rules of the programming language that you have chosen, in our case C#, which means all code must be enclosed within a method.

14.4.1.5 HTML Server controls

Adding the attribute **runat=server** to any control turns it into a server control. Recognize the <form> control? It is the same old HTML tag except it is now a server control that is executed on the server; complete with an identifier "Form1" to help address it in our code.

There is another set of controls known simply as Server controls, or as ASP Server controls that we will discuss in the next section.

14.4.1.6 Render Blocks

Render blocks come in two flavours, one to invoke methods, enclosed in <% %> symbols, this requires a semicolon terminator. And the other to evaluate an expression, enclosed in <%= %> with the expression following the =, and no semicolon is required.

The Render block is a convenient way to invoke methods and evaluate expressions, but beware that the generated code will be placed in an automatically created method called __Render_control1 giving you little control if any at all.

14.4.1.7 Render Blocks

Render blocks come in two flavours, one to invoke methods, enclosed in <% %>

14.4.2 The code behind

You must have noticed that in the preceding example we had HTML tags, server side code and Response.Write() output all in the same page, which perfectly legal, but just about! I mean to say we don't recommend this style of coding, as it the legacy style of classic ASP. In order not to be so outdated, we will use code behind, which simply means that we will remove the C# code and put it in a separate file, which we name whatever we like,

and reference in the page. File naming in VS.NET is done automatically deriving from the ASPX file name, by appending the language extension to the end of its name, .cs in this case, for C#.

Let us start with the referencing part, which we accomplish in either one of two ways:

1. The manual way by adding a "src=filename" tag to the directive line as follows:

```
<%@ Page language="c#" src="WebForm1.aspx.cs"%>
```

2. The VS.NET way, which will add a **Codebehind** attribute that is used for design-time only, coupled with the **Inherits** attribute which is used to reference the compiled assembly, and inherit from the page class. The following is generated when using VS.NET for our example:

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
    AutoEventWireup="false" Inherits="CSWebApp01.WebForm1"%>
```

Then we create a WebForm1.aspx.cs file using the editor of our choice, and save it in the same folder as the ASPX file. The file should contain the following:

1. Reference to the appropriate namespaces, as follows:

```
using System;
using System.Web;
using System.Web.UI;
```

The System.Web namespace provides us with all the ASP.NET page features, such as `HttpApplication`, `HttpSession`, `HttpServer`, `HttpRequest` and `HttpResponse` objects. You should know these objects if you have some ASP experience, as they correspond to the `Application`, `Session`, `Server`, `Request` and `Response` intrinsic objects of ASP 3.0.

At this stage we are mainly interested in the `HttpRequest` and `HttpResponse` objects, which enable the interaction between a web browser and the web server, sort of like the IO stream of the console application.

2. Create a namespace, using our project name. This should look familiar!

```
namespace MyFirstWebApp
{
```

3. Create a class deriving from the `System.Web.UI.Page` base class that in turn inherits from the control class, which defines several events, such as `Load` and `Unload`.

```
public class WebForm1 : System.Web.UI.Page
{
```

4. Move the code from the Declaration block in the ASPX page.

```
public void vTest()
{
```

```

        Response.Output.Write(this.GetType().ToString() + "<BR>");
        Response.Write(this.GetType().BaseType.BaseType + "<BR>");
    }
    public int iTest()
    {
        return 13;
    }
}

```

5. No Main() method is required, just in case you may be wondering

Finally we clean up our ASPX page from the Code declaration block, and remain with the following:

```

<HTML>
<body>
<form id="Form1" method="post" runat="server">
    <%=iTest()%>
    <br>
    <%=vTest();%>
</form>
<%
    Response.Write("This code will be part of the Method <BR>");
    Response.Write("Don't use this type of code often");
%>
</body>
</HTML>

```

Which is what a user interface UI code should look like. You can obviously see the power of the code behind notion. You could for example be working on a sophisticated algorithm placing the code in one of the methods, above, while your colleague is working on the presentation layer. The code could obviously invoke other code residing on the same machine or anywhere on the network. Are you thinking data base access using ADO.NET?

14.4.2.1 Event handling

A major use of code behind is responding to events, which in syntax looks pretty much the same as the windows forms events code, but the internal architecture is different and so is the number and sequence of events.

An ASP.NET page class raises four events in a strict sequence that can be subscribed to, by using delegates. These events are Init, Load, PreRender and Unload, respectively. Load or Page_Load event the most useful of the four events, and the most widely used.

We can wire event handlers to in two ways, the easy way and the efficient way:

1. The easy way that relies on the system's AutoEventWireup feature, is to declare methods named: Page_Init, Page_Load, Page_PreRender or Page_Unload for Init, Load, PreRender and Unload events respective. The system takes care of wiring the proper event at runtime.

2. The more efficient way, requires a more intimate knowledge of both the Page class and the event coding techniques. Each of the four events are defined as events you can subscribe to, and as virtual functions, which you can override. VS.NET uses a mix of both methods. It first overrides the OnInit() method, and calls InitializeComponent(), where the Load event is subscribed to directly. Here is the code:

```
override protected void OnInit(EventArgs e)
{
    InitializeComponent();
}

private void InitializeComponent()
{
    this.Load += new System.EventHandler(this.Page_Load);
}
```

Either method is just fine, but you need understand which one is that you are adopting, and indicate that in the page directives, as VS.NET does automatically, by setting the AutoEventWireup attribute to false as in:

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
    AutoEventWireup="false" Inherits="cSWebApp01.WebForm1"%>
```

Now let us add an event handler to the above code to display a string when in response to Page Load event.

1. The ASPX page's directive line should set the AutoEventWireup attribute to "true" as follows:

```
<%@Page language="c#" Src="WebForm1.aspx.cs" AutoEventWireup="true"%>
```

2. The code behind needs to change as follows:

```
namespace MyFirstWebApp
{
    public class WebForm1 : System.Web.UI.Page
    {
        void Page_Load(Object sender, EventArgs e)
        {
            Response.Output.Write("Page Load event was fired.<BR>");
        }

        public void vTest()
        {
            Response.Output.Write(this.GetType().ToString() + "<BR>");
            Response.Write(this.GetType().BaseType.BaseType + "<BR>");
        }
        public int iTest()
        {
            return 13;
        }
    }
}
```

In the case of VS.NET it would have done the following automatically:

3. The ASPX page's directive line should set the AutoEventWireup attribute to "false" as follows:

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
    AutoEventWireup="false" Inherits="CSWebApp01.WebForm1"%>
```

4. The code behind needs to change as follows:

```
namespace MyFirstWebApp
{
    public class WebForm1 : System.Web.UI.Page
    {
        void Page_Load(Object sender, EventArgs e)
        {
            Response.Output.Write("Page Load event was fired.<BR>");
        }

        public void vTest()
        {
            Response.Output.Write(this.GetType().ToString() + "<BR>");
            Response.Write(this.GetType().BaseType.BaseType + "<BR>");
        }
        public int iTest()
        {
            return 13;
        }
        override protected void OnInit(EventArgs e)
        {
            InitializeComponent();
            base.OnInit(e);
        }
        private void InitializeComponent()
        {
            this.Load += new System.EventHandler(this.Page_Load);
        }
    }
}
```

You have seen how we handle events that are raised by the page class on the server. ASP.NET provides a programming model that enables you to handle events raised on the client, such as in response to clicking a button, much the same way, by wiring an event handler to each object.

Let us take a look at the Web Forms and controls:

14.5 Web Forms and Server Controls

In ASP.NET the form is simply your HTML document that is rendered by the web browser, on the client machine. What is really powerful though is the way you program these forms.

ASP.NET provides a model very similar to the Windows Forms programming model, where you as a programmer lay out controls on a form, and write code to handle events that are raised in response to user interaction with these controls.

So instead of you worrying about writing the HTML stream that went through the Response object, as we did in ASP 3.0 or other CGI environments, you would send objects that know how to render themselves through the Response pipe, with ASP.NET presenting you with the plumbing.

14.5.1 Server Controls

ASP.NET provides us with controls that are sent to the browser as HTML. There two sets of controls, namely HTML controls, and ASP controls. The first being closely mapped to the standard HTML controls, where the second is a more rich representation of the same.

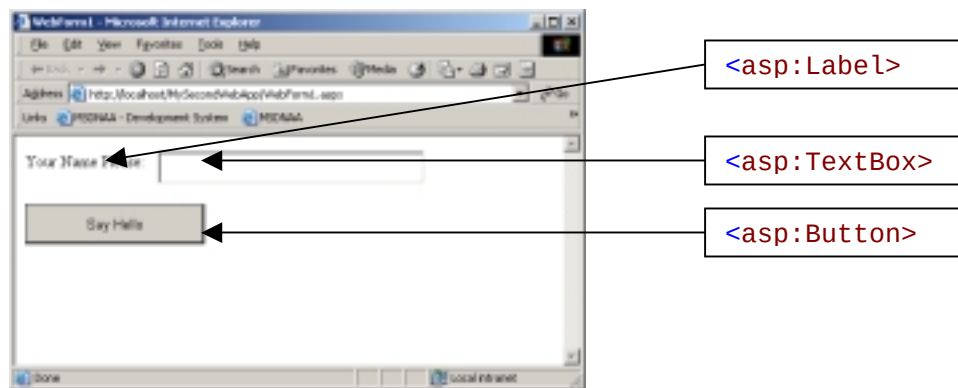
The main difference is in how much functionality each one has, so a simple text box or label control is similar in both, but if you take something like a Calendar control, which is implemented in the ASP.NET Server controls, you will know what I mean.

Enough talking already and let us look at an example:

```
<%@ Page language="c#" %>
<HTML>
<body>
  <form id="Form1" method="post" runat="server">
    <P>
      <asp:Label id="Label1" runat="server" Width="128px"
        Height="32px">Your Name Please:</asp:Label>

      <asp:TextBox id="TextBox1" runat="server" Width="264px"
        Height="32px" />
    </P>
    <P>
      <asp:Button id="Button1" runat="server" Width="178px"
        Height="40px" Text="Say Hello"></asp:Button>
    </P>
  </form>
</body>
</HTML>
```

The above code produces the following when invoked from a browser:



Every server control must be enclosed in a <form> </Form> block, since we are developing Web Forms!

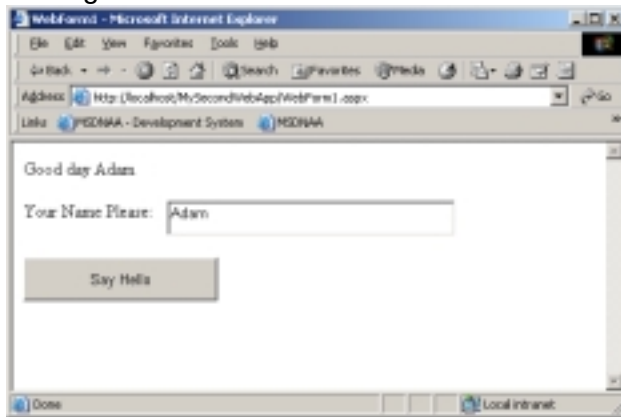
Then we lay out the controls inside the form, using either the asp syntax, or the html syntax. The above code uses the asp syntax.

The control declaration comprises of `<asp:ControlType>` followed by a set of attributes, that are more or less similar to standard HTML tag attributes. Then we have a terminator of `</asp:ControlType>` we could also use a short form of termination by simply adding a forward slash (/) at the end of the attribute string of the declaration, right before the closing `/>`, the `<asp:TextBox ... />` employs this format.

Now let us make this a little more interesting, by wiring the button's click event to a handler that we create, as follows:

```
this.Button1.Click += new System.EventHandler(this.Button1_Click);  
...  
private void Button1_Click(object sender, System.EventArgs e)  
{  
    Response.Write("Good day " + TextBox1.Text);  
}
```

The result of typing “Adam” in the text box and pushing the Button looks like the figure below.



Now let us analyze the sequence of activities that take place in the process of executing the preceding code:

1. The browser send an HTTP request with the URL:
<http://localhost/MyFirstWebApp/WebForm1.aspx>
2. The server sends the WebForm1.aspx is compiled and a page class instance (object) of the WebForm1 class is created and executed by the CLR as prescribed in section 14.3.4 “the process”.
3. The output, which is an HTML representation of the results is sent back to the browser
4. The browser renders the HTML tags to their visual representation and the user can interact with it. At this point, the connection is severed with the server
5. When the user types something and clicks the button, the form is “Posted” back to the server; with a little more than what the user has entered.

6. Sent with the user data is internal set of fields comprising of a “VIEWSTATE” object that holds all state information, which maintains the values of server controls throughout the session. Also sent with the post is event information included in `EVENTTARGET` and `EVENTARGUMENT`. All of these items are embedded as a set of hidden fields in the form.
7. A page-derived class is created at this point, and a sequence of events is raised as follows:
 - a. Init event is raised
 - b. State of server side controls is restored using values in the `VIEWSTATE` object.
 - c. Page Load event is fired
 - d. Server control events such as Button Click event are raised.
 - e. The PreRender event is raised, followed by Render process
 - f. Rendered information is sent via an `HTTPResponse`
 - g. Page Unload event is fired signaling end of life for the page class
8. The page class is discarded, and the browser receives the new output.

The code that is created by ASP.NET after parsing our ASPX page and will be executed by the CLR looks like this (Code edited for clarity):

```
//-----
// <autogenerated>
//-----

namespace ASP {
    using System;
    using System.Collections;
    using System.Collections.Specialized;
    using System.Configuration;
    using System.Text;
    using System.Text.RegularExpressions;
    using System.Web;
    using System.Web.Caching;
    using System.Web.SessionState;
    using System.Web.Security;
    using System.Web.UI;
    using System.Web.UI.WebControls;
    using System.Web.UI.HtmlControls;
    using ASP;

    [System.Runtime.CompilerServices.CompilerGlobalScopeAttribute()]
    public class WebForm1_aspx : WebApplicationcx.WebForm1,
        System.Web.SessionState.IRequiresSessionState {

        // The main page class
        public WebForm1_aspx() {
            System.Collections.ArrayList dependencies;
            if ((ASP.WebForm1_aspx.__initialized == false)) {
                ASP.WebForm1_aspx.__stringResource =
                    System.Web.UI.TemplateControl.ReadStringResource(typeof(ASP.WebForm1_aspx));
                dependencies = new System.Collections.ArrayList();
                dependencies.Add("c:\\inetpub\\wwwroot\\WebApplicationcx\\WebForm1.aspx");
                ASP.WebForm1_aspx.__fileDependencies = dependencies;
                ASP.WebForm1_aspx.__initialized = true;
            }
            this.Server.ScriptTimeout = 30000000;
        }

        // Directives attributes
        protected override bool SupportAutoEvents {
            get {
```

```

        return false;
    }
}

protected ASP.Global_asax ApplicationInstance {
    get {
        return ((ASP.Global_asax)(this.Context.ApplicationInstance));
    }
}

public override string TemplateSourceDirectory {
    get {
        return "/WebApplicationcx";
    }
}

// The Label control
private System.Web.UI.Control __BuildControlLabel1() {
    System.Web.UI.WebControls.Label __ctrl;

    #line 13 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __ctrl = new System.Web.UI.WebControls.Label();

    this.Label1 = __ctrl;

    #line 13 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __ctrl.ID = "Label1";

    System.Web.UI.IParserAccessor __parser = ((System.Web.UI.IParserAccessor)(__ctrl));

    #line 13 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __parser.AddParsedSubObject(new System.Web.UI.LiteralControl("You Name Please"));

    return __ctrl;
}

// TextBox control
private System.Web.UI.Control __BuildControlTextBox1() {
    System.Web.UI.WebControls.TextBox __ctrl;

    #line 13 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __ctrl = new System.Web.UI.WebControls.TextBox();

    this.TextBox1 = __ctrl;

    #line 13 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __ctrl.ID = "TextBox1";

    return __ctrl;
}

//Button Control
private System.Web.UI.Control __BuildControlButton1() {
    System.Web.UI.WebControls.Button __ctrl;

    #line 14 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __ctrl = new System.Web.UI.WebControls.Button();
    this.Button1 = __ctrl;

    #line 14 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __ctrl.ID = "Button1";

    #line 14 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __ctrl.Width = System.Web.UI.WebControls.Unit.Parse("144px",
        System.Globalization.CultureInfo.InvariantCulture);
}

```

```

        #line 14 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
        __ctrl.Text = "Say Hello";

        return __ctrl;
    }

// The Form control
private System.Web.UI.Control __BuildControlfrom1() {
    System.Web.UI.HtmlControls.HtmlForm __ctrl;

    #line 12 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __ctrl = new System.Web.UI.HtmlControls.HtmlForm();

    this.from1 = __ctrl;

    #line 12 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __ctrl.ID = "from1";

    #line 12 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __parser.AddParsedSubObject(new System.Web.UI.LiteralControl("\r\n\t\t\t<P>"));

    #line 12 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    this.__BuildControlLabel1();

    #line 12 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __parser.AddParsedSubObject(this.Label1);

    #line 12 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    this.__BuildControlTextBox1();

    #line 12 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __parser.AddParsedSubObject(this.TextBox1);

    #line 12 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __parser.AddParsedSubObject(new System.Web.UI.LiteralControl("</P>\r\n\t\t\t\t"));

    #line 12 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __parser.AddParsedSubObject(this.Button1);

    #line 1 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    this.__BuildControlfrom1();

    #line 1 "c:\inetpub\wwwroot\WebApplicationcx\WebForm1.aspx"
    __parser.AddParsedSubObject(this.from1);

}

protected override void FrameworkInitialize() {
    SetStringResourcePointer(ASP.WebForm1_aspx.__stringResource, 396);
    this.__BuildControlTree(this);
    this.FileDependencies = ASP.WebForm1_aspx.__fileDependencies;
    this.EnableViewStateMac = true;
    this.Request.ValidateInput();
}
}
}

```

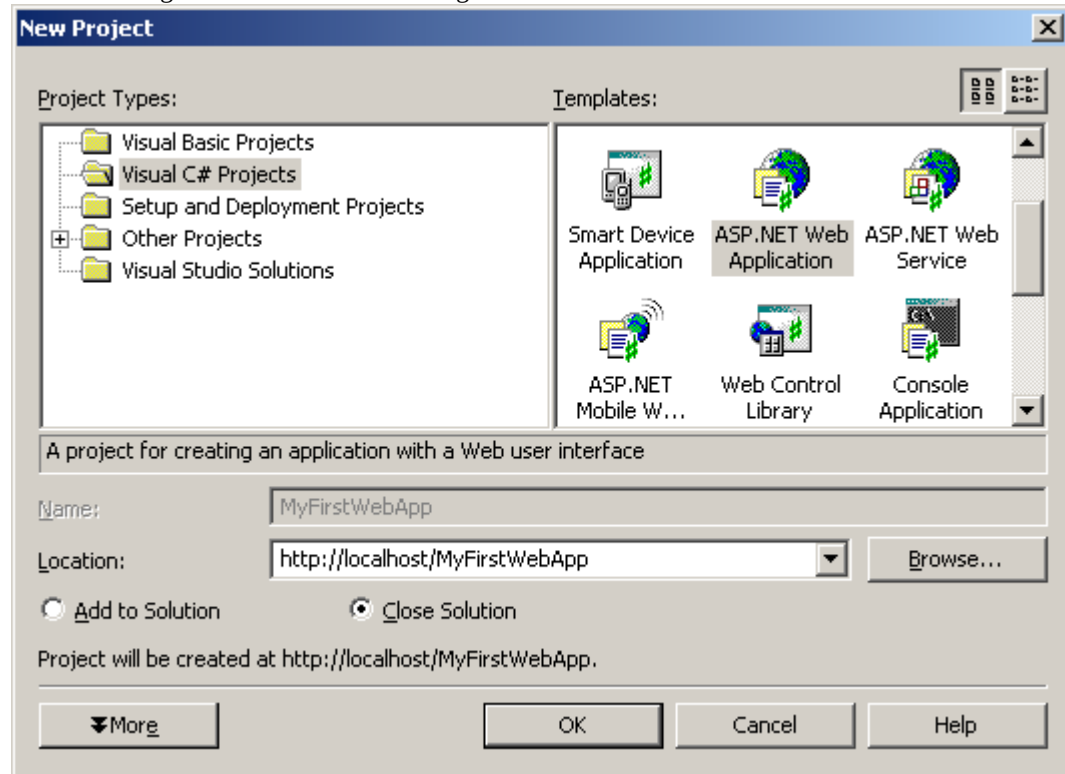
Okay, so now that you know what is in ASPX files, and understood how they are served, it is up to you to create them. All you need is some HTML knowledge, basic ASP.NET understanding, C#, an editor and a bit of luck.

14.6 Using VS.NET to develop Web Forms

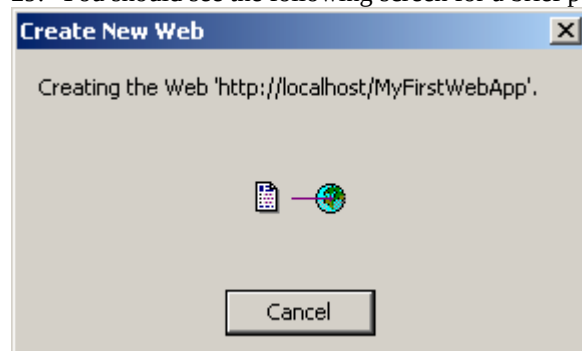
14.6.1 Creating a project

To create a new project,

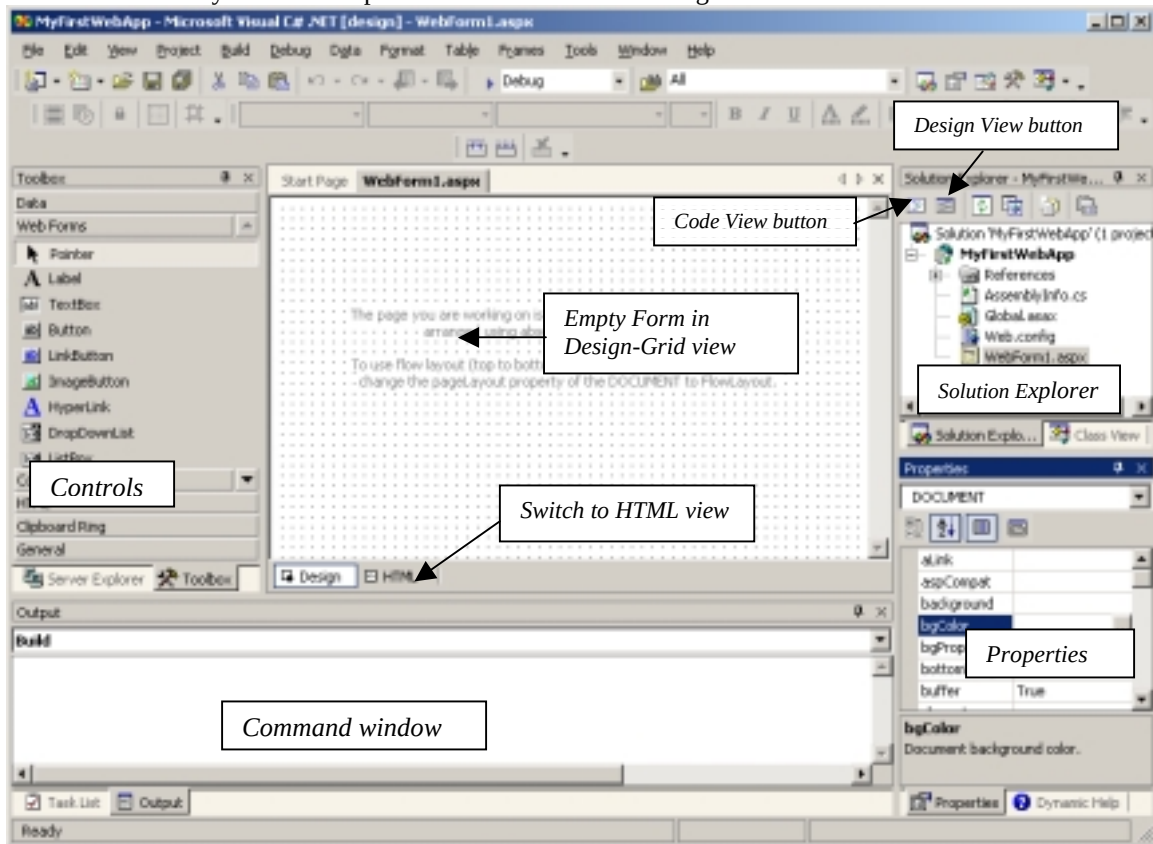
21. Start VS.NET.
22. Make sure you have the “Start Page” open (use menu item: **Help → Show Start Page**)
23. Click “New Project” hyperlink at the bottom of the screen
24. You will get a screen similar to the figure below



25. Click on “Visual C# Projects” on the upper-left explorer-style folder. This will ensure that we select the right language.
26. Select “ASP.NET Web Application” In the right-side panel. This will ensure we are creating a ASP.NET Web Application.
27. In the “Location:” text box type “myFirstWebApp”. The <http://localhost> represents your local web server, if you intend on using a remote server then change it here.
28. Press OK.
29. You should see the following screen for a brief period, depending on the machine resources



30. Then you will end up with a screen similar to the figure below:



14.6.2 Adding a Web Form

When you create a new ASP.NET Web Application project, VS.NET automatically creates a Form for you. You can also add a new form to an existing project, using menus and tool bars.

The form page will be initially displayed in a GridLayout, which helps position controls accurately around the screen, but this will generate server side attributes that will specify the x and y coordinates of the controls.

The alternative layout is FlowLayout which is like a blank page with little layout help; hence you have to use white spaces, carriage return and tables to position the controls where you want them.

You can alternate between the two layout modes using the "PageLayout" property of the form.

14.6.3 Working with page properties

You can modify the properties of a form once created, using the visual designer, by accessing the properties form. Press F4, select View→Properties window from the main menu, or simply press the  icon from the toolbars.

The properties window located at the lower right hand side of the VS.NET IDE is similar to the familiar VB or Access property sheet. Comprising of a name-value pairs, you select the property that you want to modify and change its value in the text field next to its name.

The ASP.NET designer property window shows a drop down list at the top, the default selection in this list “DOCUMENT” which is the main form, or the equivalent of the HTML document. The second item, accessed by clicking downward-pointing arrow, is the class name that corresponds to the name of the class in the code behind.

Each one of the two elements has totally different set of properties, the first being user interface related, such as colour, font, encoding and so on. The events icon is inactive, because the Document doesn’t handle events on the server.

The second however has, being the class that derives from `System.Web.UI.Page` has no properties, but the event icon will be active, which if you click it, reveals the list of events that are fired by the page. You will also notice that there are five more events listed, which we didn’t discuss previously. These events belong to the

14.6.3.1 Modifying the Title property of a page

Say we want to give our page a more meaningful name.

1. Select the form’s design view and click anywhere on the form
2. Press F4 and get to the properties window, make sure the DOCUMENT is selected at the top line
3. Select the **Title** property, erase the current “WebForm1”, and type the new name, something original such as “My first ASPX page”
4. This name will be displayed at the web explorer window’s title bar.

14.6.3.2 Modifying the Page Layout

As mentioned in section 14.6.2 the page layout can be switched back and forth between GridLayout and FlowLayout.

1. Select the form’s design view and click anywhere on the form
2. Press F4 and get to the properties window, make sure the DOCUMENT is selected at the top line
3. Select the **PageLayout** property, double click the value field to alternate between the two values, or use the drop down arrow to make your pick
4. The change will take effect immediately. You can see it visually.

14.6.4 The code behind the form

Now let us examine the code that was generated for us by the VS.NET To access the corresponding code, right click the form’s name in the solution explorer and select “View Code”. You can also switch between the designer

and code views using the corresponding buttons at the top of the solution explorer, as indicated on the figure above.

The code generated by VS.NET, which is listed below, features a template that you will see over and over when using the IDE to develop Web Forms applications, so let us take a closer look at it:

9. We have the usual referencing of namespaces by virtue of the `using` keyword. Notice that there are some namespaces, added by VS.NET, which we you may never use, but they are there. Examples are `Collections`, `ComponentModel` and `Data`.
10. Create the project namespace, which is by default the same as the project name that we typed in the project creation dialog.
11. A class `WebForm1` which inherits (:) from the `System.Web.UI.Page` base class is created for us. The `Form1` name is the name of the visual form, and the source file `Form1.cs`. But they don't have to be the same.
12. An event handler method `Page_Load()` that will be executed in response to the event of page Load.
13. Then we have the implementation of the overridden `OnInit()` method is invoked internally when the `Init` event is fired by the page class. This method invokes the `InitializeComponent()` method. You override a method that has been declared in a base class, by redefining its action, while maintaining its signature, and us the keyword `override`.
14. Then we have a collapsible region, where the `InitializeComponent()` method is implemented. The contents of this method will keep changing, as you work on the designer. We have the code that wired the Load event to our `Page_Load` handler.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.UI;
using System.Web.SessionState;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace MyFirstWebApp
{
    /// <summary>
    /// Summary description for WebForm1.
    /// </summary>
    public class WebForm1 : System.Web.UI.Page
    {
        private void Page_Load(object sender, System.EventArgs e)
        {
            // Put user code to initialize the page here
        }
    }
}
```

```

        #region Web Form Designer generated code
        override protected void OnInit(EventArgs e)
        {
            //
            // CODEGEN: This call is required by the ASP.NET Web Form
Designer.
            //
            InitializeComponent();
            base.OnInit(e);
        }

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.Load += new System.EventHandler(this.Page_Load);
        }
    }
}
}

```

14.6.5 Running an application

Running a web based application involves viewing the page within an internet browser such as internet explorer. You have two options of doing this:

1. Manual deployment
 - a. You deploy the ASPX files along with any accompanying files such as code behind files and images to a folder that accessible through a web server.
 - b. Define this folder as a virtual directory and asp application using the web server's control panel
 - c. Using a web browser request the URL, as in:
<http://localhost/App/page.aspx>
2. Using VS.NET
 - a. If the purpose is testing while developing then you simply right click the document (form) and select "View in Browser", or just click F5, which will debug and run an external web browser. The difference here is the first option opens an internal browser window in a tab, and will not allow you to debug.
 - b. If this for final deployment then you have two options:
 - i. The xcopy method, which involves an approach similar to (1), the manual deployment
 - ii. Create a deployment package using VS.NET deployment project. This is out of scope for this text, however you can

learn how to create a deployment project from the VS.NET documentation or using the MSDN online help.

To do this you can deploy the ASPX files along with any accompanying files such as code behind files and images.

You can run

14.6.6 Adding an event handler to a Form

To add an event handler for events other than the default, go through the following steps.

7. Switch to designer view by clicking () in the Solution explorer window
8. Click the form to make it active
9. Go to the properties window and select the class name instead of the "DOCUMENT" from the drop down list at the top of the window.
10. Click the Events button () , located at the top of the Properties window
11. Just as with the properties, click the event that you want to create an event handler for.
12. In the text box to the right of the event name, type the name of the handler and press ENTER.

The name could be any valid C# method name, but using the naming convention adopted by VS.NET is a good practice. If you were handling the PreRender event, for instance, then you may want to call the handler `MyForm01_ PreRender`. If you accept this naming convention, then you actually don't have to type the name. Just double click the text box and VS.NET will type it for you.

13. The preceding step will have a similar effect to double-clicking the form, except now you will be taken into a handler for an event that you chose, with a name that you typed.

14.6.7 Adding controls to a Form

Double-clicking a control in the toolbox adds it to the active form.

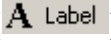
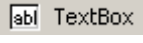
The code generated by the designer will typically carry out the following tasks:

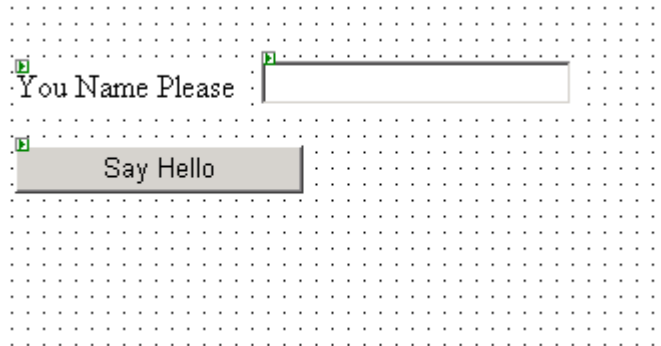
5. Create an instance of the control in the code behind, and
6. Persist the control in the HTML document

Once the control has been laid out on the form, you can move it around using the mouse, change its properties, and create event handlers for it, using the same techniques discussed for Forms.

Let us create a small application using some of the common controls in the toolbox.

14.6.7.1 A simple application

4. Create a new project, and call it MyFirstRealApp
5. Add the controls, and set their properties as follows:
 1. Add a Label control by double clicking on ( Label)
 2. Select the newly added control, and press F4
 3. Change the Text property, in the properties sheet to "Enter your Name: "
6. Add a TextBox control by double clicking on ( TextBox)
7. Empty its Text property, by deleting its content in the properties
8. Resize it by dragging its rightmost corner horizontally
9. Add a Button control by double clicking on ( Button)
10. Change the Text property, in the properties sheet to "Say Hello!"
11. Re-arrange the controls by moving them around to take the following shape:



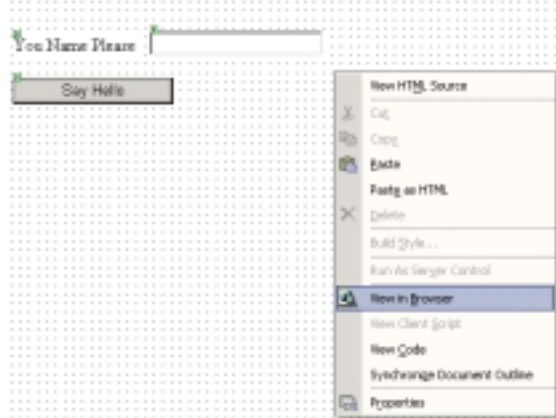
Hint: you can press the ALT key while moving the mouse to override snapping to grid, resulting in a finer move/resize.

You can also change the form's property of `snapToGrid` to `False` if you need more control over the positioning of controls.

6. Create an Event handler for the button control as follows:
 1. Double click the button control
 2. Type the following code in the Button1_Click event handler

```
if (TextBox1.Text != "") {
    Response.Write("Good day " + TextBox1.Text);
}
else
{
    Response.Write ("Type your name PLEASE!");
}
```
 3. Switch back to the Designer view

4. Change the behavior of pressing Enter by changing the TextChanged property of the TextBox as follows:
 - Click the TextBox to select it
 - Click F4 or use the toolbar to get to the property window
 - Click the events button (🔧), located at the top of the Properties window
 - Change the value of TextChanged event to Button1_Click
5. Press F5 to run the program or Right click and choose “View in Browser” and try it out



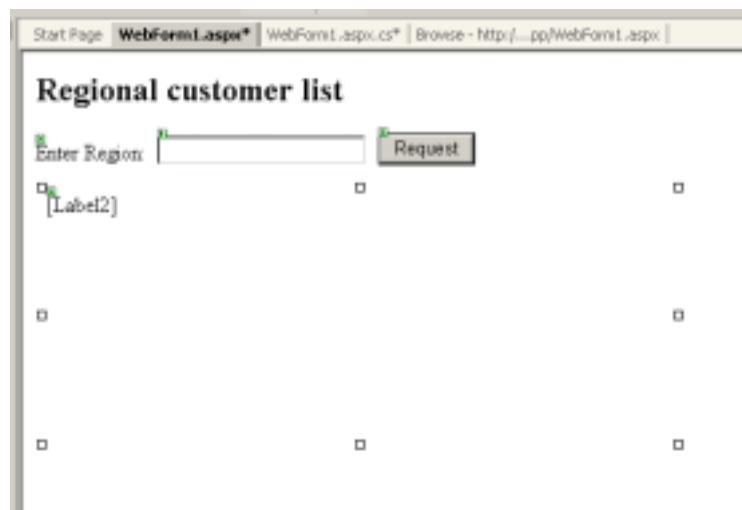
6. Press the button, enter a name and then press the button again.

Further enhancements to the program would include, clearing out the contents of the textbox after the message has been displayed, checking for a valid name (say make sure it is only alphabets).

14.6.7.2 A more useful application

Let us create a program that uses web forms to display information from a data source using ADO.NET.

1. Create a new project, and call it MyFirstRealApp
2. Add the controls, and set their properties as follows:
 1. Add a Label control by double clicking on (**A** Label)
 2. Select the newly added control, and press F4
 3. Change the Text property, in the properties sheet to “Enter Region:”
 4. Add a TextBox control by double clicking on (**ab** TextBox)
 5. Empty its Text property, by deleting its content in the properties
 6. Resize it by dragging its rightmost corner horizontally
 7. Add a Button control by double clicking on (**ab** Button)
 8. Change the Text property, in the properties sheet to “Say Hello!”
 9. Add a second Label control by double clicking on (**A** Label) this will be used to display the results and error messages.
 10. Select the newly added control, and press F4
 11. Empty the Text property, in the properties sheet by simply deleting its contents.
 12. Re-arrange the controls by moving them around to take the following shape:



3. Create an Event handler for the button control as follows:
 1. Double click the button control
 2. Type the following code in the Button1_Click event handler

```
if (TextBox1.Text.Length == 2)
{
    DataHandler(TextBox1.Text);
}
else
{
```

```

        Label2.Text = "Please enter 2 letter Region code e.g. BC";
        Label2.Visible = true;
    }

```

3. Create the DataHandler method as follows (If you don't feel comfortable with the code you should read the ADO.NET chapter):

```

public void DataHandler(string region)
{
    OleDbConnection oCon = new OleDbConnection("Data
        Source=CIMSERVER\\NETSDK;" +
        "Initial Catalog=Northwind;Integrated
        Security=SSPI;" + "Provider=SQLOLEDB");

    OleDbCommand oleCmd = new
        OleDbCommand("SELECT * FROM Customers " +
        "Where Region='" + region + "'", oCon);

    // Create the DataReader object.
    OleDbDataReader myDataReader = null;

    try
    {
        // Open the Connection object.
        oCon.Open();

        // Fill the DataReader
        myDataReader = oleCmd.ExecuteReader( );

        // Iterate through the records.
        while(myDataReader.Read())
        {
            // Display customer data in label control
            Label2.Text += myDataReader["CustomerID"] + " - ";
            Label2.Text += myDataReader.GetValue(2) + " - ";
            Label2.Text += myDataReader.GetValue(3) + " - ";
            Label2.Text += myDataReader.GetValue(4) + "<BR> ";
            Label2.Visible = true;
        }

        // Close the DataReader
        myDataReader.Close();

        // Close the connection
        oCon.Close();
    }
    catch (Exception e)
    {
        Label2.Text = e.ToString();
    }
}

```

4. Finally while you are in the code window add the following code to the Page_load event handler:
 // hide the output area

```
Label2.Visible = false;  
  
// Viewstate maintains value, hence clearing them  
Label2.Text = "";
```

5. Switch back to the Designer view
6. Change the behavior of pressing Enter by changing the TextChanged property of the TextBox as follows:
7. Click the TextBox to select it
8. Click F4 or use the toolbar to get to the property window
9. Click the events button () , located at the top of the Properties window
10. Change the value of TextChanged event to Button1_Click
11. Press F5 to run the program or Right click and choose "View in Browser" and try it out
12. Enter a region name such as BC or WA and then press the button or hit the Enter key.

Further enhancements to the above code could include using a drop down list control to display the regions, and let the user select one of them. Another enhancement could be to allow the user to change some values and write them back to the data source.

Appendix:

Compiler switches:

csc /debug+ /out:exefilename.exe sourfilename.cs

/? Or /Help display help

/main:classname specify the class name, which contains the Main() method

/warnaserror treats all warnings as errors and aborts compilation

/target specifies type of application generated

/checked specifies whether arithmetic overflow will generate a
runtime exception

/doc produce XML documentation file