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);
```