Dot Net Technology Notes (BCA 602)

BCA VI

UNIT I & UNIT II

Ms. POONAM VERMA

Syllabus

- ***The .Net Framework:Introduction***
- Introduction
- Common Language Runtime (CLR)
- Common Type System (CTS)
- Common Language Specification (CLS)
- Microsoft Intermediate Language (MSIL)
- Just-In –Time Compilation
- Framework Base Classes.
- ***C -Sharp Language (C#):***
- Introduction
- Data Types, Identifiers, Variables, Constants and Literals
- Array and Strings
- Object and Classes
- Inheritance and Polymorphism
- Operator Overloading
- Interfaces, Delegates and Events
- Type conversion.
- ***C# Using Libraries***
- Namespace- System
- Input-Output,
- Multi-Threading,
- Networking and sockets,
- Managing Console I/O Operations,
- Windows Forms,
- Error Handling.
- ***Advance Features using C#***
- Web Services,
- Window Services,
- Asp.net Web Form Controls,
- ADO.Net.
- Distributed Application in C#,
- Unsafe Mode,
- Graphical Device interface with C#.
- ***Assemblies and Attributes***
- .Net Assemblies
- Attributes
- Generic.

Unit-1
Introduction to .Net framework

1. 1 Introduction to Dot Net Framework

The .NET is the technology from Microsoft, on which all other Microsoft technologies will be depending on in future.

It is a major technology change. Just like the computer world moved from DOS to Windows, now they are moving to .NET. But don't be surprised if you find anyone saying that "I do not like .NET and I would stick with the good old COM and C++". There are still lot of people who like to use the bullock-cart instead of the latest Honda car.

.NET technology was introduced by Microsoft, to catch the market from the SUN's Java. Few years back, Microsoft had only VC++ and VB to compete with Java, but Java was catching the market very fast. With the world depending more and more on the Internet/Web and java related tools becoming the best choice for the web applications, Microsoft seemed to be loosing the battle. Thousands of programmers moved to java from VC++ and VB. To recover the market, Microsoft announced .NET.

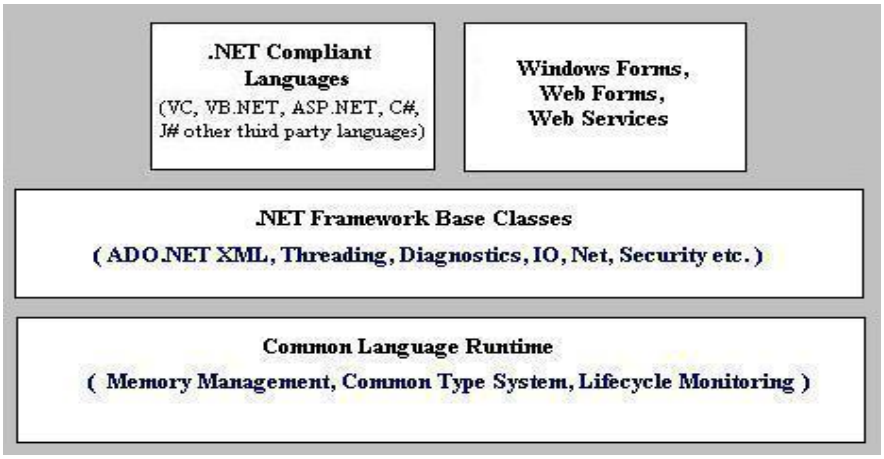
.NET framework comes with a single class library. And thats all programmers need to learn!! Whether they write the code in C# or VB.NET or J#, it doesn't matter, you just use the .NET class library. There is no classes specific to any language. There is nothing more you can do in a language, which you can't do in any other .NET language. You can write code in C# or VB.NET with the same number of lines of code, same performance and same efficiency, because everyone uses same .NET class library.

Features of .NET

- It is a platform neutral framework.
- It is a layer between the operating system and the programming language.
- It supports many programming languages, including VB.NET, C# etc.
- .NET provides a common set of class libraries, which can be accessed from any .NET based programming language. There will not be separate set of classes and libraries for each language. If you know any one .NET language, you can write code in any .NET language.
- In future versions of Windows, .NET will be freely distributed as part of operating system and users will never have to install .NET separately.

Major Components of .NET

The diagram given below describes various components of .NET Framework.



The .NET framework can only be exploited by languages that are compliant with .NET. Most of Microsoft languages have been made to fully comply with .NET.
.NET also introduces Web Forms, Web Services and Windows Forms. The reason why they have been shown separately and not as a part of a particular language is that these
technologies can be used by any .NET compliant language. For example Windows Forms is used by VC, VB.NET, C# all as a mode of providing GUI.
The next component of .NET is the .NET Framework Base Classes. These are the common class libraries (much like Java packages) that can be used by any .NET compliant language. These classes provide the programmers with a high degree of functionality that they can use in their programs. For example their are classes to handle reading, writing and manipulating XML documents, enhanced ADOs etc.
The bottom most layer is the CLR - the common runtime language.

Origin of .net Technology

1. OLE Technology
2. COM Technology
3. .net Technology

OLE Technology(Object Linking and Embedding)

- Easy interprocess communication
- Embed documents from one application into another application
- To enable one application to manipulate objects located in another application
- Ex: inter operability between various products such as MS word and MSExcel

COM Technology(Component Object Model)

- Monolithic approach leads to many problem of maintainability and testing
- A program is broken into number of independent components where each one offers a particular service
- Each component can be developed and tested independently and then integrated into main system.
- Benefits:
- Reduces the overall complexity of software.
- Enables distributed development across multiple organization or departments.
- Enhances software maintainability

.net Technology

- Third generatiom component model
- IPC in COM is replaced by Intermediate Language(IL or MSIL)
- Interoperability by compiling code into IL.
- Metadata

1.2 Common Language Runtime (CLR)

The CLR is the heart of .NET framework. It is .NET equivalent of Java Virtual Machine (JVM). It is the runtime that converts a MSIL (Micro Soft Intermediate Language) code into the host machine language code, which is then executed appropriately.
The CLR provides a number of services that include:

- Loading and execution of codes
- Memory isolation for application
- Verification of type safety
- Compilation of IL into native executable code
- Providing metadata
- Automatic garbage collection
- Enforcement of Security

- Interoperability with other systems
- Managing exceptions and errors
- Provide support for debugging and profiling

1.3 Common Type System (CTS)

The language interoperability, and .NET Class Framework, are not possible without all the language sharing the same data types. What this means is that an "int" should mean the same in VB, VC++, C# and all other .NET compliant languages. Same idea follows for all the other data types. This is achieved through introduction of Common Type System (CTS).

CTS, much like Java, defines every data type as a Class. Every .NET compliant language must stick to this definition. Since CTS defines every data type as a class; this means that only Object-Oriented (or Object-Based) languages can achieve .NET compliance. Given below is a list of CTS supported data types:

Data Type	**Description**
System.Byte	1-byte unsigned integer between 0-255
System.Int16	2-bytes signed integer in the following range: 32,678 to 32,767
System.Int32	4-byte signed integer containing a value in the following range: -2,147,483,648 to 2,147,483,647
System.Int64	8-byte signed integer containing a value from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
System.Single	4-byte floating point. The value limits are: **for negative values:** *-3.402823E38 to - 1.401298E-45* **for positive values:** 1.401298E-45 TO 30402823E38
System.Double	8-bytes wide floating point. The value limits are: **for negative values:** *-1.79769313486231E308 to - 4.964065645841247E-324* **for positive values:** *4.964065645841247E-324 to 1.79769313486232E308*
System.Object	4-bytes address reference to an object
System.Char	2-bytes single Unicode Character.
System.String	string of up to 2 billion Unicode characters.
System.Decimal	12-bytes signed integer that can have 28 digits on either side of decimal.
System.Boolean	4-Bytes number that contains true(1) or false (0)

1.4 Common Language Specification (CLS)

One of the obvious themes of .NET is unification and interoperability between various programming languages. In order to achieve this; certain rules must be laid and all the languages

must follow these rules. In other words we can not have languages running around creating their own extensions and their own fancy new data types. CLS is the collection of the rules and constraints that every language (that seeks to achieve .NET compatibility) must follow. Microsoft has defined three level of CLS compatibility/compliance. The goals and objectives of each compliance level have been set aside. The three compliance levels with their brief description are given below:

Compliant producer

The component developed in this type of language can be used by any other language.

Consumer

The language in this category can use classes produced in any other language. In simple words this means that the language can instantiate classes developed in other language. This is similar to how COM components can be instantiated by your ASP code.

Extender

Languages in this category can not just use the classes as in CONSUMER category; but can also extend classes using inheritance.

Languages that come with Microsoft Visual Studio namely Visual C++, Visual Basic and C#; all satisfy the above three categories. Vendors can select any of the above categories as the targeted compliance level(s) for their languages.

1.5 Microsoft Intermediate Language (MSIL)

A .NET programming language (C#, VB.NET, J# etc.) does not compile into executable code; instead it compiles into an intermediate code called Microsoft Intermediate Language (MSIL). As a programmer one need not worry about the syntax of MSIL - since our source code in automatically converted to MSIL. The MSIL code is then send to the CLR (Common Language Runtime) that converts the code to machine language which is then run on the host machine.

MSIL is similar to Java Byte code. A Java program is compiled into Java Byte code (the .class file) by a Java compiler, the class file is then sent to JVM which converts it into the host machine language.

Managed Code

The role of CLR doesn't end once we have compiled our code to MSIL and a JIT compiler has compiled this to native code. Code written using the .NET framework, is managed code when it is executed. This stage is usually referred to as being at runtime. This means that the CLR looks after our applications, by managing memory, handling security, allowing cross language debugging and so on. By contrast, applications that do not run under the control of the CLR are said to be unmanaged and certain languages such as C++ can be used to write such applications, that for example, to access low level functions of the operating systems. However in C# we can only write code that runs in a managed environment.

Unified classes

The term .NET framework refers to the group of technologies that form the development foundation for the Microsoft .NET platform. The key technologies in this group are the run time and the class libraries.

The run time is responsible for managing your code and providing services to it while it executes, playing a role similar to that of the Visual Basic 6.0 run time.

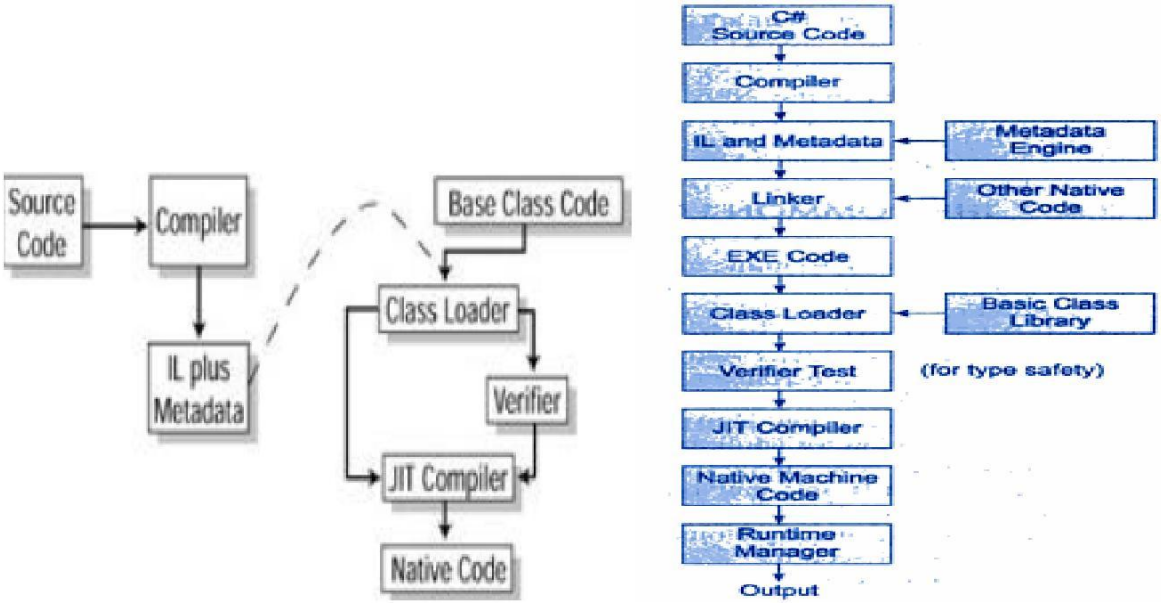
The .NET programming languages including Visual Basic .NET , Microsoft Visual C# and C++ managed extensions and many other programming languages from various vendors utilize .NET services and features through a common set of unified classes.

The .NET unified classes provide foundation of which you build your applications, regardless of the language you use. Whether you simply concating a string, or building a windows Services or a multiple-tier web-based applications, you will be using these unified classes.

The unified classes provide a consistent method of accessing the platforms functionality. Once you learn to use the class library, you'll find that all tasks follow the same uniform architecture, you no longer need to learn and master different API architecture to write your applications.
By building your applications on a unified, integrated framework, you maximize your return on the time you spend learning this framework, and you end up with more robust applications that are easy to deploy and maintain.

1.6 Just In Time Compiler

- To make it easy for language writers to port their languages to .NET, Microsoft developed a language akin to assembly language called Microsoft intermediate language (MSIL). To compile applications for .NET, compilers take source code as input and produce MSIL as output.
- MSIL itself is a complete language that you can write applications in. However, as with assembly language, you would probably never do so except in unusual circumstances. Because MSIL is its own language, each compiler team makes its own decision about how much of the MSIL it will support. However, if you're a compiler writer and you want to create a language that does interoperate with other languages, you should restrict yourself to features specified by the CLS.



- You write source code in C# and compile it using the C# compiler (csc.exe) into an EXE.
- The C# compiler outputs the MSIL code and a manifest into a read-only part of the EXE that has a **standard PE (Win32-portable executable) header.** When the compiler creates the output, it also imports a function named **_ *CorExeMain* from the .NET runtime.**
- When the application is executed, the operating system loads the PE, as well as any dependent dynamic-link libraries (DLLs), such as the one that exports the _ *CorExeMain* function (mscoree.dll), just as it does with any valid PE.

1.7 Framework Base Classes

The .NET Framework has an extensive set of class libraries. This includes classes for:

- **Data Access:** High Performance data access classes for connecting to SQL Server or any other OLEDB provider.
- **XML Supports:** Next generation XML support that goes far beyond the functionality of MSXML.
- **Directory Services:** Support for accessing Active Directory/LDPA using ADSI.
- **Regular Expression :** Support for above and beyond that found in Perl 5.
- **Queuing Supports:** Provides a clean object-oriented set of classes for working with MSMQ.

These class libraries use the CLR base class libraries for common functionality.

Base Class Libraries

The Base class library in the .NET Framework is huge. It covers areas such as:

- **Collection :** The System.Collections namespaes provides numerous collection classes.
- **Thread Support:** The System.Threading namespace provides support for creating fast, efficient, multi-threaded appliction.
- **Code Generation:** The System.CodeDOM namespace provides classes for generating source files in numerous language. ASP.NET uses these classes when converting ASP.NET pages into classes, which are subsequently compiled.
- **IO:** The System.IO provides extensive support for working with files and all other stream types.
- **Reflection:** The System.Reflection namespace provides support for load assemblies, examining the type with in assemblies, creating instances of types, etc.
- **Security:** The System.Security namespace provides support for services such as authentication, authorization, permission sets, policies, and cryptography. These base services are used by application development technologies like ASP.NET to build their security infrastructure.

The list of support base classes goes on forever in .NET, but if you ever find yourself lost looking for a specific class, you can use the WinCV tool to locate it. You can run this from the Start bar Run menu. The file is typically located in the c:\program files\ Microsoft.Net\FrameworkSDK\Bin directory.

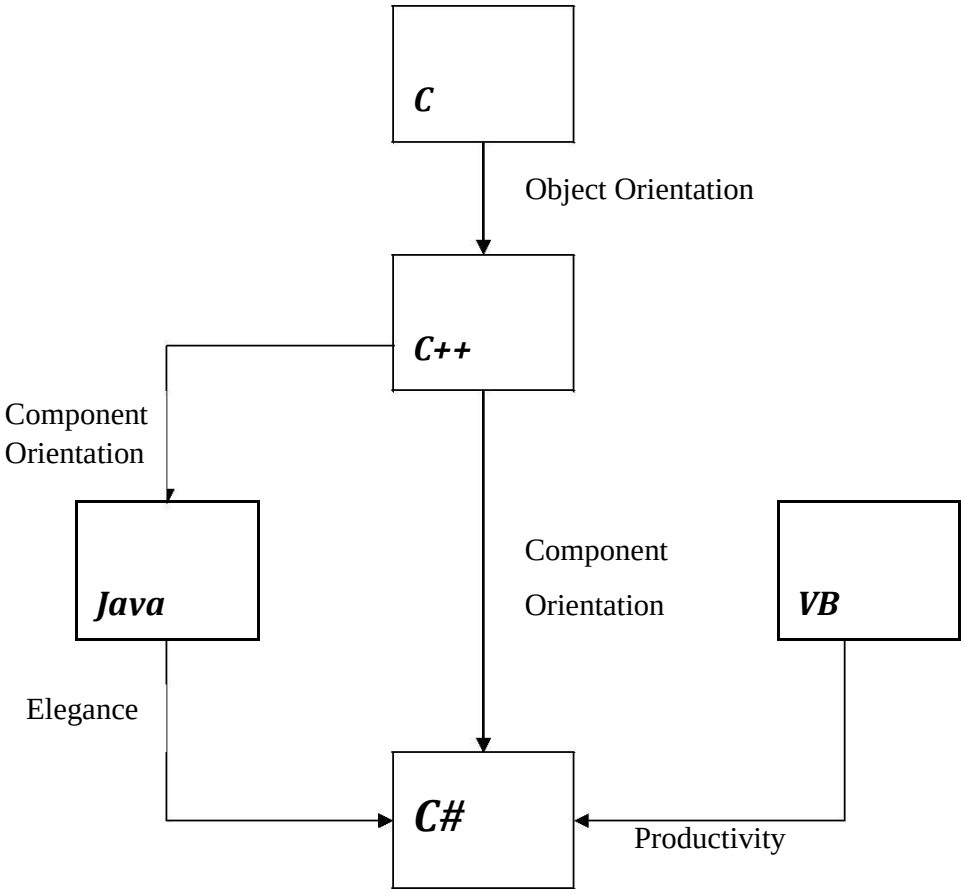
Unit-II

C -Sharp Language (C#)

2.1 Introduction

Microsoft Corporation, developed a new computer programming language C# pronounced as 'C-Sharp'. C# is a simple, modern, object oriented, and type safe programming language derived from C and C++. C# is a purely object-oriented language like as Java. It has been designed to support the key features of .NET framework.

Like Java, C# is a descendant language of C++ which is descendant of C language.



C# modernize C++ by enhancing some of its features and adding a few new features. C# borrows Java's features such as grouping of classes, interface and implementation together in one file so the programmers can easily edit the codes. C# also handles objects using reference, the same way as Java.

C# uses VB's approach to form designing, namely, dragging controls from a tool box, dropping them onto forms, and writing events handlers for them.

Comparing C# to C++ and Java

C# versus Java

C# and Java are both new-generation languages descended from a line including C and C++. Each includes advanced features, like garbage collection, which remove some of the low level maintenance tasks from the programmer. In a lot of areas they are syntactically similar.

Both C# and Java compile initially to an intermediate language: C# to Microsoft Intermediate Language (MSIL), and Java to Java bytecode. In each case the intermediate language can be run -

by interpretation or just-in-time compilation - on an appropriate 'virtual machine'. In C#, however, more support is given for the further compilation of the intermediate language code into native code.
C# contains more primitive data types than Java, and also allows more extension to the value types. For example, C# supports 'enumerations', type -safe value types which are limited to a defined set of constant variables, and 'structs', which are user-defined value types.
Unlike Java, C# has the useful feature that we can overload various operators.
Like Java, C# gives up on multiple class inheritance in favour of a single inheritance model extended by the multiple inheritance of interfaces. However, polymorphism is handled in a more complicated fashion, with derived class methods either 'overriding' or 'hiding' super class methods C# also uses 'delegates' - type-safe method pointers. These are used to implement event-handling. In Java, multi -dimensional arrays are implemented solely with single-dimensional arrays (where arrays can be members of other arrays. In addition to jagged arrays, however, C# also implements genuine rectangular arrays.

C# versus C++

Although it has some elements derived from Visual Basic and Java, C++ is C#'s closest relative. In an important change from C++, C# code does not require header files. All code is written inline. As touched on above, the .NET runtime in which C# runs performs memory management, taking care of tasks like garbage collection. Because of this, the use of pointers in C# is much less important than in C++. Pointers can be used in C#, where the code is marked as 'unsafe', but they are only really useful in situations where performance gains are at an absolute premium.
Speaking generally, the 'plumbing' of C# types is different from that of C++ types, with all C# types being ultimately derived from the 'object' type. There are also specific differences in the way that certain common types can be used. For instance, C# arrays are bounds checked unlike in C++, and it is therefore not possible to write past the end of a C# array.
C# statements are quite similar to C++ statements. To note just one example of a difference: the 'switch' statements has been changed so that 'fall-through' behavior is disallowed.
As mentioned above, C# gives up on the idea of multiple class inheritance. Other differences relating to the use of classes are: there is support for class 'properties' of the kind found in Visual Basic, and class methods are called using the . operator rather than the :: operator.

Features of C#

1. Simplicity

All the Syntax of java is like C++. There is no preprocessor, and much larger library. C# code does not require header files. All code is written inline.

2. Consistent behavior

C# introduced an unified type system which eliminates the problem of varying ranges of integer types. All types are treated as objects and developers can extend the type system simply and easily.

3. Modern programming language

C# supports number of modern features, such as:

- Automatic Garbage Collection
- Error Handling features
- Modern debugging features
- Robust Security features

4. Pure Object- Oriented programming language

In C#, every thing is an object. There are no more global functions, variable and constants.
It supports all three object oriented features:

- Encapsulation

- Inheritance
- Polymorphism

5. Type Safety

Type safety promotes robust programming. Some examples of type safety are:

- All objects and arrays are initialized by zero dynamically
- An error message will be produced , on use of any uninitialized variable
- Automatic checking of array out of bound and etc.

6. Feature of Versioning

Making new versions of software module work with the existing applications is known as versioning. Its achieve by the keywords **new** and **override.**

7. Compatible with other language

C# enforces the .NET common language specifications (CLS) and therefore allows inter-operation with other .NET language.

8. Inter-operability

C# provides support for using COM objects, no matter what language was used to author them. C# also supports a special feature that enables a program to call out any native API.

A Simple C# Program

Let's begin in the traditional way, by looking at the code of a Hello World program (note that the tabulation and line numbers are included just for the sake of readability).

```
1.  Using System;
2.  public class HelloWorld
3.  {
4.      public static void Main()
5.      {
6.          // This is a single line comment
7.          /* This is a
8.          multiple
9.          line comment */
10.         Console.WriteLine("Hello World! ");
11.     }
12. }
```

- The first thing to note about C# is that it is case-sensitive. You will therefore get compiler errors if, for instance, you write 'console' rather than 'Console'.
- The second thing to note is that every statement finishes with a semicolon (;) or else takes a code block within curly braces.

Explanation of Program

Line 1 : using System;

we are using the System namespace (namespaces are also covered in chapter 7). The point of this declaration is mostly to save ourselves time typing. Because the 'Console' object used in line 10 of the code actually belongs to the 'System' namespace, its fully qualified name is 'System.Console'. However, because in line 1 we declare that the code is using the System namespace, we can then leave off the 'System.' part of its name within the code.

Line 2: public class HelloWorld

As C# is an object-oriented language, C# programs must be placed in classes (classes are discussed in chapter 5 but if you are new to object orientation we suggest that you first read some introductory material). This line declares the class to be named 'HelloWorld'.

Line 4: public static void Main()

When compiled and run, the program above will automatically run the 'Main' method declared and begun in this line. Note again C#'s case-sensitivity - the method is 'Main' rather than 'main'.

Line 3,11 and 5,12 :

These lines are uses the '{' for starting braces and '}' for closing braces of block. **Lines 6-9** : Comments

('//' uses for single line and '/* -- - - */' uses for multiple line comments)

These lines of the program are ignored by the compiler, being comments entered by the programmer for his own benefit.

Line 6 shows a single line comment, in which everything on the line after the two forward slashes is ignored by the compiler.

Lines 7-9 demonstrate a multi-line comment, in which everything between the opening /* and closing */ is ignored, even when it spans multiple lines.

Line 10:

The statement on this line calls the 'WriteLine' method of the Console class in the System namespace. It should be obvious how this works in the given example - it just prints out the given string to the 'Console' (on PC machines this will be a DOS prompt).

Instruction for Saving the Program

In order to run the program, it must first be saved in a file. Unlike in Java, the name of the class and the name of the file in which it is saved do not need to match up, although it does make things easier if you use this convention. In addition, you are free to choose any extension for the file, but it is usual to use the extension '.cs'.

Writing program in Computer

There are two ways of program writing in

computer • Using Text Editor

Using Visual Studio.NET

2.2 Data Types. Identifiers, Variables, Constants and Literals

Identifiers & Variables

Identifiers refer to the names of variables, functions arrays, classes, etc. created by programmer. They are fundamental requirement of any language. Each language has its own rules for naming these identifiers.

To name the variables of your program, you must follow strict rules. In fact, everything else in your program must have a name.

There are some rules you must follow when naming your objects. On this site, here are the rules we will follow:

- The name must start with a letter or an underscore
- After the first letter or underscore, the name can have letters, digits, and/or underscores
- The name must not have any special characters other than the underscore
- The name cannot have a space

C# is case-sensitive. This means that the names Case, case, and CASE are completely different. For example, the main function is always written Main.

C# Keywords

C# uses a series of words, called keywords, for its internal use. This means that you must avoid naming your objects using one of these keywords. They are:

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

Data types

C# is a type-safe language. Variables are declared as being of a particular type, and each variable is constrained to hold only values of its declared type.

Variables can hold either value types or reference types, or they can be pointers. Here's a quick recap of the difference between value types and reference types.

- where a variable v contains a value type, it directly contains an object with some value. No other variable v' can directly contain the object contained by v (although v' might contain an object with the same value).

- where a variable v contains a reference type, what it directly contains is something which refers to an object. Another variable v' can contain a reference to the same object referred to by v.

Value Types

C# defines the following value types:

- Primitives int i;
- Enum enum state { off, on }
- Struct struct Point{ int x, y; }

It is possible in C# to define your own value types by declaring enumerations or structs. These user-defined types are mostly treated in exactly the same way as C#'s predefined value types, although compilers are optimized for the latter. The following table lists, and gives information about, the predefined value types. Because in C# all of the apparently fundamental value types are in fact built up from the (actually fundamental) object type, the list also indicates which System types in the .Net framework correspond to these pre-defined types.

C# Type	**.Net Framework (System) type**	**Signed?**	**Bytes Occupied**	**Possible Values**
sbyte	System.Sbyte	Yes	1	-128 to 127
short	System.Int16	Yes	2	-32768 to 32767
int	System.Int32	Yes	4	-2147483648 to 2147483647
long	System.Int64	Yes	8	-9223372036854775808 to 9223372036854775807
byte	System.Byte	No	1	0 to 255
ushort	System.Uint16	No	2	0 to 65535
uint	System.UInt32	No	4	0 to 4294967295

ulong	System.Uint64	No	8	0 to 18446744073709551615
float	System.Single	Yes	4	Approximately ±1.5 x 10^{-45} to ±3.4 x 10^{38} with 7 significant figures
double	System.Double	Yes	8	Approximately ±5.0 x 10^{-324} to ±1.7 x 10^{308} with 15 or 16 significant figures
decimal	System.Decimal	Yes	12	Approximately ±1.0 x 10^{-28} to ±7.9 x 10^{28} with 28 or 29 significant figures
char	System.Char	N/A	2	Any Unicode character (16 bit)
bool	System.Boolean	N/A	1 / 2	true or false

In the following lines of code, two variables are declared and set with integer values.

```
int x = 10;
int y = x;
y = 20; // after this statement x holds value 10 and y holds value 20
```

Reference Types

The pre-defined reference types are object and string, where object - is the ultimate base class of all other types. New reference types can be defined using 'class', 'interface', and 'delegate' declarations. There fore the reference types are :

Predefined Reference Types

- Object
- String

User Defined Reference Types

- Classes
- Interfaces
- Delegates
- Arrays

Reference types actually hold the value of a memory address occupied by the object they reference. Consider the following piece of code, in which two variables are given a reference to the same object (for the sake of the example, this object is taken to contain the numeric property 'myValue').

```
object x = new
object(); x.myValue =
10; object y = x ;
y.myValue = 20; // after this statement both
                x.myValue // and y.myValue equal 20
```

This code illustrates how changing a property of an object using a particular reference to it is reflected in all other references to it. Note, however, that although strings are reference types, they work rather more like value types. When one string is set to the value of another, eg

```
string s1 = "hello";
string s2 = s1;
```

Then s2 does at this point reference the same string object as s1. However, when the value of s1 is changed, for instance with

```
s1 = "goodbye";
```

what happens is that a new string object is created for s1 to point to. Hence, following this piece of code, s1 equals "goodbye", whereas s2 still equals "hello".
The reason for this behaviour is that string objects are 'immutable'. That is, the properties of these objects can't themselves change. So in order to change what a string variable references, a new string object must be created.

Boxing

C# allows you convert any value type to a corresponding reference type, and to convert the resultant 'boxed' type back again. The following piece of code demonstrates boxing. When the second line executes, an object is initiated as the value of 'box', and the value held by i is copied across to this object. It is interesting to note that the runtime type of box is returned as the boxed value type; the 'is' operator thus returns the type of box below as 'int'.

```
int i = 123;
object box = i;
if (box is int)
{Console.Write("Box contains an int");} // this line is printed
```

When boxing occurs, the contents of value type are copied from stack into memory allocated into the managed heap. The new reference type created contains a copy of the value type, and can be used by other types that expect an object reference. The value contained in the value type and the created reference types are not associated in any way (except that they contain the same values). If we change the original value type, the refernce type is not affected.

The following code explicitly **unboxes** a reference type into a value type:

```
object o;
int i = (int) o;
```

When unboxing occurs, memory is copied from the managed heap to the stack.

2.3 Array and Strings

Arrays

An array is a group or collection of similar values. An array contains a number of variables, which are accessed through computed indices. The various value contained in an array are also called the elements of array. All elements of an array have to be of same type, and this type is called the element type of the array. The element of an array can be of any type including an array type.
An array has a rank that determines the number of indices associated wth each array elements. The rank of an array is also referred as the dimension of the array. An array may be :

- Single Dimensional
- Multi Dimensional

An array with a rank of one is called single-dimensional array, and an array with a rank greater than one is called a multi dimensional array.
Each dimension of array has an associated length, which is an integer number greater than or equal to zero. For a dimension of length n, indices can range from 0 to n-1. in C#, array types are categorized under the reference types alongside with classes and interfaces.

Single Dimensional Array

Single -dimensional arrays have a single dimension (ie, are of rank 1). The process of creation of arrays is basically divided into three steps:

1. Declaration of Array
2. Memory Allocation for Array
3. Initialization of Array

Declaration of Array

To declare an array in C# place a pair of square brackets after the variable type. The syntax is given below :

```
type[] arrayname;
```

For Example:

```
int[] a; float[]
marks;
double[] x;
int[] m,n;
```

You must note that we do not enter the size of the arrays in the declaration.

Memory Allocation for Array

After declaring an array, we need to allocate space and defining the size. Declaring arrays merely says what kind of values the array will hold. It does not create them. Arrays in C# are objects, and you use the new keyword to create them. When you create an array, yu must tell the compiler how many components will be stored in it. Here is given the syntax:

```
arrayname = new type[size];
```

For Example:

```
a = new int[5];
      marks = new float[6];
      x = new double[10];
      m = int[100];
n = int [50];
```

It is also possible to combine the two steps, declaration and memory allocation of array, into one as shown below:

```
int[] num = new int [5];
```

Initialization of Array

This step involves placing data into the array. Arrays are automatically assigned the default values associated with their type. For example, if we have an array of numerical type, each element is set to number 0. But explicit values can be assigned as and when desired.

Individual elements of an array are referenced by the array name and a number that represents their position in the array. He number you use to identify them are called subscripts or indexes into the array.

Subscripts are consecutive integers beginning with 0. thus the array "num" above has components **num[0], num[1], num[2], num[3],** and **num[4].**

The initialization process is done using the array subscripts as shown:

```
arrayname[subscript] = value;
```

For Example:

```
num[0] = 5;
num[1] = 15;
num[2] = 52;
num[3] = 45;
num[4] = 57;
```

We can also initialize arrays automatically in the same way as the ordinary variables when they are declared, as shown below:

```
type[] arrayname = { list of values };
```

the list of variables separated by commas and defined on both ends by curly braces. You must note that no size is given in this syntax. The compiler space for all the elements specified in the list.

For Example:

```
int[] num = {5,15,52,45,57};
```

You can combine all the steps, namely declaration, memory allocation and initialization of arrays like as:

```
int[] num = new int [5] {5,15,52,45,57};
```

You can also assign an array object to another. For Example

```
int[] a = { 10, 20,
30}; int[] b;
b=a;
```

The above example is valid in C#. Both the array will have same values.

Example

```
using system;
class Number
{
        public static void Main()
        {
                int [] num = {10, 20, 30, 40,
                50}; int n = num.Length;
                // Length is predefined attribute to access the size of
                array Console.Write(" Elements of array are :");
                for(int i=0; i<n; i++)
                {
                        Console.WriteLine(num[i]);
                }

                int sum =0;
                for(int i=0; i<n; i++)
                {
                        sum = sum + num[i]);
                }
                Console.WriteLine(" The sum of elements :"+sum);
}
```

OUTPUT:

Elements of array
are: 10 20 30 40 50

The sum of elements :150

Multi Dimensional Array

C# supports two types of multidimensional arrays:

- Rectangular Array
- Jagged Array

Rectangular Arrays

A rectangular array is a single array with more than one dimension, with the dimensions' sizes fixed in the array's declaration. The following code creates a 2 by 3 multi-dimensional array:

```
int[,] squareArray = new int[2,3];
```

As with single-dimensional arrays, rectangular arrays can be filled at the time they are declared. For instance, the code

```
int[,] squareArray = {{1, 2, 3}, {4, 5, 6}};
```

creates a 2 by 3 array with the given values. It is, of course, important that the given values do fill out exactly a rectangular array.

The **System.Array** class includes a number of methods for determining the size and bounds of arrays. These include the methods **GetUpperBound(int i)** and **GetLowerBound(int i)**, which

return, respectively, the upper and lower subscripts of dimension i of the array (note that i is zero based, so the first array is actually array 0).
For instance, since the length of the second dimension of squareArray is 3, the

```
expression squareArray.GetLowerBound(1)
```

returns 0, and the expression

```
squareArray.GetUpperBound(1)
```

returns 2.
System.Array also includes the method **GetLength(int i)**, which returns the number of elements in the ith dimension (again, zero based).
The following piece of code loops through squareArray and writes out the value of its elements.

```
for(int i = 0; i < squareArray.GetLength(0); i++)
  for (int j = 0; j < squareArray.GetLength(1); j++)
     Console.WriteLine(squareArray[i,j]);
```

A foreach loop can also be used to access each of the elements of an array in turn, but using this construction one doesn't have the same control over the order in which the elements are accessed.

Jagged Arrays

Using jagged arrays, one can create multidimensional arrays with irregular dimensions. This flexibility derives from the fact that multidimensional arrays are implemented as arrays of arrays. The following piece of code demonstrates how one might declare an array made up of a group of 4 and a group of 6 elements:

```
int[][] jag = new int[2][];
jag[0] = new int [4];
jag[1] = new int [6];
```

The code reveals that each of jag[0] and jag[1] holds a reference to a single-dimensional int array. To illustrate how one accesses the integer elements: the term jag[0][1] provides access to the second element of the first group.
To initialise a jagged array whilst assigning values to its elements, one can use code like the following:

```
int[ ][ ] jag = new int[ ][ ] {new int[ ] {1, 2, 3, 4}, new int[ ] {5, 6, 7, 8, 9, 10}};
```

Be careful using methods like GetLowerBound, GetUpperBound, GetLength, etc. with jagged arrays. Since jagged arrays are constructed out of single-dimensional arrays, they shouldn't be treated as having multiple dimensions in the same way that rectangular arrays do.
To loop through all the elements of a jagged array one can use code like the following:

```
for (int i = 0; i < jag.GetLength(0); i++)
   for (int j = 0; j < jag[i].GetLength(0);
      j++) Console.WriteLine(jag[i][j]);
```

or

```
for (int i = 0; i < jag.Length; i++)
   for (int j = 0; j < jag[i].Length; j++)
      Console.WriteLine(jag[i][j]);
```

Strings

A string is an empty space, a character, a word, or a group of words that you want the compiler to consider "as is", that is, not to pay too much attention to what the string is made of, unless you explicitly ask it to. This means that, in the strict sense, you can put in a string anything you want. Primarily, the value of a string starts with a double quote and ends with a double-quote. An example of a string is "Welcome to the World of C# Programming!". You can include such a string in the **Console.Write()** method to display it on the console. Here is an example:

Example using
System; class
BookClub
{
 static void Main()
  {
   Console.WriteLine("Welcome to the World of C# Programming!");
  }
}

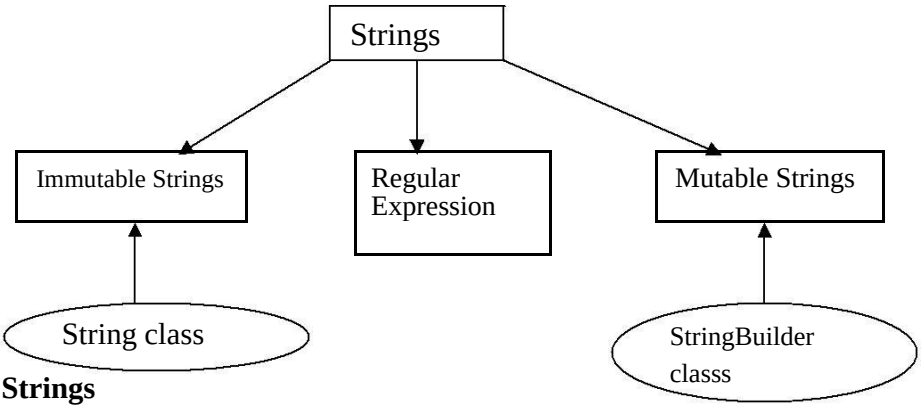
OUTPUT:
Welcome to the World of C# Programming!

Types of String

There are two types of string in C#:

1) **Immutable strings**
2) **Mutable strings**

The immutable strings are can't be modify and mutable strings are modifiable.C# also supports a feature of *regular expression* that can be used for complex strings manipulations and pattern matching.



Handling of Strings

We can create immutable strings using **string** or **String** objects in number of ways. There are a some techniques to handling the immutable strings:

Assigning String

```
string s1;
s1 = "Welcome";
```

or

```
string s1 = "Welcome";
```

Copying String

```
string s2 = s1;
```

or

```
string s2 = string.Copy(s1);
```

Concatenating Strings

```
string s3 = s1 + s2;
```

or

```
string s3 = string.Concat(s1,s2);
```

Reading from Console

```
string s1 = Console.ReadLine();
```

Converting Number to String

```
int num = 100;
string s1= num.ToString();
```

Inserting String

```
string s1 = Wel;
string s2 = s1.insert(3,"come");
// s2 = Welcome
string s3 = s1.insert(3,"don");
// s3 = Weldon;
```

Comparing Strings

```
int n = string. Compare(s1,s2);
```

This statement will perform case-sensitive comparison and returns integer values for different conditions. Such as:

- If s1 is equal to s2 it will return zero.
- If s1 is greater than s2 it will return positive integer (1).
- If s1 is less than s2 it will return negative integer(-1). Or you can use following statement:

```
bool a = s2.Equals(s1); bool
b = string.Equal(s1,s2);
```

Above statements will return a Boolean value **true** (if equal) or **false**(if not equal).

Or you can also use the "==" operator for comparing the strings. Like as:

```
if ( s1 == s2)
  Console.Write(" both are equal");
```

In this statement, it will return a Boolean value **true** (if equal) or **false**(if not equal).

Mutable String

Mutable strings are those strings, which can be modify dynamically. This type of strings are created using **StringBuilder** class. For Example:

```
StringBuilder s1 = new StringBuilder("Welcome");
StringBuilder s2 = new StringBuilder( );
```

The string **str1** is created with an initial size of seven characters and **str2** is created as an empty string. They can grow dynamically as more character added to them. Mutual string are referred as a *dynamic strings*.
The StringBuilder class supports many methods that are useful for manipulating dynamic strings. Some of the most common methods are listed below:

Method	Operation
Append( )	Append a string
AppendFormat( )	Append string using specific format
EnsureCapacity( )	Ensure sufficient size
Insert( )	Insert a string at a specified position
Remove( )	Remove specified character
Replace( )	Removes previous string with new one

StringBuilder also provides some attributes to access some properties of strings, such as:

Attributes	Puspose
Capacity	To retrieve or set the number of characters the object can hold
Length	To retrieve or set the length
MaxCapacity	To retrieve maximum capacity of the object
[ ]	To get or set a character at a specified position

Example

```
using System.Text;    //For using StringBuilder
using System;

class StrMethod
{
        public static void Main( )
        {
                StringBuilder s = new StringBuilder("C");
                Console.WriteLine(" Stored String is :"+ s);
                Console.WriteLine("Length of string is :"+s.Length);

                s.Append("Sharp ");
                // appending the string s

                Console.WriteLine(" After Append String is :"+ s);
                Console.WriteLine("Length of string is :"+s.Length);

                s.Insert(7,"Language");
                // inserting the string at last in s

                Console.WriteLine("After Insertion String is:"+ s);
                Console.WriteLine("Length of string is :"+s.Length);

                int n = s.Length;

                s[n] = "!";
```

```
            Console.WriteLine(" At Last String is :"+ s);
        }
}
```

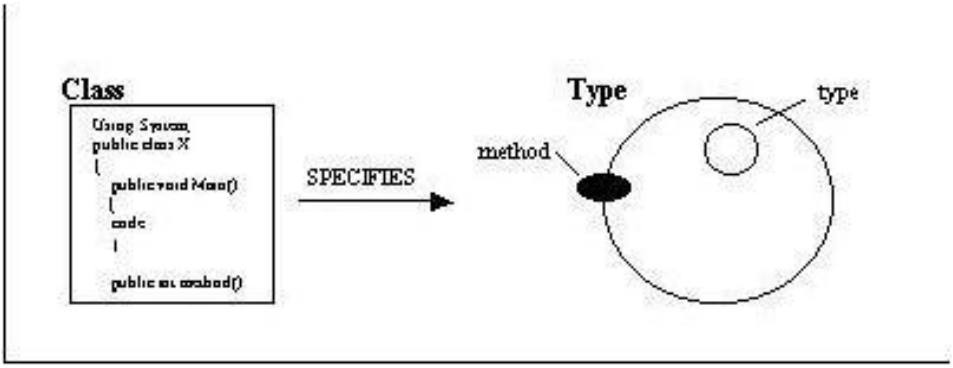
OUTPUT:

Stored String is : C
Length of String is : 1

After Append string is : CSharp
After Insertion String is : CSharp Language
At Last String is : CSharp Language!

2.4 Object and Classes

As we noted previously, one can create new reference types by defining classes. Classes provide 'templates' from which these direct instances are generated. Where we appeal to the relation between a class and its corresponding reference type instances we shall say that a class specifies the type (also that the class specifies the constitutive elements of the type).
Any type is made up of elements, which we term type members. There are two main kinds of type members that a class can specify. Firstly, a class can specify other types - both value and reference. This idea, that types can contain other types, is known within the literature on object orientation as 'containment', or else 'aggregation'. Where a type contains another reference type, we shall call it the containing type of the latter.
The second, main kind of type members that a class can specify are methods, functions designed for reading and manipulating the value and reference types an instance contains.



Objects in C# are created from types, just like a variable. This type of an object is known as class. we can use class definition to instantiate objects, which means to create a real named instance of a class.

Declaration of Classes

Class is an user-defined data type. To create a class, you start with the **class** keyword followed by a name and its body delimited by curly brackets. . The following is an example of a very simple class declaration

```
class classname
{
   // class-body
}
```

The class-body contains the member data and member function of the class. C++ programmer must note that there is no semicolon after the closing brace of the class.

Members of Class

Class is a mechanism to implement the encapsulation, it bind the data and a function in a single unit. Therefore data and functions are the members of class, which is known as member data and member function.

There is an example of full class :

```
class Circle
{
      double radius;
      public void get_radius(double r)
      {
          radius = r;
        }
      public double area()
      {
        return ( 3.14 * r *r);
         }
}
```

Member Access Modifiers

Access modifiers provide the accessibility control for the members of classes to outside the class. They also provide the concept of data hiding. There are five member access modifiers provided by the C# Language.

Modifier	***Accessibility***
private	Members only accessible with in class
public	Members may accessible anywhere outside class
protected	Members only accessible with in class and derived class
internal	Members accessible only within assembly
protected internal	Members accessible in assembly, derived class or containing program

By default all member of class have **private** accessibility. If we want a member to have any other accessibility, then we must specify a suitable access modifier to it individually.

For Example:

```
class Demo
{
   public int a;
   internal int x;
   protected    double d;
   float m;     // private by default
  }
```

We cannot declare more than one member data or member function under an accessibility modifier.

Read Only Member

When creating a member variable of a class, one of the decisions you make consists of deciding how the field would get its value(s). Sometimes you will allow the clients of the class to change the values of the field. In some other cases, you may want the field to only hold or present the value without being able to change it. This can still allow the clients to access the field and its value but on a read-only basis.

To create a field whose value can only be read, precede its data type, during declaration, with the **readonly** keyword. Here is an example:

```
public readonly double PI;
```

After declaring the variable, you should initialize it. You have two main alternatives. You can initialize the field when declaring it. Here is an example:

Example

```
using System;
namespace Geometry
{
	class Circle
	{
		public double Radius;

		public Circle(double rad)
		{
			Radius = rad;
		}

		public readonly double PI = 3.14159;
	}
	class Exercise
	{
		static int Main()
		{
		   Circle circ = new Circle(24.72);

		    Console.WriteLine("Circle Characteristics");
		    Console.WriteLine("Radius: {0}", circ.Radius);
			Console.WriteLine("PI:    {0}\n", circ.PI);

			return 0;
		}
	}
}
```

OUTPUT:

```
Circle Characteristics
Radius: 24.72
PI:    3.14159
```

Object Creation

In C# objects are created using the **new** keyword. Actually it **new** is an operator, creates an object of the specified class and returns a reference to that object. Here is an example :

```
public class Exercise
{
   public void Welcome()
   {
            Console.WriteLine("This is Exercise class");
   }
}

public class Class1
{
   static void Main()
   {
            Exercise exo = new Exercise();
   }
}
```

Constructor and Destructor

When you declare a variable of a class, a special method must be called to initialize the members of that class. This method is automatically provided for every class and it is called a constructor.

Default Constructor

Whenever you create a new class, a constructor is automatically provided to it. This particular constructor is called the default constructor. You have the option of creating it or not. Although a constructor is created for your class, you can customize its behavior or change it tremendously.

A constructor holds the same name as its class and doesn't return any value, not even **void**. Here is an example:

Like every method, a constructor can be equipped with a body. In this body, you can access any of the member variables (or method(s)) of the same class. When introducing classes other than the main class, we saw that, to use such a class, you can declare its variable and allocate memory using the new operator. You can notice that we always included the parentheses when declaring such a variable. Here is an example:

```
public class Class1
{
   static void Main()
   {
   Exercise exo = new Exercise();
   }
}
```

In this case, the parentheses indicate that we are calling the default constructor to instantiate the class.

Consider the following Example:

Example

using System;

```
public class Exercise
{
   public void Welcome()
   {
          Console.WriteLine("The wonderful world of C# programming");
   }

   public Exercise()
   {
          Console.WriteLine("The Exercise class is now available");
   }
}

public class Class1
{
   static void Main()
   {
          Exercise exo = new Exercise();
   }
}
```

OUTPUT:

The Exercise class is now available

This shows that, when a class has been instantiated, its constructor is the first method to be called. For this reason, you can use a constructor to initialize a class, that is, to assign default values to its member variables. When a constructor is used to initialize a variable declared for a class. That constructor is referred to as an instance constructor.

Parameterized Constructor

In the previous section, we saw that there is always a default constructor for a new class that you create; you just the option of explicitly creating one or not. The default constructor as we saw it doesn't take arguments: this is not a rule, it is simply assumed. Instead of a default constructor, you may want to create a constructor that takes an argument. Here is an example:

```
using System;
public class Quadrilateral
{    public    Quadrilateral(double    side)
            {…….
            }
}
public class Class1
{
  static void Main()
  {

    Quadrilateral Square = new Quadrilateral(6.55);

   }
}
```

Static Constructor

Like the above described instance constructors, a static constructor is used to initialize a class. The main difference is that a static constructor works internally, in the class. Therefore, it is not used to initialize a variable of the class and you can never declare a variable of a class using a static constructor.

To make a constructor static, when creating it, type the **static** keyword to its left. Here is an example:

```
using System;

public class Quadrilateral
{
        static Quadrilateral()
        {

        }
}

public class Class1
{
        static void Main()
        {

           /* Use the default constructor to initialize
           an instance of the class */

           Quadrilateral Square = new Quadrilateral();
        }
}
```

In the above class, a static constructor is created for the class but the default constructor is still available and it is used to instantiate the class.

Constructor Overloading

A constructor is the primary method of a class. It allows the programmer to initialize a variable of a class when the class is instantiated. A constructor that plays this role of initializing an instance of a class is also called an instance constructor. Most of the time, you don't even need to create a constructor, since one is automatically provided to any class you create. Sometimes too, as we have seen in some classes, you need to create your own class as you judge it necessary. And sometimes, a single constructor may not be sufficient. For example, when creating a class, you may decide, or find out, that there must be more than one way for a user to initialize a variable.

Like any other method, a constructor can be overloaded. In other words, you can create a class and give it more than one constructor. The same rules used on overloading regular methods also apply to constructors: the different constructors must have different number of arguments or a different number of arguments.

Example

```
using System;
public class Applicant
{
        public string FullName;
        public string Address;
        public string City;
        public string State;
```

```
        public string ZIPCode;
        public string Sex;
        public string
        DateOfBirth; public int
        Weight; public string
        Height; public int Race;

        // The default constructor, used to initialize an
        // Applicant instance without much
        information public Applicant()
        {
                this.FullName = "Unknown";
                this.Sex = "Ungenerated";
        }

        // A constructor that is passed only one
        argument public Applicant(string n)
        {
                this.FullName = n;
        }

        // A constructor with more than one argument
        // This type is suitable to completely initialize a
        variable public Applicant(string n, string s, string dob)
        {
                this.FullName = n;
                this.Sex = s;
                this.DateOfBirth = dob;
        }
}

public class Exercise
{
  static Applicant RegisterPersonalInformation()
   {
        string name;
        char sex;
        string gender =
        null; string dob;


        Console.WriteLine(" ** Motor Vehicle Administration
        **"); Console.WriteLine("Applicant's Registration");
        Console.Write("Full Name: ");
        name = Console.ReadLine();

        do
        {
                Console.Write("Sex(F=Female/M=Male):
                "); sex = char.Parse(Console.ReadLine());

                if( (sex != 'f') && (sex != 'F') && (sex !=
                        'm') && (sex != 'M') )
                Console.WriteLine("Please enter a valid character");

        }while( (sex != 'f') && (sex != 'F') &&
              (sex != 'm') && (sex != 'M') );

                if( (sex == 'f') || sex == 'F' )
                        gender = "Female";
```

```
                    else if( (sex == 'm') || (sex == 'M') )
                            gender = "Male";

                    Console.Write("Date of Birth(mm/dd/yyyy): ");
                    dob = Console.ReadLine();

                    Applicant person = new Applicant(name, gender, dob);

                    return person;
            }

static void Show(Applicant person)
{
            Console.WriteLine("\n ** Motor Vehicle Administration **");
            Console.WriteLine(" --- Applicant's Personal Information ---");
            Console.WriteLine("Full Name:   {0}", person.FullName);
            Console.WriteLine("Sex:         {0}", person.Sex);
            Console.WriteLine("Date of Birth: {0}\n", person.DateOfBirth);
}

public static int Main()
{
            Applicant App = new Applicant();

            App = RegisterPersonalInformation();
            Show(App);

            return 0;
 }
}
```

OUTPUT:

** Motor Vehicle Administration
** Applicant's Registration
Full Name: Dominique Monay
Sex(F=Female/M=Male): d
Please enter a valid character
Sex(F=Female/M=Male): M
Date of Birth(mm/dd/yyyy): 06/10/1972

** Motor Vehicle Administration **
--- Applicant's Personal Information ---
Full Name: Dominique Monay
Sex: Male
Date of Birth: 06/10/1972

Destructor

While a constructor is a method used to initialize an instance of a class, a destructor is used to destruct an instance of class when that variable is not used anymore. Like the constructor, the destructor has the same name as the class. To indicate that the method is a destructor, its name is preceded with a tilde.

Example

```
using System;

class SampleClass
```

```
{
        // Constructor
        public SampleClass()
        {
                Console.WriteLine("SampleClass - Constructor");
        }

        ~SampleClass()
        {
                Console.WriteLine("Destructor of SampleClass");
        }
}

public class NewProject
{

        static void Main()
        {
                SampleClass Sample = new SampleClass();

                Console.WriteLine("Welcome");
        }
}
```

OUTPUT:

SampleClass -
Constructor Welcome
Destructor of SampleClass

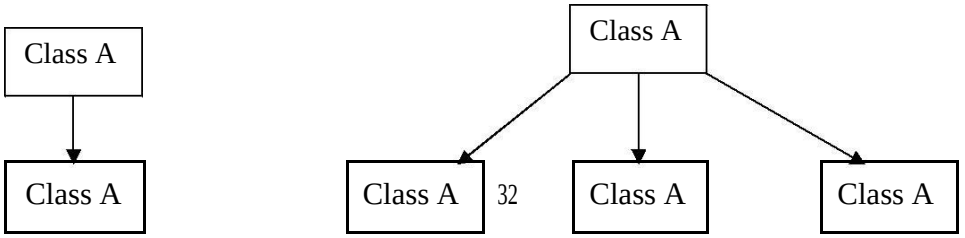
Like a (default) constructor, a destructor is automatically created for your class but you can also create it if you want. A class can have only one constructor. If you don't create it, the compiler would create it for your class. If you create it, the compiler would not create another. A destructor cannot have an access level. A destructor is called when the memory that a class was used is no longer needed. This is done automatically by the compiler. For this reason, you will hardly need to create a constructor, since its job is automatically taken care of behind the scenes by the compiler.

2.5 Inheritance and Polymorphism

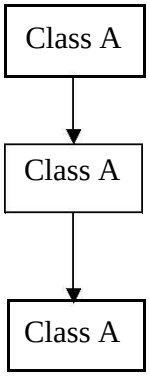
Inheritance means taking an existing class and adding functionality by deriving a new class from it.
The class you start with is called the *base class*, and the new class you create is called the *derived class*.
When you derive a class from another class, the new class gets all the functionality of the base class plus whatever new features you add. You can add data members and functions to the new class, but you cannot remove anything from what the base class offers.

In C# inheritance may be implemented in different combinations as illustrated in figure 6.1and they include :

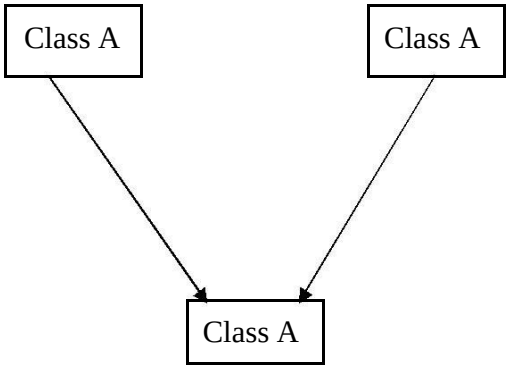
1) Single Inheritance
2) Multilevel Inheritance
3) Multiple Inheritance
4) Hierarchical Inheritance



Single inheritance



Hierarchical inheritance



Multilevel inheritance

Multiple inheritance

Figure

The Multiple inheritance does not directly implemented by C#. But we can implement the concept of multiple inheritance using interface.

Derived class definition

A derived class can be defined with specifying its relationship with the base class. The syntax and general form of derived class is:

```
class derived-class-name : base-class-name
 {
    members of
  class };
```

Class Modifiers

When implementing inheritance, it is important to understand how establish accessibility level for our classes and their members. Class modifier is used to decide which parts of the system can create class objects.

Basically there are four different - optional - class modifiers. These are :

- public
- internal
- protected
- private

These are used to specify the access levels of the types defined by the classes. The following different access levels can be specified with above four modifiers:

- protected internal
- new
- abstract
- sealed

public

The 'public' keyword identifies a type as fully accessible to all other types. This is the implicit accessibility of enumeration members and interface members.

internal

If a class is declared as 'internal', the type it defines is accessible only to types within the same assembly (a self-contained 'unit of packaging' containing code, metadata etc.). This is the default access level of non-nested classes.

protected

If a class is declared as 'protected', its type is accessible by a containing type and any type that inherits from this containing type. This modifier should only be used for internal classes (ie. classes declared within other classes).

private

Where a class is declared as 'private', access to the type it defines is limited to a containing type only. This modifier should only be used for internal classes (ie. classes declared within other classes).

protected internal

The permissions allowed by this access level are those allowed by the 'protected' level plus those allowed by the 'internal' level. The access level is thus more liberal than its parts taken individually. This modifier should only be used for internal classes (ie. classes declared within other classes).

new

The 'new' keyword can be used for 'nested' classes. A nested class is one that is defined in the body of another class; it is in most ways identical to a class defined in the normal way, but its access level cannot be more liberal than that of the class in which it is defined. A nested class should be declared using the 'new' keyword just in case it has the same name as (and thus overrides) an inherited type.

abstract

A class declared as 'abstract' cannot itself be instanced - it is designed only to be a base class for inheritance.

sealed

A class declared as 'sealed' cannot be inherited from other classes.

Abstract Classes

In C#, you can create a class whose role is only meant to provide fundamental characteristics for other classes. This type of class cannot be used to declare a variable of the object. Such a class is referred to as abstract. Therefore, an abstract class can be created only to serve as a parent class for others. To create an abstract class, type the **abstract** keyword to the left of its name. Here is an example:

```
abstract class Ball
{
        protected int TypeOfSport;
        protected string Dimensions;
}
```

The important features of abstract class are:

- The abstract class may contain the one or more abstract methods
- The abstract class cannot be instantiate in the other classes
- If your are inheriting the abstract class in other class, you should implement all the abstract methods in the derived class.

Example

```
using System;
abstract class First
{
        protected string s = " ";
        public abstract void SayHello();
}

class Second : First
{
        public override void SayHello()
        {
                s = "Hello World";
                Console.WriteLine("{0}",s);
        }
}

class mainclass
{
         public static void Main()
        {
                Second S = new Second();

                S.SayHello();
        }
}
```

OUTPUT:

Hello World

In class First we define abstract method called **SayHello().** But there is no implementation. The class Second derived from the class First and inherits the abstract class, where we overriding the **SayHello()** method. Overriding is discussed in the section of polymorphism in more detail.

Sealed Class

Any of the classes we have used so far in our lessons can be inherited from. If you create a certain class and don't want anybody to derive another class from it, you can mark it as sealed. In other words, a sealed class is one that cannot serve as base for another class.

To mark a class as sealed, type the **sealed** keyword to its left. Here is an

```
example: public sealed class Ball
{
        public int TypeOfSport;
        public string Dimensions;
}
```

Polymorphism

Polymorphism is a feature to use one name in many forms. There Polymorphism can be achieved in following ways in c#:

- Method Overloading

- Method Overriding
- Method Hiding

Method overloading ,discussed in chapter 5, means one method name with different arguments. Method overriding and hiding makes use of the following three method-head keywords

- new
- virtual,
- override

The main difference between hiding and overriding relates to the choice of which method to call where the declared class of a variable is different to the run-time class of the object it references. This point is explained further below.

Method Overriding

Suppose that we define a Square class which inherits from a Rectangle class (a square being a special case of a rectangle). Each of these classes also specifies a 'getArea' instance method, returning the area of the given instance.

For the Square class to 'override' the Rectangle class' getArea method, the Rectangle class' method must have first declared that it is happy to be overridden. One way in which it can do this is with the **'virtual'** keyword. So, for instance, the Rectangle class' getArea method might be specified like this:

```
public virtual double getArea()
   return length * width;
}
```

To override this method the Square class would then specify the overriding method with the **'override'** keyword. For example:

```
public override double getArea()
{
   return length * length;
}
```

Note that for one method to override another, the overridden method must not be static, and it must be declared as either **'virtual', 'abstract'** or **'override**'. Furthermore, the access modifiers for each method must be the same.

The major implication of the specifications above is that if we construct a new Square instance and then call its 'getArea' method, the method actually called will be the Square instance's getArea method. So, for instance, if we run the following code:

```
Square sq = new Square(5);
double area = sq.getArea();
```

then the getArea method called on the second line will be the method defined in the Square class. There is, however, a more subtle point. To show this, suppose that we declare two variables in the following way:

```
Square sq = new Square(4);
Rectangle r = sq;
```

Here variable r refers to sq as a Rectangle instance (possible because the Square class derives from the Rectangle class). We can now raise the question: if we run the following code

```
double area = r.getArea();
```

then which getArea method is actually called - the Square class method or the Rectangle class method?

The answer in this case is that the Square class method would still be called. Because the Square class' getArea method 'overrides' the corresponding method in the Rectangle class, calls to this method on a Square instance always 'slide through' to the overriding method.

Method Hiding

Where one method 'hides' another, the hidden method does not need to be declared with any special keyword. Instead, the hiding method just declares itself as **'new'**. So, where the Square class hides the Rectangle class' getArea method, the two methods might just be written thus:

```
public double getArea() // in Rectangle
{
   return length * width;
}
```

```
public new double getArea() // in Square
{
   return length * length;
}
```

Note that a method can 'hide' another one without the access modifiers of these methods being the same. So, for instance, the Square's getArea method could be declared as private, viz:

```
private new double getArea()
{
   return length * length;
}
```

This leads us to an important point. A 'new' method only hides a super-class method with a scope defined by its access modifier. Specifically, where the access level of the hiding method is 'private', as in the method just described, this method only hides the super-class method for the particular class in which it is defined.

To make this point more concrete, suppose that we introduced a further class, SpecialSquare, which inherits from Square. Suppose further that SpecialSquare does not overwrite the getArea method. In this case, because Square's getArea method is defined as private, SpecialSquare inherits its getArea method directly from the Rectangle class (where the getArea method is public).

The final point to note about method hiding is that method calls do not always 'slide through' in the way that they do with virtual methods. So, if we declare two variables thus:

```
Square sq = new Square(4);

Rectangle r = sq;
```

then run the code

```
double area = r.getArea();
```

the getArea method run will be that defined in the Rectangle class, not the Square class.

Properties

This Chapter teaches C# Properties. Our objectives are as follows:

- Understand What Properties Are For.
- Implement a Property.
- Create a Read-Only Property.
- Create a Write-Only Property.
- Create an auto-implemented property.

Overview of Properties

Properties provide the opportunity to protect a field in a class by reading and writing to it through the property. In other languages, this is often accomplished by programs implementing specialized getter and setter methods. C# properties enable this type of protection while also letting you access the property just like it was a field.

Another benefit of properties over fields is that you can change their internal implementation over time. With a public field, the underlying data type must always be the same because calling code depends on the field being the same. However, with a property, you can change the implementation. For example, if a customer has an ID that is originally stored as an int, you might have a requirements change that made you perform a validation to ensure that calling code could never set the ID to a negative value. If it was a field, you would never be able to do this, but a property allows you to make such a change without breaking code. Now, lets see how to use properties.

Creating Read-Only Properties

Properties can be made read-only. This is accomplished by having only a *get* accessor in the property implementation.

Creating a Write-Only Property

You can assign values to, but not read from, a write-only property. A write-only property only has a *set* accessor.

Indexers

Indexers are real easy. They allow your class to be used just like an array. On the inside of a class, you manage a collection of values any way you want. These objects could be a finite set of class members, another array, or some complex data structure. Regardless of the internal implementation of the class, its data can be obtained consistently through the use of indexers. Here's an example.

An Example

```
using System;

/// <summary>
///     A simple indexer example.
/// </summary>
class IntIndexer
{
   private string[] myData;

   public IntIndexer(int size)
   {
      myData = new string[size];

      for (int i=0; i < size; i++)
      {
```

```
            myData[i] = "empty";
        }
    }
    public string this[int pos]
    {
        get
        {
            return myData[pos];
        }
        set
        {
            myData[pos] = value;
        }
    }

    static void Main(string[] args)
    {
        int size = 10;

        IntIndexer myInd = new IntIndexer(size);

        myInd[9] = "Some Value";
        myInd[3] = "Another Value";
        myInd[5] = "Any Value";

        Console.WriteLine("\nIndexer Output\n");

        for (int i=0; i < size; i++)
        {
            Console.WriteLine("myInd[{0}]: {1}", i, myInd[i]);
        }
    }
}
```

Overloaded Indexers that accept different types

Additionally, Indexers can be overloaded. In listing 11-2, we modify the previous program to accept **overloaded Indexers that accept different types.**

Listing 11-2. Overloaded Indexers:

OvrIndexer.cs using System;

```
/// <summary>
///     Implements overloaded indexers.
/// </summary>
class OvrIndexer
{
    private string[] myData;
    private int arrSize;

    public OvrIndexer(int size)
    {
```

```
    arrSize = size;
    myData = new string[size];

    for (int i=0; i < size; i++)
    {
        myData[i] = "empty";
    }
}

public string this[int pos]
{
    get
    {
        return myData[pos];
    }
    set
    {
        myData[pos] = value;
    }
}

public string this[string data]
{
    get
    {
        int count = 0;

        for (int i=0; i < arrSize; i++)
        {
            if (myData[i] == data)
            {
                count++;
            }
        }
        return count.ToString();
    }
    set
    {
        for (int i=0; i < arrSize; i++)
        {
            if (myData[i] == data)
            {
                myData[i] = value;
            }
        }
    }
}

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

```
        int size = 10;
        OvrIndexer myInd = new OvrIndexer(size);

        myInd[9] = "Some Value";
        myInd[3] = "Another Value";
        myInd[5] = "Any Value";

        myInd["empty"] = "no value";

        Console.WriteLine("\nIndexer Output\n");

        for (int i=0; i < size; i++)
        {
            Console.WriteLine("myInd[{0}]: {1}", i, myInd[i]);
        }

        Console.WriteLine("\nNumber of \"no value\" entries: {0}", myInd["no value"]);
    }
}
```

An indexer with multiple parameters

An Indexer signature is specified by the number and type of parameters in an Indexers parameter list. The class will be smart enough to figure out which Indexer to invoke, based on the number and type of arguments in the Indexer call. An indexer with multiple parameters would be implemented something like this:

```
    public object this[int param1, ..., int paramN]
    {
        get       {
            // process and return some class data       }
        set       {
            // process and assign some class data       }
    }
```

ref ,out and params

The **out** method parameter keyword on a method parameter causes a method to refer to the same variable that was passed into the method. Any changes made to the parameter in the method will be reflected in that variable when control passes back to the calling method.

Declaring an **out** method is useful when you want a method to return multiple values. A method that uses an **out** parameter can still return a value. A method can have more than one **out** parameter.

To use an **out** parameter, the argument must explicitly be passed to the method as an **out** argument. The value of an **out** argument will not be passed to the **out** parameter.

A variable passed as an **out** argument need not be initialized. However, the **out** parameter must be assigned a value before the method returns.

A property is not a variable and cannot be passed as an **out** parameter.

An overload will occur if declarations of two methods differ only in their use of **out**. However, it is not possible to define an overload that only differs by **ref** and **out**. For example, the following overload declarations are valid:

```
class MyClass
{
  public void MyMethod(int i) {i = 10;}
  public void MyMethod(out int i) {i = 10;}
```

```
}
```
while the following overload declarations are
invalid: class MyClass
```
{
  public void MyMethod(out int i) {i = 10;}
  public void MyMethod(ref int i) {i = 10;}
}
```
Example:
```
// cs_out.cs
using System; public
class MyClass
{
  public static int TestOut(out char i)
  {
    i = 'b';
    return -1;
  }

  public static void Main()
  {
    char i; // variable need not be initialized
    Console.WriteLine(TestOut(out i));
    Console.WriteLine(i);
  }
}
```
Output:
```
-1
B
```

ref Keyword:

The ref method parameter keyword on a method parameter causes a method to refer to the same variable that was passed into the method. Any changes made to the parameter in the method will be reflected in that variable when control passes back to the calling method.

```
using System; public
class MyClass
{
  public static void TestRef(ref char i)
  {
    // The value of i will be changed in the calling
    method i = 'b';
  }

  public static void TestNoRef(char i)
  {
    // The value of i will be unchanged in the calling
    method i = 'c';
  }

    public static void Main()
  {
```

```
    char i = 'a'; // variable must be initialized
    TestRef(ref i); // the arg must be passed as
    ref Console.WriteLine(i);
    TestNoRef(i);
    Console.WriteLine(i);
  }
}
```

OUTPUT:

b

b

Parameter Arrays(Params)

- A parameter declared with a params modifier is a parameter array. If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type.
- For example, the types string[] and string[][] can be used as the type of a parameter array, but the type string[,] cannot.
- It is not possible to combine the params modifier with the modifiers ref and out.
- A parameter array permits arguments to be specified in one of two ways in a method invocation:
  - The argument given for a parameter array can be a single expression of a type that is implicitly convertible to the parameter array type.
  - Alternatively, the invocation can specify zero or more arguments for the parameter array, where each argument is an expression of a type that is implicitly convertible to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

```
using System;
 class Test {
static void F(params int[] args) {
Console.Write("Array contains {0} elements:",
 args.Length); foreach (int i in args)
 Console.Write(" {0}",
i); Console.WriteLine();
}
static  void  Main()
{ int[] arr = {1, 2,
3}; F(arr);
F(10, 20, 30,
 40); F();
 } }
```

2.6 Operator Overloading

Overloading

Another important and exciting feature object-oriented programming is Operator overloading. C# supports the concept of operator overloading. Operator overloading is a concept in which operator

can defined to work with the user-defined data types such as structs and classes in the same way as the pre-defined data types.

Example using
System; public
class Item
{
public int i;
public Item(int j)
{
i = j;
}

public static Item operator + (Item x , Item y)

{
System.Console.WriteLine("operator +" + x.i +" "+
y.i); Item z = new Item(x.i+y.i);
return z;
}
}

public class Test
{
public static void Main()
{ Item a = new Item(10);
Item b = new Item(5);
Item c;
c = a + b ;
System.Console.WriteLine(c.i);
}
}

Output:
operator + 10 5
15

There are many operators that cannot be overloaded. Which are listed below:

- **Conditional Operator** & &, ||
- **Compound Assignment** +=, -=, *=, /=, %=
- **Other Operators** [], (), = , ?:, ->, new, sizeof, types of, is, as

2.7 Interfaces, Delegates and Events.

Interface

Imagine you start creating a class and, while implementing or testing it, you find out that this particular class can be instead as a general base that other classes can be derived from. An

interface is a special class whose purpose is to serve as a template that actual classes can be based on.
An interface is primarily created like a class: it has a name, a body and can have members. To create an interface, instead of the class keyword, you use the interface keyword. By convention, the name of an interface starts with I. Here is an example:

```
interface ICourtDimensions
{
}
```

An interface is mostly used to lay a foundation for other classes. For this reason, it is the prime candidate for class derivation. To derive from an interface, use the same technique we have applied in inheritance so far. Here is an example of a class named SportBall that derives from an interface named ISportType:

```
public class SportBall : ISportType
{
        int players;
        string sport;
}
```

Just as you can derive a class from an interface, you can create an interface that itself is based on another interface. Here is an example:

```
public interface ISportType : IBall
{
}
```

The C# language doesn't allow multiple inheritance which is the ability to create a class based on more than one class. Multiple inheritance is allowed only if the bases are interfaces. To create multiple inheritance, separate the names of interface with a comma. Here is an example:

```
public interface ISportType : IBall, ICourtDimensions
{
}
```

You can also involve a class as parent in a multiple inheritance scenario but there must be only one class. Here is an example in which a class called Sports derives from one class and various interfaces:

```
public interface Sports: Player, IBall, ICourtDimensions
{
}
```

Example

```
// implementation of multiple
interface using System;

interface Add
{
        int sum();
}
interface Multiply
{
        int mul();
}
```

```
class calculate : Add, Multiply
{
        int a,b;
        public calculate( int x, int y)
        {
                a = x;
                b = y;
        }
                public int sum()
        {
                return ( a + b );
        }
        public int mul()
        {
                return ( a * b );
        }
}
class MyInterface
{
        public static void Main()
        {

                calculate cal = new calculate(5,10);

                Add A = (Add) cal;              // casting
                Console.WriteLine("Sum : " + A.sum());

                Multiply M = (Multiply) cal;
                Console.WriteLine("Multiplication :" + M.mul());
        }
}
```

OUTPUT:

Sum : 15
Multiplication : 50

Delegates

The C and C++ languages are having the concept of function pointer. This was even more useful when programming for the Microsoft Windows operating systems because the Win32 library relies on the concept of callback functions. Callback functions are used in Microsoft Windows programming to process messages. For this reason and because of their functionality, callback functions were carried out in the .NET Framework but they were defined with the name of delegate.

A delegate is a special type of user-defined variable that is declared globally, like a class. In fact, a delegate is created like an interface but as a method. Based on this, a delegate provides a **template** for a method, like an interface provides a template for a class. Like an interface, a delegate is not defined. Its role is to show what a useful method would look like. To support this concept, a delegate can provide all the necessary information that would be used on a method. This includes a return type, no argument or one or more arguments.

Delegate Declaration and Instantiation

Delegates can be specified on their own in a namespace, or else can be specified within another class. In each case, the declaration specifies a new class, which inherits from **System.MulticastDelegate**.

Each delegate is limited to referencing methods of a particular kind only. The type is indicated by the delegate declaration - the input parameters and return type given in the delegate declaration must be shared by the methods its delegate instances reference. To illustrate this: a delegate specified as below can be used to refer only to methods which have a single String input and no return value:

```
public delegate void Print (String s);
```

Suppose, for instance, that a class contains the following method:

```
public void realMethod (String myString)
{
   // method code
}
```

Another method in this class could then instantiate the 'Print' delegate in the following way, so that it holds a reference to 'realMethod':

```
Print delegateVariable = new Print(realMethod);
```

We can note two important points about this example. Firstly, the unqualified method passed to the delegate constructor is implicitly recognised as a method of the instance passing it. That is, the code is equivalent to:

```
Print delegateVariable = new Print(this.realMethod);
```

We can, however, in the same way pass to the delegate constructor the methods of other class instances, or even static class methods. In the case of the former, the instance must exist at the time the method reference is passed. In the case of the latter (exemplified below), the class need never be instantiated.

```
Print delegateVariable = new Print(ExampleClass.exampleMethod);
```

Example

```
using System;
delegate int operation(int x, int y)
{
        delegate int Operation(int x, int
 y); class MathOpr
        {
        public static int Add(int a, int b)
                {
                        return(a + b);
                }
        public static int Sub(int a, int b)
                {
                        return( a – b);
                }
        public static int Mul(int a, int b)
                {
                        return( a * b);
```

```
        }
    }
    class Test
    {
        Operation opr1 = new Operation (Mathopr.Add);
        Operation opr2 = new Operation (Mathopr.Sub);
        Operation opr3 = new Operation (Mathopr.Mul);

        //invoking of delegates int
        ans1 = opr1(200, 100); int
        ans2 = opr2(200, 100); int
        ans3 = opr3(20,10);

        Console.WriteLine("\n Addition :"+ ans1);
        Console.WriteLine("\n Subtraction :"+ ans2);
        Console.WriteLine("\n multiplication :"+ ans3);
    }
}
```

OUTPUT:

Adiition : 300
Subtraction :100
Multiplication : 200

Multicast Delegates

The second thing to note about the example is that all delegates can be constructed in this fashion, to create a delegate instance which refers to a single method. However, as we noted before, some delegates - termed '**multicast delegates**' - can simultaneously reference multiple methods. These delegates must - like our Print delegate - specify a 'void' return type.
One manipulates the references of multicast delegates by using addition and subtraction .
The following code gives some examples:

```
Print s = null;
s = s + new Print (realMethod);
s += new Print (otherRealMethod);
```

The - and -= operators are used in the same way to remove method references from a delegate. The following code gives an example of the use of multicast delegates.

Example

```
using System;
delegate void
MultiDel(); class MD
{ static public void Hello()
    {
        Console.WriteLine("Hello");
    }
    static public void Show()
    {
        Console.WriteLine(" Hi");
    }
}
```

```
class Test
{
        public static void Main()
        {
                //delegate instances
                MultiDel M1 = new MultiDel(MD.Display);
                MultiDel M2 = new MultiDel(MD.Show);

                //combine the delegates
                MultiDel M3 = M1 + M2;
                MultiDel M4 = M2 + M1;

                //extracting the delegates
                MultiDel M5 = M3 – M2;
                MultiDel M6 = M4 – M1;

                //invoking
                delegates M3();
                M4();
                M5();
                M6();
        }
}
```

OUTPUT:

```
Hello
Hi

Hi
Hello

Hello

Hi
```

Events

In object-oriented languages, objects expose encapsulated functions called methods. Methods are encapsulated functions which run when they are invoked.

Sometimes, however, we think of the process of method invocation more grandly. In such a case, the method invocation is termed an 'event', and the running of the method is the 'handling' of the event. An archetypal example of an event is a user's selection of a button on a graphical user interface; this action may trigger a number of methods to 'handle' it.

What distinguishes events from other method invocations is not, however, that they must be generated externally. Any internal change in the state of a program can be used as an event. Rather, what distinguishes events is that they are backed by a particular 'subscription-notification' model. An arbitrary class must be able to 'subscribe to' (or declare its interest in) a particular event, and then receive a 'notification' (ie. have one of its methods run) whenever the event occurs.

Delegates (in particular multicast delegates) are essential in realizing this subscription-notification model. The following example describes how Class 2 subscribes to an event issued by Class 1.

1. Class 1 is an issuer of E-events. It maintains a public multicast delegate D.

2. Class 2 wants to respond to E-events with its event-handling method M. It therefore adds onto D a reference to M.
3. When Class 1 wants to issue an E-event, it calls D. This invokes all of the methods which have subscribed to the event, including M.
The 'event' keyword is used to declare a particular multicast delegate (in fact, it is usual in the literature to just identify the event with this delegate). The code below shows a class EventIssuer, which maintains an event field myEvent. We could instead have declared the event to be a property instead of a field . To raise the myEvent event, the method onMyEvent is called (note that we are checking in this method to see if myEvent is null - trying to trigger a null event gives a run-time error).

```
public class EventIssuer
{
public delegate void EventDelegate(object from, EventArgs
   args); public event EventDelegate myEvent;
   protected virtual void onMyEvent(EventArgs args)
   {
      if (myEvent!=null)
         myEvent(this, args);
   }
```

A class which wanted to handle the events issued by an EventIssuer ei with its method handleEvents would then subscribe to these events with the code:
ei.myEvent += new EventIssuer.EventDelegate(handleEvents);

Tips For Events
The code above demonstrates some points about event-handling which are not enforced by the language architecture, but are used throughout the .Net framework as good practice.
1. When you want to raise an event in code, you don't tend to trigger the class's event object directly. Rather, you call a 'protected, virtual' method to trigger it (cf. the onMyEvent method above).
2. By convention, when events are raised they pass two objects to their subscribers. The first is a reference to the class raising the event; the second is an instance of the System.EventArgs class which contains any arbitrary data about the event.
3. If an event is not interested in passing data to subscribers, then its defining delegate will still reference an EventArgs object (but a null value will be passed by the event). If an event should pass data to its subscribers, however, then it is standard to use a specific class which derives from the EventArgs class to hold this data.
4. When you write a class which inherits from an event-raising base class, you can 'intercept' an event by overriding the method used to raise it. The following code illustrates such an intercept - classes which subscribe to the event will never receive notifications about it.

```
protected override void onMyEvent(EventArgs args)
   {
      Console.WriteLine("hello");  }
```

If you want subscribers to continue to receive notifications despite such an 'intercepting' method, however, then you can call the base class method as in the following:

```
   protected override void onMyEvent(EventArgs
   args) { Console.WriteLine("hello");
      base.onMyEvent(args);  }
```

2.8 Type conversion

There are two types of conversions:

1. Implicit Conversion
2. Explicit Conversion

Implicit Conversion : In implicit conversion the compiler will make conversion for us without asking.

char -> int -> float is an example of data compatibility. Complier checks for type compatibility at compilation.

Explicit Conversion: In explicit conversion we specifically ask the compiler to convert the value into another data type.CLR checks for data compatibility at runtime.Explicit conversion is carried out using casts. When we cast one type to another, we deliberately force the compiler to make the transformation.Casting of big data type into small may lead to loosing of data.

Microsoft .NET provides three ways of type conversion:

1. **Parsing(int.Parse()**
2. **Convert Class(Convert.ToInt32())**
3. **Explicit Cast Operator ()**

Parsing

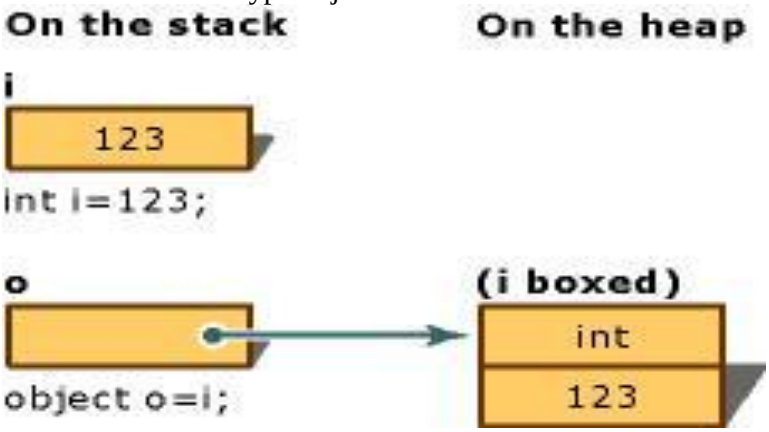
Parsing is used to convert string type data to primitive value type. For this we use parse methods with value types.

Convert Class: One primitive type to another primitive type.

This class contains different static methods like ToInt32(), ToInt16(), ToString(), ToDateTime() etc used in type conversion.

Boxing and unboxing

Boxing and unboxing is an important concept in C# type system. With Boxing and unboxing one can link between value-types and reference-types by allowing any value of a value-type to be converted to and from type object.



Boxing

- Boxing is a mechanism in which value type is converted into reference type.
- It is implicit conversion process in which object type (super type) is used.
- In this process type and value both are stored in object type

Unboxing

- Unboxing is a mechanism in which reference type is converted into value.
- It is explicit conversion process.

Example

```
int     i, j;
object  obj;
string  s;
i  = 32;
obj = i;     // boxed copy!
i   = 19;
j   = (int) obj; // unboxed!
s = j.ToString();  // boxed!
s = 99.ToString(); // boxed!
```

Difference Between Int32.Parse(), Convert.ToInt32(), and Int32.TryParse(),(int)

Int32.Parse (string s) method converts the string representation of a number to its 32-bit signed integer equivalent.

– When s is a null reference, it will throw ArgumentNullException.
– If s is other than integer value, it will throw FormatException.
– When s represents a number less than MinValue or greater than MaxValue, it will throw OverflowException.

- (int) will only convert types that can be represented as an integer (ie double, long, float, etc) although some data loss may occur.
- string s1 = "1234";
- string s2 = "1234.65";
- string s3 = null;
- string s4 =123456789123456789123456789123456789123456789";
- int result;
- bool success;
- result = Int32.Parse(s1); *//-- 1234*
- result = Int32.Parse(s2); *//-- FormatException*
- result = Int32.Parse(s3); *//-- ArgumentNullException*
- result = Int32.Parse(s4); *//-- OverflowException*

Convert.ToInt32(string):This method converts the specified string representation of 32-bit signed integer equivalent. This calls in turn **Int32.Parse** () method.

– When s is a null reference, it will return 0 rather than throw ArgumentNullException.
– If s is other than integer value, it will throwFormatException.
– When s represents a number less than MinValue or greater than MaxValue, it will throw OverflowException. For example:

- result = Convert.ToInt32(s1); *//-- 1234*
- result = Convert.ToInt32(s2); *//-- FormatException*
- result = Convert.ToInt32(s3); *//-- 0*
- result = Convert.ToInt32(s4); *//-- OverflowException*

Int32.TryParse(string, out int): This method converts the specified string representation of 32-bit signed integer equivalent to out variable, and returns true if it is parsed successfully, false otherwise.

– When s is a null reference, it will return 0 rather than throwArgumentNullException.
– If s is other than an integer value, the out variable will have 0 rather thanFormatException.
– When s represents a number less than MinValue or greater than MaxValue, the outvariable will have 0 rather than OverflowException.
– success = Int32.TryParse(s1, out result); *//-- success => true; result => 1234*

- success = Int32.TryParse(s2, out result); *//-- success => false; result => 0*
- success = Int32.TryParse(s3, out result); *//-- success => false; result => 0*
- success = Int32.TryParse(s4, out result); *//-- success => false; result => 0*
- Convert.ToInt32 is better than **Int32.Parse** since it returns 0 rather than an exception. But again, according to the requirement, this can be used. TryParse will be the best since it always handles exceptions by itself.

Unit-III
C# Using Libraries

3.1 Namespace- System

System Namespace in fundamental namespace for c# application. It contain all the fundamental classes and base classes which are required in simple C# application. These classes and sub classes defines reference data type, method and interfaces. Some classes provide some other feature like data type conversion, mathematical function.

Some functionality provided by System namespace

- Commonly-used value
- Mathematics
- Remote and local program invocation
- Application environment management
- Reference data types
- Events and event handlers
- Interfaces Attributes Processing exceptions
- Data type conversion
- Method parameter manipulation

Some Classes provide by System namespace

- AccessViolationException
- Array
- ArgumentNullException
- AttributeUsageAttribute
- Buffer
- Console
- Convert
- Delegate
- Exception
- InvalidCastException

Some interfaces provided by System namespace

- Public interface ICloneable
- Public interface IComparable
- Public interface IComparable<T>
- Public interface IConvertible
- Public interface ICustomFormatter
- Public interface IDisposable
- Public interface IEquatable<T>
- Public interface IFormatProvider

MATH EXAMPLE

```
using System; class
Pythagorean { static
void Main() {
double s1;
double s2;
double hypot;
string str;
Console.WriteLine("Enter length of first side:
"); str = Console.ReadLine();
s1 = Double.Parse(str);
Console.WriteLine("Enter length of second side: ");
```

```
str = Console.ReadLine();
s2 = Double.Parse(str);
hypot = Math.Sqrt(s1*s1 + s2*s2);
Console.WriteLine("Hypotenuse is " + hypot);
}
}
```

Sorting and Searching,Reverse,Copy Arrays

Using Sort( ), you can sort an entire array, a range within an array, or a pair of arrays that contain corresponding key/value pairs. Once an array has been sorted, you can efficiently search it using BinarySearch( ).

```
using System; class
SortDemo { static
void Main() {
int[] nums = { 5, 4, 6, 3, 14, 9, 8, 17, 1, 24, -1, 0
}; Console.Write("Original order: ");
foreach(int i in nums)
Console.Write(i + " ");  Console.WriteLine();
Array.Sort(nums);
Console.Write("Sorted order: ");
foreach(int i in nums)
Console.Write(i + " ");   Console.WriteLine();
int idx = Array.BinarySearch(nums, 14);
Console.WriteLine("Index of 14 is " + idx); } }
```

The IComparable and IComparable<T> Interfaces

- Many classes will need to implement either the IComparable or IComparable<T> interface because they enable one object to be compared to another (for the purpose of ordering) by various methods defined by the .NET Framework
- **IComparable is especially easy to implement because it consists of just this one method:**
- int CompareTo(object *obj)*
- This method compares the invoking object against the value in *obj*. *It returns greater than zero* if the invoking object is greater than *obj, zero if the two objects are equal, and less than* zero if the invoking object is less than *obj*.

StringBuilder in C#

Once created a string cannot be changed. A StringBuilder can be changed as many times as necessary. It yields astonishing performance improvements. It eliminates millions of string copies. Many C# programs append or replace strings in loops. There the StringBuilder type becomes a necessary optimization. It uses the new keyword for StringBuilder. Use the new keyword to make your StringBuilder. This is different from regular strings. StringBuilder has many overloaded constructors. continuing on it calls the instance Append method. This method adds the contents of its arguments to the buffer in the StringBuilder. Every argument to StringBuilder will automatically have its ToString method called. It calls AppendLine, which does the exact same thing as Append, except with a new line on the end. Next, Append and Append Line call themselves. This shows terse syntax with StringBuilder. Finally ToString returns the buffer. You will almost always want ToString. It will return the contents as a string.

Example

```
using System;
using System.Text;
class Program
{
   static void Main()
   {
      StringBuilder builder = new StringBuilder();
      // Append to StringBuilder.
      for (int i = 0; i < 10; i++)
      {
         builder.Append(i).Append(" ");
      }
      Console.WriteLine(builder);
   }
}
```

3.2 Input-Output

C# programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ. Thus, the I/O classes and methods can be applied to many types of devices. For example, the same methods that you use to write to the console can also be used to write to a disk file.

Byte Streams and Character Streams

At the lowest level, all C# I/O operates on bytes. This makes sense because many devices are byte oriented when it comes to I/O operations. Frequently, though, we humans prefer to communicate using characters. Recall that in C#, **char** is a 16-bit type, and **byte** is an 8-bit type. If you are using the ASCII character set, then it is easy to convert between **char** and **byte**; just ignore the high-order byte of the **char** value. But this won't work for the rest of the Unicode characters, which need both bytes (and possibly more). Thus, byte streams are not perfectly suited to handling character-based I/O. To solve this problem, the .NET Framework defines several classes that convert a byte stream into a character stream, handling the translation of **byte**-to-**char** and **char**-to-**byte** for you automatically.

The Predefined Streams

Three predefined streams, which are exposed by the properties called **Console.In**, **Console.Out**, and **Console.Error**, are available to all programs that use the **System** namespace. **Console.Out** refers to the standard output stream. By default, this is the console. When you call **Console.WriteLine( )**, for example, it automatically sends information to **Console.Out**. **Console.In** refers to standard input, which is, by default, the keyboard. **Console.Error** refers to the standard error stream, which is also the console by default. However, these streams can be redirected to any compatible I/O device. The standard streams are character streams. Thus, these streams read and write characters.

System.IO Namespace

- **BinaryReader Class:** Reads primitive data types as binary values in a specific encoding.
- **BinaryWriter Class :** Writes primitive types in binary to a stream and supports writing strings in a specific encoding.
- **BufferedStream Class :** Adds a buffering layer to read and write operations on another stream. This class cannot be inherited.

- **Directory Class:** Exposes static methods for creating, moving, and enumerating through directories and subdirectories. This class cannot be inherited.
- **DirectoryInfo Class:** Exposes instance methods for creating, moving, and enumerating through directories and subdirectories. This class cannot be inherited.
- **File Class:**Provides static methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects.
- **FileInfo :**Provides properties and instance methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects. This class cannot be inherited.
- **FileStream**: Exposes a Stream around a file, supporting both synchronous and asynchronous read and write operations.
- **IOException** Class
- **Path** Class
- **Stream** Class: Provides a generic view of a sequence of bytes.
- **StreamReader:** Implements a TextReader that reads characters from a byte stream in a particular encoding.
- **StreamWriter :** Implements a TextWriter for writing characters to a stream in a particular encoding.
- **StringReader :**Implements a TextReader that reads from a string.
- **StringWriter:** Implements a TextWriter for writing information to a string. The information is stored in an underlying StringBuilder.
- **TextReader:** Represents a reader that can read a sequential series of characters.
- **TextWriter:**Represents a writer that can write a sequential series of characters. This class is abstract.
  –

Creating and writing text on a File

```
namespace IOTest{
 class Program  {
      static void Main(string[] args)      {
         StreamWriter sw;
        sw= File.CreateText("d:/workspace/Hello.txt");
       sw.WriteLine("Hello Mca Students This is your basic
       IO"); //sw.Flush();
        //sw.Close();
        Console.WriteLine("Please show the file in d drive ");
      } } }

class TextFileWriter  {
      static void Main(string[] args)      {
         TextWriter tw = new StreamWriter("date.txt");
         tw.WriteLine(DateTime.Now);
        tw.Close();
      }  }
class TextFileReader  {
      static void Main(string[] args)      {
         Textreader tr = new StreamReader("date.txt");
         Console.WriteLine(tr.ReadLine()); tr.Close();

      }
   }
```

Listing A Directory

```
class DirTest1{
static void Main(){
System.Console.WriteLine("sub directries in this directory
        "); string[] dirs = Directory.GetDirectories("C:\\");
        int count = dirs.Length;
        for (int i=0; i<count; i++)
        System.Console.WriteLine(dirs[i]);
 System.Console.WriteLine("Files in this dierectory are");
        string[] files = Directory.GetFiles("C:\\");
        int count1 = files.Length;
        for (int i=0; i<count1; i++)
        System.Console.WriteLine(files[i]);
}}
```

Formatted Strings in C#

The parameter placeholder {0}, {1} etc in Console.Write is handled by the string formatting. The placeholder has to have the index of the parameter but can also include formatting information. This is used in Console.WriteLine and also string.format hence its inclusion here.

Layout of the PlaceHolder

The three parts are

index, alignment : format

So {0:X} (x means Hexadecimal) or {0,10} meaning output in a width of 10. The alignment value specifies teh width and left or right alignment by using - (left aligned) or + (right aligned) numbers. 10 Means right aligned in a width of 10, -6 means left aligned in a width of 6. Alignment is ignored if the output exceeds the width.

This provides a large number of formatting examples.

List of Numeric Formats

- C or C - For Currency. Uses the cultures currency symbol.
- D or D - Integer types. Add a number for 0 padding eg D5.
- E or e - Scientific notation.
- F or f - Fixed Point.
- G or g - Compact fixed-point/scientific notation.
- N or n - Number. This can be enhanced by a NumberFormatInfo object.
- P or p - Percentage.
- R or r - Round-trip. Keeps exact digits when converted to string and back.
- X or x - Hexadecimal. x - uses abcdef, X use ABCDEF.

Dates can also be specified either using Standard Format strings

- O or o - YYYY-MM-dd:mm:ss:fffffffzz
- R or r - RFC1123 eg ddd, dd MMM yyyy HH:ss GMT
- s - sortable . yyyy-MM-ddTHH:mm:ss
- u - Universal Sort Date - yyyy-MM-dd HH:mm:ssZ or Format specifiers.

Example

```
using System;
using System.Text;
using System.Globalization;
```

```
namespace ex6
{
   class Program
   {
      static void Main(string[] args)
      {
         // Numeric Formatting Examples
         // integer
   int i = 5678;
   string s = string.Format("{0,10:D}", i); // Into a string right aligned 10 width
   Console.WriteLine("{0,10:D}", i);       // or output direct
   Console.WriteLine("{0,10:D7}", i);       // or output direct with leading 0

         // double and currency in various
  formats double d=47.5;
  double bigd = 19876543.6754;
  Console.WriteLine("{0,15:C2}", d); // In Uk = £47.50 right aligned in 15 width

  Console.WriteLine("{0,15:N10}", bigd); // Number
  Console.WriteLine("{0,15:E3}", d); // Scientific 4.750E+001
  Console.WriteLine("{0,15:F5}", d); // Fixed Point 47.50000
  Console.WriteLine("{0,15:G4}", d); // Compact 47.5
  Console.WriteLine("{0,10:P2}", d/100.0); // %
  Console.WriteLine("{0,15:R}", bigd); // Roundtrip - not a digit lost

         // Hex-a-diddly-decimal
  Console.WriteLine("{0,10:x8}", i); // lowercase 0000162e
  Console.WriteLine("{0,10:X8}", i); // uppercase 0000162E


         // Date formats
  DateTime dt = DateTime.Now;
         // Standards
Console.WriteLine("{0:O}", dt); // O or o yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffzz
Console.WriteLine("{0:R}", dt); // R or r ddd, dd MMM yyyy HH':'mm':'ss 'GMT'
Console.WriteLine("{0:s}", dt); // s yyyy'-'MM'-'dd'T'HH':'mm':'ss
Console.WriteLine("{0:u}", dt); // u yyyy'-'MM'-'dd HH':'mm':'ss'Z'

         // Using date/time specifiers
Console.WriteLine("{0:t}", dt); // short time
Console.WriteLine("{0:T}", dt); // long time
Console.WriteLine("{0:d}", dt); // short date
Console.WriteLine("{0:D}", dt); // long date
Console.WriteLine("{0:f}", dt); // long date / short time
Console.WriteLine("{0:F}", dt); // long date / long time
Console.WriteLine("{0:g}", dt); // short date / short time
Console.WriteLine("{0:G}", dt); // short date / long time
Console.WriteLine("{0:o}", dt); // Round Trip

      // roll your own...
```

```
Console.WriteLine("{0:dd/mm/yyyy HH:MM:ss}", dt);//custom - what most people use (UK)!
Console.WriteLine("{0:mm/dd/yyyy HH:MM:ss}", dt);//custom - what most people use (US)!
Console.WriteLine("{0:yyyy/mm/dd HH:MM:ss}", dt);//custom - (Japan) Good for sorting!

Console.WriteLine("{0:dd MMM yyyy HH:MM:ss}", dt); // custom - month (UK)
Console.WriteLine("{0:MMM dd yyyy HH:MM:ss}", dt); // custom - month (US)
Console.WriteLine("{0:yyyy MMM dd HH:MM:ss}", dt); // custom - (Japan)
Console.ReadKey();
      }
   }
}
```

3.3 Multi-Threading

Thread :

- A **thread** is defined as the execution path of a program. Each thread defines a unique flow of control. If your application involves complicated and time consuming operations then it is often helpful to set different execution paths or threads, with each thread performing a particular job.
- Threads are **lightweight processes**. One common example of use of thread is implementation of concurrent programming by modern operating systems. Use of threads saves wastage of CPU cycle and increase efficiency of an application.

Thread Life Cycle

The life cycle of a thread starts when an object of the System.Threading.Thread class is created and ends when the thread is terminated or completes execution.

Following are the various states in the life cycle of a thread:

- **The Unstarted State**: it is the situation when the instance of the thread is created but the Start method has not been called.
- **The Ready State**: it is the situation when the thread is ready to run and waiting CPU cycle.
- **The Not Runnable State**: a thread is not runnable, when:
  - Sleep method has been called
  - Wait method has been called
  - Blocked by I/O operations
- **The Dead State**: it is the situation when the thread has completed execution or has been aborted.

The Main Thread

In C#, the **System.Threading.Thread** class is used for working with threads. It allows creating and accessing individual threads in a multithreaded application. The first thread to be executed in a process is called the **main** thread.

When a C# program starts execution, the main thread is automatically created. The threads created using the **Thread** class are called the child threads of the main thread. You can access a thread using the **CurrentThread** property of the Thread class.

The following program demonstrates main thread execution: using System;

```
using System.Threading;

namespace MultithreadingApplication
{
   class MainThreadProgram
   {
```

```
      static void Main(string[] args)
      {
         Thread th = Thread.CurrentThread;
         th.Name = "MainThread";
         Console.WriteLine("This is {0}",
         th.Name); Console.ReadKey();
      }
   }
}
```

OUTPUT:
This is MainThread

Creating Threads: Threads are created by creating the object of Thread. The extended Thread class then calls the **Start()**method to begin the child thread execution. The following program demonstrates the concept:

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
   class ThreadCreationProgram
   {
      public static void CallToChildThread()
      {
         Console.WriteLine("Child thread starts");
      }

      static void Main(string[] args)
      {
         ThreadStart childref = new
         ThreadStart(CallToChildThread); Console.WriteLine("In
         Main: Creating the Child thread"); Thread childThread =
         new Thread(childref); childThread.Start();
         Console.ReadKey();
      }
   }
}
```

OUTPUT:
In Main: Creating the Child
thread Child thread starts

Managing Threads
The Thread class provides various methods for managing threads.
The following example demonstrates the use of the **sleep()** method for making a thread pause for a specific period of time.

```
using System;
using System.Threading;

namespace MultithreadingApplication
```

```
{
   class ThreadCreationProgram
   {
      public static void CallToChildThread()
      {
         Console.WriteLine("Child thread starts");
         // the thread is paused for 5000
         milliseconds int sleepfor = 5000;
         Console.WriteLine("Child Thread Paused for {0}
                  seconds", sleepfor / 1000);
         Thread.Sleep(sleepfor);
         Console.WriteLine("Child thread resumes");
      }

      static void Main(string[] args)
      {
         ThreadStart childref = new
         ThreadStart(CallToChildThread); Console.WriteLine("In
         Main: Creating the Child thread"); Thread childThread =
         new Thread(childref); childThread.Start();
         Console.ReadKey();
      }
   }
}
```

OUTPUT:

```
In Main: Creating the Child
thread Child thread starts
Child Thread Paused for 5
seconds Child thread resumes
Destroying Threads
```

The **Abort()** method is used for destroying threads. The runtime aborts the thread by throwing a **ThreadAbortException**. This exception cannot be caught, the control is sent to the *finally* block, if any.

The following program illustrates this:

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
   class ThreadCreationProgram
   {
      public static void CallToChildThread()
      {
         try
         {

            Console.WriteLine("Child thread starts");
```

```
            // do some work, like counting to 10
            for (int counter = 0; counter <= 10; counter++)
            {
               Thread.Sleep(500);
               Console.WriteLine(counter);
            }
            Console.WriteLine("Child Thread Completed");

         }
         catch (ThreadAbortException e)
         {
            Console.WriteLine("Thread Abort Exception");
         }
         finally
         {
            Console.WriteLine("Couldn't catch the Thread Exception");
         }

      }

      static void Main(string[] args)
      {
         ThreadStart childref = new
         ThreadStart(CallToChildThread); Console.WriteLine("In
         Main: Creating the Child thread"); Thread childThread =
         new Thread(childref); childThread.Start();
         //stop the main thread for some
         time Thread.Sleep(2000);
         //now abort the child
         Console.WriteLine("In Main: Aborting the Child
         thread"); childThread.Abort();
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result: In Main: Creating the Child thread
Child thread
starts 0 1 2

In Main: Aborting the Child
thread Thread Abort Exception
Couldn't catch the Thread Exception

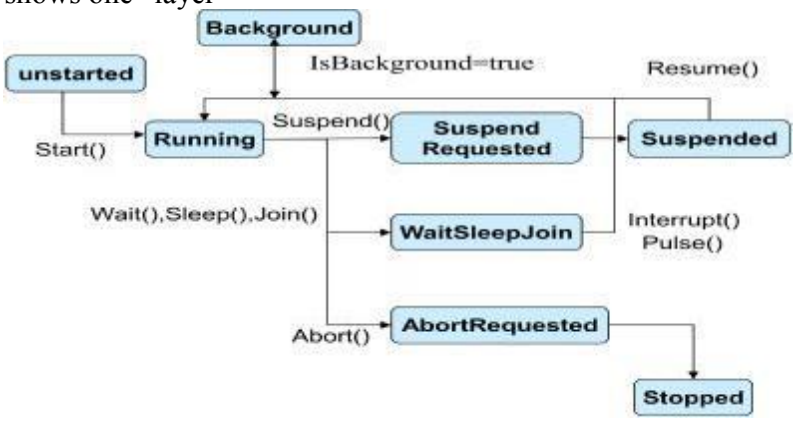
ThreadPriority and ThreadState:

A thread's Priority property determines how much execution time it gets relative to other active threads in the operating system, on the following scale:

This becomes relevant only when multiple threads are simultaneously active.Think carefully before elevating a thread's priority — it can lead to problems such as resource starvation for other threads.Elevating a thread's priority doesn't make it capable of performing real-time work, because it's still throttled by the application's process priority. To perform real-time work, you must also elevate the process priority using the Process class in System.Diagnostics (we didn't tell you how to do this):

```
public enum ThreadPriority
        { Highest,
        AboveNormal,
        Normal,
        BelowNormal,
        Lowest,
}
```

Thread States: You can query a thread's execution status via its **ThreadState** property. This returns a flags enum of type **ThreadState**, which combines three "layers" of data in a bitwise fashion. Most values, however, are redundant, unused, or deprecated. The following diagram shows one "layer"




```
public enum ThreadState {
        Background,
        Unstarted,
        Running,
        WaitSleepJoin,
        SuspendRequested,
        Suspended,
        AbortRequested,
        Stopped
}
```

The following code strips a ThreadState to one of the four most useful values: Unstarted, Running, WaitSleepJoin, and Stopped:

```
public static ThreadState SimpleThreadState (ThreadState ts)
{
 return ts & (ThreadState.Unstarted |
          ThreadState.WaitSleepJoin
          | ThreadState.Stopped);
}
```

The ThreadState property is useful for diagnostic purposes, but unsuitable for synchronization, because a thread's state may change in between testing ThreadState and acting on that information.

Foreground and Background Threads

A *foreground thread* runs indefinitely, whereas a *background thread* stops as soon as the last foreground thread has stopped. You can use the IsBackground property to determine or change the background status of a thread.

Multithreading with Forms and Controls

While multithreading is best suited to running procedures and class methods, you can also use it with forms and controls. If you do so, be aware of the following points:

- Whenever possible, execute the methods of a control only on the thread with which it was created. If you must call a method of a control from another thread, you must use Invoke to call the method.
- Do not use the **SyncLock** (Visual Basic) or **lock** (C#) statement to lock threads that manipulate controls or forms. Because the methods of controls and forms sometimes call back to a calling procedure, you can end up inadvertently creating a deadlock—a situation in which two threads wait for each other to release the lock, causing the application to halt.

Synchronizing threads

Synchronization constructs can be divided into four categories:

1. Simple blocking methods: These wait for another thread to finish or for a period of time to elapse. Sleep, Join, and Task.Wait are simple blocking methods.

2.Locking constructs: These limit the number of threads that can perform some activity or execute a section of code at a time. *Exclusive* locking constructs are most common—these allow just one thread in at a time, and allow competing threads to access common data without interfering with each other. The standard exclusive locking constructs are lock (Monitor.Enter/Monitor.Exit), Mutex, and SpinLock. The nonexclusive locking constructs are Semaphore, SemaphoreSlim, and the reader/writer locks.

3. Signaling constructs: These allow a thread to pause until receiving a notification from another, avoiding the need for inefficient polling. There are two commonly used signaling devices: event wait handles and Monitor's Wait/Pulse methods. Framework 4.0 introduces the CountdownEvent and Barrier classes.

4. Nonblocking synchronization constructs: These protect access to a common field by calling upon processor primitives. The CLR and C# provide the following nonblocking constructs: Thread.MemoryBarrier, Thread.VolatileRead, Thread.VolatileWrite, the volatile keyword, and the Interlocked class.

The lock

The **lock** (C#) and **SyncLock** (Visual Basic) statements can be used to ensure that a block of code runs to completion without interruption by other threads. This is accomplished by obtaining a mutual-exclusion lock for a given object for the duration of the code block.

A **lock** or **SyncLock** statement is given an object as an argument, and is followed by a code block that is to be executed by only one thread at a time. For example:

```
public class TestThreading
{
  private System.Object lockThis = new System.Object();

  public void Process()
  {
```

```
        lock (lockThis)
        {
            // Access thread-sensitive resources.
        }
    }

}
```

The argument provided to the **lock** keyword must be an object based on a reference type, and is used to define the scope of the lock.

Monitors

Like the **lock** and **SyncLock** keywords, monitors prevent blocks of code from simultaneous execution by multiple threads. The Enter method allows one and only one thread to proceed into the following statements; all other threads are blocked until the executing thread calls Exit. This is just like using the**lock** keyword. For example:

```
lock (x)
{
   DoSomething();
}
```

This is equivalent to:

```
System.Object obj = (System.Object)x;
System.Threading.Monitor.Enter(obj);
try
{
   DoSomething();
}
finally
{
   System.Threading.Monitor.Exit(obj);
}
```

Using the **lock** (C#) keyword is generally preferred over using the Monitorclass directly, both because **lock** is more concise, and because **lock** insures that the underlying monitor is released, even if the protected code throws an exception. This is accomplished with the **finally** keyword, which executes its associated code block regardless of whether an exception is thrown.

Synchronization Events and Wait Handles

Using a lock or monitor is useful for preventing the simultaneous execution of thread-sensitive blocks of code, but these constructs do not allow one thread to communicate an event to another. This requiressynchronization events, which are objects that have one of two states, signaled and un-signaled, that can be used to activate and suspend threads. Threads can be suspended by being made to wait on a synchronization event that is unsignaled, and can be activated by changing the event state to signaled. If a thread attempts to wait on an event that is already signaled, then the thread continues to execute without delay.

There are two kinds of synchronization events: AutoResetEvent, and ManualResetEvent. They differ only in that AutoResetEvent changes from signaled to unsignaled automatically any time it activates a thread. Conversely, a ManualResetEvent allows any number of threads to be activated by its signaled state, and will only revert to an unsignaled state when its Reset method is called.

Threads can be made to wait on events by calling one of the wait methods, such as WaitOne, WaitAny, orWaitAll. WaitHandle.WaitOne() causes the thread to wait until a single event becomes signaled,WaitHandle.WaitAny() blocks a thread until one or more indicated events become

signaled, andWaitHandle.WaitAll() blocks the thread until all of the indicated events become signaled. An event becomes signaled when its Set method is called.
In the following example, a thread is created and started by the Main function. The new thread waits on an event using the WaitOne method. The thread is suspended until the event becomes signaled by the primary thread that is executing the Main function. Once the event becomes signaled, the auxiliary thread returns. In this case, because the event is only used for one thread activation, either the AutoResetEventor ManualResetEvent classes could be used.using System;
using System.Threading;

```
class ThreadingExample
{
   static AutoResetEvent autoEvent;

   static void DoWork()
   {
      Console.WriteLine(" worker thread started, now waiting on event...");
      autoEvent.WaitOne();
      Console.WriteLine("  worker thread reactivated, now exiting...");
   }

   static void Main()
   {
      autoEvent = new AutoResetEvent(false);

      Console.WriteLine("main thread starting worker
      thread..."); Thread t = new Thread(DoWork);
      t.Start();

      Console.WriteLine("main thread sleeping for 1 second...");
      Thread.Sleep(1000);

      Console.WriteLine("main thread signaling worker
      thread..."); autoEvent.Set();
   }
}
```

Mutex Object

A mutex is similar to a monitor; it prevents the simultaneous execution of a block of code by more than one thread at a time. In fact, the name "mutex" is a shortened form of the term "mutually exclusive." Unlike monitors, however, a mutex can be used to synchronize threads across processes. A mutex is represented by the Mutex class.
When used for inter-process synchronization, a mutex is called a named mutex because it is to be used in another application, and therefore it cannot be shared by means of a global or static variable. It must be given a name so that both applications can access the same mutex object.
Although a mutex can be used for intra-process thread synchronization, using Monitor is generally preferred, because monitors were designed specifically for the .NET Framework and therefore make better use of resources. In contrast, the Mutex class is a wrapper to a Win32 construct. While it is more powerful than a monitor, a mutex requires interop transitions that are more computationally expensive than those required by the Monitor class

Interlocked Class

You can use the methods of the Interlocked class to prevent problems that can occur when multiple threads attempt to simultaneously update or compare the same value. The methods of this class let you safely increment, decrement, exchange, and compare values from any thread.

ReaderWriter Locks

In some cases, you may want to lock a resource only when data is being written and permit multiple clients to simultaneously read data when data is not being updated. The ReaderWriterLock class enforces exclusive access to a resource while a thread is modifying the resource, but it allows non-exclusive access when reading the resource. ReaderWriter locks are a useful alternative to exclusive locks, which cause other threads to wait, even when those threads do not need to update data.

Deadlocks

Thread synchronization is invaluable in multithreaded applications, but there is always the danger of creating a deadlock, where multiple threads are waiting for each other and the application comes to a halt. A deadlock is analogous to a situation in which cars are stopped at a four-way stop and each person is waiting for the other to go. Avoiding deadlocks is important; the key is careful planning. You can often predict deadlock situations by diagramming multithreaded applications before you start coding.

3.4 Networking and sockets

The .NET framework provides two namespaces, **System.Net** and **System.Net.Sockets** for network programming. The classes and methods of these namespaces help us to write programs, which can communicate across the network. The communication can be either connection oriented or connectionless. They can also be either stream oriented or data-gram based. The most widely used protocol TCP is used for stream-based communication and UDP is used for data-grams based applications.

The **System.Net.Sockets.Socket** is an important class from the **System.Net.Sockets** namespace. A Socket instance has a local and a remote end-point associated with it. The local end-point contains the connection information for the current socket instance.

There are some other helper classes like **IPEndPoint**, **IPADdress**, **SocketException** etc, which we can use for Network programming. The .NET framework supports both synchronous and asynchronous communication between the client and server. There are different methods supporting for these two types of communication. A synchronous method is operating in blocking mode, in which the method waits until the operation is complete before it returns. But an asynchronous method is operating in non-blocking mode, where it returns immediately, possibly before the operation has completed.

Dns Class

The System.net namespace provides this class, which can be used to creates and send queries to obtain information about the host server from the Internet Domain Name Service (DNS). Remember that in order to access DNS, the machine executing the query must be connected to a network. If the query is executed on a machine, that does not have access to a domain name server, a System.Net.SocketException is thrown. All the members of this class are static in nature. The important methods of this class are given below.

```
public static IPHostEntry GetHostByAddress(string address);
```

Where address should be in a dotted-quad format like "202.87.40.193". This method returns an IPHostEntry instance containing the host information. If DNS server is not available, the method returns a SocketException.

```
public static string GetHostName();
```

This method returns the DNS host name of the local machine.
In my machine Dns.GetHostName() returns *Arora* which is the DNS name of my machine.

```
public static IPHostEntry Resolve(string hostname);
```

This method resolves a DNS host name or IP address to a IPHostEntry instance. The host name should be in a dotted-quad format like 127.0.01 or www.microsoft.com.

IPHostEntry Class

This is a container class for Internet host address information. This class makes no thread safety guarantees. The following are the important members of this class.

- AddressList property: Gives an IPAddress array containing IP addresses that resolve to the host name.
- Aliases property: Gives a string array containing DNS name that resolves to the IP addresses in AddressList property.

The following program shows the application of the above two classes.

```
using System;
using System.Net;
using System.Net.Sockets;

class MyClient
{
        public static void Main()
        {

    IPHostEntry IPHost = Dns.Resolve("www.hotmail.com");
        Console.WriteLine(IPHost.HostName);

    string []aliases = IPHost.Aliases;
        Console.WriteLine(aliases.Length);
            IPAddress[] addr = IPHost.AddressList;
        Console.WriteLine(addr.Length);

    for(int i= 0; i < addr.Length ; i++)
        {
        Console.WriteLine(addr[i]);
        }
           }
 }
```

IPEndPoint Class

This class is a concrete derived class of the abstract class EndPoint. The IPEndPoint class represents a network end point as an IP address and a port number. There is couple of useful constructors in this class:

```
        IPEndPoint(long addresses, int port);
        IPEndPoint (IPAddress addr, int port);
        IPHostEntry IPHost = Dns.Resolve("www.c-
        sharpcorner.com"); Console.WriteLine(IPHost.HostName);
        string []aliases = IPHost.Aliases;
        IPAddress[] addr = IPHost.AddressList;
        Console.WriteLine(addr[0]);
```

```
EndPoint ep = new IPEndPoint(addr[0],80);
```

Socket Programming: Synchronous Clients

The steps for creating a simple synchronous client are as follows.

1. Create a Socket instance.
2. Connect the above socket instance to an end-point.
3. Send or Receive information.
4. Shutdown the socket
5. Close the socket

The Socket class provides a constructor for creating a Socket instance.

```
public Socket (AddressFamily af, ProtocolType pt, SocketType st)
```

Where AddressFamily, ProtocolType and SocketTYpe are the enumeration types declared inside the Socket class.

The AddressFamily member specifies the addressing scheme that a socket instance must use to resolve an address. For example AddressFamily.InterNetwork indicates that an IP version 4 addresses is expected when a socket connects to an end point.

The SocketType parameter specifies the socket type of the current instance. For example SocketType.Stream indicates a connection-oriented stream and SocketType.Dgram indicates a connectionless stream.

The ProtocolType parameter specifies the ptotocol to be used for the communication. For example ProtocolType.Tcp indicates that the protocol used is TCP and ProtocolType.Udp indicates that the protocol using is UDP.

```
public Connect (EndPoint ep);
```

The Connect() method is used by the local end-point to connect to the remote end-point. This method is used only in the client side. Once the connection has been established the Send() and Receive() methods can be used for sending and receiving the data across the network.

The Connected property defined inside the class Socket can be used for checking the connection. We can use the Connected property of the Socket class to know whether the current Socket instance is connected or not. A property value of true indicates that the current Socket instance is connected.

```
IPHostEntry IPHost = Dns.Resolve("www.yahoo.com");
Console.WriteLine(IPHost.HostName);
string []aliases = IPHost.Aliases;
IPAddress[] addr = IPHost.AddressList;
Console.WriteLine(addr[0]);
EndPoint ep = new IPEndPoint(addr[0],80);

Socket sock = new Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
sock.Connect(ep);
if(sock.Connected)
Console.WriteLine("OK");
```

The Send() method of the socket class can be used to send data to a connected remote socket.

```
public int Send (byte[] buffer, int size, SocketFlags flags);
```

Where byte[] parameter storing the data to send to the socket, size parameter containing the number of bytes to send across the network. The SocketFlags parameter can be a bitwise

combination of any one of the following values defined in the System.Net.Sockets.SocketFlags enumerator.

```
SocketFlags.None
SocketFlags.DontRoute
SocketFlags.OutOfBnd
```

The method Send() returns a System.Int32 containing the number of bytes send.Remember that there are other overloaded versions of Send() method as follows.

```
public int Send (byte[] buffer, SocketFlags
flags); public int Send (byte[] buffer);
public int Send (byte[] buffer,int offset, int size, SocketFlags flags);
```

The Receive() method can be used to receive data from a socket.

```
public int Receive(byte[] buffer, int size, SocketFlags flags);
```

Where byte[] parameter storing the data to send to the socket, size parameter containing the number of bytes to send across the network. The SocketFlags parameter can be a bitwise combination of any one of the following values defined in the System.Net.Sockets.SocketFlags enumerator explained above.
The overloaded versions of Receive() methods are shown

```
below. public int Receive (byte[] buffer, SocketFlags
flags); public int Receive (byte[] buffer);
public int Receive (byte[] buffer,int offset, int size, SocketFlags flags);
```

When the communication across the sockets is over, the connection between the sockets can be terminated by invoking the method ShutDown()

```
public void ShutDown(SocketShutdown how);
```

Where ‘how’ is one of the values defined in the SocketSHutdown enumeration. The value SoketShutdown.Send means that the socket on the other end of the connection is notified that the current instance would not send any more data. The value SoketShutdown.Receive means that the socket on the other end of the connection is notified that the current instance will not receive any more data and the value SoketShutdown.Both means that both the action are not possible. Remember that the ShutDown() method must be called before the Close(0 method to ensure that all pending data is sent or received.
A socket can be closed by invoking the method

```
Close(). public void Close();
```

This method closes the current instance and releases all managed and un-managed resources allocated by the current instance. This method internally calls the Dispose() method with an argument of ‘true’ value, which frees both managed and un-managed resources used by the current instance.

```
protected virtual void Dispose(bool);
```

The above method closes the current instance and releases the un-managed resources allocated by the current instance and exceptionally release the managed resources also. An argument value of ‘true’ releases both managed and un-managed resources and a value of ‘false’ releases only un-managed resources.
The following program can send an HTTP request to a web server and can read the response from the web server.

Example

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
class MyClient{
public static void Main(){
IPHostEntry IPHost = Dns.Resolve("www.google.com");
Console.WriteLine(IPHost.HostName);

string []aliases = IPHost.Aliases;
IPAddress[] addr = IPHost.AddressList;
Console.WriteLine(addr[0]);

EndPoint ep = new IPEndPoint(addr[0],80);
Socket sock = new Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
sock.Connect(ep);
if(sock.Connected)
        Console.WriteLine("OK");
Encoding ASCII = Encoding.ASCII;
string Get = "GET / HTTP/1.1\r\nHost: " + "www. google.com" + "\r\nConnection:
Close\r\n\r\n"; Byte[] ByteGet = ASCII.GetBytes(Get);
Byte[] RecvBytes = new Byte[256];
sock.Send(ByteGet, ByteGet.Length, 0);
Int32 bytes = sock.Receive(RecvBytes, RecvBytes.Length, 0);

Console.WriteLine(bytes);
String strRetPage = null;
strRetPage = strRetPage + ASCII.GetString(RecvBytes, 0,
bytes); while (bytes > 0) {
          bytes = sock.Receive(RecvBytes, RecvBytes.Length, 0);
        strRetPage = strRetPage + ASCII.GetString(RecvBytes, 0,
        bytes); Console.WriteLine(strRetPage );
        }
sock.ShutDown(SocketShutdown.Both);
sock.Close();
}}
```

3.5 Managing Console I/O Operations

Console Input

In previous Chapters, we saw that the **Console** class allows using the **Write()** and the **WriteLine()** functions to display things on the screen. While the **Console.Write()** method is used to display something on the screen, the **Console** class provides the **Read()** method to get a value from the user. To use it, the name of a variable can be assigned to it. The syntax used is:

```
        VariableName = Console.Read();
```

This simply means that, when the user types something and presses Enter, what the user had typed would be given (the word is *assigned*) to the variable specified on the left side of the assignment operator.

Read() doesn't always have to assign its value to a variable. For example, it can be used on its own line, which simply means that the user is expected to type something but the value typed by the user would not be used for any significant purpose. For example some versions of C# (even including Microsoft's C# and Borland C#Builder) would display the DOS window briefly and disappear. You can use the **Read()** function to wait for the user to press any key in order to close the DOS window.

Besides **Read()**, the **Console** class also provides the **ReadLine()** method. Like the **WriteLine()** member function, after performing its assignment, the **ReadLine()** method sends the caret to the next line. Otherwise, it plays the same role as the **Read()** function.

```
string FirstName;
Console.Write("Enter First Name: ");
FirstName = Console.ReadLine();
```

In C#, everything the user types is a string and the compiler would hardly analyze it without your explicit asking it to do so. Therefore, if you want to get a number from the user, first request a string. After getting the string, you must convert it to a number. To perform this conversion, each data type of the .NET Framework provides a mechanism called **Parse**. To use **Parse()**, type the data type, followed by a period, followed by Parse, and followed by parentheses. In the parentheses of Parse, type the string that you requested from the user. Here is an example:

```
using System;
namespace GeorgetownCleaningServices
{
  class OrderProcessing
  {
        static void Main()
        {
              int Number;
              string strNumber;
              strNumber = Console.ReadLine();
              Number = int.Parse(strNumber);
        }
  }
}
```

Console Output

Instead of using two **Write()** or a combination of **Write()** and **WriteLine()** to display data, you can convert a value to a string and display it directly. To do this, you can provide two strings to the **Write()** or **WriteLine()** and separate them with a comma:

1. The first part of the string provided to **Write()** or **WriteLine()** is the complete string that would display to the user. This first string itself can be made of different sections:
    a. One section is a string in any way you want it to display
    b. Another section is a number included between an opening curly bracket "{" and a closing curly bracket "}". This combination of "{" and "}" is referred to as a placeholder
    c. You can put the placeholder anywhere inside of the string. The first placeholder must have number 0. The second must have number 1, etc. With this technique, you can create the string anyway you like and use the placeholders anywhere inside of the string

The second part of the string provided to **Write()** or **WriteLine()** is the value that you want to display. It can be one value if you used only one placeholder with 0 in the first string. If you used different placeholders, you can then provide a different value for each one of them in this second part, separating the values with a comma

Example using
System; class
Exercise{
```
   static void Main()  {
      String FullName = "Anselme
      Bogos"; int Age = 15;
      double HSalary = 22.74;

      Console.WriteLine("Full Name: {0}", FullName);
      Console.WriteLine("Age: {0}", Age);
      Console.WriteLine("Distance: {0}", HSalary);

      Console.WriteLine();
   }}
```
OUTPUT:
Full Name: Anselme
Bogos Age: 15
Distance: 22.74
As mentioned already, the numeric value typed in the curly brackets of the first part is an ordered number. If you want to display more than one value, provide each incremental value in its curly brackets. The syntax used is:

```
Write("To Display {0} {1} {2} {n}", First, Second, Third, nth);
```

You can use the sections between a closing curly bracket and an opening curly bracket to create a meaningful sentence.
The **System** namespace provides a specific letter that you can use in the **Write()** or **WriteLine()**'s placeholder for each category of data to display. To format a value, in the placeholder of the variable or value, after the number, type a colon and one of the appropriate letter from the following table. If you are using **ToString()**, then, in the parentheses of **ToString()**, you can include a specific letter or combination inside of double-quotes. The letters and their meanings are:

Character		Used For
c	C	Currency values
d	D	Decimal numbers
e	E	Scientific numeric display such as $1.45e^5$
f	F	Fixed decimal numbers
g	G	General and most common type of numbers
n	N	Natural numbers
r	R	Roundtrip formatting
x	X	Hexadecimal formatting
p	P	Percentages

Example
```
using System;
class Exercise
{
   static void Main()
   {
```

```
        double Distance =
        248.38782; int Age = 15;
        int NewColor = 3478;
        double HSal = 22.74, HrsWork = 35.5018473;
        double WeeklySal = HSal * HrsWork;

    Console.WriteLine("Distance: {0}", Distance.ToString("E"));
    Console.WriteLine("Age: {0}", Age.ToString());
    Console.WriteLine("Color: {0}", NewColor.ToString("X"));
Console.WriteLine("Weekly Salary: {0} for {1} hours",
WeekSal.ToString("c"), HrsWork.ToString("F"));
            Console.WriteLine();
    }
}
```

OUTPUT:

```
Distance:
2.483878E+002 Age: 15
Color: D96
Weekly Salary: $807.31 for 35.50 hours
```

To specify the amount of space used to display a string, you can use its placeholder in **Write()** or **WriteLine()**. To do this, in the placeholder, type the 0 or the incrementing number of the placer and its formatting character if necessary and if any. Then, type a comma followed by the number of characters equivalent to the desired width. Here are examples:

```
using System;
class Exercise{
            static void Main() {
                        String FullName = "Anselme Bogos";
                        int Age = 15;
                        double Marks = 22.74;
                        Console.WriteLine("Full Name: {0,20}", FullName);
                        Console.WriteLine("Age:{0,14}", Age.ToString());
                  Console.WriteLine("Marks:{0:C,8}", Marks.ToString());
                        Console.WriteLine();
            }
}
```

OUTPUT:

```
Full Name:       Anselme Bogos
Age:          15
Marks: 22.74
```

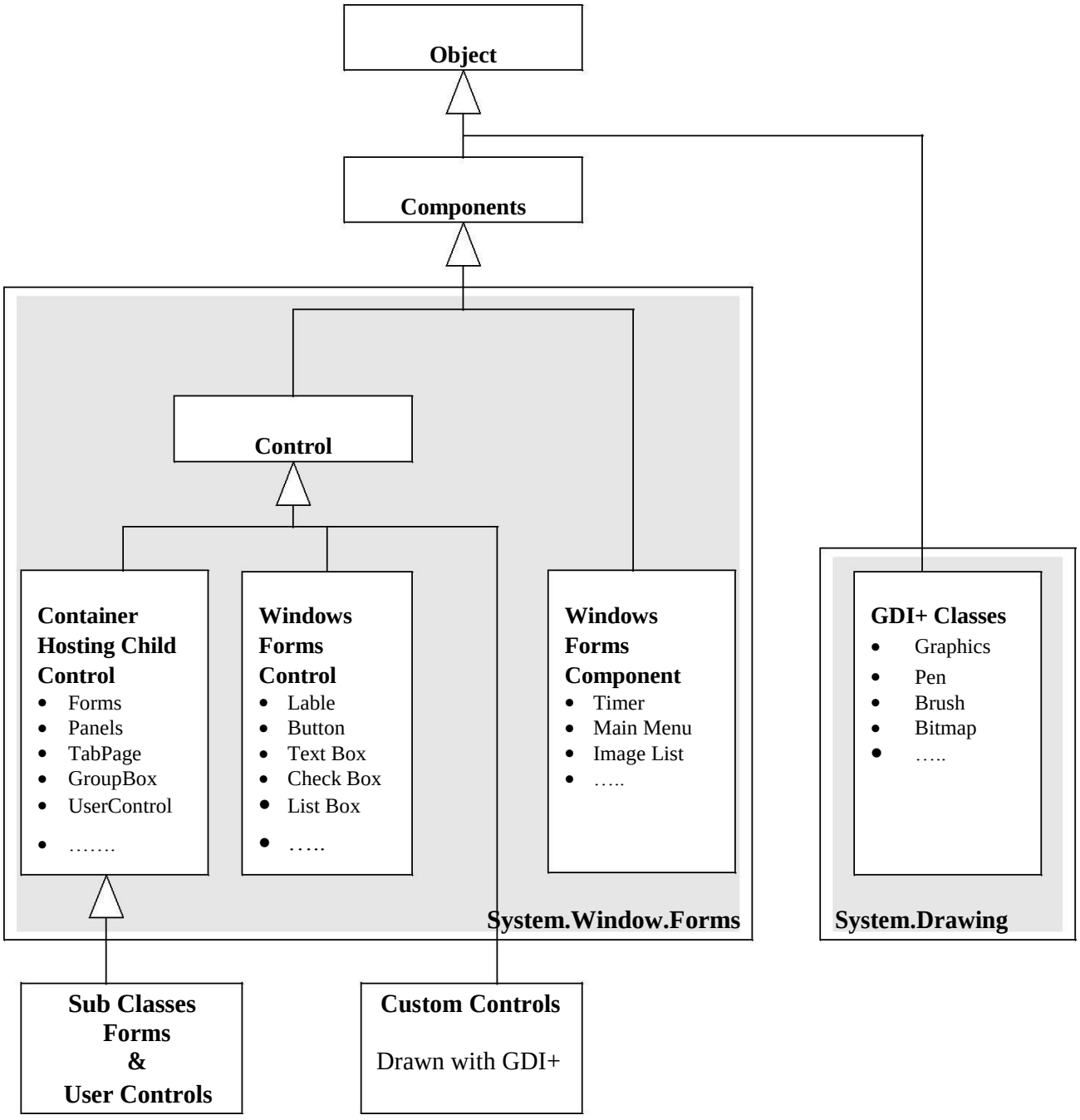
The sign you provide for the width is very important. If it is positive, the line of text is aligned to the right. This should be your preferred alignment for numeric values. If the number is negative, then the text is aligned to the left.

3.6 Windows Forms

The Windows Forms is a collection of classes and types that encapsulate and extend the Win32 API in an organized object model. In other words, the components used to create Windows GUI applications are provided as .NET classes and types that form part of an orderly hierarchy.

This hierarchy is defined by inheritance: Simple reusable classes such as *Component* are provided, and then used as a base from which more sophisticated classes are derived. We can draw a useful overview by representing the inheritance hierarchy in a treelike diagram. Figure 8.1 summarizes at a high level the classes that include Windows Forms and GDI+.

Figure : A Summary of Window Forms and GDI+ Classes



The arrows represent inheritance: *Control* assumes all the functionality of *Component*, which assumes all the functionality of *Object*. Following Table provides a quick and practical summary of the four essential classes on which the Windows Forms types are based.

Class	Role	Why We Need It
Object	Acts as a base class for all types in the .NET Framework.	For a tidy unified type system, and to provide core functionality available to all types (such as *ToString*).
Component	Provides the basics of containership, facilitates hosting in a visual designer, and defines a protocol for resource disposal.	So Visual Studio's Designer can host a wide variety of controls and components in a generic way, to provide a base from which you can write nonvisual components, and to allow the cleanup of Windows handles and file handles in a timely and reliable manner.
Control	Provides the core functionality for a visual control that responds to mouse and keyboard messages, accepts focus, and can participate in drag-and-drop operations.	As a common superclass for all controls, such as textboxes, labels, and buttons, allowing them to be treated in a consistent manner, as well as providing a base from which you can derive your own custom controls.
Form	Defines a class representing a window to which you can add controls.	To provide a base class with standard windowing and containership functionality that you can subclass to create forms in your application.

Table : Core Classes

Creating a Windows Forms application is largely just a matter of instantiating and extending the Windows Forms and GDI+ classes. In a nutshell, you typically complete the following steps:

1. Create a new project defining the structure of a Windows Forms application.
2. Define one or more Forms (classes derived from the *Form* class) for the windows in your application.
3. Use the Designer to add controls to your forms (such as textboxes and checkboxes), and then configure the controls by setting their properties and attaching event handlers.
4. Add other Designer-managed components, such as menus or image lists.
5. Add code to your form classes to provide functionality.
6. Write custom controls to meet special requirements, using GDI+ classes to handle low-level graphics.

Creating Windows Forms Application

The first step to building a Windows Forms application is creating a project. A Windows Forms project is just like any other type of project in that it consists of a grouping of source code files, a list of references to required .NET code libraries, and an appropriate configuration of compilation and debugging options. When you use Visual Studio to create a project from a template, it sets all of this up for you, providing a "skeleton" appropriate to the template you've selected. In the case of Windows Forms, this consists of the following:

- A project of Output Type *Windows Application*. You can view or change this in the **Project | Properties** dialog box.

- References to the .NET assemblies required for typical Windows Forms applications (covering most of the types in the *Windows Forms* namespace). You can see a list of the project references in the Solution Explorer.
- A blank form, called *Form1* (a C# class with the structure required for a visually editable form).
- A *Main* method in *Form1* that instantiates and displays the form.

Let's start the walkthrough by creating a new Windows Forms project. From the main menu, choose **File | New | Project**, click **Visual C# Projects** , and choose the **Windows Application** template (see Figure 8.2). Change the project name to **SimpleApp** and click **OK**.

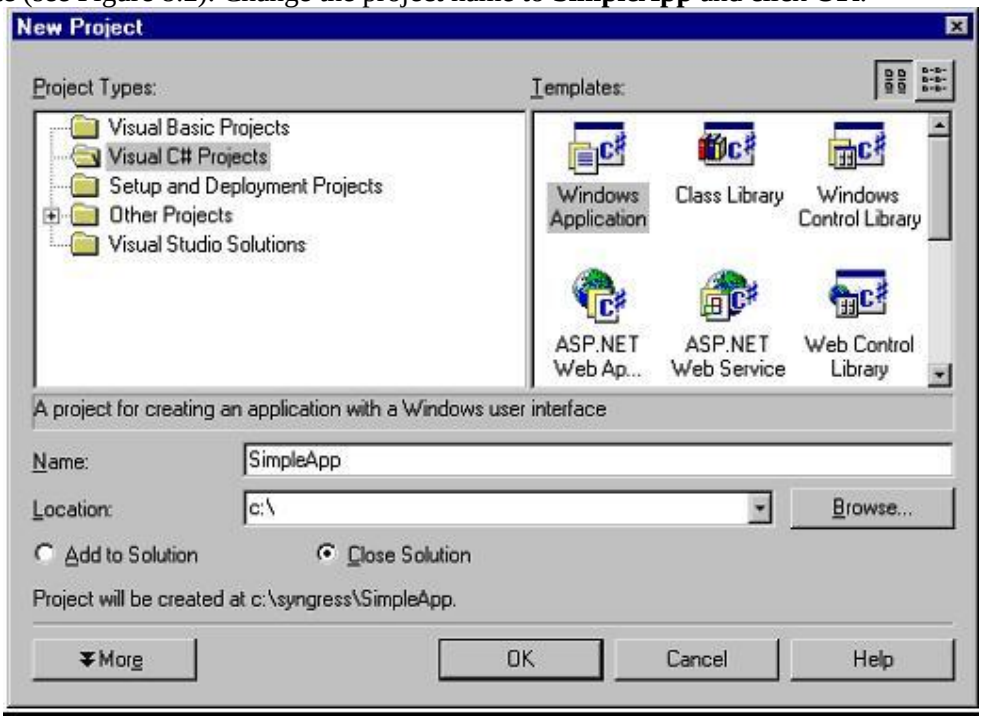


Figure 8.2 Creating a New Windows Forms Project

Adding Controls

Once we've created the project, Visual Studio opens the main form (*Form1*) in the Designer—the visual editor for our C# form class. Basically, a form created in Visual Studio is just a C# file, defining a class based on *System.Windows.Forms.Form*, containing code to add and configure the controls created visually. Visual Studio is a "two-way tool" meaning that we can work with the same code either visually (using the Designer) or programmatically (in the Code Editor).

Let's use the Designer to add a few controls to *Form1*. We can add controls and components from the toolbox window and then configure them using the Properties window.

1. From the toolbox, add a Label control to the form. By default, Visual Studio will name the control *Label1*.
2. From the Properties Window (F4) change *label1*'s *Text* property to **Favorite CD**, and change its *AutoSize* property to **True** .This tells the control to size itself according to the metrics of the font and width of the text.
3. Now add a **TextBox** from the toolbox onto the form, and position it below the label. Enlarge it horizontally and clear its *Text* property.
4. Add another label to the form, setting its *Text* property to **Favorite Style**, and *AutoSize* property to **True**.

5. Add a **ComboBox** and position it below the *Favorite Style* label. Clear its *Text* property.
6. Select the combo's *Items* property, and then click the ellipses on the right to open the String Collection Editor. Type in a few styles of music—each on a separate line
7. Click **OK**, and then press **F5** to save, compile, and run the application.

Adding an Event Handler

Let's add some functionality to the form.

1. Add a **Button** and **ListBox** to the form.
2. Select the button, and change its *Text* property to **Update**. Then click the **lightning icon** in the Properties window to switch to the Events View.

   Think of these events as "hooks" into which we can attach our own methods. You can either double-click on an event to create a new event-handling method, or use the drop-down list to connect into an existing compatible method.

3. Double-click on the **Click** event. Visual Studio will write a skeleton event-handling method, wiring it to the event. It will then place you in the Code Editor, inside the empty method definition:

```
private void button1_Click(object sender, System.EventArgs e)
{

}
```

The .NET convention for event handling requires two parameters: a sender parameter of type *object*, and an event arguments parameter of type *EventArgs*—or a descendant of *EventArgs*. The sender parameter tells us which control fired the event (this is useful when many controls have been wired to the same event-handling method). The second parameter is designed to supply special data about the event. In the case of *Click***,** we have a standard *EventArgs* object, and this contains no useful information—it's just there to meet the protocol required to support more sophisticated events (such as *KeyPress* or *MouseDown*).

The actual name for this method (*button1_Click*) is just a convenient identifier generated by Visual Studio; Windows Forms doesn't impose any particular naming convention.

4. Add the following code to the event handler:

```
private void button1_Click(object sender, System.EventArgs e)
{
  listBox1.Items.Clear();
  listBox1.Items.Add ("Fav CD: " + textBox1.Text);
  listBox1.Items.Add ("Fav Style: " + comboBox1.Text);
}
```

Here we're manipulating our list box through its *Items* property. *Items* returns a collection object, having methods to add and remove items from its list. Note how we access each control through its name—this is possible because the Designer creates class fields matching the names of each control. You can see these declarations at the top of the class definition.

5. Press **F5** to compile and run the program

Adding Controls at Runtime

Sometimes it's necessary to add controls without the help of the Designer. For instance, you might want some controls to appear on a form only when a particular button is clicked.
In reading how to programmatically add controls, it's very helpful to examine a visually created form in the Code Editor. If you expand the Designer Generated Code region, you'll see a method

called *InitializeComponent* containing all the code that creates and configures each of the form's visual components.

Here are the four steps to programmatically adding a control or component:

1. Add a class field declaration for the new control.
2. Instantiate the control.
3. Configure the control by setting its properties and adding event handlers, if required.
4. Add the control to the form's *Controls* collection (or alternatively, to the Controls collection of a container control, such as a *GroupBox*).

Example: Create a new form, add a button, and then have a textbox appear when the user clicks the button:

1. Create a new Windows Forms project called *SimpleApp2* and add a *Button* control from the toolbox onto the new form.
2. Press **F7** to open the Code Editor, and locate *button1*'s declaration. Below this, add a similar declaration for our new textbox, as follows (you can exclude the *System.Windows.Forms* prefix if your form has the appropriate *using* statement):
   private System.Windows.Forms.Button button1;
   private System.Windows.Forms.TextBox myTextBox;
   You need to understand that this declaration doesn't actually create a textbox. All it does is instruct the compiler, once our form is instantiated, to create a field that can *reference* (point to) a textbox object—one that does not yet exist. This declaration exists so as to provide a convenient way to refer to the control throughout the lifetime of the form. In the cases where we don't need to explicitly reference the control after its been created, we can do away with this declaration.
3. Return to the Designer, and double-click on the button. This is a quick way to attach an event handler to the button's default event (Click).
4. Add the following code to the button's event handler:

```
private void button1_Click(object sender, System.EventArgs e)
{
  // Create the actual textbox and assign its reference to
  myTextBox this.myTextBox = new TextBox();

  // Position the control
  myTextBox.Location = new Point (30, 20);

  // Put the control on the form.
  this.Controls.Add (myTextBox);
}
```

5. Press **F5** to test the application

Attaching an Event Handler at Runtime

Let's suppose we want to set up our newly created textbox so that when it's right-clicked, a message box appears. We need to add an event handler to the textbox at runtime, and there are two steps to this:

- Writing the event-handling method.
- Attaching the method to the control's event.
  - In our case, we'll need to attach to the textbox's *MouseDown* event (because there's no specific right-click event). First, we need to write the event-handling method, with

parameters of the correct type for a *MouseDown* event. You can determine an event's signature in two ways:

- Look for the event in the Microsoft documentation, and then click on its delegate (in our case, *MouseEventHandler*).
- Using the Designer, add a dummy control of the type we're attaching to, create an appropriate event handler, and then delete the dummy control. The event-handling method will still be there—with the correct signature. All we need to do is rename it.

1.

```
void myTextBox_MouseDown (object sender, MouseEventArgs e)
{
  if (e.Buttons == MouseButtons.Right)
     // Show is a static method of System.Windows.Forms.MessageBox
     MessageBox.Show ("Right Click!");
}
```

2. Next, we attach this method to *myTextBox*'s *MouseDown* event. Return to the *button1_Click* method and add the following line of code:

```
myTextBox.MouseDown += new MouseEventHandler (myTextBox_MouseDown)
```

On the left-hand side, **myTextBox.MouseDown** is the event to which we're attaching, using the += operator. On the right-hand side, we're creating a new *MouseEventHandler* delegate instance: in other words, an object containing a pointer to a method (*myTextBox_MouseDown*) conforming to *MouseEventHandler*'s signature.

3. Test the application.

3.7 Error Handling

There are no exceptions in C and in C++ one can get away from using them with error handling functions such as exit() and terminate(). In C# these functions are absent and we introduce exceptions which take their place. The exception handling in C#, and Java is quite similar. When a program has a bug we can intercept it in the flow of execution by inserting an error handling statement. To catch a particular type of exception in a piece of code, you have to first wrap it in a 'try' block and then specify a 'catch' block matching that type of exception. When an exception occurs in code within the 'try' block, the code execution moves to the end of the try box and looks for an appropriate exception handler. For instance, the following piece of code demonstrates catching an exception specifically generated by division by zero:

```
try{
   int zero = 0;
   res = (num / zero);
}
catch (System.DivideByZeroException e){
   Console.WriteLine("Error: an attempt to divide by zero");
}
```

You can specify multiple catch blocks (following each other), to catch different types of exception. A complication results, however, from the fact that exceptions form an object hierarchy, so a particular exception might match more than one catch box. What you have to do here is put catch boxes for the more specific exceptions before those for the more general exceptions. At most one

catch box will be triggered by an exception, and this will be the first (and thus more specific) catch box reached.

.

Example

```
//first exception handling
program using System;
class OutOfRange: Exception
{
}

class Demo
   { int n;
   public int []array;

   public Demo ( int n)   {
      this.array = new int[n];
      this.n = n;
   }

   public void show_element (int i)   {
      try        {
         if (i == 0) throw ( new OutOfRange());       }
      catch (Exception e)        {
         Console.WriteLine("Exception : {0}", e);       }
       Console.WriteLine (array [i]);
   }
}
class Test {
   public static void Main() {
      Demo test = new Demo (3);
      test.array [1] = 2;
      test.array [2] = 3;
      test.show_element (0);
   }
}
```

Exception	When Occured
System.ArithmeticException	A base class for exceptions that occur during arithmetic operations, such as System.DivideByZeroException and System.OverflowException.
System.ArrayTypeMismatchException	Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.
System.DivideByZeroException	Thrown when an attempt to divide an integral value by zero occurs.
System.IndexOutOfRangeException	Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
System.InvalidCastException	Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.
System.NullReferenceException	Thrown when a null reference is used in a way that causes the referenced object to be required.
System.OutOfMemoryException	Thrown when an attempt to allocate memory (via new) fails.
System.OverflowException	Thrown when an arithmetic operation in a checked context overflows.
System.StackOverflowException	Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.
System.TypeInitializationException	Thrown when a static constructor throws an exception, and no catch clauses exists to catch it.

Unit-4:
Advance Features using C#

4.1 Web Services

Web service is simply an application that exposes a Web-accessible API. That means you can invoke this application programmatically over the Web. Using SOAP Protocol XML+HTTP =SOAP (Simple Object Access protocol)

Web services allow applications to share data. Web services can be called across platforms and operating systems regardless of programming language. .NET is Microsoft's platform for XML Web services.

Web Service Application

Like Pay per view T.V. Channel we can make Software as pay per use, using Web service. For example, let say Infovision Inc developed Expensive Software for 3D Virtual Modeling. Lot of company does not want acquire the licensee because it is expensive and they need to pay for Support etc… instead of this if Infovision makes this software as a WebService, most of the company will use as pay per use.

Another example Credit card validation we can expose as a webservice. (Good example for webService)

Using VisualStudio.NET we can easily create WebServices. ASP.NET WebService this project type will be used to create WebService.

The client of a Web service can be a rich Windows application created using Windows Forms, WPF, Silverlight, or an ASP.NET application using Web Forms. A Windows PC, a UNIX system, or a mobile device can be used to consume (use) the Web service. With the .NET Framework, Web services can be consumed in every application type.

There are several web service available with .Net Framework, such as:

1) **Validation Controls** such as:
    1. E-mail address validator,
    2. Regular expression validator,
    3. Range Validator, etc.
2) **Login Controls** such as:
    1. Create user
    2. Delete user
    3. Manage users, etc.

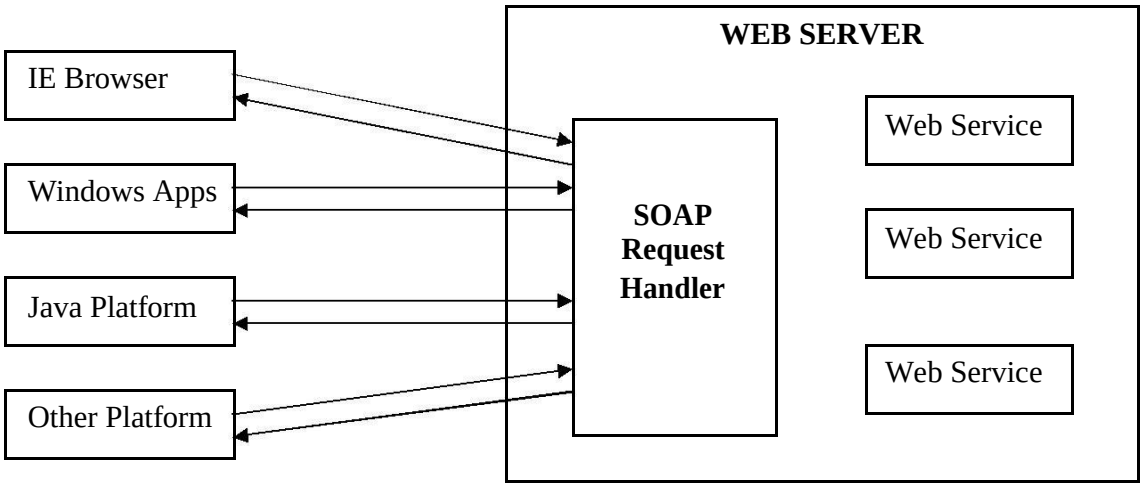
Some Web services are also available on internet , which are free and offer application-components like:

- Currency Conversion,
- Weather Reports,
- Language Translation,
- Search Engines,
- Document Convertor, etc.

Some are paid and can be use by authorised sites, such as:

- Credit and Debit card payment
- Net Banking, etc.

Web Services Architecture



Web Service Architecture

Creating Web Service

To create and expose ASP.NET Web Servies by authoring and saving text files with the file extension "asmx" within the virtual path of an ASP.NET Web Application.

To understand the concept of Web Services we have given an example of Web Service, which provides the current time of day on its machine.

Declaring WebMethod methods

A WebMethod represents a method for web. WebMethod has six properties they are :

1) Description
2) MessageName
3) EnableSession
4) CacheDuration
5) TransactionOption
6) BufferResponse

```
[WebMethod]
Public string SayHello ()
{
return "Hello
Neeraj Tripathi";
}
```

Description

Both the [WebService], and [WebMethod] attributes have a Description Property. With this property we can associate documentation with our web Service and WebMethod. For example you can use Description Attribute to describe the Webmethod.

```
[WebMethod (Description="This method will add three
integer")] Public int Add (int a, int b, int c)
{
return a+b+c;
}
```

MessageName

This is property useful, when we want to overload the WebMethod.
For Example

```
[WebMethod]
Public int Add (int a, int b, int c)
{
return a+b+c;
}
[WebMwthod (MessageName=”Add”)]
public int Add(int a)
{
  int s = 0 ;
  for(int i = 1; i <= a;
    i++) s = s + i;
  return s;
 }
```

EnableSession

This Property used for to enable the session in WebServices.
(Remember WebServiecs uses HTTP protocol this is stateless) .if we want to maintain the state between client and server we need to use this property. Default Enablesession is false

```
[WebMethod (EnableSession=true)]
Public string SayHiToMS ()
{
return " Hello to .NET ";
}
```

CacheDuration

When we cache the previously accessed result. Next time the same user or different user asks we can serve from cache. Here result cached 60 milliseconds. if we invoke this method with in 60 milliseconds it will print same time . This will increase the Web Service performance.

```
[WebMethod (CacheDuration =
60)] Public string ShowTime()
{
return
DateTime.Now.ToShortTimeString();
}
```

TransactionOption

TransactionOption Can be one of five modes:

- Disabled
- NotSupported

- Supported
- Required
- RequiresNew

Even though there are five modes, web methods can only participate as the root object in a transaction. This means both Required and RequiresNew result in a new transaction being created for the web method. The Disabled, NotSupported, and Supported settings result in no transaction being used for the web method. The TransactionOption property of a web method is set to Disabled by default.

BufferResponse

This property is boolean control. WebMethod whether or not to buffer the method's response.

4.2 Window Service

Windows Services is previously called as NT Service. The Idea of creating a windows service application is two fold one is to create a long running application and the other is service applications are the application that run directly in the windows session itself (This can be verified by accessing the current or home directory of the service application which default to .winnt\system32). One more advantage of the Windows Service Application, which makes it more useful, compared to other application is that, Service application can be made to run in the security context of a specific user account.

There are basically two types of Services that can be created in .NET Framework. Services that are only service in a process are assigned the type Win32OwnProcess. Services that share a process with another service are assigned the type Win32ShareProcess.The type of the service can be queried. There are other types of services which are occasionally used they are mostly for hardware, Kernel, File System.

Lifecycle of the Service Application.

In a typical application the Main method is the entry pointand for what ever instantiation has to happen will happen here matters. In Caseof Service, the Main method remains to the entry module by the actual functionality of the service application start only in the Main method by triggering a overridden method called OnStart() which actually starts the Service. Prior to OnStart() method another significant variance is compared to the other application is service should be installed onto the system on which it will run. This process executes the installers for the service project and loads the service into the Services Control Manager for that computer. The Services Control Manager is the central utility provided by Windows to administer services following which the main and Onstart() is followed to fall inline.

A Service application can be in any one of the states listed they are viz., Running, Paused or Stopped. The Services also has the capability to queue the commands and also remain in those states. Such states can be queried, some of the significant states are like ContinuePending, which means continue command is in queue and yet to get executed. Similarly StartPending, StopPending, PausePending.

Window SERVICE application development can be divided to two phases. One is the development of Service functionality and the last phases is about the development. The 3 Main classes involved in Service development are:

- System.ServiceProcess.ServiceBase
- System.ServiceProcess.ServiceProcessInstaller
- ServiceController

System.ServiceProcess.ServiceBase , is the class in which we override the methods for implementation, Table 9.1 shows some methods which are provide for override.

Method	Override to
OnStart	Indicate what actions should be taken when your service starts running. You must write code in this procedure for your service to perform useful work.
OnPause	Indicate what should happen when your service is paused.
OnStop	Indicate what should happen when your service stops running.
OnContinue	Indicate what should happen when your service resumes normal functioning after being paused.
OnShutDown	Indicate what should happen just prior to your system shutting down, if your service is running at that time.
OnCustomCommand	Indicate what should happen when your service receives a custom command. For more information on custom commands.
OnPowerEvent	Indicate how the service should respond when a power management event is received, such as a low battery or suspended operation.
Run	The main entry point for the service, u sally in the Main method

Developing Window Service

To develop and run a Window Service application on .NET frame, you have to follow the following steps.

Step 1: Create Skeleton of the Service

Step 2: Add functionality to your service

Step 3: Install and Run the Service

Step 4: Start and Stop the Service

Create Skeleton of the Service

To create a new Window Service, pick Windows Service option from your Visual C# Projects, give your service a name, and click OK, see figure 9.2.

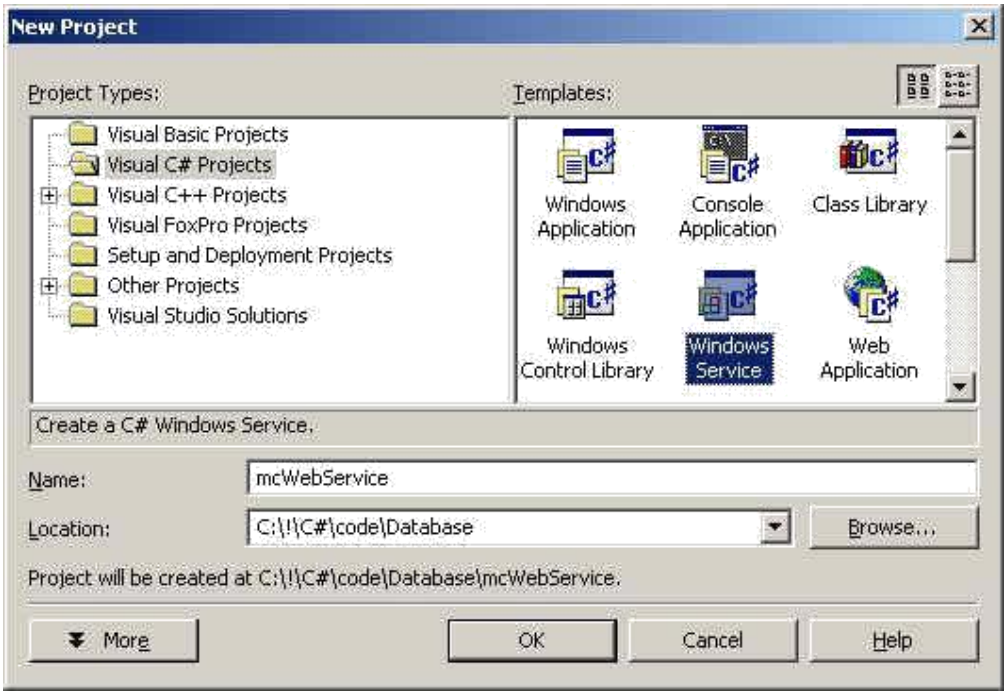


Figure 9.2 Create New Window Service

The result looks like figure 9.3. The Wizard adds WinService1.cs class to your project.

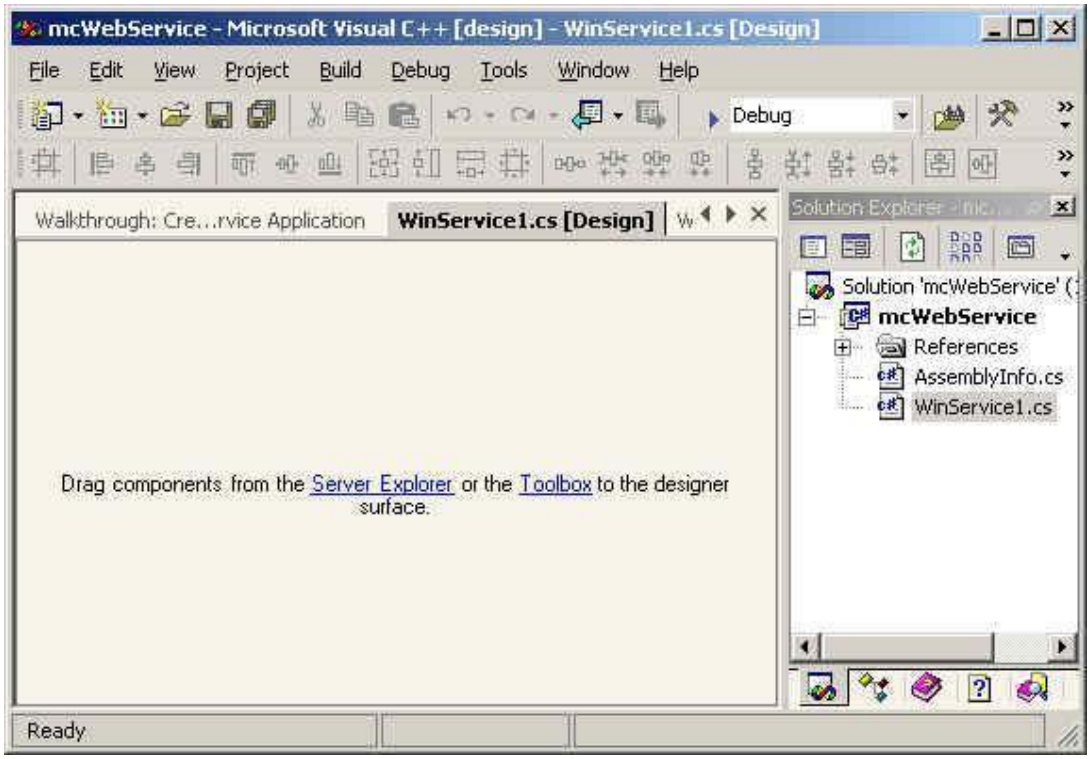


Figure 9.3 WinService1.cs

Set your **ServiceName** to your own name so it would be easier to recognize your service during testing see figure 9.4, OR you can set this property programmatically using this line this.ServiceName = "mcWinService"; (This is the name you will be looking for later).

Figure 9.4 Setting the name of Service

The default code of WinService1.cs added by the Wizard looks like here

```
namespace mcWebService
{
using System;
using System.Collections;
using System.Core;
using System.ComponentModel;
using System.Configuration;
using System.Data;
using System.Web.Services;
using System.Diagnostics;
using System.ServiceProcess;

public class WinService1 : System.ServiceProcess.ServiceBase
{
// <summary>
// Required designer variable.
// </summary>
private System.ComponentModel.Container components;

public WinService1()
{
// This call is required by the WinForms Component
Designer. InitializeComponent();
// TODO: Add any initialization after the InitComponent call
}

// The main entry point for the
process static void Main()
{
System.ServiceProcess.ServiceBase[] ServicesToRun;

// More than one user Service may run within the same process. To add
// another service to this process, change the following line to
// create a second service object. For example,
//
// ServicesToRun = New System.ServiceProcess.ServiceBase[] {new
WinService1(), new MySecondUserService()};
//
ServicesToRun = new System.ServiceProcess.ServiceBase[] { new WinService1()
}; System.ServiceProcess.ServiceBase.Run(ServicesToRun);
}

// <summary>
// Required method for Designer support - do not modify
// the contents of this method with the code editor.
// </summary>

private void InitializeComponent()
{
components = new System.ComponentModel.Container();
this.ServiceName = "WinService1";
}

// <summary>
// Set things in motion so your service can do its work.
// </summary>
```

```
protected override void OnStart(string[] args)
{
// TODO: Add code here to start your service.
}

// <summary>
// Stop this service.
// </summary>

protected override void OnStop()
{
// TODO: Add code here to perform any tear-down necessary to stop your service.
}
}
}
```

Add functionality to your service

As you saw WebService1.cs, there are two overridden functions OnStart and OnStop. The OnStart function executes when you start your service and the OnStop function gets execute when you stop a service. I write some text to a text file when you start and stop the service.

```
protected override void OnStart(string[] args)
{
  FileStream fs = new FileStream(@"c:\temp\mcWindowsService.txt" ,
FileMode.OpenOrCreate, FileAccess.Write);
StreamWriter m_streamWriter = new StreamWriter(fs);
m_streamWriter.BaseStream.Seek(0, SeekOrigin.End);
m_streamWriter.WriteLine(" mcWindowsService: Service Started
\n"); m_streamWriter.Flush();
m_streamWriter.Close();

}

// <summary>
// Stop this service.
// </summary>
protected override void OnStop()
{
FileStream fs = new FileStream(@"c:\temp\mcWindowsService.txt" ,
FileMode.OpenOrCreate, FileAccess.Write);
StreamWriter m_streamWriter = new StreamWriter(fs);
m_streamWriter.BaseStream.Seek(0, SeekOrigin.End);
m_streamWriter.WriteLine(" mcWindowsService: Service Stopped
\n"); m_streamWriter.Flush();
m_streamWriter.Close();
}
```

Install and Run the Service

Build of this application makes one exe, **mcWinService.exe**. You need to call installutil to register this service from command line.

```
installutil C:\mcWebService\bin\Debug\mcWebService.exe
```

You can use /u option to uninstall the service.

```

```

```
installutil /u
C:\mcWebService\bin\Debug\mcWebService.exe
```

Now Run the application.

Start and Stop the Service

For start and stop the service you need to go to the Computer Management. You can use Manage menu item by right clicking on My Computer, see figure 9.5.

Figure 9.5 Computer Management

Under **Services and Applications**, you will see the service **mcWinService**. Start and Stop menu item starts and stops the service, see figure 9.6.

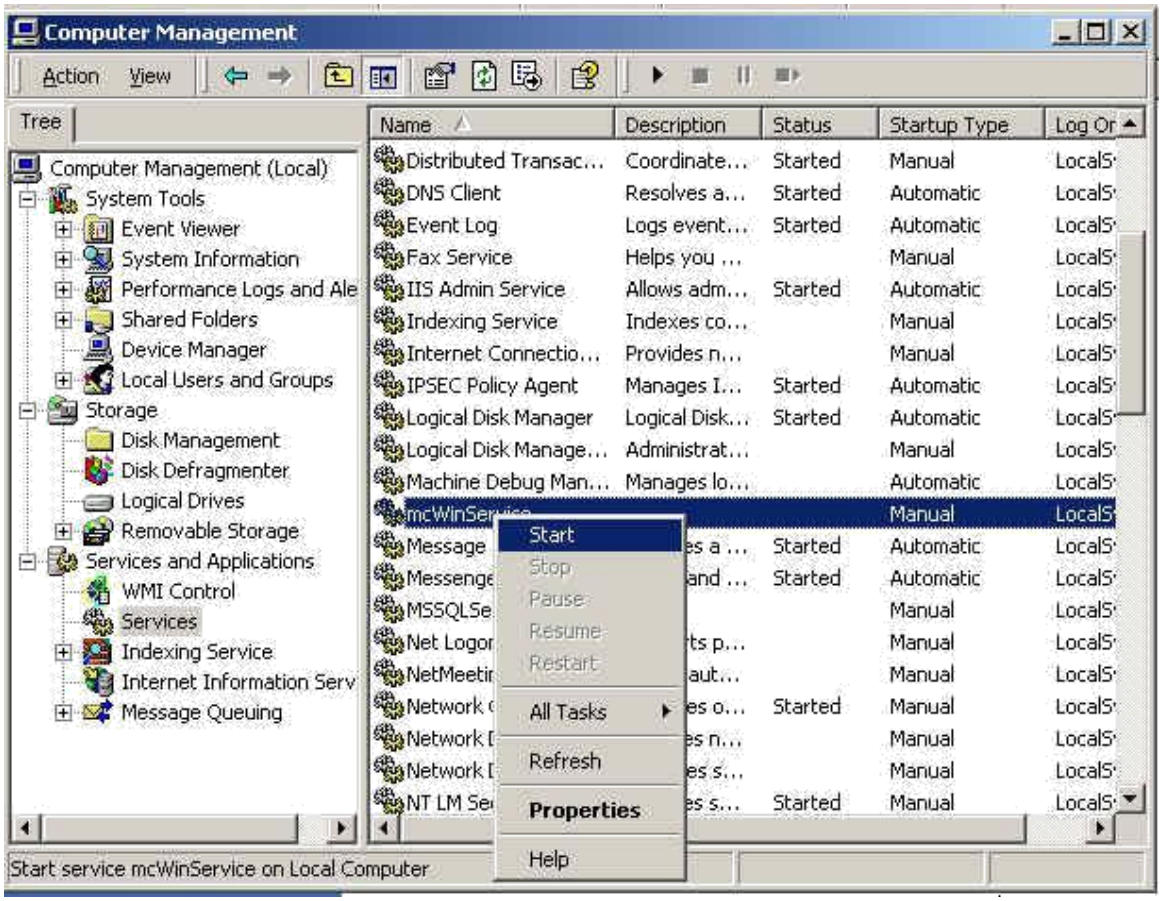


Figure 9.6 Start the Service

4.3ASP.NET and Web controls

ASP.NET is a compiled, .NET-based environment; you can author applications in any .NET compatible language, including Visual Basic .NET, C#, and JScript .NET. Additionally, the entire .NET Framework is available to any ASP.NET application. Developers can easily access the benefits of
these technologies, which include the managed common language runtime environment, type safety, inheritance, and so on.

FEATURES OF ASP.NET

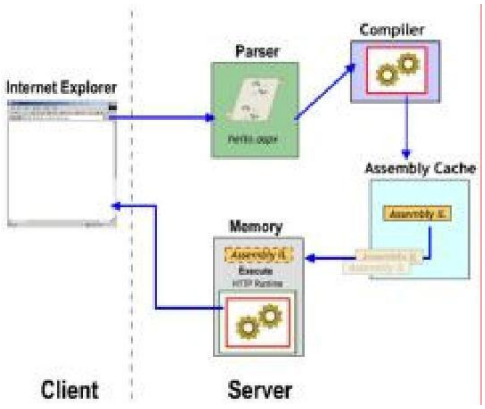
ASP.NET has better language support, a large set of new controls, XML-based components, and better user authentication. ASP.NET provides increased performance by running compiled code. ASP.NET code is not fully backward compatible with ASP. Instead of that ASP.NET having following major features:

- Better language support
- Programmable controls
- Event-driven programming
- XML-based components
- User authentication, with accounts and roles
- Higher scalability
- Increased performance - Compiled code
- Easier configuration and deployment
- Not fully ASP compatible

Language Support
ASP.NET uses ADO.NET.
ASP.NET supports full Visual Basic, not
VBScript. ASP.NET supports C# (C sharp) and
C++. ASP.NET supports JScript.

ASP.NET Controls
ASP.NET contains a large set of HTML controls. Almost all HTML elements on a page can be defined as ASP.NET control objects that can be controlled by scripts.
ASP.NET also contains a new set of object-oriented input controls, like programmable list-boxes and validation controls.
A new data grid control supports sorting, data paging, and everything you can expect from a dataset control.

ASP.NET EXECUTION MODEL
When the client requests a Web page for the first time, the following set of events take place:
1. The client browser issues a GET HTTP request to the server.
2. The ASP.NET parser interprets the source code.
3. If the code was not already compiled into a dynamic-link library (DLL), ASP.NET invokes the compiler.
4. Runtime loads and executes the Microsoft intermediate language (MSIL) code.



When the user requests the same Web page for the second time, the following set of events take place:
1. The client browser issues a GET HTTP request to the server.
2. Runtime loads and immediately executes the MSIL code that was already compiled during the user's first access attempt.

ASP.NET PAGE LIFE CYCLE
The lifetime of an ASP.NET page is filled with events. A .NET technical interview might begin with this question. A series of processing steps takes place during this page life cycle. Following tasks are performed:
• Initialization
• Instantiation of controls
• Restoration and Maintainance of State
• Running Event Handlers
• Rendering of data to the browser
The life cycle may be broken down into Stages and Events. The stages reflect the broad spectrum of tasks performed. The following stages take place:

1. **Page Request:** This is the first stage, before the page life cycle starts. Whenever a page is requested, ASP.NET detects whether the page is to be requested, parsed and compiled or whether the page can be cached from the system.
2. **Start:** In this stage, properties such as Request and Response are set. Its also determined at this stage whether the request is a new request or old, and thus it sets the IsPostBack property in the Start stage of the page life cycle.
3. **Page Initialization:** Each control of the page is assigned a unique identification ID. If there are themes, they are applied. Note that during the Page Initialization stage, neither postback data is loaded, nor any viewstate data is retrieved.
4. **Load:** If current request is a postback, then control values are retrieved from their viewstate.
5. **Validation:** The validate method of the validation controls is invoked. This sets the IsValid property of the validation control.
6. **PostBack Event Handling:** Event handlers are invoked, in case the request is a postback.
7. **Rendering:** Viewstate for the page is saved. Then render method for each control is called. A textwriter writes the output of the rendering stage to the output stream of the page's Response property.
8. **Unload:** This is the last stage in the page life cycle stages. It is invoked when the page is completely rendered. Page properties like Respone and Request are unloaded.

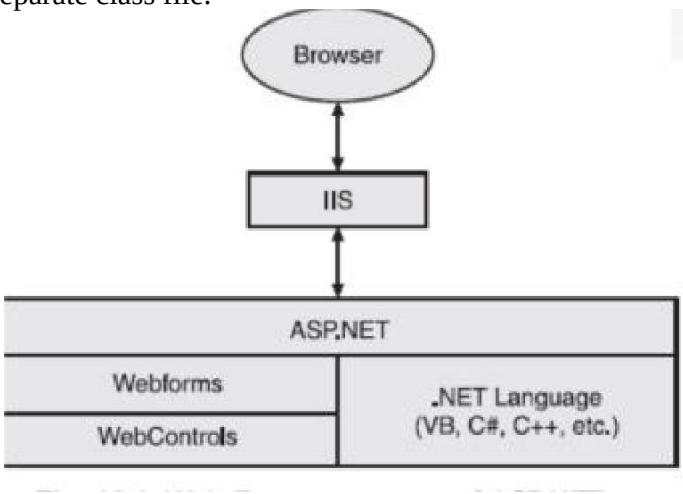
Note that each stage has its own events within it. These events may be used by developers to handle their code. Listed below are page events that are used more frequently.

• **PreInit:** Checks the IsPostBack property. To create or recreate dynamic controls. To set master pages dynamically. Gets and Sets profile propety values.
• **Init:** Raised after all controls are initialized, and skin properties are set.
• **InitComplete:** This event may be used, when we need to be sure that all initialization tasks are complete.
• **PreLoad:** If processing on a control or a page is required before the Load event.
• **Load:** invokes the OnLoad event on the page. The same is done for each child control on the page. May set properties of controls, create database connections.
• **Control Events:** These are the control specific events, such as button clicks, listbox item selects etc.
• **LoadComplete:** To execute tasks that require that the complete page has been loaded.
• **PreRender:** Some methods are called before the PreRenderEvent takes place, like EnsureChildControls, data bound controls that have a dataSourceId set also call the DataBind method. Each control of the page has a PreRender event. Developers may use the prerender event to make final changes to the controls before it is rendered to the page.
• **SaveStateComplete:** ViewState is saved before this event occurs. However, if any changes to the viewstate of a control is made, then this is the event to be used. It cannot be used to make changes to other properties of a control.
• **Render:** This is a stage, not an event. The page object invokes this stage on each control of the page. This actually means that the ASP.NET server control's HTML markup is sent to the browser.
• **Unload:** This event occurs for each control. It takes care of cleanup activities like wiping the database connectivities.

WEB FORM

Web Forms are the heart and soul of ASP.NET. Web Forms are the User Interface (UI) elements that give your Web applications their look and feel. Web Forms are similar to Windows Forms in that they provide properties, methods, and events for the controls that are placed onto them. However, these UI elements render themselves in the appropriate markup language required by the request, e.g., HTML. If you use Microsoft Visual Studio® .NET, you will also get the familiar drag-and drop interface used to create your UI for your Web application. Web Forms are made up

of two components: the visual portion (the ASPX file), and the code behind the form, which resides in a separate class file.



The Purpose of Web Forms

Web Forms and ASP.NET were created to overcome some of the limitations of ASP. These new strengths include:

• Separation of HTML interface from application logic

• A rich set of server-side controls that can detect the browser and send out appropriate markup language such as HTML

• Less code to write due to the data binding capabilities of the new server-side .NET controls

• Event-based programming model that is familiar to Microsoft Visual Basic programmers

• Compiled code and support for multiple languages, as opposed to ASP which was interpreted as Microsoft Visual Basic Scripting (VBScript) or Microsoft Jscript.

•Allows third parties to create controls that provide additional functionality

The categories of controls available are as follows:

• **Standard:** Common controls that make up 90 percent of all pages.

• **Data:** Controls used to connect to data sources (databases or XML files).

• **Validation:** Controls that can be added to a page to validate user input (for example, to ensure that certain text boxes contain data or that data has been entered in the correct format).

• **Navigation:** Controls used to provide a simple and quick solution to making a site navigable (for example, dynamic menus and breadcrumbs of hyperlinks).

• **Login:** A set of controls that make it simple to move from a completely open site to one that has personalized areas.

• **WebParts:** Controls that make it possible to create Sharepoint-style sites with dragand-droppable sections, known as Web Parts, which enable the user to rearrange their view of a site.

• **HTML:** Simple HTML elements.

12.2. STANDARD CONTROLS

Standard Controls are basic controls which are used for designing user interface for webpages. User can input data through various controls on a user interface at the client browser and can display output through these controls received from the server.

Here are some of the most commonly used controls which are grouped under the standard tab Toolbox:

• **TextBox control:** Used for entering text on a page, commonly seen on order forms on shopping sites, or for logging in to a site.

• **Button control:** From submitting an order to changing preferences on a web site, clicking a button on a page normally causes information to be sent to the server, which reacts to that information and displays a result.
• **Label control:** Used for displaying simple text in a specified position on a page. The Label control is an easy way to change the text on part of a page in response to user interaction.
• **Hyperlink control:** Used for providing hyperlink functionality on a page that enables navigation to other parts of a site, or to other resources on the Internet.
• **Image control:** Used for displaying images on a page. The server can change the image that is displayed in the control programmatically in response to user input.
• **DropDown List control:** Used for offering the user a list of options to choose from; collapses when not in use to save space.
• **Listbox control:** Used for offering a fixed-size list of items to choose from.
• **CheckBox and Radio Button controls:** Used for selecting options with either a yes/no or "this one out of many" style, respectively.

VALIDATION CONTROLS

These controls make page validation much easier and reduce the amount of code that the developer must write to perform page validation. The ASP.NET team reviewed numerous Web sites to determine the most common types of validation that were taking place on the Web. Most developers were reinventing the wheel to perform validation, so the ASP.NET team decided that Web developers needed a set of validation controls to add to their toolbox. From the start, these controls were designed to detect the version of the browser when used in client-side validation and then render the correct version of HTML for that client browser. These research efforts lead to the development of the six controls covered in this chapter. The examples in this chapter will take a look at each control and explain the most commonly used properties for each control. However, keep in mind that all of the controls share basic properties, such as font, fore color, back color, and so on, so this chapter won't discuss those properties in detail. Everything in the .NET Framework is a class, and the validation controls are no exception.

All validation controls in the .NET Framework are derived from the BaseValidator class. This class serves as the base abstract class for the validation controls and provides the core implementation for all validation controls that derive from this class. Validation controls always perform validation checking on the server. Validation controls also have complete client-side implementation that allows browsers that support DHTML to perform validation on the client. Client-side validation enhances the validation scheme by checking user input as the user enters data. This allows errors to be detected on the client before the form is submitted, preventing the round trip necessary for server -side validation. In addition, more than one validator may be used on a page to validate different aspects.

1. **RequiredFieldValidator Control**
   Use the RequiredFieldValidator control when a value is required for an input element on the Web page. This control checks whether the value of the associated input control is different from its initial value.
2. **CompareValidator Control**
   Use the CompareValidator control to make sure that a value matches a specified value. This control compares the value of an input control to another input control or a constant value using a variety of operators and types. You can also use this control to make sure that your input value is of a specific type: integer, string, and so on.
3. **RangeValidator Control**
   Use the RangeValidator control to determine whether a value falls within the specified range. It checks whether the value of the associated input control is within some minimum and maximum, which can be a constant value or the value of another control.
4. **RegularExpressionValidator Control**

Use the RegularExpressionValidator control to check a value against a regular expression. It checks whether the value of the associated input control matches the pattern of a regular expression.

5. **CustomValidator Control**
   Use the CustomValidator control to perform user-defined custom validation. This control allows custom code to perform validation on the client and/or server.
6. **ValidationSummary Control**
   Use the ValidationSummary control to capture all the validation errors from the other controls and display them on the page as a list, a bulleted list, or in single paragraph format. The errors can be displayed inline and/or in a pop-up message box.

4.4 ADO.NET

ADO.NET is the new database technology used in .NET platform. ADO.NET is the next step in the evolution of Microsoft ActiveX Data Objects (ADO). It does not share the same programming model, but shares much of the ADO functionality. The ADO.NET as a marketing term that covers the classes in the System.Data namespace. ADO.NET is a set of classes that expose the data access services of the .NET Framework. ADO.NET is a natural evolution of ADO and is built around N-Tier
application development. ADO.NET has been created with XML at its core.

CONNECTED VS DISCONNECTED

For much of the history of computers, the only environment available was the connected environment. With the advent of the Internet, disconnected work scenarios have become commonplace, and with the increasing use of handheld devices, disconnected scenarios are becoming nearly universal. Laptop, notebook, and other portable computers allow you to use applications when you are disconnected from servers or databases. In many situations, people do not work entirely in a connected or disconnected environment, but rather in an environment that combines the two approaches.

Connected

A connected environment is one in which a user or an application is constantly connected to a data source. A connected scenario offers the following advantages:

• A secure environment is easier to maintain.
• Concurrency is easier to control.
• Data is more likely to be current than in other scenarios.

A connected scenario has the following disadvantages:

• It must have a constant network connection.
• Scalability

Disconnected

A disconnected environment is one in which a user or an application is not constantly connected to a source of data. Mobile users who work with laptop computers are the primary users in disconnected environments. Users can take a subset of data with them on a disconnected computer, and then merge changes back into the central data store.

A disconnected environment provides the following advantages:

• You can work at any time that is convenient for you, and can connect to a data source at any time to process requests.
• Other users can use the connection.
• A disconnected environment improves the scalability and performance of applications.

A disconnected environment has the following disadvantages:

• Data is not always up to date.
• Change conflicts can occur and must be resolved.

ADVANTAGES OF ADO.NET

ADO.NET provides the following advantages over other data access models and components:

Interoperability. ADO.NET uses XML as the format for transmitting data from a data source to a local in-memory copy of the data.
Maintainability. When an increasing number of users work with an application, the increased use can strain resources. By using n-tier applications, you can spread application logic across additional tiers. ADO.NET architecture uses local in-memory caches to hold copies of data, making it easy for additional tiers to trade information.
Programmability. The ADO.NET programming model uses strongly typed data. Strongly typed data makes code more concise and easier to write because Microsoft Visual Studio .NET provides statement completion.
Performance. ADO.NET helps you to avoid costly data type conversions because of its use of strongly typed data.
Scalability. The ADO.NET programming model encourages programmers to conserve system resources for applications that run over the Web. Because data is held locally in in memory caches, there is no need to retain database locks or maintain active database connections for extended periods.

.NET DATA PROVIDER

A .NET data provider is used for connecting to a database, executing commands, and retrieving results. Those results are either processed directly, or placed in an ADO.NET **DataSet** in order to be exposed to the user in an ad-hoc manner, combined with data from multiple sources, or remoted between tiers. The .NET data provider is designed to be lightweight, creating a minimal layer between the data source and your code, increasing performance without sacrificing functionality. The ADO.NET object model includes the following data provider classes:
1. SQL Server .NET Data Provider
2. OLE DB .NET Data Provider
3. Oracle .NET Data Provider
4. ODBC .NET Data Provider
5. Other Native .NET Data Provider

1. SQL Server Data Provider: To use the SQL Server .NET Data Provider, you need to include the **System.Data.SqlClient** namespace in your applications. Using this provider is more efficient than using the OLE DB .NET Data Provider because it does not pass through an OLE DB or Open Database Connectivity (ODBC) layer. It provides optimized access to SQL Server 2000 and SQL Server 7.0 databases.
2. OLE DB Data Provider: To use the OLE DB .NET Data Provider, you need to include the **System.Data.OleDb** namespace in your applications. .NET Provides access to SQL Server versions 6.5 and earlier. It also provides access to other databases, such as Oracle, Sybase, DB2/400, and Microsoft Access.
3. Oracle .NET Data Provider: To use the Oracle Database, a native Oracle .NET data driver is the best choice**.** To use the Oracle .NET Data Provider, you need to include the **System.Data.OracleClient** namespace in your applications. Oracle itself also provides a .NET data provider, referenced as Oracle.DataAccess.Client. This is a separate download that you have to get from Oracle. Whether you use a .NET data provider from the database vendor or just use the one provided with the .NET framework is your choice.
4. ODBC .NET Data Provider: If you have a data source for which no native or OLE DB provider is available, the ODBC .NET data provider is good alternative because most database provide an ODBC interface. It is referenced with this *using* directive: **System.Data.Odbc**
5. Other Native .NET Data Provider: If there is a native .NET data provider available specifically for your database, then you may want to use that .NET data provider instead. Many other database vendors and third-party companies provide native .NET data providers; the choice between using the native providers and using something generic like the ODBC provider will depend on your circumstances. If you value portability over

performance, then go generic. If you want to get the best performance or make the best use of a particular database's features, go native.

Type of class	Purpose	Name in System.Data.OleDb namespace	Name in System.Data SqlClient Namespace
Connection	Establishes and manages a connection to a database. Very similar to the Connection object in ADO.	OleDbConnection	SQLConnection
Command	Carries out an operation while connected to the database. Conceptually similar to the Command object in ADO, but it contains a different object model and has new functionality.	OleDbCommand	SQLCommand
DataReader	Presents a stream of data generated by a query operation on a database. Fulfills similar function as a forward-only, read-only Recordset in ADO.	OleDbDataReader	SQLDataReader
DataAdapter	Fetches data from a database and transfers it into a DataSet and then later transfers changes in the data back into the original data store.	OleDbDataAdapter	SQLDataAdapter

Program to Display the content of Database

```
using System;
using System.Data;
using System.Data.OleDb;
class test
{
public static void Main()
{
OleDbConnection conn = new OleDbConnection(@"Provider =
Microsoft.Jet.OLEDB.4.0; Data Source=c:\lalit\data.mdb");
OleDbDataAdapter adp= new OleDbDataAdapter("Select * from
emp",conn); DataTable tbl =new DataTable();
adp.Fill(tbl);
foreach(DataRow row in tbl.Rows)
Console.WriteLine("{0,10}{1,10}{2,10}",row[0],row[1],row[2]);
}
}
```

4.5 Distributed Applications

Distributed applications are those applications which uses distributed components. Distributed components can be accessed across the network, even thogh it may be different types of application (Web, standalone, PDA's etc.). You can increase the scalability of your application by making component distributed. Commonly, we have three steps to construct a simple distributed application using C#:

1. Create **remoting** objects, which will either inherit MarshalByRefObject or be serialized.
2. Make a **host** (server) who will provide remoting server using channel for remoting client to call for.
3. Make **clients** who will call the remote objects on server, which will establish a connection channel to the appointed port on server and then using the Well- Known objects.

Remote Object

A remote object is implemented in a class that derives from System.MarshalByRefObject. A client doesn't call the methods directly; instead a proxy object is used to invoke methods on the remote object. Every public method that we define in the remote object class is available to be called from clients. All the objects passed by value must be serializable.
To create MyRemoteObject.dll Take .NET IDE New Project->YourName.Samples (for example, RemotingSamples.cs). (By Default.,Class1.cs will come. Change that file name MyRemoteObject.cs.)

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
namespace RemotingSamples
{
public class RemoteObject : MarshalByRefObject
{
//////////////////////////////////////////////////constructor
public RemoteObject()
{
Console.writeline("Remote object activated");
}
//////////////////////////////////////////////////return message
reply public String ReplyMessage(String msg)
{
Console.WriteLine("Client : "+msg);
//print given message on console
return "Server : Yeah! I'm here";
}
}
}
```

The remote object must be compiled as follows to generate remoteobject.dll which is used to generate server and client executable.

csc /t:library /debug /r:System.Runtime.Remoting.dll RemotingSamples.cs

Server

The remote object needs a server process where it will be instantiated. This server has to create a channel and put it into listening mode so that clients can connect to this channel. For creating server, Take .NET IDE-> New Project named "server". This is console application is the server application used to register remote object to be access by client application. First, of all choose channel to use and register it, supported channels are HTTP, TCP and SMTP. We have used here TCP. Then register the remote object specifying its type.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
namespace RemotingSamples
```

```
{
public class Server
{
//////////////////////////////////////////////////////
///constructor
public Server()
{
}
//////////////////////////////////////////////////////
///main method
public static int Main(string [] args)
{
//select channel to communicate TcpChannel
chan = new TcpChannel(8085);
ChannelServices.RegisterChannel(chan); //register channel //register remote object

RemotingConfiguration.RegisterWellKnownServiceType( Type.GetType("RemotingSamples.RemoteObject,object"),

"RemotingServer", WellKnownObjectMode.SingleCall); //inform console

Console.WriteLine("Server
Activated"); return 0;
}
}
}
```

The server must be compiled as follows to produce server.exe.

csc /debug /r:remoteobject.dll /r:System.Runtime.Remoting.dll server.cs

Client

This is the client application and it will call remote object method. First, of all client must select the channel on which the remote object is available, activate the remote object and then call proxy's object method return by remote object activation.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using RemotingSamples;
namespace RemotingSamples
{
public class Client
{
//////////////////////////////////////////////////////
///constructor
public Client()
{
}
//////////////////////////////////////////////////////
///main method
public static int Main(string [] args)
{
```

```
//select channel to communicate with server
TcpChannel chan = new TcpChannel();
ChannelServices.RegisterChannel(chan);
RemoteObject remObject = (RemoteObject)
Activator.GetObject(typeof(RemotingSamples.RemoteObject),
"tcp://localhost:8085/RemotingServer");
if (remObject==null)
Console.WriteLine("cannot locate
server"); else
remObject.ReplyMessage("You
there?"); return 0;
}
}
}
```

The client must be compiled as follows in order to produce client.exe

csc/debug/r:remoteobject.dll /r:System.Runtime.Remoting.dll client.cs

Advantage and Disadvantage of Remoting

Lease-Based Lifetime

Distributed garbage collection of objects is managed by a system called 'leased based lifetime'. Each object has a lease time, and when that time expires the object is disconnected from the .NET runtime remoting infrastructure. Net remoting takes a lease-base lifetime of the object that is scalable.

Call Context

Additional information can be passed with every method call that is not part of the argument with the help of SOAP Header.

Distributed Identities

If we pass a reference to a remote object, we will access the same object using this reference.

Advantage Over Web Services

1. It works using purely Common Type System.
2. It supports high speed binary over top/ip communication.

Advantage Over COM/DCOM

1. It does no have extra interface language (IDL)
2. It works using purely managed code
3. It's using Common Type System. No Safearrays etc

Disadvantages

1. It is not an open standard like web services.
2. It is not as widespread and established ad DCOM.
3. Less support for transactions, load balancing compared with DCOM.

4.6 Unsafe Mode

When you want to compile program using command line switch you type the program name after the compiler name; for example if your program name is prog1.cs then you will compile this:

```
csc prog1.cs
```

This works fine for safe code while you are programming. Microsft added one more switch to command line compiler of C# for writing unsafe code. Now if you want to write unsafe code then you have to specify the /unsafe command line switch with command line compiler otherwise the compiler gives an error. if you want to write unsafe code in your program then you compile your programas follows:

csc /unsafe prog1.cs

Here *prog1.cs* is the name of the program. If you compile your program which has unsafe code without using the /unsafe switch then compiler gives error.

To set this compiler option in the Visual Studio development environment following steps are required :

- Open the project's **Properties** page.
- Click the **Build** property page.
- Select the **Allow Unsafe Code** check box.

4.7 GDI APPLICATIONS

Using Graphical Device Interface (GDI) objects in earlier versions of Visual Studio was a pain. In Visual Studio .NET, Microsoft has taken care of most of the GDI problems and have made it easy to use. GDI+ is the next evolution of GDI in .NET Visual Studio. GDI+ resides in **System.Drawing.dll** assembly. All GDI+ classes are reside in the **System.Drawing**, **System.Text**, **System.Printing**, **System.Internal**, **System.Imaging**, **System.Drawing2D** and **System.Design** namespaces.
The first class we must discuss is the **Graphics** class. After the Graphics class, we will discuss other useful GDI+ classes and structures such as Pen, Brush, and Rectangle.

The Graphics Class

The **Graphics** class encapsulates GDI+ drawing surfaces. Before drawing any object (for example circle, or rectangle) we have to create a surface using Graphics class. Generally we use Paint event of a Form to get the reference of the graphics. Another way is to override **OnPaint** method.
Here is how you get a reference of the Graphics object:

```
private void form1_Paint(object sender, PaintEventArgs e)
{
Graphics g = e.Graphics;
}
OR
protected override void OnPaint(PaintEventArgs e)
{
Graphics g = e.Graphics;
}
```

Once you have the Graphics reference, you can call any of this class's members to draw various objects. Here are some of Graphics class's methods:

Methods	Description
DrawArc	Draws an arc from the specified ellipse.
DrawBezier	Draws a cubic bezier curve.
DrawBeziers	Draws a series of cubic Bezier curves.
DrawClosedCurve	Draws a closed curve defined by an array of points.
DrawCurve	Draws a curve defined by an array of points.
DrawEllipse	Draws an ellipse.
DrawImage	Draws an image.
DrawLine	Draws a line.
DrawPath	Draws the lines and curves defined by a GraphicsPath.
DrawPie	Draws the outline of a pie section.
DrawPolygon	Draws the outline of a polygon.
DrawRectangle	Draws the outline of a rectangle.
DrawString	Draws a string.
FillEllipse	Fills the interior of an ellipse defined by a bounding rectangle.
FillPath	Fills the interior of a path.
FillPie	Fills the interior of a pie section.
FillPolygon	Fills the interior of a polygon defined by an array of points.
FillRectangle	Fills the interior of a rectangle with a Brush.
FillRectangles	Fills the interiors of a series of rectangles with a Brush.
FillRegion	Fills the interior of a Region.

After creating a **Graphics** object, you can use it draw lines, fill shapes, draw text and so on.
The major objects are:

Object	Uses
Brush	Used to fill enclosed surfaces with patterns, colors, or bitmaps.
Pen	Used to draw lines and polygons, including rectangles, arcs, and pies
Font	Used to describe the font to be used to render text
Color	Used to describe the color used to render a particular object. In GDI+ color can be alpha blended

Example of Drawing an Arc

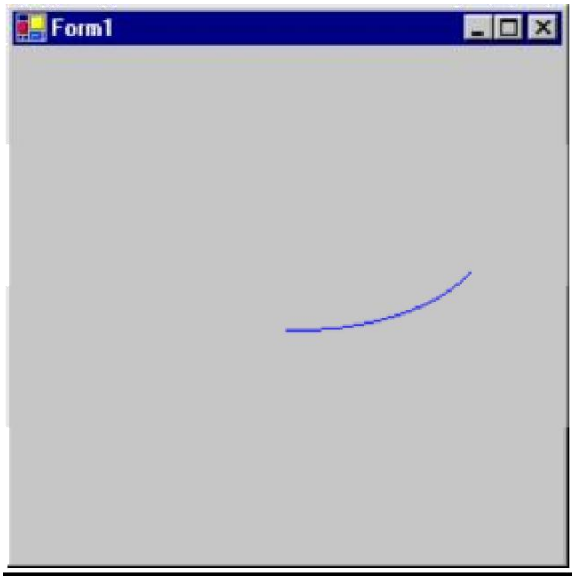
DrawArc function draws an arc. This function takes four arguments. First is the Pen. You create a pen by using the **Pen** class. The Pen constructor takes at least one argument, the color or the brush of the pen. Second argument width of the pen or brush is optional.

Pen pn = new Pen(Color.Blue); or Pen pn = new Pen(Color.Blue, 100);

The second argument is a rectangle. You can create a rectangle by using Rectangle structure. The Rectangle constructor takes four int type arguments and they are left and right corners of the rectangle.

```
Rectangle rect = new Rectangle(50, 50, 200, 100);
protected override void OnPaint(PaintEventArgs pe)
{
Graphics g = pe.Graphics ;
Pen pn = new Pen( Color.Blue );
Rectangle rect = new Rectangle(50, 50, 200, 100);
g.DrawArc( pn, rect, 12, 84 );
}
```

The output looks like figure



UNIT-5

Assemblies and Attributes

5.1 Assemblies

What is an assembly?

- An Assembly is a logical unit of code
- Assembly physically exist as DLLs or EXEs
- One assembly can contain one or more files
- The constituent files can include any file types like image files, text files etc. along with DLLs or EXEs
- When you compile your source code by default the exe/dll generated is actually an assembly
- Unless your code is bundled as assembly it can not be used in any other application
- When you talk about version of a component you are actually talking about version of the assembly to which the component belongs.
- Every assembly file contains information about itself. This information is called as Assembly Manifest.

What is assembly manifest?

- Assembly manifest is a data structure which stores information about an assembly
- This information is stored within the assembly file(DLL/EXE) itself
- The information includes version information, list of constituent files etc.

What is private and shared assembly?

The assembly which is used only by a single application is called as private assembly. Suppose you created a DLL which encapsulates your business logic. This DLL will be used by your client application only and not by any other application. In order to run the application properly your DLL must reside in the same folder in which the client application is installed. Thus the assembly is private to your application.

Suppose that you are creating a general purpose DLL which provides functionality which will be used by variety of applications. Now, instead of each client application having its own copy of DLL you can place the DLL in 'global assembly cache'. Such assemblies are called as shared assemblies.

What is Global Assembly Cache?

Global assembly cache is nothing but a special disk folder where all the shared assemblies will be kept. It is located under <drive>:\WinNT\Assembly folder.

How assemblies avoid DLL Hell?

As stated earlier most of the assemblies are private. Hence each client application refers assemblies from its own installation folder. So, even though there are multiple versions of same assembly they will not conflict with each other. Consider following example :

- You created assembly Assembly1
- You also created a client application which uses Assembly1 say Client1
- You installed the client in C:\MyApp1 and also placed Assembly1 in this folder
- After some days you changed Assembly1
- You now created another application Client2 which uses this changed Assembly1
- You installed Client2 in C:\MyApp2 and also placed changed Assembly1 in this folder
- Since both the clients are referring to their own versions of Assembly1 everything goes on smoothly

Now consider the case when you develop assembly that is shared one. In this case it is important to know how assemblies are versioned. All assemblies has a version number in the form:

major.minor.build.revision

If you change the original assembly the changed version will be considered compatible with existing one if the major and minor versions of both the assemblies match.

When the client application requests assembly the requested version number is matched against available versions and the version matching major and minor version numbers and having most latest build and revision number are supplied.

How do I create shared assemblies?

Following steps are involved in creating shared assemblies :

3) Create your DLL/EXE **source code**
4) Generate unique assembly name using **SN utility**
5) Sign your DLL/EXE with the private key by modifying **AssemblyInfo** file
6) **Compile** your DLL/EXE
7) Place the resultant DLL/EXE in global assembly cache using **GAC utility**

How do I create unique assembly name?

Microsoft now uses a public-private key pair to uniquely identify an assembly. These keys are generated using a utility called SN.exe (SN stands for shared name). The most common syntax of is :

sn -k mykeyfile.snk

Where k represents that we want to generate a key and the file name followed is the file in which the keys will be stored.

How do I sign my DLL/EXE?

Before placing the assembly into shared cache you need to sign it using the keys we just generated. You mention the signing information in a special file called AssemblyInfo. Open the file from VS.NET solution explorer and change it to include following lines :

[assembly:AssemblyKeyFile("file_path")]

Now recompile the project and the assembly will be signed for you.

Note : You can also supply the key file information during command line compilation via /a.keyfile switch.

How do I place the assembly in shared cache?

Microsoft has provided a utility called AL.exe to actually place your assembly in shared cache.

GACUtil /i my_dll.dll

Now your dll will be placed at proper location by the utility.

Hands On...

Now, that we have understood the basics of assemblies let us apply our knowledge by developing a simple shared assembly.

In this example we will create a C#.NET component called **SampleGAC** (GAC stands for Global Assembly Cache). We will also create a key file named **sample.snk.** We will sign our component with this key file and place it in Global Assembly Cache.

- **Step 1 : Creating our sample component**

Here is the code for the component. It just includes one method which returns a string. using System;

```
namespace BAJComponents
{
      public class Sample
            {
             public string GetData()
                        {
                  return "hello world";
```

```
        }
    }
}
```

- **Step 2 : Generate a key file**

To generate the key file issue following command at command prompt. sn -k sample.snk
This will generate the key file in the same folder

- **Step 3 : Sign your component with the key**

```
using System;
[assembly:AssemblyKeyFile("C:\Sample.snk")]
namespace BAJComponents
{
    public class Sample
        {
        public string GetData()
                    {
            return "hello world";
          }
    }
}
```

- **Step 4 : Compile your file again**

csc /out:sampleGAC.dll /target:library sampleGAC.cs

- **Step 4 : Host the signed assembly in Global Assembly Cache**

We will use GAC utility to place the assembly in Global Assembly Cache. GACUtil /i sampleGAC.dll
After hosting the assembly just go to WINNT\Assembly folder and you will find your assembly listed there. Note how the assembly folder is treated differently that normal folders.

- **Step 5 : Test that our assembly works**

Now, we will create a sample client application which uses our shared assembly. Just create a sample code as listed below :

```
using System;
using BAJComponents;
public class SampleTest
{
    public static void Main()
    {
        Sample x = new sample()
        string s =x.GetData()
        Console.WriteLine(s)
    }
}
```

Compile above code using :
csc sampletest.cs /t:exe /r:<assembly_dll_path_here>
Now, copy the resulting EXE in any other folder and run it. It will display "Hello World" indicating that it is using our shared assembly.

5.2 GENERICS

Generics are a new feature in the .Net framework 2.0+ which make it possible to design classes and methods that do not specify data types until the class or method is declared and instantiated by client code.

Generics are supplied by the System.Collection.Generics namesapce.

Nullable Types

System.Nullable<int> a;

"a" can have an int value as well as "null" value.

Means: a = **null**; // valid statement.

You can test nullable types like

as: if(a == **null**)

{

}

OR

if(a.**HasValue**)

{

}

The short hand of **System.Nullable<int>** is simply **int? .** So you can use like

as: **int? a;** instead of **System.Nullable<int> a;**

Some problem with

code: int? op1 = 5;

int result = op1 * 2;

will not compile.

To get result use following

code: int? op1 = 5;

int result = (int) op1 *

2; or

int result = op1.Value * 2;

---

int? op1 = null;

int? op2 = 5;

int? result = op1 * op2;

result = null

By using a generic type parameter you can write a single class that other client code can use without incurring the cost or risk of casts or boxing operations.

The ?? Operator (*null coalescing operator*)

it is a binary operator that enables you to supply an alternative value to use for expresions that might evaluate null

op1 ?? op2;

equivalent

op1 == null ? op2 : op1 ;

int? op1 = null;

int result = op1 * 2 ??

5; *result = 10*

GENERIC METHOD

using System;

using System.Collections.Generic;

```
class Generics
{
   static void Main(string[] args)
   {
      // create arrays of various types
      int[] intArray = { 1, 2, 3, 4, 5, 6 };
      double[] doubleArray = { 1.0, 2.0, 3.0, 4.0, 5.0
      }; char[] charArray = { 'A', 'B', 'C', 'D', 'E' };

      DisplayArray(intArray);
      DisplayArray(doubleArray);
      DisplayArray(charArray);

      Console.ReadLine();
   }

   // generic method displays array of any type
   static void DisplayArray<E>(E[] array)
   {
      Console.WriteLine("Display array of type " + array.GetType() +
      ":"); foreach (E element in array)
         Console.Write(element + " ");
   }
}
```

GENERIC CLASS

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

class Generics
{
   static void Main(string[] args)
   {
      MyGeneric<int> mygenericint = new
      MyGeneric<int>(); mygenericint.GenericField = 13;
      mygenericint.GenericMethod(42);

      MyGeneric<string> mygenericstring = new
      MyGeneric<string>(); mygenericstring.GenericField = "xxx";
      mygenericstring.GenericMethod("xxx");

      // These lines will cause a compile error
      MyGeneric<int> mygenericint2 = new
      MyGeneric<int>(); mygenericint2.GenericField = "xxx";
      mygenericint2.GenericMethod("xxx");
   }
}

public class MyGeneric<T>
{
```

```
    public T GenericField;
    public void GenericMethod(T t)
    {
        Console.WriteLine("GenericMethod parameter type: " + t.GetType());
    }
}
```

5.3 Attributes

Introduction

Usinng an attribute is a way to add special meaning to our method and cause it to act in a certain way. Before this was available, developers didn't have a way to define their own attributes. DotNet paved the way for developers and opened new horizons to conquer. Attributes are like adding behaviours to methods, classes, properties structures, events, modules, and so forth. It means we can enforce certain constraints on those methods, classes, properties and vice-versa to behave in the way specify to them.

One more added feature is that, before DotNet, if we made a class or DLL and a newer version came along, the older version that should be removed or should not be used still exists. This causes a problem called DLL Hell. DotNet solved this problem by the concept of versioning. This means that the same DLL with the same name, containing some new methods with older methods can co-exist with a different version number.

Let us start to learn to use attributes in C# code:

Case 1

I had a method in my old DLL. Now, I have updated my DLL and added two new methods and one new method that is the upgraded version of previous DLL.

What should happen is that my previous programs that use this DLL should work properly. Or, if I intend to use the old method in the new program, it should tell me that the old method is obsolete and I should use the new method.

Before DotNet, this wasn't possible. DotNet has removed this hurdle from the path of developers. Let us see how can we do this stuff:

```
Using System;
Namespace MyExample
{
  Class MyAttribute
  {
    [Obsolete()]
    public static void SaySomething(string str)
    {
      Console.WriteLine(str);
    }
    public static void Talk(string str)
    {
      Console.WriteLine(str);
    }

    static void Main()
    {
      SaySomething("Hello to Sufyan");
```

```
      Console.ReadLine();
    }
  }
}
```

Here, when we call SaySomething("Hello to Sufyan"), it will execute the code but in the output window a warning message will be displayed, showing us that the SaySomething method is obsolete.

Still, it doesn't ask us to use the new talk method SaySomething instead. To display a customized message, we will customize the obsolete attribute like this:

```
[Obsolete("SaySomething() method is now
      Obsolete. Please use Talk()")]
```

By using this approach, we can add additional meaning to our attribute.

> **Note:** We can have multiple attribute statements before a method or class. In such cases, if one of the statements is true, it will allow access to that particular method.

There are many attributes that come with DotNet.

- Conditional
- DllImport
- Obsolete
- Serializable

Conditional attribute

By using the Conditional attribute, we are following a scenario that if a particular condition is true, the user will have access to that specific method. Suppose you wanted your specific method to run only if Internet Explorer is found on the system. Applying this type of security on a method was not easy in the past. Now, it is possible by applying conditional attributes.

Case 2

Suppose you want a method to run if program is in Debug mode. We will write the conditional statement as follows:

```
[Conditional(.DEBUG.)]
public static void Help(String str)
{
  Console.WriteLine(str);
}
```

If one calls this method in Main(), it will run only if the program is in Debug mode. If you turn it off in the release mode, it will not run.

DLL Import

Before DotNet, if one wanted to access the core Windows API, he could add a reference and use the library provided by Windows SDK. In DotNet, if we want to access those core features of DotNet, we use DllImport. Component DLLs aren't accessed this way. They are accessed by making a reference to them. Normal DLLs are accessed by using the DllImport attribute.

Case 3

```
using System;
System.Runtime.InteropServices;
class Beeper
{
  [DllImport(.kernel32.dll.)]
  Public static extern bool Beep(int frequency,int
  duration); static void Main()
  {
    Beep(1000,111);
  }
```

}
So far, we have learned how to use attributes in DotNet. Still, we haven't learn how to create our own custom attributes with specified behaviour that we assign.

Steps in Creating a Custom Attribute

1. Define the attribute's usage.
2. Extend our class with AttributeClass.
3. Define the behaviours to the class.

Attribute behaviours can be of these types:

- All—Any application element
- Assembly—Attribute can be applied to an assembly
- Class—Attribute can be applied to a class
- Constructor—Attribute can be applied to a constructor
- Delegate—Attribute can be applied to a delegate
- Enum—Attribute can be applied to an enumeration
- Event—Attribute can be applied to an event
- Field—Attribute can be applied to a field
- Interface—Attribute can be applied to an interface
- Method—Attribute can be applied to a method
- Module—Attribute can be applied to a module
- Parameter—Attribute can be applied to a parameter
- Property—Attribute can be applied to a property
- ReturnValue—Attribute can be applied to a return value
- Struct—Attribute can be applied to a structure

The second step is to extend the class with the Attribute class. Basically, our attribute is a class that defines the behaviours of our attribute. For example:

```
...
public class
MySpecialAttribute:Attribute {...
```

A point to be noted here is that the name of Attribute we are going to make is MySpecial.
We didn't suffix the word Attribute after MySpecial. DotNet automatically suffixed it.
Let us create a sample custom attribute that will execute the method if the regKey provided to the Attribute parameter is correct:

```
using System;
namespace RegKeyAttributeTestor
{
  [AttributeUsage(AttributeTargets.Method|AttributeTargets.Struct,
              AllowMultiple=false,Inherited=false]
  public class MyAttribute:Attribute
  {
    private string regKey="a12nf";
    public MyAttribute(string regKey)
    {
      if(this.regKey==regKey)
      {
        Console.WriteLine("Permitted to use this App");
      }
      else
      {
        Console.WriteLine("Not registered to use this
```

```
      }
    }
  } //End Attribute class code
  class useAttrib
  {
    [MyAttribute("hello")]
    public static string SayHello(string str)
    {
      return str;
    }
    static void Main()
    {
      Console.WriteLine(SayHello("Hello to
      Sufyan")); Console.ReadLine();
    }
  }

}
AttributeUsage(AttributeTargets.Method|AttributeTargets.Struct,
          AllowMultiple=false, Inherited=false)]
```

Here, multiple Attribute targets are declared by using "|" between different targets. Allows multiple=false means multiple attributes can be used with this attribute. Inherited=false shows that if some class extends a class that uses this attribute and calls a method that is bound to this attribute, that class has no access to this attribute unless this property is set to true.

Reading Metadata from Assemblies

We first write our custom attribute:

```
using System;

namespace Sufyan
{
  [AttributeUsage(AttributeTargets.Method,AllowMultiple=false,
             Inherited=false)]
  public class RegKeyAttribute : Attribute
  {
    private string regKey;

    public RegKeyAttribute(string VRegKey)
    {
      this.regKey = VRegKey;
      if (this.regKey==.hello.)
      {
        Console.WriteLine("Aha! You're Registered");
      }
      else
      {
        Console.WriteLine("Oho! You're not Registered");
      }
    }
  }
}
```

Now, we shall write our code to reference it through Assembly;.

Steps

First, compile the first example as a Class library; it will generate a .dll file. We will place this .dll file in the bin folder of our second example we are going to make now.

```
using System;
using System.Reflection;
using Sufyan;

namespace AdvancedDotNet1
{
  class Classic
  {
    static void Main(string[] args)
    {
      Assembly suf = Assembly.Load("RegKey");
      Type KIA=suf.GetType();
      []KO=Attribute.GetCustomAttributes(KIA);
      Object Regist =KO[0];
      Console.WriteLine("Registeration Code is :"+
                  Regist.ToString() );
      Console.ReadLine();
    }
  }
}
```

System.Reflection is used to retrieve info from the assembly metadata.

To use this in another class, verify that a specific method will execute if that particular developer is registered with me and I have issued him the license key to use.

```
class useAttrib
  {
    [RegKeyAttribute ("hello")]
    public static string SayHello(string str)
    {
      return str;
    }
    static void Main()
    {
      Console.WriteLine(SayHello("Hello to
      Sufyan")); Console.ReadLine();
    }
  }
```

Have a close look at this code. I have provided the Hello string as a parameter to the attribute. When we run this application, if we pass a string other then "hello", it will say that you are not registered. We can further modify it according to our need.