CHAPTER 2
C# BASICS

2.1 Introduction

C#, pronounced as C Sharp, is an object-oriented programming language developed by Microsoft in the early 2000s, led by Anders Hejlsberg. It is part of the .NET framework and is intended to be a simple general-purpose programming language that can be used to develop different types of applications, including console, windows, web and mobile applications.

C# is designed for Common Language Infrastructure (CLI), which consists of the executable code and runtime environment that allows use of various high-level languages on different computer platforms and architectures.

2.1.1 Features of C#

C# has the following features.

i. **Simplicity**: C# is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc. C# code does not require header files. All code is written inline.

ii. **Modern programming language**: C# programming is based upon the current trend and it is very powerful and simple for building scalable, interoperable and robust applications. It supports number of modern features, such as
    - Automatic Garbage Collection;
    - Error Handling features;
    - Modern debugging features; and
    - Robust Security features.

iii. **Object- Oriented programming language**: In C#, everything is an object. There are no more global functions, variable and constants. It supports all three object oriented feature:
    - Encapsulation;
    - Inheritance; and
    - Polymorphism.

iv. **Compatible with other programming languages**: C# enforces the .NET common language specifications (CLS) and therefore allows interoperation with other .NET language.

v. **Structured Programming Language**: C# is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

vi. **Rich Library**: C# provides a lot of inbuilt functions that makes the development fast.

vii. **Fast Speed**: The compilation and execution time of C# language is fast.

viii. **Type Safe**: C# type safe code can only access the memory location that it has permission to execute. Therefore it improves a security of the program.

ix. **Feature of Versioning**: Making new versions of software module work with the existing applications is known as versioning. It is achieve by using the keywords new and override.

2.1.2 Simple C# Program

Example 1:

```
using System;
namespace HelloWorld
{
//My first C# program to display the words Hello World
class HelloWorld
{
static void Main(string[] args)
{
Console.WriteLine("Hello, World!");
}
}
}
```

2.1.3 Structure of a C# Program

C# program consists of the following things:

i. **Directive**

The “using” keyword is used to contain the System namespace in the program. Every program has multiple “using” statements.

ii. **Namespace**

A namespace is simply a grouping of related code elements. These elements include classes, interfaces, enums and structs. C# comes with a large amount of pre-written code that are organised into different namespaces. The System namespace contains code for methods that allow us to interact with our users. The WriteLine( ) method was used in the previous example.

These are pre-written namespaces provided by Microsoft but we can declare our own namespace. An advantage of declaring namespaces is that it prevents naming conflicts. Two or more code elements can have the same name as long as they belong to different namespaces.

The program in **Example 2** defines two namespaces, both of which contain a class named MyClass. This is allowed in C# as the two classes belong to different namespaces (First and Second).

Example 2:

```
namespace First
{
      class MyClass
      {
      }
}
namespace Second
{
      class MyClass
      {
      }
}
```

iii. Class Declaration

The class HelloWorld contains the data and method definitions that our program uses.

iv. The Main ( ) Method

This is the entry point for all C# programs. The main method states what the class does when executed.

v. Statements and Expressions

The **WriteLine** is a method of the Console class distinct in the System namespace. This statement causes the message "Hello, World!" to be displayed on the screen.

vi. Comments

Comments start with two forward slashes (//). Alternatively, we can also use /* …*/ for multiline comments. Comments make our code readable to other programmers. They are ignored by the compiler.

2.1.4 Important things to note in C#

i. C# is case sensitive;
ii. C# program execution starts at the Main method;
iii. All C# expression and statements must end with a semicolon (;)
iv. File name is different from the class name. This is unlike Java.

2.2 Data Types

The variables in C#, are categorized into the following types:

i. Value types;
ii. Reference types; and
iii. Pointer types.

Value types – Value type variables can be assigned a value directly. They are derived from the class System.ValueType.

The value types directly contain data. Some examples are *int*, *char*, and *float*, which store numbers, alphabets, and floating point numbers, respectively.

Example 3:

```
int i = 75;

float f = 53.005f;

double d = 2345.7652;

bool b = true;
```

Reference types – Reference types do not contain the actual data stored in a variable, but they contain a reference to the variables. In other words, they refer to a memory location. 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. Therefore the reference types are:

Predefined Reference Types

i. Object; and
ii. String.

User Defined Reference Types

i. Classes;
ii. Interfaces;
iii. Delegates; and
iv. Arrays.

Object Type is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or user defined types.

String Type allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type.

Example 4:

String str = “Programming Class”;

Char to String

```
    string s1 = "hello";
    char[] ch = { 'c', 's', 'h', 'a', 'r', 'p' };
    string s2 = new string(ch);
    Console.WriteLine(s1);
    Console.WriteLine(s2);
```

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;
```

2.3 Variable

A variable is a name given to a memory location that can be manipulated by our programs. All operations done on variables affect the memory location. The value stored in a variable can be changed during program execution.

2.3.1 Type of Variables

- Local variables
- Instance variables or Non – Static Variables
- Static Variables or Class Variables
- Constant Variables
- Read-only Variables

Local Variable

A variable defined within a block or method or constructor is called local variable.

Example 5:

```
    static void Main(String args[])
    {
    // Declare local variable
    int age = 24;
    Console.WriteLine("Student age is : " + age);
    }
```

Instance Variables

As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed. Unlike local variables, we may use access specifiers for instance variables.

Example 6:

```
class Marks {
      //    These variables are instance variables.
      //    These variables are in a class and
      //    are not inside any function
      int Marks;
      // Main Method
      public static void Main(String[] args)
  {
      // first object
      Marks obj1 = new Marks();
      obj1.Marks = 90;
      // second object
      Marks obj2 = new Marks();
      obj2.Marks = 95;
      //    displaying      marks      for      first      object
      Console.WriteLine("Marks      for      first      object:");
      Console.WriteLine(obj1.Marks);
      //    displaying      marks      for      second      object
      Console.WriteLine("Marks      for      second      object:");
      Console.WriteLine(obj2.Marks);
  }
}
```

Static Variables or Class Variables

Static variables are also known as Class variables. These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method, constructor or block.

To access static variables use class name, there is no need to create any object of that class.

Example 7:

```
class Emp {
      //    static variable salary static double salary;
      static String name = "E.Kumaran";
      //    Main Method
      public static void Main(String[] args)
```

```
    {
    //   accessing static variable without object
         Emp.salary = 50000;
    Console.WriteLine(Emp.name  +  "'s  average  salary:"  +
Emp.salary);
         }
    }
```

Constants Variables

A variable declared using the keyword "const" is a constant variable. The value of constant variables cannot be changed during the execution of the program. This means that, we cannot assign values to the constant variable at run time. Instead, it must be assigned at the compile time.

Example 8:

```
Const int max=500;
```

Read-Only Variables

A read-only variable is declared by using the readonly keyword then it will be read-only variables and these variables can't be modified like constants but after initialization.

It's not compulsory to initialize a read-only variable at the time of the declaration, they can also be initialized under the constructor.

Example 9:

```
Class rc
{
    readonly int k;     // readonly variables
    //   constructor
    Public rc()
    {    // initializing readonly variable k
    this.k = 90;
    }
```

2.4 Type conversion

Type conversion is converting one type of data to another type. It is also known as **Type Casting**. In C#, type casting has two forms:

i. Implicit type conversion: smaller to larger integral types.

```
int i=100; long l=i;
```

ii. Explicit type conversion: Larger to small integral types.

```
double d = 5673.74;
int i;
//    cast double to
int. i = (int)d;
```

2.4.1 Boxing

Boxing is the process of converting a Value Type variable (char, int etc.) into a Reference Type variable (object). Value Type variables are always stored in Stack memory, while Reference Type variables are stored in Heap memory.

Example 10:

```
int num = 23; // 23 will assigned to num
Object Obj = num; // Boxing
```

2.4.2 Unboxing

Unboxing is the process of converting a Reference Type variable (object) to a Value Type variable.

Example 11:

```
object o = 245;
int j = (int)o;
```

2.5 Input Statements

The Console class in the System namespace provides a function ReadLine( ) for accepting input from the user to be stored into a variable.

Example 12:

```
int num;
Double r;
num = Convert.ToInt32(Console.ReadLine());
r = Convert.ToDouble(Console.ReadLine());
string s = console.ReadLine();
Char c = Convert.ToChar(Console.ReadLine());
```

2.5.1 Operators

Operators can be categorized based upon their different functionality:

- Arithmetic Operators
- Relational Operators
- Logical Operators

- Bitwise Operators
- Assignment Operators
- Conditional Operator

Arithmetic Operators

These are used to perform arithmetic/mathematical operations on operands. The Binary Operators that fall under this category are:

- Addition: '+' operator
- Subtraction: '-' operator
- Multiplication: '*' operator
- Division: '/' operator
- Modulus: '%' operator

Example 13:

```
// Addition
      result = (x + y);
      Console.WriteLine("Addition Operator: " + result);
// Subtraction
      result = (x - y);
      Console.WriteLine("Subtraction Operator: " + result);


//    Multiplication
            result = (x * y);
            Console.WriteLine("Multiplication    Operator:    "+
      result);
// Division
      result = (x / y);
      Console.WriteLine("Division Operator: " + result);
// Modulo
      result = (x % y);
      Console.WriteLine("Modulo Operator: " + result);
```

Relational Operators

Relational operators are used for comparison of two values. Let's see them one by one:

- '=='(Equal To) operator
- '!='(Not Equal To) operator o '>'(Greater Than) operator
- <'(Less Than) operator
- '>='(Greater Than Equal To) operator
- '<='(Less Than Equal To) operator

Example 14:

```
bool result;
int x = 5, y = 10;
// Equal to Operator
     result = (x == y);
     Console.WriteLine("Equal to Operator: " + result);
// Greater than Operator
     result = (x > y);
     Console.WriteLine("Greater than Operator: " + result);
// Less than Operator
     result = (x < y);
     Console.WriteLine("Less than Operator: " + result);
// Greater than Equal to Operator
     result = (x >= y);
     Console.WriteLine("Greater than or Equal to: "+ result);
// Less than Equal to Operator
     result = (x <= y);
     Console.WriteLine("Lesser than or Equal to: "+ result);
// Not Equal To Operator
     result = (x != y);
     Console.WriteLine("Not Equal to Operator: " + result);
```

Logical Operators

They are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. They are described below:

- Logical AND: The '&&' operator
- Logical OR: The '||' operator
- Logical NOT: The '!' operator

Example 15:

```
bool a = true, b = false, result;
// AND operator
     result = a && b;
     Console.WriteLine("AND Operator: " + result);
// OR operator
     result = a || b;
     Console.WriteLine("OR Operator: " + result);
// NOT operator
```

```
      result = !a;
      Console.WriteLine("NOT Operator: " + result);
```

2.5.2 Control Statements

A control statement allows structured control of the sequence of application statements, such as loops, and conditional tests. It also allows switching the input stream to another file, opening and closing files, and various other operations. These include Decision-Making Statements and Looping Statements.

Decision Making Statements (if, if-else, if-else-if ladder, nested if, switch, nested switch).

Looping in programming language is a way to execute a statement or a set of statements multiple number of times depending on the result of condition to be evaluated to execute statements. The result condition should be true to execute statements within loops. Loops are mainly divided into two categories:

i. Entry Controlled Loops: The loops in which conditions to be tested are present at the beginning of the body of the loop are known as Entry Controlled Loops. They include "while loop" and "for loop".

Example 16:

```
int x = 1;
// Exit when x becomes greater than 4

      while (x <= 4)
          {
            Console.WriteLine("GeeksforGeeks");
// Increase the value of x for next iteration
            x++;
}
```

ii. Exit Controlled Loops: The loops in which the testing condition is present at the end of loop body are termed as Exit Controlled Loops. do-while is an exit controlled loop.

Example 17:

```
int x = 21;
      do
          {
//    The line will be printed even if the condition is false
            Console.WriteLine("GeeksforGeeks");
            x++;
      }
      while (x < 20);
```

2.6 Structure

Structure is a value type and a collection of variables of different data types under a single unit. It is almost similar to a class because both are user-defined data types and both hold a bunch of different data types.

Example 18:

```
public struct Person
{
//    Declaring different data types
      public string Name;
      public int Age;
      public int Weight;
}
static void Main(string[] args)
{
//    Declare P1 of type Person Person P1;
// P1's data
      P1.Name = "Keshav Gupta";
      P1.Age = 21;
      P1.Weight = 80;
// Displaying the values
      Console.WriteLine("Data Stored in P1 is " +
                        P1.Name + ", age is " +
                        P1.Age + " and weight is " +
                        P1.Weight);
}
```

Difference between Class and Structure

Class	**Structure**
Classes are of reference types.	Structs are of value types.
All the reference types are allocated on heap memory.	All the value types are allocated on stack memory.
Allocation of large reference type is cheaper than allocation of large value type.	Allocation and de-allocation is cheaper in value type as compare to reference type.
Class has limitless features.	Struct has limited features.
Class is generally used in large programs.	Struct are used in small programs.
Classes can contain constructor or destructor.	Structure does not contain constructor or destructor.

Class	Structure
Classes used new keyword for creating instances.	Struct can create an instance, without new keyword.
A Class can inherit from another class.	A Struct is not allowed to inherit from another struct or class
The data member of a class can be protected.	The data member of struct can't be protected.
Function member of the class can be virtual or abstract.	Function member of the struct cannot be virtual or abstract.

2.7 Class and Object

Class and Object are the basic concepts of Object Oriented Programming which revolve around the real-life entities.

A **class** is a user-defined blueprint or prototype from which objects are created. Basically, a class combines the fields and methods.

An object consists of :

- **State:** It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Example 19:

```
using System;
     class A
             {
              public int x=100;
             }
     class mprogram
              {
              static void Main(string[] args)
              {
              A a = new A();
           Console.WriteLine("Class A variable x is " +a.x);
              }
             }
```

2.7.1 Static Class

A static class can only contain static data members, static methods, and a static constructor.It is not allowed to create objects of the static class. **Static classes are sealed,** cannot inherit a static class from another class.

Note: Not allowed to create objects.

Syntax

```
static class Class_Name
{
//    static data members
//    static method
}
```

Example 20:

```
static class Author {
//    Static data members of Author
      public static string A_name = "Ankita";
      public static string L_name = "CSharp";
      public static int T_no = 84;
//    Static method of Author
      public static void details()
{
      Console.WriteLine("The details of Author is:");
}
}
//    Main Method
static public void Main(){
//    Calling static method of Author
                 Author.details();
//    Accessing the static data members of Author
      Console.WriteLine("Author name : {0} ", Author.A_name);
      Console.WriteLine("Language : {0} ", Author.L_name);
      Console.WriteLine("Total      number      of      articles
      :{0}",Author.T_ no);
}
```

2.7.2 Partial Class

A partial class is a special feature of C#. It provides a special ability to implement the functionality of a single class into multiple files and all these files are combined into a single class file when the application is compiled using the partial modifier keyword. The partial modifier can be applied to a class, method, interface or structure.

Advantages:

- It avoids programming confusion (in other words better readability).
- Multiple programmers can work with the same class using different files.
- Even though multiple files are used to develop the class all such files should have a common class name.

Example 21:

```
Filename: partial1.cs
using System;
partial class A
{
     public void Add(int x,int y)
     {
     Console.WriteLine("sum is {0}",(x+y));
     }
}
Filename: partial2.cs
using System;
partial class A
{
     public void Substract(int x,int y)
    {
    Console.WriteLine("Subtract is {0}", (x-y));
    }
 }
 Filename joinpartial.cs
 class Demo
 {
   public static void Main()
   {
    A obj=new A();
    obj.Add(7,3);
    obj.Substract(15,12);
   }
 }
```

2.8 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 within declared class itself
public	Members may be accessible anywhere outside declared class
protected	Members only accessible by the subclasses in other package or any class within the package of the protected members' 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.

Example 22:

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

2.9 Inheritance

Inheritance supports the concept of “reusability” one class is allowed to inherit the features (fields and methods) of another class.

i. Single Inheritance
ii. Multilevel Inheritance
iii. Multiple Inheritance (interface)
iv. Hierarchical Inheritance

Important terminology:

- Super Class: The class whose features are inherited is known as super class (or a base class or a parent class).
- Sub Class: The class that inherits the other class is known as subclass (or a derived class, extended class, or child class). The subclass can add its own fields and methods

- Reusability: To create a new class and there is already a class that includes some of the code that need to derive new class from the existing class.

Example 23:

```
Class A

{

Int x;

Void display()

{

System.Consolw.WriteLine("x="+x);

}

Class B : A

{

Display();

}
```

2.10 Interface

C# allows the user to inherit one interface into another interface. When a class implements the inherited interface.

Example 24:

```
using System;
//    declaring an interface
public interface A {
//    method of interface
      void mymethod1();
      void mymethod2();
}
//    The methods of interface A
//    is inherited into interface B
public interface B : A {
//    method of interface B
      void mymethod3();
}
//    Below class is inheriting
//    only interface B
//    This class must
//    implement both interfaces
```

```
class Geeks : B
{
//    implementing the method
//    of interface A
      public void mymethod1()
{
      Console.WriteLine("Implement method 1");
}
//    Implement the method
//    of interface B
      public void mymethod3()
{
      Console.WriteLine("Implement method 3");
}
}
```

2.11 Sealed classes

Sealed classes are used to restrict the users from inheriting the class. A class can be sealed by using the sealed keyword. The keyword tells the compiler that the class is sealed, and therefore, cannot be extended. No class can be derived from a sealed class.

Example 25:

```
sealed class SealedClass {
      // Calling Function
      public int Add(int a, int b)
      {
            return a + b;
      }
}
```

Important

- Sealed class is used to stop a class to be inherited. You cannot derive or extend any class from it.
- Sealed method is implemented so that no other class can overthrow it and implement its own method.
- The main purpose of the sealed class is to withdraw the inheritance attribute from the user so that they can't attain a class from a sealed class. Sealed classes are used best when you have a class with static members.

2.12 Method Overloading

Method Overloading is the common way of implementing polymorphism. It is the ability to redefine a function in more than one form. A user can implement function overloading by defining two or more functions in a class sharing the same name. i.e. the methods can have the same name but with different parameters list.

Example 26:

```
//    adding two integer values
      public int Add(int a, int b)
      {
            int sum = a + b;
            return sum;
      }
// adding three integer values
      public int Add(int a, int b, int c)
      {
            int sum = a + b + c;
            return sum;
      }
```

2.13 Method Overriding

Method Overriding is a technique that allows the invoking of functions from another class (base class) in the derived class. Creating a method in the derived class with the same signature as a method in the base class is called as method overriding.

Three types of keywords for Method Overriding:

i. virtual keyword: This modifier or keyword use within base class method. It is used to modify a method in base class for overridden that particular method in the derived class.
ii. override: This modifier or keyword use with derived class method. It is used to modify a virtual or abstract method into derived class which presents in base class.
iii. base Keyword: This is used to access members of the base class from derived class.

Example 27:

```
//    method overriding
using System;
//    base class
public class web {
      string name = "Method";
//    declare as virtual public
      virtual void showdata()
      {
      Console.WriteLine("Base class: " + name);
      }
}
class stream : web {
      string s = "Method";
      public override void showdata()
      {
      base.showdata();
      Console.WriteLine("Sub Class: " + s);
      }
}
class mc {
      static void Main()
{
      stream E = new stream();
E.showdata();
}
}
```

2.14 Array

An array is simply a collection of data that are normally related to each other and each data item is called an element of the array.

Syntax :

```
type [ ] < Name_Array > = new < datatype > [size];
```

Examples

```
int[] intArray1 = new int[5];
int[] intArray2 = new int[5]{1, 2, 3, 4, 5};
int[] intArray3 = {1, 2, 3, 4, 5};
```

2.14.1 Jagged Arrays

A jagged array is an array of arrays such that member arrays can be of different sizes. In other words, the length of each array index can differ. A jagged array is sometimes called an “array of arrays”.

Syntax:

```
data_type[][] name_of_array = new data_type[rows][]
```

Example 28:

```
//    Declare the Jagged Array of four elements:
      int[][] jagged_arr = new int[4][];
//    Initialize the elements
      jagged_arr[0] = new int[] {1, 2, 3, 4};
      jagged_arr[1] = new int[] {11, 34, 67};
      jagged_arr[2] = new int[] {89, 23};
      jagged_arr[3] = new int[] {0, 45, 78, 53, 99};
```

2.14.2 ArrayList

ArrayList is a powerful feature of C# language. It is the non-generic type of collection which is defined in System.Collections namespace. ArrayList are dynamic hence, we can add any number of items without specifying the size of it.

ArrayList is defined under System.Collections namespace. So, when you use Arraylist in your program you must add System.Collections namespace.

Syntax:

```
ArrayList list_name = new ArrayList();
```

Example 29:

```
// Creating ArrayList
ArrayList My_array = new ArrayList();
//    This ArrayList contains elements of different types
My_array.Add(112.6);
My_array.Add("C# program");
My_array.Add(null);
My_array.Add('Q');
My_array.Add(1231);
//    Access the array list
foreach(var elements in My_array)
      {
      Console.WriteLine(elements);
      }
```

Note: Array List allows you to add, insert, remove and change elements.

2.15 Indexers

An indexer allows an instance of a class or struct to be indexed as an array. If the user will define an indexer for a class, then the class will behave like a virtual array. Array access operator i.e ([ ]) is used to access the instance of the class which uses an indexer.

Syntax:

```
[access_modifier] [return_type] this
[argument_list] {
      get
      {
      // get block code
      }
      set
      {
      // set block code
      }
}
```

Example 30:

```
public string this[int index]
{
      get
      {
      return val[index];
      }
      set
      {
      val[index] = value;
      }
IndexerCreation ic = new IndexerCreation();
      ic[0] = "C";
      ic[1] = "CPP";
      ic[2] = "CSHARP";
```

2.16 Properties

Properties are the special type of class members that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called accessors.

Accessors : The block of "set" and "get"

There are different types of properties based on the "get" and set accessors:

i. Read and Write Properties: When property contains both get and set methods.
ii. Read-Only Properties: When property contains only get method.
iii. WriteOnly Properties: When property contains only set method.
iv. Auto Implemented Properties: When there is no additional logic in the property accessors.

2.17 Delegates

A delegate is a reference type variable that holds the reference to a method. The reference can be changed at runtime. Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the System.Delegate class.

A delegate will call only a method which agrees with its signature and return type. A method can be a static method associated with a class or can be instance method associated with an object, it doesn't matter.

Syntax:

```
[modifier]      delegate      [return_type]      [delegate_name]
([parameter_list]);
```

Example 31:

```
public delegate void addnum(int a, int b);
using System;
class TestDelegate {
      delegate int NumberChanger(int n);
         int num = 10;
         public static int AddNum(int p) {
            num += p;
            return num;
      }
         public static int MultNum(int q) {
            num *= q;
            return num;
      }
         public static int getNum() {
            return num;
      }
      static void Main(string[] args) {
//create delegate instances
            NumberChanger   nc1   =   new   NumberChanger(AddNum);
            NumberChanger nc2 = new NumberChanger(MultNum);
//calling the methods using the delegate objects nc1(25);
```

```
        Console.WriteLine("Value of Num: {0}", getNum()); nc2(5);
        Console.WriteLine("Value   of   Num:   {0}",   getNum());
        Console.ReadKey();
}
}
}
```

2.18 Events

Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

2.18.1 Delegates with Events

C# and .NET supports event driven programming via delegates.The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class. The class containing the event is used to publish the event. This is called the publisher class. Some other class that accepts this event is called the subscriber class. Events use the publisher-subscriber model.

A publisher is an object that contains the definition of the event and the delegate. The event-delegate association is also defined in this object. A publisher class object invokes the event and it is notified to other objects.

A subscriber is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

To declare an event inside a class, first of all, you must declare a delegate type for the even as:

```
public delegate string BoilerLogHandler(string str);
```

Following are the key points about Events

i. Event Handlers in C# return void and take two parameters.
ii. The First parameter of Event - Source of Event means publishing object.
iii. The Second parameter of Event - Object derived from EventArgs.
iv. The publishers determines when an event is raised and the subscriber determines what action is taken in response.
v. An Event can have so many subscribers.
vi. Events are basically used for the single user action like button click.
vii. If an Event has multiple subscribers then event handlers are invoked synchronously.

CHAPTER 3
C#- USING LIBRARIES

3.1 Namespace

A namespace is designed for providing a way to keep one set of names separate from another. The class names declared in one namespace does not conflict with the same class names declared in another.

Syntax:

```
namespace namespace_name {
      // code declarations
}
```

The using keyword states that the program is using the names in the given namespace. For example, we are using the System namespace in our programs. It can also avoid prepending of namespaces with the using namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace.

Example 1:

```
using System;
using first_space;
using second_space;
namespace first_space {
      class abc {
            public void func() {
                  Console.WriteLine("Inside first_space");
            }
      }
}
namespace second_space {
      class efg {
            public void func() {
                  Console.WriteLine("Inside second_space");
            }
      }
}
      class TestClass {
            static void Main(string[] args) {
                  abc fc = new abc();
                  efg sc = new efg();
                  fc.func();
```

```
                    sc.func();
                    Console.ReadKey();
        }
}
```

3.2 I/O Stream

A stream is linked to a physical device by the I/O system.

Type of streams

i. Byte Streams − It includes Stream, FileStream, MemoryStream and BufferedStream.
ii. Character Streams − It includes Textreader-TextWriter, StreamReader, StreamWriter and other streams.
iii. 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.

3.3 File I/O

A file is a collection of data stored in a disk with a specific name and a directory path. When a file is opened for reading or writing, it becomes a stream.

The stream is basically the sequence of bytes passing through the communication path. There are two main streams: the input stream and the output stream. The input stream is used for reading data from file (read operation) and the output stream is used for writing into the file (write operation).

The System.IO namespace has various classes that are used for performing numerous operations with files, such as creating and deleting files, reading from or writing to a file, closing a file etc.

3.3.1 FileStream Class

The FileStream class in the System.IO namespace helps in reading from, writing to and closing files. This class derives from the abstract class Stream.

Using System.io;

To create a FileStream object to create a new file or open an existing file.

Syntax

```
FileStream <object_name> = new FileStream( <file_name>, <FileMode Enumerator>,<FileAccess Enumerator>, <FileShare Enumerator>);
```

Example 2:

```
FileStream F = new FileStream("sample.txt", FileMode.Open,
FileAccess.Read, FileShare.Read);
```

3.3.2 FileMode

The FileMode enumerator defines various methods for opening files. The members of the FileMode enumerator are –

i. Append – It opens an existing file and puts cursor at the end of file, or creates the file, if the file does not exist.
ii. Create – It creates a new file.
iii. CreateNew – It specifies to the operating system, that it should create a new file.
iv. Open – It opens an existing file.
v. OpenOrCreate – It specifies to the operating system that it should open a file if it exists, otherwise it should create a new file.
vi. Truncate – It opens an existing file and truncates its size to zero bytes

3.3.3 FileAccess

FileAccess enumerators have members: Read, ReadWrite and Write.

3.3.4 FileShare

Fileshare members include:

i. Inheritable – It allows a file handle to pass inheritance to the child processes.
ii. None – It declines sharing of the current file.
iii. Read – It allows opening the file for reading.
iv. ReadWrite – It allows opening the file for reading and writing.
v. Write – It allows opening the file for writing.

Example 3:

```
using System;
using System.IO;
namespace FileIOApplication {
  class Program {
    static void Main(string[] args) {
     FileStream F = new FileStream("test.dat",
     FileMode.OpenOrCreate, FileAccess.ReadWrite);
     for (int i = 1; i <= 20; i++)
       { F.WriteByte((byte)i);
      }
     F.Position = 0;
```

```
        for (int i = 0; i <= 20; i++) {
           Console.Write(F.ReadByte() + " ");
        }
        F.Close();
        Console.ReadKey();
      }
    }
}
```

3.4 Multitasking

Multitasking is the simultaneous execution of multiple tasks or processes over a certain time interval. Windows operating system is an example of multitasking because it is capable of running more than one process at a time like running Google Chrome, Notepad, VLC player etc. at the same time.

3.5 Thread

A thread is a lightweight process, or in other words, a thread is a unit which executes the code under the program. Every program by default carries one thread to execute the logic of the program and the thread is known as the Main Thread.

3.5.1 Multithreading

It is a process which contains multiple threads within a single process. Here each thread performs different activities. For example, we have a class and this call contains two different methods, now using multithreading each method is executed by a separate thread. So the major advantage of multithreading is it works simultaneously, means multiple tasks executes at the same time. And also maximizing the utilization of the CPU because multithreading works on time-sharing concept mean each thread takes its own time for execution and does not affect the execution of another thread, this time interval is given by the operating system.

Example 4:

```
public class GFG {
// static method one
public static void method1()
{
// It prints numbers from 0 to 10
   for (int I = 0; I <= 10; I++) {
      Console.WriteLine("Method1 is : {0}", I);
//   When the value of I is equal to 5 then
//    this method sleeps for 6 seconds
   if (I == 5) {
      Thread.Sleep(6000);
     }
   }
}
// static method two
public static void method2()
{
// It prints numbers from 0 to 10
   for (int J = 0; J <= 10; J++) {
      Console.WriteLine("Method2 is : {0}", J);
    }
}
// Main Method
static public void Main()
{
//    Creating and initializing threads
  Thread thr1 = new Thread (method1);
  Thread thr2 = new Thread (method2);
  thr1.Start();
  thr2.Start();
    }
}
```

Advantages of Multithreading

i. It executes multiple process simultaneously.
ii. Maximize the utilization of CPU resources.
iii. Time sharing between multiple processes.

3.6 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 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.

The .NET framework supports both synchronous and asynchronous communication between the client and server.

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.

3.6.1 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).

Example 5:

```
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]);
}
}
}
```

3.6.2 Socket

Sockets are the most powerful networking mechanism available in .NET—HTTP is layered on top of sockets, and in most cases WCF is too. Sockets provide more or less direct access to the underlying TCP/IP network services.

The basic idea of a socket has been around for decades, and appears in many operating systems. The central concept is to present network communication through the same abstractions as file I/O.

Streams are concerned with the body of an HTTP request or response. With sockets, the streams are at a lower level, encompassing all the data.

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)
```

3.7 Data handling

DBMS - Database Management System (DBMS) is a software system that enables users to define, create, maintain and control access to the database.

3.7.1 Database languages

Database languages are special-purpose languages, which allow one or more of the following tasks, sometimes distinguished as sublanguages:

- Data control language (DCL) – controls access to data.
- Data definition language (DDL) – defines data types such as creating, altering, or dropping and the relationships among them.
- Data manipulation language (DML) – performs tasks such as inserting, updating, or deleting data occurrences.
- Data query language (DQL) – allows searching for information and computing derived information.

3.7.2 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.

Connected Vs Disconnected

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:

i. A secure environment is easier to maintain.
ii. Concurrency is easier to control.
iii. Data is more likely to be current than in other scenarios. A connected scenario has the following disadvantages:
iv. It must have a constant network connection.
v. 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:

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

A disconnected environment has the following disadvantages:

i. Data is not always up to date.
ii. 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:

i. Interoperability. ADO.NET uses XML as the format for transmitting data from a data source to a local in-memory copy of the data.
ii. 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.
iii. 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.

iv. Performance. ADO.NET helps you to avoid costly data type conversions because of its use of strongly typed data.

v. 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 memory caches, there is no need to retain database locks or maintain active database connections for extended periods.

3.8 .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:

i. SQL Server .NET Data Provider
ii. OLE DB .NET Data Provider
iii. Oracle .NET Data Provider
iv. ODBC .NET Data Provider
v. Other Native .NET Data Provider

3.9 Windows Forms/Applications

The Windows Forms is a collection of classes and types that encapsulate and extend the Win32 API in an organized object model. The .NET Framework contains an entire subsystem devoted to Windows programming called Windows Forms. The primary support for Windows Forms is contained in the System.Windows.Forms namespace. A form encapsulates the basic functionality necessary to create a window, display it on the screen, and receive messages. A form can represent any type of window, including the main window of the application, a child window, or even a dialog box.

3.9.1 The Form Class

Form contains significant functionality of its own, and it inherits additional functionality.

Two of its most important base classes which supports the .NET component model are:

- System.ComponentModel.Component; and
- System.Windows.Forms.Control.

The Control class defines features common to all Windows controls.

3.9.2 Creating Windows Form Application

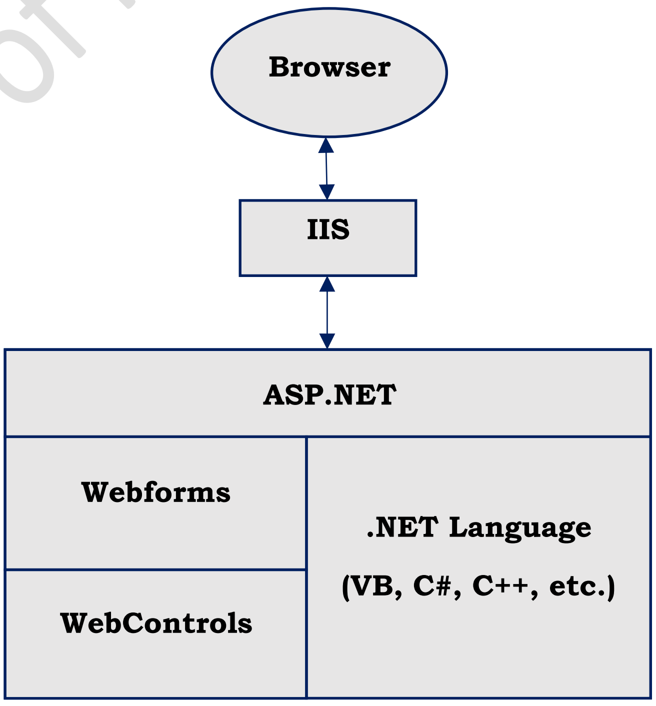
Creating a Windows Forms application is largely just a matter of instantiating and extending the Windows Forms and GDI+ classes. In a nutshell, you typically complete the following steps:

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. Compile and run the program

3.9.3 Web Forms/Application

Web Forms are the heart and soul of ASP.NET. Web Forms are the User Interface (UI) elements that give 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.

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:

1. Separation of HTML interface from application logic;
2. A rich set of server-side controls that can detect the browser and send out appropriate markup language such as HTML;
3. Less code to write due to the data binding capabilities of the new server-side .NET controls;
4. Event-based programming model that is familiar to Microsoft Visual Basic programmers;
5. Compiled code and support for multiple languages, as opposed to ASP which was interpreted as Microsoft Visual Basic Scripting (VBScript) or Microsoft Jscript.; and
6. Allows third parties to create controls that provide additional functionality.

3.9.4 Exception Handling

An exception is an unwanted or unexpected event, which occurs during the execution of a program (at runtime) that disrupts the normal flow of the program's instructions. This unwanted event is known as Exception.

Errors

i. Errors are unexpected issues that may arise during computer program execution.
ii. Errors cannot be handled.
iii. All Errors are exceptions.

Exceptions

i. Exceptions are unexpected events that may arise during run-time.
ii. Exceptions can be handled using try-catch mechanisms.
iii. All exceptions are not errors.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: try, catch, finally, and throw.

try − A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.

catch − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

finally − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.

throw − A program throws an exception when a problem shows up. This is done using a throw keyword.

Exception Hierarchy

All the exceptions are derived from the base class Exception which gets further divided into two branches as ApplicationException and SystemException.

- SystemException is a base class for all CLR or program code generated errors.
- ApplicationException is a base class for all application related exceptions.

There are different kinds of exceptions which can be generated in C# program:

- Divide By Zero exception: It occurs when the user attempts to divide by zero;
- Out of Memory exceptions: It occurs when then the program tries to use excessive memory;
- Index out of bound Exception: Accessing the array element or index which is not present in it;
- Stackoverflow Exception: Mainly caused due to infinite recursion process; and
- Null Reference Exception: Occurs when the user attempts to reference an object which is of NULL type.

Example 6:

```
public void division(int num1, int num2) {
      try {
            result = num1 / num2;
      }
      catch (DivideByZeroException e) {
            Console.WriteLine("Exception caught: {0}", e);
      }
```

CHAPTER 4
ADVANCED FEATURES USED IN C#

4.1 Web Services -WS

A web service is any piece of software that makes itself available over the internet and uses a standardized XML messaging system. XML is used to encode all communications to a web service.

Web services are self-contained, modular, distributed, dynamic applications that can be described, published, located, or invoked over the network to create products, processes, and supply chains. These applications can be local, distributed, or web-based. Web services are built on top of open standards such as TCP/IP, HTTP, Java, HTML, and XML.

Web services are XML-based information exchange systems that use the Internet for direct application-to-application interaction. These systems can include programs, objects, messages, or documents. A web service is a collection of open protocols and standards used for exchanging data between applications or systems.

4.1.1 Components of Web Services

The basic web services platform is XML + HTTP. All the standard web services work using the following components −

i. SOAP (Simple Object Access Protocol)
ii. UDDI (Universal Description, Discovery and Integration)
iii. WSDL (Web Services Description Language)

Working process

A web service enables communication among various applications by using open standards such as HTML, XML, WSDL, and SOAP. A web service takes the help of −

i. XML to tag the data
ii. SOAP to transfer a message
iii. WSDL to describe the availability of service.

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.

4.1.2 Web Service Applications

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

Validation Controls

i. E-mail address validator,
ii. Regular expression validator,

iii. Range Validator, etc.

Login Controls

i. Create user
ii. Delete user
iii. Manage users, etc.

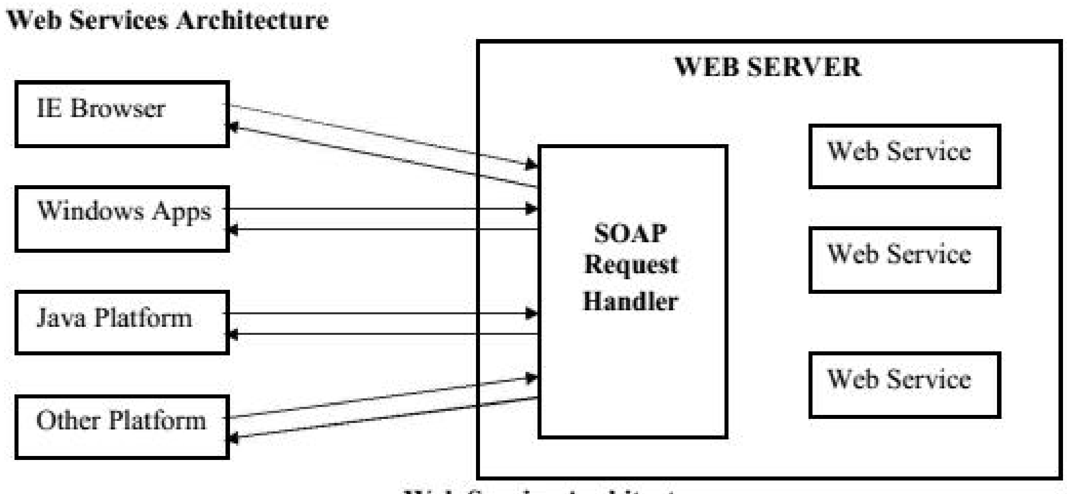
Some Web services are also available on internet, which are free and offer application-components like:

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

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

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

4.1.3 Web Service Architecture



Web Service Architecture

4.1.4 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:

i. Description
ii. MessageName
iii. EnableSession
iv. CacheDuration
v. TransactionOption
vi. BufferResponse

4.1.5 Create web service

URL: https://www.tutorialspoint.com/asp.net/asp.net_web_services.htm

4.2 Windows Services

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.

Windows Services are non-UI software applications that run in the background. Windows services are usually started when an operating system boots and scheduled to run in the background to execute some tasks. Windows services can also be started automatically or manually. You can also manually pause, stop and restart Windows services.

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.

The Main method for your service application must issue the Run command for the services your project contains. The Run method loads the services into the Services Control Manager on the appropriate server. If you use the Windows Services project template, this method is written for you automatically.

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:

i. System.ServiceProcess.ServiceBase
ii. System.ServiceProcess.ServiceProcessInstaller
iii. ServiceController

4.2.1 Developing Window Service

To develop and run a Window Service application on .NET frame to the he 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

4.2.2 Creating a Windows Service Application

https://docs.microsoft.com/en-us/dotnet/framework/windows-services/walkthrough-creating-a-windows-service-application-in-the-component-designer

https://dotnetcoretutorials.com/2019/09/19/creating-windows-services-in-net-core-part-1-the-microsoft-way/

4.3 Reflection

Reflection objects are used for obtaining type information at runtime. The classes that give access to the metadata of a running program are in the System.Reflection namespace.

The System.Reflection namespace contains classes that allow you to obtain information about the application and to dynamically add types, values, and objects to the application.

Applications of Reflection

Reflection has the following applications −

i. It allows view attribute information at runtime.
ii. It allows examining various types in an assembly and instantiate these types.
iii. It allows late binding to methods and properties
iv. It allows creating new types at runtime and then performs some tasks using those types.

Reflection is a process to get metadata of a type at runtime. The System.Reflection namespace contains required classes for reflection such as:

Class	**Description**
Assembly	describes an assembly which is a reusable, versionable, and self-describing building block of a common language runtime application
AssemblyName	Identifies an assembly with a unique name
ConstructorInfo	Describes a class constructor and gives access to the metadata
MethodInfo	Describes the class method and gives access to its metadata
ParameterInfo	Describes the parameters of a method and gives access to its metadata
EventInfo	Describes the event info and gives accessto its metadata
PropertyInfo	Discovers the attributes of a property and provides access to property metadata
MemberInfo	Obtains information about the attributes of a member and provides access to member metadata

4.4 Viewing Metadata

The MemberInfo object of the System.Reflection class needs to be initialized for discovering the attributes associated with a class.

```
System.Reflection.MemberInfo info = typeof(MyClass);
```

C# Type Properties	
Property	**Description**
Assembly	Gets the Assembly for this type.
AssemblyQualifiedName	Gets the Assembly qualified name for this type.
Attributes	Gets the Attributes associated with the type.
BaseType	Gets the base or parent type.
FullName	Gets the fully qualified name of the type.
IsAbstract	is used to check if the type is Abstract.

C# Type Methods	
Method	**Description**
GetConstructors()	Returns all the public constructors for the Type.
GetConstructors(BindingFlags)	Returns all the constructors for the Type with specified BindingFlags.
GetFields()	Returns all the public fields for the Type.
GetFields(BindingFlags)	Returns all the public constructors for the Type with specified BindingFlags.
GetMembers()	Returns all the public members for the Type.
GetMembers(BindingFlags)	Returns all the members for the Type with specified BindingFlags.
GetMethods()	Returns all the public methods for the Type.
GetMethods(BindingFlags)	Returns all the methods for the Type with specified BindingFlags.
GetProperties()	Returns all the public properties for the Type.
GetProperties(BindingFlags)	Returns all the properties for the Type with specified BindingFlags
GetType()	Gets the current Type.
GetType(String)	Gets the Type for the given name

Example 1:

C# Reflection Example: Get Type

```
using System;
public class ReflectionExample
```

```
{
      public static void Main()
      {
            int a = 10;
      Type type = a.GetType();
      Console.WriteLine(type);
      }
}
```

4.5 COM

Component Object Model (COM) is a method to facilitate communication between different applications and languages. COM is used by developers to create re-usable software components, link components together to build applications, and take advantage of Windows services. COM objects can be created with a variety of programming languages. Object-oriented languages, such as C++, provide programming mechanisms that simplify the implementation of COM objects. The family of COM technologies includes COM+, Distributed COM (DCOM) and ActiveX® Controls.

COM (Component Object Model) was the first programming model that provided component based approach to software development. This component based approach of COM allowed us to develop small, logical reusable and standalone modules that integrates into a single application. But these components could not be displayed over a network. So these drawback produce another model that is DCOM (Distributed COM).

The DCOM programming model enables you to display COM components over a network and easily distribute applications across platforms. DCOM components also help in two-tier client/server applications. These models also have some drawbacks that help the development of the COM+ approach.

4.5.1 Creating COM components in .NET

The following steps explain the way to create the COM server in C#:

1. Create a new Class Library project.
2. Create a new interface, say IManagedInterface, and declare the methods required. Then provide the Guid (this is the IID) for the interface using the GuidAttribute defined in System.Runtime.InteropServices. The Guid can be created using the Guidgen.exe.
3. Define a class that implements this interface. Again provide the Guid (this is the CLSID) for this class also.
4. Mark the assembly as ComVisible. For this go to AssemblyInfo.cs file and add the following statement [assembly: ComVisible (true)]. This gives the accessibility of all types within the assembly to COM.
5. Build the project. This will generate an assembly in the output path. Now register the assembly using regasm.exe (a tool provided with the

.NET Framework) - regasm \bin\debug\ComInDotNet.dll \tlb:ComInDotNet.tlb this will create a TLB file after registration.
6. Alternatively this can be done in the Project properties --> Build --> check the Register for COM interop.

The COM server is ready. Now a client has to be created. It can be in any language. If the client is .NET, just add the above created COM assembly as a reference and use it. Localization

4.5.2 Call COM components from .NET

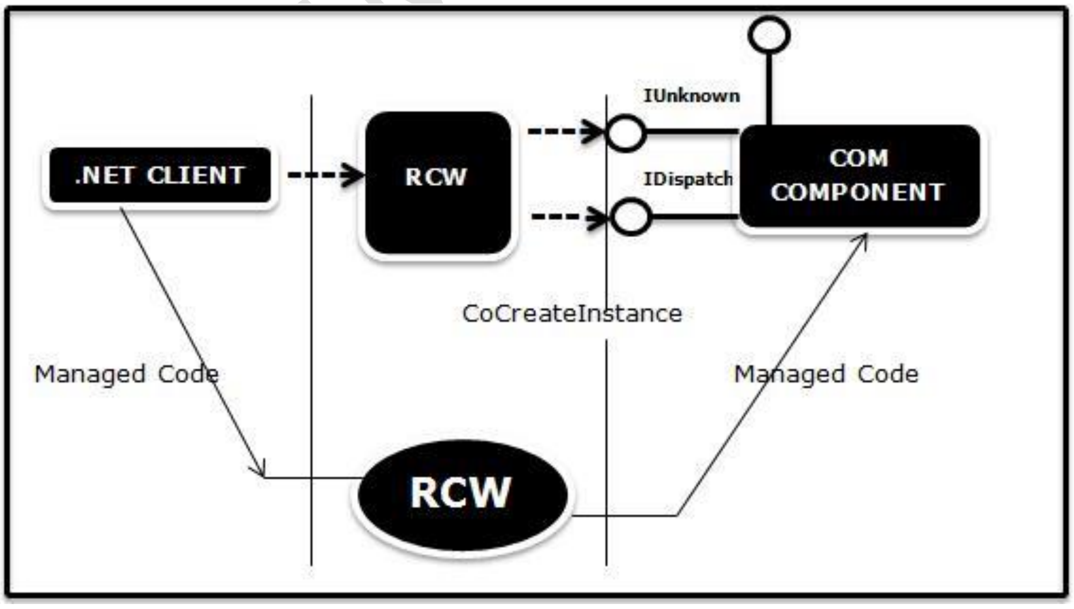
COM components and .NET Components have a different internal architecture. For both of them to communicate with each other, the inter-operation feature is required, this feature is called interoperability. Enterprises that have written their business solutions using the old native COM technology need a way for re-using these components in the new .NET environment.

.NET components communicate with COM using RCW (Runtime Callable Wrapper)

RCW: RCW Means Runtime Callable Wrappers, The Common Language Runtime (CLR) exposes COM objects through a proxy called the Runtime Callable Wrapper (RCW). Although the RCW appears to be an ordinary object to .NET clients, its primary function is to marshal calls between a .NET client and a COM object.

How to use a COM component

- Right click the Project and click on Add References.
- Select the COM tab
- And at last select COM component



4.6 Globalization and Localization

Globalization is the process of designing and developing applications that function for multiple cultures.

Localization is the process of customizing your application for a given culture and locale.

4.6.1 Globalization

Globalization involves designing and developing a world-ready app that supports localized interfaces and regional data for users in multiple cultures. Before beginning the design phase, you should determine which cultures your app will support. Although an app targets a single culture or region as its default, you can design and write it so that it can easily be extended to users in other cultures or regions.

As developers, we all have assumptions about user interfaces and data that are formed by our cultures. For example, for an English-speaking developer in the United States, serializing date and time data as a string in the format MM/dd/yyyy hh:mm:ss seems perfectly reasonable. However, deserializing that string on a system in a different culture is likely to throw a FormatException exception or produce inaccurate data. Globalization enables us to identify such culture-specific assumptions and ensure that they do not affect our app's design or code.

Strings

The handling of characters and strings is a central focus of globalization, because each culture or region may use different characters and character sets and sort them differently. This section provides recommendations for using strings in globalized apps.

Use Unicode internally

By default, .NET uses Unicode strings. A Unicode string consists of zero, one, or more Char objects, each of which represents a UTF-16 code unit. There is a Unicode representation for almost every character in every character set in use throughout the world.

4.6.2 Localization

Localization is the process of translating an application's resources into localized versions for each culture that the application will support. You should proceed to the localization step only after completing the Localizability Review step to verify that the globalized application is ready for localization.

An application that is ready for localization is separated into two conceptual blocks: a block that contains all user interface elements and a block that contains executable code. The user interface block contains only localizable user-interface elements such as strings, error messages, dialog boxes, menus, embedded object resources, and so on for the neutral culture. The code block contains only the application code to be used by all supported cultures. The common language runtime supports a satellite assembly resource model that separates an application's executable code from its resources. For more information about implementing this model, see Resources in .NET.

For each localized version of your application, add a new satellite assembly that contains the localized user interface block translated into the appropriate language for the target culture. The code block for all cultures should remain the same. The combination of a localized version of the user interface block with the code block produces a localized version of your application.

The Windows Software Development Kit (SDK) supplies the Windows Forms Resource Editor (Winres.exe) that allows you to quickly localize Windows Forms for target cultures.

Cultures and Locales

The language needs to be associated with the particular region where it is spoken, and this is done by using locale (language + location). For example: fr is the code for French language. fr-FR means French language in France. So, for specifies only the language whereas fr-FR is the locale. Similarly, fr-CA defines another locale implying French language and culture in Canada. If we use only fr, it implies a neutral culture (i.e., location neutral).

Set culture information

Application level -In web.config file

```
<configuration>

<system.web>

<globalization culture="fr-FR" uiCulture="fr-FR"/>
</system.web>

</configuration>
```

4.7 Resource Files

A resource file is an XML file that contains the strings that you want to translate into different languages or paths to images.

The resource file contains key/value pairs. Each pair is an individual resource. Key names are not case sensitive.

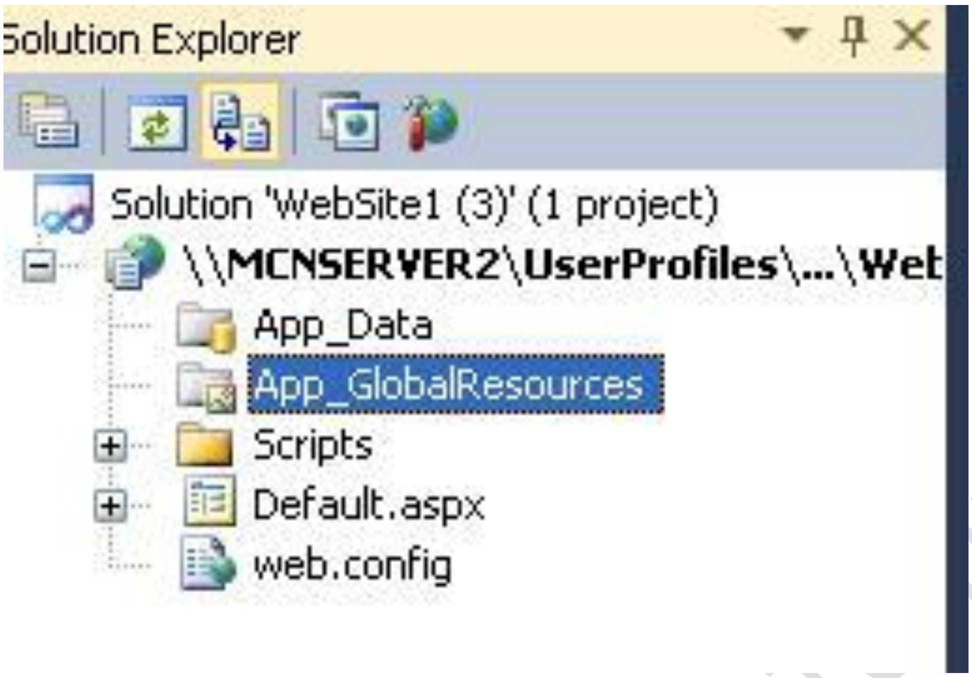
e.g. A resource file might contain a resource with the key Button1 and the value Submit

Resource files in ASP. NET have a .resx extension. At run time, the .resx file is compiled into an assembly.

4.7.1 Global Resource Files

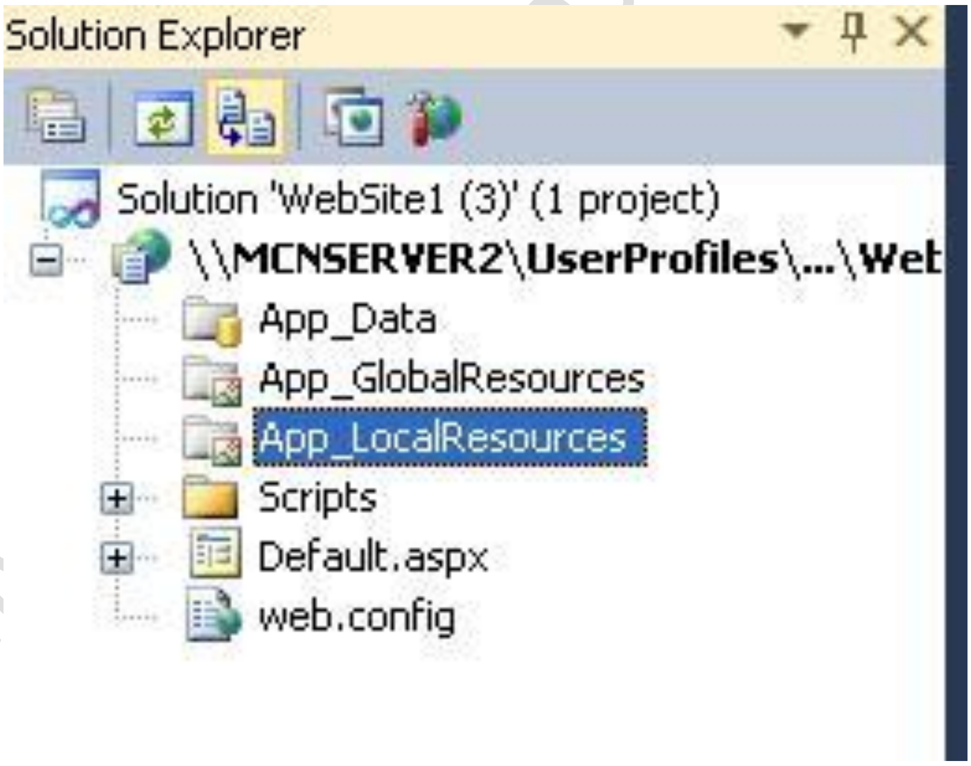
You create a global resource file by putting it in the reserved folder App_GlobalResources at the root of the application.

Any .resx file that is in the App_GlobalResources folder has global scope.



4.7.2 Local Resource Files

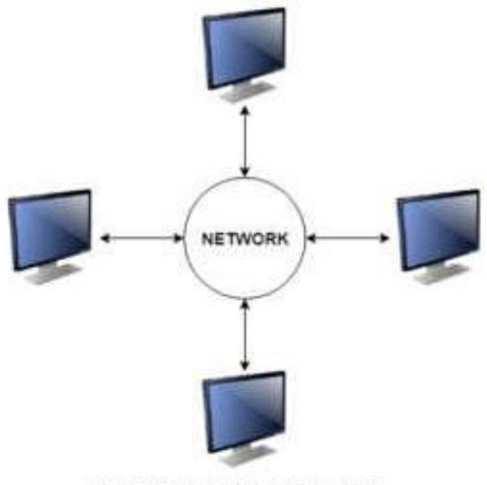
A local resources file is one that applies to only one ASP. NET page or user control (an ASP. NET file that has a file-name extension of .aspx, .ascx, or .master).



CHAPTER 5
DISTRIBUTED APPLICATION

5.1 Distributed System

A distributed system contains multiple nodes that are physically separate but linked together using the network. All the nodes in this system communicate with each other and handle processes in tandem. Each of these nodes contains a small part of the distributed operating system software.



DISTRIBUTED OPERATING SYSTEM

5.1.1 Types of Distributed Systems

The nodes in the distributed systems can be arranged in the form of client/server systems or peer to peer systems. Details about these are as follows:

Client/ Server Systems

In client server systems, the client requests a resource and the server provides that resource. A server may serve multiple clients at the same time while a client is in contact with only one server. Both the client and server usually communicate via a computer network and so they are a part of distributed systems.

Peer to Peer Systems

The peer to peer systems contains nodes that are equal participants in data sharing. All the tasks are equally divided between all the nodes. The nodes interact with each other as required as share resources. This is done with the help of a network.

Advantages of Distributed Systems

i. All the nodes in the distributed system are connected to each other. So nodes can easily share data with other nodes.
ii. More nodes can easily be added to the distributed system i.e. it can be scaled as required.

iii. Failure of one node does not lead to the failure of the entire distributed system. Other nodes can still communicate with each other.
iv. Resources like printers can be shared with multiple nodes rather than being restricted to just one.

Disadvantages of Distributed Systems

i. It is difficult to provide adequate security in distributed systems because the nodes as well as the connections need to be secured.
ii. Some messages and data can be lost in the network while moving from one node to another.
iii. The database connected to the distributed systems is quite complicated and difficult to handle as compared to a single user system.
iv. Overloading may occur in the network if all the nodes of the distributed system try to send data at once.

Refer URL:

https://www.tutorialspoint.com/software_architecture_design/distributed_architecture.htm

5.2 Distributed Applications

Enterprises and users demand distributed applications. Distributed applications allow objects to talk across process boundaries. Often, distributed applications also meet the following objectives:

i. Establish communication between objects that run in different application domains and processes, whether on the same computer or across the Internet.
ii. Enable enterprise application integration by establishing communication between objects that run on heterogeneous architectures.
iii. Enable application availability by making sure that portions of an application run even if some components are busy or have failed.
iv. Provide increased security and scalability by dividing the application into several layers (or tiers).

5.3 Evolution of Distributed Applications

A well-designed distributed application has the potential to be more connected, more available, more scalable, and more robust than an application where all components run on a single computer. This is a desirable model for an enterprise application.

Traditionally, there have been several efforts to design frameworks for developing distributed applications. A few well-known frameworks are Distributed Computing

Environment/Remote Procedure Calls (DEC/RPC), Microsoft Distributed Component Object Model (DCOM), Common Object Request Broker Architecture (CORBA), and Java Remote Method Invocation (RMI). Some of these implementations are widely deployed in enterprises.

However, modern business requirements are different from those of earlier days. Today, businesses seek solutions that can be developed rapidly, that integrate well with their legacy applications, and that interoperate well with their business partners. Each of the technologies already mentioned failed to satisfy one or more of these requirements.

In 2000, Microsoft introduced the .NET Framework for designing next-generation distributed applications. As you'll explore more in this book, the .NET Framework is specifically targeted to meet the needs of modern business, whether the need is rapid development or integration or interoperability.

5.4 Using the .NET Framework to Develop Distributed Applications

The .NET Framework provides various mechanisms to support distributed application development. Most of this functionality is present in the following three namespaces of the Framework Class Library (FCL):

i. The System.Net Namespace—This namespace includes classes to create standalone listeners and custom protocol handlers to start from scratch and create your own framework for developing a distributed application. Working with the System.Net namespace directly requires a good understanding of network programming.

ii. The System.Runtime.Remoting Namespace—This namespace includes the classes that constitute the .NET remoting framework. The .NET remoting framework allows communication between objects living in different application domains, whether or not they are on the same computer. Remoting provides an abstraction over the complex network programming and exposes a simple mechanism for inter-application domain communication. The key objectives of .NET remoting are flexibility and extensibility.

iii. The System.Web.Services Namespace—This namespace includes the classes that constitutes the ASP.NET Web services framework. ASP.NET Web services allow objects living in different application domains to exchange messages through standard protocols such as HTTP and SOAP. ASP.NET Web services, when compared to remoting, provide a much higher level of abstraction and simplicity. The key objectives of ASP.NET Web services are the ease of use and interoperability with other systems.

iv. Both .NET remoting and ASP.NET Web services provide a complete framework for designing distributed applications. Most programmers will use either .NET remoting or ASP.NET Web services rather than build a distributed programming framework from scratch with the System.Net namespace classes.

v. The functionality offered by .NET remoting and ASP.NET Web services appears very similar. In fact, ASP.NET Web services are actually built on the .NET remoting infrastructure. It is also possible to use .NET remoting to design Web services. Given the amount of similarity, how do you choose one over the other in your project? Simply put, the decision depends on the type of application you want to create. You'll use

vi. .NET Remoting when both the end points (client and server) of a distributed application are in your control. This might be a case when an application has been designed for use within a corporate network.

vii. ASP.NET Web services when one end point of a distributed application is not in your control. This might be a case when your application is interoperating with your business partner's application.

Refer URL:

https://www.pearsonitcertification.com/articles/ar ticle.aspx?p=31490

5.5 XML

XML stands for Extensible Markup Language. It is a text-based markup language derived from Standard Generalized Markup Language (SGML).

XML tags identify the data and are used to store and organize the data, rather than specifying how to display it like HTML tags, which are used to display the data. XML is not going to replace HTML in the near future, but it introduces new possibilities by adopting many successful features of HTML.

There are three important characteristics of XML that make it useful in a variety of systems and solutions −

i. XML is extensible − XML allows you to create your own self-descriptive tags, or language, that suits your application.
ii. XML carries the data, does not present it − XML allows you to store the data irrespective of how it will be presented.
iii. XML is a public standard − XML was developed by an organization called the World Wide Web Consortium (W3C) and is available as an open standard.

5.6 Serialization

Serialization is the process of converting an object into a stream of bytes. In this article, I will show you how to serialize object to XML in C#. XML serialization converts the public fields and properties of an object into an XML stream.

XML serialization converts (serializes) the public fields and properties of an object, and the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to a serial format (in this case, XML) for storage or transport.

Because XML is an open standard, the XML stream can be processed by any application, as needed, regardless of platform. For example, XML Web services created using ASP.NET use the XmlSerializer class to create XML streams that pass data between XML Web service applications throughout the Internet or on intranets. Conversely, deserialization takes such an XML stream and reconstructs the object.

XML serialization can also be used to serialize objects into XML streams that conform to the SOAP specification. SOAP is a protocol based on XML, designed specifically to transport procedure calls using XML.

Namespaces to use XmlSerializer

```
using System.Xml.Serialization
```

5.7 Deserializing XML Data

Deserialization is the process of taking XML-formatted data and converting it to a .NET framework object: the reverse of the process shown above. Providing that the XML is well-formed and accurately matches the structure of the target type, deserialization is a relatively straightforward task.

In the example below, the XML output of the preceding examples is hard-coded into a string, but it could be fetched from a network stream or external file. The XmlSerializer class is used to deserialize the string to an instance of the Test class, and the example then prints the fields to the console. To obtain a suitable stream that can be passed into the XmlSerializer's constructor, a StringReader (from the System.IO namespace) is declared.

Refer URL: http://csharp.net-informations.com/xml/xml-serialization-tutorial.htm

5.8 Unsafe Mode

Unsafe code in C# is the part of the program that runs outside the control of the Common Language Runtime (CLR) of the .NET frameworks. The CLR is responsible for all of the background tasks that the programmer doesn't have to worry about like memory allocation and release, managing stack etc. Using the keyword "unsafe" means telling the compiler that the management of this code will be done by the programmer. Making a code content unsafe introduces stability and security risks as there are no bound checks in cases of arrays, memory related errors can occur which might remain unchecked etc.

A programmer can make the following sub-programs as unsafe:

- Code blocks
- Methods
- Types
- Class
- Struct

Example 1:

```
unsafe
    {
        int x = 10;
        int* ptr;
        ptr = &x;
// displaying value of x using pointer
```

```
        Console.WriteLine("Inside the unsafe code block");
        Console.WriteLine("The value of x is " + *ptr);
        } // end
```

The unsafe code or the unmanaged code is a code block that uses a pointer variable.

5.9 Pointers

A pointer is a variable whose value is the address of another variable i.e., the direct address of the memory location. similar to any variable or constant, you must declare a pointer before you can use it to store any variable address.

Syntax

```
type *var-name;
```

The following are valid pointer declarations:

```
int *ip; /* pointer to an integer */
double *dp;/* pointer to a double */
float *fp; /* pointer to a float */
char *ch   /* pointer to a character */
```

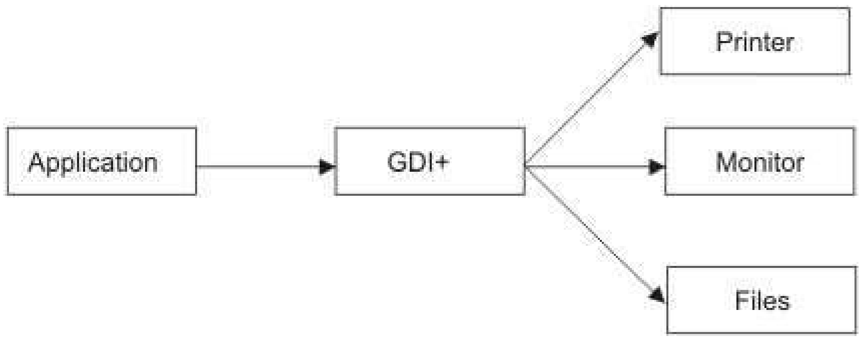
5.10 Graphical Device Interface (GDI)

Graphics Device Interface + (GDI+) is a graphical subsystem of Windows that consists of an application programming interface (API) to display graphics and formatted text on both video display and printer.

GDI+ acts as an intermediate layer between applications and device drivers for rendering two-dimensional graphics, images and text.

GDI was the tool by which what you see is what you get (WYSIWYG) capability was provided in Windows applications. GDI+ is an enhanced C++-based version of GDI. GDI+ helps the developer to write device-independent applications by hiding the details of graphic hardware. It also provides graphic services in a more optimized manner than earlier versions. Due to its object-oriented structure and statelessness, GDI+ provides an easy and flexible interface developers can use to interact with an application's graphical user interface (GUI). Although GDI+ is slightly slower than GDI, its rendering quality is better.

The GDI+ services can be categorized into 2D vector graphics, imaging and typography. Vector graphics include drawing primitives like rectangles, lines and curves. These primitives are drawn using objects of a specific class, which has all the information required. Imaging involves displaying complex images that cannot be displayed using vector graphics and performing image operations such as stretching and skewing. Simple text can be printed in multiple fonts, sizes and colors using typography services of GDI+.



The features included in GDI+ are:

- Gradient brushes used for filling shapes, paths and regions using linear and path gradient pushes
- Cardinal splines for creating larger curves formed out of individual curves
- Independent path objects for drawing a path multiple times
- A matrix object tool for transforming (rotating, translating, etc.) graphics
- Regions stored in world coordinates format, which allows them to undergo any transformation stored in a transformation matrix
- Alpha blending to specify the transparency of the fill color
- Multiple image formats (BMP, IMG, TIFF, etc.) supported by providing classes to load, save and manipulate them
- Sub-pixel anti-aliasing to render text with a smoother appearance on a liquid crystal display (LCD) screen.

5.11 Managed and Unmanaged Code

Code written in the Microsoft .NET development environment is divided into two categories: managed and unmanaged. In brief, code written in the .NET Framework that is being managed by the common language runtime (CLR) is called managed code. Code this is not being managed by the CLR is called unmanaged code.

Managed code enjoys many rich features provide by the CLR, including automatic memory management and garbage collection, cross-language integration, language independence, rich exception handling, improved security, debugging and profiling, versioning, and deployment. With the help of garbage collector (GC), the CLR automatically manages the life cycle of the objects.

When the GC finds that an object has not been used after a certain amount of time, the CLR frees resources associated with that object automatically and removes the object from the memory. You can also control the life cycle of the objects programmatically.

To write both managed and unmanaged applications using Microsoft Visual Studio .NET. You can use Visual C++ to write unmanaged code in Visual Studio .NET.Managed Extensions to C++ (MC++) is the way to write C++ managed code. Code written using C# and Visual Basic .NET is managed code.