Index

INTRODUCTION .NET Framework

The **Microsoft .NET Framework** is a software framework that can be installed on computers running Microsoft Windows operating systems. It includes a large library of coded solutions to common programming problems and a virtual machine that manages the execution of programs written specifically for the framework. The .NET Framework is a Microsoft offering and is intended to be used by most new applications created for the Windows platform.

The framework's Base Class Library provides a large range of features including user interface, data and data access, database connectivity, cryptography, web application development, numeric algorithms, and network communications. The class library is used by programmers, who combine it with their own code to produce applications.

Programs written for the .NET Framework execute in a software environment that manages the program's runtime requirements. Also part of the .NET Framework, this runtime environment is known as the Common Language Runtime (CLR). The CLR provides the appearance of an application virtual machine so that programmers need not consider the capabilities of the specific CPU that will execute the program. The CLR also provides other important services such as security, memory management, and exception handling. The class library and the CLR together constitute the .NET Framework.

Some Important Features of Dot Net

1. **Interoperability support (Interop)**Migrating to . NET from existing languages and platforms has been made much easier; Especially if that environment is COM or Java. COM, Interop is built into the framework, and C# will be very familiar for those developing in Java currently. In fact, Microsoft has a migration utility to automatically migrate existing Java source code into C#.
2. **Common language runtime (CLR)**
This is the engine that is shared among all languages supported in . NET, including C#, VB. NET, Managed C++, J#, and others to come. With the help of the CLR, the developer can write base classes in VB. NET, child classes in C#, and aggregate this tree from Managed C++ (this is just one example). You choose the language during implementation.
3. **Base class library (BCL)**
What makes Java so appealing besides the managed environment and cross-platform support is its class library. The . NET framework takes the class library concept a step further by supporting it across any language and extensible for future platform variances. Now BCL-supported features such as remoting, string manipulation, exception handling, and collection management construct is the same from any language conforming to the CLI.
4. **Common type system (CTS)**
This addresses the supported data types within the framework and how they are represented in metadata format. Each supported . NET language need only support a subset of the total data type set. Typically, it will be those types used most frequently (e. g. , integer, short, long, string, char, boolean, object, interface, struct, etc.)
5. **Simplified deployment**
Say goodbye to DLL hell and the nightmare of Windows registration. Applications can now be deployed by a simple XCOPY of the assemblies, ASP. net files, and configuration files.
6. **Full Web service and SOAP support**
Complexities are optionally hidden for building Web service providers and consumers in . NET. Details of the syntax and protocol surrounding XML Web services can be fully customized if needed, however. It is truly the best of both worlds.
7. **XML at the core**
Serialization, streaming, parsing, transforming, and schema support are only some of the "baked-in" XML features of the . NET runtime.
8. **Object-oriented ASP. NET**
Use script for your clients, not your server-based code! Leverage your existing OO framework from ASP. NET and enjoy improved Web application performance due to compiled server code.

Base Class Library (BCL)

The **Base Class Library** (**BCL**) is a standard library available to all languages using the .NET Framework. .NET includes the BCL in order to encapsulate a large number of common functions, such as file reading and writing, graphic rendering, database interaction, and XML document manipulation, which makes the programmer's job easier. It is much larger in scope than standard libraries for most other languages, including C++, and would be comparable in scope to the standard libraries of Java. The BCL is sometimes incorrectly referred to as the Framework Class Library (FCL), which is a superset including the Microsoft.* namespaces. FCL forms the main set with BCL acting as the subset. The BCL is updated with each version of the .NET Framework.

Namespaces

Some of the namespaces may or may not be officially considered part of the BCL by Microsoft, but all are included as part of the libraries that are provided with Microsoft's implementation of the .NET Framework.

- **System :** This namespace include the core needs for programming. It includes base types like String, DateTime, Boolean, and so forth, support for environments such as the console, math functions, and base classes for attributes, exceptions, and arrays.
- **System.Collections**: Defines many common containers or collections used in programming, such as lists, queues, stacks, hashtables, and dictionaries. It includes support for generics.
- **System.Diagnostics:** Gives you the ability to diagnose your application. It includes event logging, performance counters, tracing, and interaction with system processes.
- **System.Globalization:** Provides help for writing internationalized applications. "Culture-related information, including the language, the country/region, the calendars in use, [and] the format patterns for dates, currency, and numbers" can be defined.
- **System.IO:** Allows you to read from and write to different streams, such as files or other data streams. Also provides a connection to the file system.
- **System.Net:** Provides an interface "for many of the protocols used on networks today", such as HTTP, FTP, and SMTP. Secure communication is supported by protocols such as SSL.
- **System.Reflection:** Provides an object view of types, methods, and fields. You have "the ability to dynamically create and invoke types". It exposes the API to access the Reflective programming capabilities of CLR.
- **System.Runtime:** Allows you to manage the runtime behavior of an application or the CLR. Some of the included abilities are interoping with COM or other native code, writing distributed applications, and serializing objects into binary or SOAP.
- **System.Security:** "Provides the underlying structure of the common language runtime security system."This namespace allows you to build security into your application based on policy and permissions. It provides services such as cryptography.
- **System.Text:** Supports various encodings, regular expressions, and a more efficient mechanism for manipulating strings (StringBuilder).
- **System.Threading:** Helps facilitate multithreaded programming. It allows the synchronizing of "thread activities and access to data" and provides "a pool of system-supplied threads."

Common Language Runtime

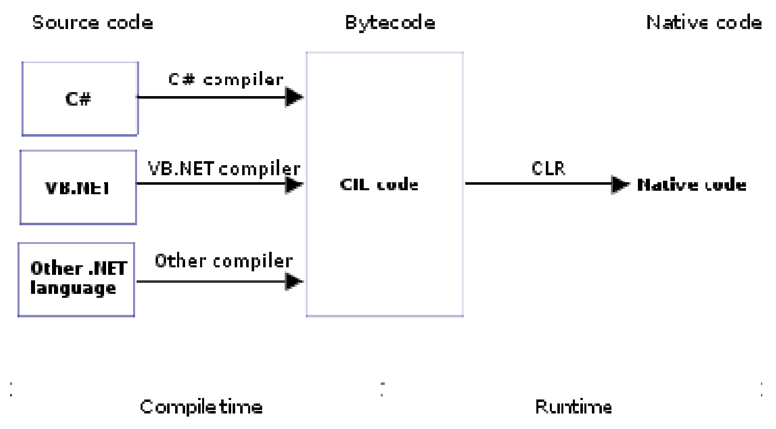
Common Language Runtime (CLR) is a core component of Microsoft's .NET initiative. It is Microsoft's implementation of the Common Language Infrastructure (CLI) standard, which defines an execution environment for program code. In the CLR code is expressed in a form of bytecode called the Common Intermediate Language (CIL, previously known as MSIL -- Microsoft Intermediate Language).

Developers using the CLR write code in a language such as C# or VB.NET. At compile time, a .NET compiler converts such code into CIL code. At runtime, the CLR's just-in-time compiler converts the CIL code into code native to the operating system. Alternatively, the CIL code can be compiled to native code in a separate step prior to runtime. This speeds up all later runs of the software as the CIL-to-native compilation is no longer necessary.

Although some other implementations of the Common Language Infrastructure run on non-Windows operating systems, Microsoft's implementation runs only on Microsoft Windows operating systems.

The CLR allows programmers to ignore many details of the specific CPU that will execute the program. It also provides other important services, including the following:

- Memory management
- Thread management
- Exception handling
- Garbage collection
- Security



PALINDROME

- A **palindrome** is a word, phrase, number or other sequence of units that can be read the same way in either direction
- The word **palindrome** is derived from the Greek palíndromos, meaning running back again (*palín* = AGAIN + *drom–, drameîn* = RUN).
- Some simple examples are:

RACECAR	DEED	LEVEL	PIP
ROTOR	CIVIC	POP	MADAM
EYE	NUN	RADAR	TOOT

Program1: Write a Program in C# to check whether a no. is a Palindrome or not.

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

namespace Program
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("find whether palindrome:");
         Console.WriteLine();
         Console.Write("enter a string:");
         String str2 = "";
         String str1 = Console.ReadLine();
         int n, len;
         len = str1.Length - 1;
         for (n = len; n >= 0; n--)
         {
            str2 = str2 + str1.Substring(n, 1);
         }
         if (str2 == str1)
         {
            Console.Write("the given string" + " " + str1 + " " + "is a palindrome");
            Console.ReadLine();
         }
        else
         {
            Console.Write("the given string" + " " + str1 + " " + "is not a palindrome");
            Console.ReadLine();
         }
      }
   }
}
```

OUTPUT:

```
1: find whether palindrome:
enter a string:MADAM
the given string MADAM is a palindrome

2: find whether palindrome:
enter a string:RAJNI
the given string RAJNI is not a palindrome
```

COMMAND LINE ARGUMENT

Command line arguments are helpful to provide parameters without exposing them to everybody. When developing with .NET and C# you can get the command line arguments from your `Main(string[] Args)` function. `Args` is in fact an array containing all the strings separated by spaces entered in the command line.

The parameter of the Main method is a String array that represents the command-line arguments. Check for the existence of the arguments by testing the Length property and iterate through the array to get the arguments

```
if (args.Length == 0)
{

Console.WriteLine("No Arguments Passed");

}else

{

Console.WriteLine("Following are the arguments");

foreach (string arg in args )

{

Console.WriteLine(arg);

}

}
```

The arguments can also be accessed as zero-based array

Program2: Write a Program in C# to demonstrate Command Line arguments processing.

```
using System;
namespace CommandLine
{
   class CommandLine
   {
      static void Main(string[] args)
      {
         Console.Write("WELCOME to ");
         Console.Write(" " + args[0]);
         Console.Write(" " + args[1]);
      }
   }
}
```

OUTPUT:

```
C:\cd WINDOWS\Microsoft.NET\Framework\v2.0.50727

C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc CommandLine.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.3053
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>CommandLine HelloWorld
WELCOME to  HelloWorld

C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>
```

QUADRATIC EQUATION

In mathematics, a **quadratic equation** is a polynomial equation of the second degree. The general form is

$$ax^2 + bx + c = 0,$$

where *x* represents a variable, and *a*, *b*, and *c*, constants, with $a \neq 0$. (If $a = 0$, the equation becomes a linear equation.). The constants *a*, *b*, and *c*, are called respectively, the quadratic coefficient, the linear coefficient and the constant term or free term.

A quadratic equation with real or complex coefficients has two solutions, called *roots*. These two solutions may or may not be distinct, and they may or may not be *real*. The roots are given by the **quadratic formula**:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

where the symbol "±" indicates that both

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

are solutions.

Program3: Write a Program in C# to find the roots of Quadratic Equation.

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

namespace QuadraticEquation
{
   class QuadraticEquation
   {
      static void Main(string[] args)
      {
         int a, b, c;
         float d, x1, x2;
         Console.WriteLine("\n Enter a,b,c :\n");
         Console.Write(" a = ");
         a = Convert.ToInt32(Console.ReadLine());
         Console.Write(" b = ");
         b = Convert.ToInt32(Console.ReadLine());
         Console.Write(" c = ");
         c = Convert.ToInt32(Console.ReadLine());
         d = b * b - 4 * a * c;
         if (d > 0)
         {
            Console.WriteLine("\n Equation has two roots.");
            x1 = (float)(-b + Math.Sqrt(d)) / (2 * a);
            x2 = (float)(-b - Math.Sqrt(d)) / (2 * a);
            Console.WriteLine("\n The roots are:\n");
            Console.WriteLine("\n root1 = "+x1+"\n root2 = "+x2);
         }
         else if (d == 0)
         {
            Console.WriteLine("\n Equation has only one root.");
            x1 = (float)-b / (2 * a);
            Console.WriteLine("\n The root is:\n");
            Console.WriteLine("\n root = " + x1);
         }
         else
         {
            Console.WriteLine("\n Equation has imaginary roots.");
            x1 = (float)-b / (2 * a);
            x2 = (float)Math.Sqrt(-d) / (2 * a);
            Console.WriteLine("\n The roots are:\n");
            Console.WriteLine("\n root1 = " + x1 + " + " + x2 + " i \n" + " root2 = " + x1 + " - " + x2 + "
i ");
}
Console.ReadLine();
        }
    }
}
```

OUTPUT:

```
 Enter a,b,c :
 a = 1
 b = -2
 c = -2

 Equation has two roots.

The roots are:

 root1 = 2.732051
 root2 = -0.7320508
```

BOXING AND UNBOXING

Boxing and unboxing is a essential concept in C# type system. With Boxing and unboxing one can link between value-types and reference-types by allowing any value of a value-type to be converted to and from type object. Boxing and unboxing enables a unified view of the type system wherein a value of any type can ultimately be treated as an object.

- Converting a value type to reference type is called Boxing.
- Unboxing is an explicit operation.

C# provides a unified type system. All types including value types derive from the type object. It is possible to call object methods on any value, even values of primitive types such as int.

The example

```
class Test
{
        static void Main() {
                int i = 1;
                object o = i;           // boxing
                int j = (int) o;        // unboxing
        }
}
```

An int value can be converted to object and back again to int.

This example shows both boxing and unboxing. When a variable of a value type needs to be converted to a reference type, an object box is allocated to hold the value, and the value is copied into the box. Unboxing is just the opposite. When an object box is cast back to its original value type, the value is copied out of the box and into the appropriate storage location.

Boxing conversions: A boxing conversion permits any value-type to be implicitly converted to the type object or to any interface-type implemented by the value-type. Boxing a value of a value-type consists of allocating an object instance and copying the value-type value into that instance.

Unboxing conversions: An unboxing conversion permits an explicit conversion from type object to any value-type or from any interface-type to any value-type that implements the interface-type. An unboxing operation consists of first checking that the object instance is a boxed value of the given value-type, and then copying the value out of the instance.

Program4: Write a Program in C# to demonstrate boxing and unBoxing .

```
using System;

class Boxing
{
   static void box(object obj)
   {
      Console.WriteLine("value" + obj);
   }
   public static void Main()
   {
      Object o;
      int a = 10;
      double d = 4.4;
      o = a; //boxing integer
      Console.WriteLine("Passing integer");
      box(a);
      Console.WriteLine("Passing Object");
      box(o);
      int x = (int)o;//Unboxing
      Console.WriteLine("Unboxing");
      Console.WriteLine("a=" + x);
      o = d; //boxing double
      Console.WriteLine("Passing double");
      box(d);
      Console.WriteLine("Passing Object");
      box(o);
      double dd = (double)o; //Unboxing
      Console.WriteLine("Unboxing");
      Console.WriteLine("d=" + dd);
      Console.ReadLine();
   }
}
```

OUTPUT:

```
Passing integer
value10
Passing Object
value10
Unboxing
a=10
Passing double
value4.4
Passing Object
value4.4
Unboxing
d=4.4
```

STACK OPERATION

Introduction:

.NET includes a Stack class inside the **System.Collections** namespace. It is efficient because it is implemented using a **System.Collections.ArrayList,** so if you need to consume a stack, it is a better idea to use the built in .NET stack class.

Definition:

A stack is a data structure that allows to add and remove objects at the same position. The last object placed on the stack is the first one to be removed following a Last In First Out (LIFO) data storing method.

Common functions:

The most common functions to manipulate a stack are:

- **Push(element)**: Adds a new object to the last position of the stack.
- **Pop()**: Returns and removes the last object of the stack.
- **Peek()**: Returns the last object without removing it.
- **IsEmpty():** Validates if the stack is empty.

Implementation:

There are many ways to implement a stack. We will use an array to keep the demonstration simple, even though we might consider using a linked list to allow dynamic resizing. We will define the maximum size of the stack at compilation time, but we might also define it at runtime. We will use an index to store the last object position (bottom of the stack) at any time.

- Each time a **Push()** operation is done, I validate if the stack has enough space, I increment the index, and add the new object.
- Each time a **Pop()** operation is done, I validate if the stack **IsEmpty()**, I decrease the index, and return the last object.
- Each time a **Peek()** operation is done, I validate if the stack **IsEmpty()**, and I return the last object.

Program5: Write a Program in C# to implement Stack operations.

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

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         Stack st = new Stack();
         Console.WriteLine("\n enter the element to be inserted into the stack : ");
         int x = Convert.ToInt32(Console.ReadLine());
         st.Push(x);
         String ans;
         int y;
         do
         {
            Console.WriteLine("\n 1.push\n2.pop\n3.exit\nenter your choice:");
            y = Convert.ToInt32(Console.ReadLine());
            switch (y)
            {
               case 1: Console.WriteLine("push operation:");
                  Console.WriteLine("\n enter the element to be inserted : ");
                  x = Convert.ToInt32(Console.ReadLine());
                  st.Push(x);
                  Console.WriteLine("\n elements after push:");
                  foreach (int i in st)
                     Console.WriteLine(i + "");
                  break;
               case 2: st.Pop();
                  Console.WriteLine("\n elements after pop:");
                  foreach (int i in st)
                     Console.WriteLine(i + "");
                  break;
               case 3:
                  Environment.Exit(-1);
                  break;
            }
            Console.WriteLine("\n do you to continue?");
            ans = Console.ReadLine();
         } while (ans == "y");
         Console.ReadLine();
      }
   }
}
```

OUTPUT:

```
enter the element to be inserted into the stack :
1

 1.push
2.pop
3.exit
enter your choice:
1
push operation:

 enter the element to be inserted :
2

 elements after push:
2
1

 do you to continue?
y

 1.push
2.pop
3.exit
enter your choice:
2

 elements after pop:
1

 do you to continue?
n
```

OPERATOR OVERLOADING

Operator overloading, also known as overloading, provides a way to define and use operators such as +, -, and / for user-defined classes or structs. It allows us to define/redefine the way operators work with our classes and structs. This allows programmers to make their custom types look and feel like simple types such as int and string. It consists of nothing more than a method declared by the keyword operator and followed by an operator. There are three types of overloadable operators called unary, binary, and conversion. Not all operators of each type can be overloaded.

Overloading Unary Operators: Unary operators are those that require only a single operand/parameter for the operation. The class or struct involved in the operation must contain the operator declaration. They include +, -, !, ~, ++, --, true, and false. When overloading unary operators, the following rules apply:

- +, -, !, or ~ must take a parameter of the defining type and can return any type
- ++ or - must take and return the defining type
- true or false must take a parameter of the defining type and can return a bool

Overloading Binary Operator: Binary operators are those that require two operands/parameters for the operation. One of the parameters has to be of a type in which the operator is declared. They include +, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, and <=.

Overloading Conversion Operator: Conversion operators are those that involve converting from one data type to another through assignment. There are implicit and explicit conversions.

- Implicit conversions are those that involve direct assignment of one type to another.
- Explicit conversions are conversions that require one type to be casted as another type in order to perform the conversion. Conversions that may cause exceptions or result in loss of data as the type is converted should be handled as explicit conversions.

Program 6: **write a Program to demonstrate Operator Overloading.**

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

namespace Operatoroverloading
{
   class Complex
   {
      private int x, y;
      public Complex() { }
      public Complex(int i, int j)
      {
         x = i;
         y = j;
      }
      public void ShowXY()
      {
         Console.WriteLine(Convert.ToString(x) + " " + Convert.ToString(y));
      }
      public static Complex operator -(Complex c)
      {
         Complex temp = new Complex();
         temp.x = -c.x;
         temp.y = -c.y;
         return temp;
      }
      public static Complex operator +(Complex c1, Complex c2)
      {
         Complex temp = new Complex();
         temp.x = c1.x + c2.x;
         temp.y = c1.y + c2.y;
         return temp;
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         Complex c1 = new Complex(10, 20);
         c1.ShowXY();
         Complex c2 = new Complex();
         c2.ShowXY();
         c2 = -c1;
         c2.ShowXY();
```

```
            Complex c3 = new Complex(30, 40);
            c3.ShowXY();
            Complex c4 = new Complex();
            c4 = c1 + c3;
            c4.ShowXY();
            Console.ReadLine();

        }
    }
}
```

OUTPUT:

```
10 20

0 0

-10 -20

30 40

40 60
```

ARRAYS

An array is a data structure that contains a number of variables of the same type. Arrays are declared with a type:

```
type[] arrayName;
```

Array Overview

An array has the following properties:

- An array can be Single-Dimensional, Multidimensional or Jagged.
- The default value of numeric array elements are set to zero, and reference elements are set to null.
- A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to **null**.
- Arrays are zero indexed: an array with `n` elements is indexed from `0` to `n-1`.
- Array elements can be of any type, including an array type.
- Array types are reference types derived from the abstract base type Array. Since this type implements IEnumerable and IEnumerable, you can use foreach iteration on all arrays in C#.

Single-Dimensional Arrays:

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

This array contains the elements from array[0] to array[4]. The new operator is used to create the array and initialize the array elements to their default values. In this example, all the array elements are initialized to zero.
An array that stores string elements can be declared in the same way.
For example:

```
            string[] stringArray = new string[6];
```

Array Initialization :

It is possible to initialize an array upon declaration, in which case, the rank specifier is not needed because it is already supplied by the number of elements in the initialization list.
For example:

```
            int[] array1 = new int[5] { 1, 3, 5, 7, 9 };
```

A string array can be initialized in the same way. The following is a declaration of a string array where each array element is initialized by a name of a day:

```
string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Multidimensional Arrays:

Arrays can have more than one dimension. For example, the following declaration creates a two-dimensional array of four rows and two columns:

int[,] array = new int[4, 2];

Also, the following declaration creates an array of three dimensions, 4, 2, and 3:

int[, ,] array1 = new int[4, 2, 3];

Array Initialization:

You can initialize the array upon declaration as shown in the following example:

```
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
int[, ,] array3D = new int[,,] { { { 1, 2, 3 } }, { { 4, 5, 6 } } };
```

<u>Program 7:</u> Write a Program in C# to find the second largest element in a single dimensional array.

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

namespace SecondLargest
{
   class SecondLargest
   {
      static void Main(string[] args)
      {
         int n, i, temp;
         Console.WriteLine("enter the no of elements:");
         n = Convert.ToInt32(Console.ReadLine());
         Console.WriteLine("enter the array elements:");
         int[] arr = new int[n];
         for (i = 0; i < n; i++)
         {
            arr[i] = Convert.ToInt32(Console.ReadLine());
         }
         int l1, l2;
         l1 = arr[0];
         l2 = arr[1];
         if (l1 < l2)
         {
            temp = l1;
            l1 = l2;
            l2 = temp;
         }
         for (i = 2; i < n; i++)
         {
            if (arr[i] > l2)
               l2 = arr[i];
            if (l2 > l1)
            {
               temp = l2;
               l2 = l1;
               l1 = temp;
            }
         }
         Console.WriteLine("\n second largest element is : " + l2);
         Console.ReadLine();
      }
   }
}
```

OUTPUT:

```
enter the no of elements:
5
enter the array elements:
99
11
77
55
88

 second largest element is : 88
```

Rectangular Arrays

- In addition to jagged arrays, Visual J# also supports rectangular arrays.
- The element type and shape of an array — including the number of dimensions it has — are part of its type.
- However, the size of the array, as represented by the length of each of its dimensions, is not part of an array's type.
- This split is made clear in the Visual J# language syntax, as the length of each dimension is specified in the array creation expression rather than in the array type.
- For instance, the declaration `int[,,] arr = new int[1, 1, 2];` is an array type of `int[,,]` and an array creation expression of `new int[1, 1, 2]`.
- It is also possible to mix rectangular and jagged arrays.
- For example, `int [,][] mixedArr = new int[,][] {{{1,2}}, {{2,3,4}}};` is a mixed two-dimensional rectangular array of jagged arrays.
- The array is a rectangular array of dimensions `[2,1]`, each element of which is a single dimensional integer array. The length of the array at `mixedArr[0, 0]` is 2, and the length of the array at `mixedArr[1, 0]` is 3.
- Examples:

```
1.   // 2-D rectangular array;
    // Explicit creation with initializer list.
           int[,] arr2 = new int[,] {{1, 2}, {3, 4}};
2.   // 2-D rectangular array;
    // Shorthand: creation with initializer list.
           int[,] arr3 = {{1, 2}, {3, 4}};

3.   // 3-D rectangular array; explicit creation.
           int[,,] arr4 = new int[1, 1, 2];
           arr4[0, 0, 0] = 1;
           arr4[0, 0, 1] = 2;
```

Program 8: **Write a Program in C# to multiply to matrices using Rectangular arrays.**

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

namespace Matrices
{
    class Matrices
    {
        int[,] a;
        int[,] b;
        int[,] c;
        public void input()
        {
            Console.WriteLine("\n size of matrix 1 : ");
            Console.WriteLine("enter the no of rows in matrix 1 : ");
            int m = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine();
            Console.WriteLine("\n enter the no of col in matrix 1 : ");
            int n = Convert.ToInt32(Console.ReadLine());
            a = new int[m, n];
            Console.WriteLine("\n enter the elements of matrix 1: ");
            for (int i = 0; i < a.GetLength(0); i++)
            {
                for (int j = 0; j < a.GetLength(1); j++)
                {
                    a[i, j] = Convert.ToInt32(Console.ReadLine());
                }
            }
            Console.WriteLine("\n size of matrix 2 : ");
            Console.WriteLine("enter the no of rows in matrix 2 : ");
            m = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine();
            Console.WriteLine("\n enter the no of col in matrix 2 : ");
            n = Convert.ToInt32(Console.ReadLine());
            b = new int[m, n];
            Console.WriteLine("\n enter the elements of matrix 2: ");
            for (int i = 0; i < b.GetLength(0); i++)
            {
                for (int j = 0; j < b.GetLength(1); j++)
                {
                    b[i, j] = Convert.ToInt32(Console.ReadLine());
                }
            }
        }
```

```
public void display()
        {
            Console.WriteLine("matrix 1");
            for (int i = 0; i < a.GetLength(0); i++)
            {
                for (int j = 0; j < a.GetLength(1); j++)
                {
                    Console.Write("\t" + a[i, j]);
                }
                Console.WriteLine();
                Console.WriteLine();
            }
            Console.WriteLine("matrix 2");
            for (int i = 0; i < b.GetLength(0); i++)
            {
                for (int j = 0; j < b.GetLength(1); j++)
                {
                    Console.Write("\t" + b[i, j]);
                }
                Console.WriteLine();
                Console.WriteLine();
            }
            Console.WriteLine("\n resultant matrix after multiplying matrix1 and matrix2:");
            for (int i = 0; i < c.GetLength(0); i++)
            {
                for (int j = 0; j < c.GetLength(1); j++)
                {
                    Console.Write("\t" + c[i, j]);
                }
                Console.WriteLine();
                Console.WriteLine();
            }

        }
        public void multiply()
        {
            if (a.GetLength(1) == b.GetLength(0))
            {
                c = new int[a.GetLength(0), b.GetLength(1)];
                for (int i = 0; i < c.GetLength(0); i++)
                {
                    for (int j = 0; j < c.GetLength(1); j++)
                    {
                        c[i, j] = 0;
                        for (int k = 0; k < a.GetLength(1); k++)
                            c[i, j] = c[i, j] + a[i, k] * b[k, j];
                    }
                }
            }
            else
            {
                Console.WriteLine("\n no of col in matrix1 is not equal to no of rows in matrix2");
                Console.WriteLine("\n therefore mul is not possible");
                Environment.Exit(-1);
            }
        }
    }
    class Multiplication
    {
        public static void Main(string[] args)
        {
            Matrices m = new Matrices();
            m.input();
            m.multiply();
            m.display();
            Console.ReadLine()
        }
    }
}
```

OUTPUT:

```
 size of matrix 1 :
enter the no of rows in matrix 1 : 2
 enter the no of col in matrix 1 : 2
enter the elements of matrix 1:
3       3       3       3

size of matrix 2 :
enter the no of rows in matrix 2 : 2
enter the no of col in matrix 2 : 2
 enter the elements of matrix 2:
4       4       4       4
matrix 1
        3       3

        3       3
matrix 2
        4       4

        4       4

 resultant matrix after multiplying matrix1 and matrix2:
        24      24

        24      24
```

Jagged Arrays

A jagged array is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes. A jagged array is sometimes called an "array of arrays." The following examples show how to declare, initialize, and access jagged arrays.

The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

```
int[][] jaggedArray = new int[3][];
```

Before you can use `jaggedArray`, its elements must be initialized. You can initialize the elements like this:

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Each of the elements is a single-dimensional array of integers. The first element is an array of 5 integers, the second is an array of 4 integers, and the third is an array of 2 integers.

It is also possible to use initializers to fill the array elements with values, in which case you do not need the array size. For example:

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

You can also initialize the array upon declaration like this:

```
int[][] jaggedArray2 = new int[][]
{
        new int[] {1,3,5,7,9},
        new int[] {0,2,4,6},
        new int[] {11,22}
};
```

<u>Program 9:</u> Find the sum of all the elements present in a jagged array of 3 inner arrays.

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

namespace JaggedArray
{
    class JaggedArray
    {
        static void Main(string[] args)
        {
            int[][] arr = new int[3][];
            int l1, l2, l3, resadd = 0, i, j;
            Console.WriteLine("Enter length of 1st array   :");
            l1 = Convert.ToInt32(Console.ReadLine());
            arr[0] = new int[l1];
            Console.WriteLine("Enter elements of 1st array :");
            for (i = 0; i < l1; i++)
                arr[0][i] = Convert.ToInt32(Console.ReadLine());
            Console.Clear();

            Console.WriteLine("Enter length of 2nd array   :");
            l2 = Convert.ToInt32(Console.ReadLine());
            arr[1] = new int[l2];
            Console.WriteLine("Enter elements of 2nd array :");
            for (i = 0; i < l2; i++)
                arr[1][i] = Convert.ToInt32(Console.ReadLine());
            Console.Clear();

            Console.WriteLine("Enter length of 3rd array   :");
            l3 = Convert.ToInt32(Console.ReadLine());
            arr[2] = new int[l3];
            Console.WriteLine("Enter elements of 3rd array :");
            for (i = 0; i < l3; i++)
                arr[2][i] = Convert.ToInt32(Console.ReadLine());
            Console.Clear();

            Console.WriteLine("\n THE CURRENT ELEMENTS IN THE ARRAY ARE:\n");
            for (i = 0; i < 3; i++)
            {
                Console.WriteLine();
                for (j = 0; j < arr[i].Length; j++)
                {
                    Console.Write(" " + arr[i][j]);
                    resadd = resadd + arr[i][j];
                }
            }
            Console.WriteLine("\n\n SUM OF ALL ELEMENTS IN THE JAGGED ARRAY IS = " +
resadd);
            Console.ReadLine();
        }
    }
}
```

OUTPUT:

```
 Enter length of 1st array   : 3

Enter elements of 1st array :
1
2
3

Enter length of 1st array   : 3

Enter elements of 1st array :
4
5
6

Enter length of 1st array   : 3

Enter elements of 1st array :
7
8
9

THE CURRENT ELEMENTS IN THE ARRAY ARE:

1 2 3
4 5 6
7 8 9

 SUM OF ALL ELEMENTS IN THE JAGGED ARRAY IS = 45
```

Program 10: Write a Program to reverse a given string using C#.

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

namespace Reverse
{
    class Reverse
    {
        static void Main(string[] args)
        {
            String inv;
            String outv = "";
            Console.WriteLine(" Enter a string :\n");
            inv = Console.ReadLine();
            for (int i = inv.Length - 1; i >= 0; i--)
            {
                outv = outv + inv.Substring(i, 1);
            }
            Console.WriteLine("The reversed string is: " + outv);
            Console.ReadLine();
        }
    }
}
```

OUTPUT:

```
Enter a string :

RAJNI

The reversed string is: INJAR
```

TRY, CATCH, FINALLY

- We can use the **Try...Catch...Finally** statements for structured exception handling.
- This allows you to execute a particular block of statements if a specified exception occurs while your code is running.
- When this happens, the code is said to *throw* the exception, and you *catch* it with the appropriate **Catch** statement.
- Execution enters the **Try...Catch...Finally** block with the statement block following the **Try** statement.
- If Visual Basic completes this block without an exception being generated, it looks for the optional **Finally** statement at the end.
- If you have supplied the **Finally** statement, Visual Basic executes the statement block following it.
- In any case, control is then transferred to the statement following the **End Try** statement.
- If an exception occurs, Visual Basic examines the **Catch** statements in the order they appear within **Try...Catch...Finally**.
- If it finds a **Catch** statement that handles the generated exception, it executes the corresponding statement block.
- When it has finished executing the **Catch** block, it executes the **Finally** block if it is present. Execution then proceeds to the statement following the **End Try** statement.
- A **Catch** statement handles an exception that is of the same type as the type declared in the **Catch** statement, or of a type derived from it.
- If you supply a **Catch** statement that does not specify an exception type, it handles any exception derived from the **Exception** class.
- If such a statement is the last **Catch** statement, it can catch any exception that you did not handle in the preceding **Catch** blocks.

Program 11: Using Try, Catch and Finally blocks write a program in C# to demonstrate error handling.

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ErrorHandling
{
   class ErrorHandling
   {
      static int m = 10;
      static int n = 0;
      static void Division()
      {
         try
         {
            int k = m / n;
         }
         catch (ArgumentException e)
         {
            Console.WriteLine("Exception caught:" + e.Message);
         }
         finally
         {
            Console.WriteLine("Inside division method");
            Console.ReadLine();
         }
      }
      static void Main(string[] args)
      {
         try
         {
            Division();
         }
         catch (DivideByZeroException e)
         {
            Console.WriteLine("Exception caught:" + e.Message);
         }
         finally
         {
            Console.WriteLine("inside main method");
            Console.ReadLine();
         }
      }
   }
}
```

OUTPUT:

```
Inside division method
Exception caught:Attempted to divide by zero.
inside main method
```

Switch Statement

The **switch** statement is a control statement that handles multiple selections and enumerations by passing control to one of the **case** statements within its body. It takes the following form:

```
switch (expression)
{
   case constant-expression:
      statement
      jump-statement
   [default:
      statement
      Jump-statement]
}
```

Where:

expression

An integral or string type expression.

statement

The embedded statement(s) to be executed if control is transferred to the **case** or the **default**.

jump-statement

A jump statement that transfers control out of the **case** body.

constant-expression

Control is transferred to a specific **case** according to the value of this expression.

The ***switch*** statement is a decision structure that allows for a selection to be made among a mutually exclusive set of alternatives.

- Each alternative must be specified with a *case* label.
- Each alternative must be terminated with a *break;* statement.

The values examined by the *switch* statement in C# must be either integers or strings.

An optional *default* label can be used to instruct the program to execute a block of code if none of the case conditions are met.

Program 12: **Design a simple calculator using Switch Statement in C#.**

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

namespace Calculator
{
   class Calculator
   {
      static void Main(string[] args)
      {
         int a, b, x;
         string c;
         double s = 0;

         do
         {
            Console.WriteLine("Enter value of A : ");
            a = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Enter value of B : ");
            b = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Enter The OPeration \n1: Addition\n2: Substraction\n3: 
Multiplication\n4: Division ");
            x = Convert.ToInt32(Console.ReadLine());
            switch (x)
            {
               case 1: s = a + b; break;
               case 2: s = a - b; break;
               case 3: s = (double)a * b; break;
               case 4: s = (double)a / b;
                  if (b == 0)
                  {
                     Console.WriteLine("denominator value should be greater than zero");

                  }
                  break;
               default: Console.WriteLine("Invaluid Input : "); break;
            }
            Console.WriteLine("The result is = " + s);
            Console.WriteLine("Do you want to Continue (Y/N)");
            c = Console.ReadLine();
         } while (c == "y");

      }
   }
}
```

OUTPUT:

```
Enter value of A: 5
Enter value of B : 4

Enter The OPeration
1: Addition
2: Substraction
3: Multiplication
4: Division
1
The result is = 9
Do you want to Continue (Y/N): y

Enter value of A : 4
Enter value of B : 5

Enter The OPeration
1: Addition
2: Substraction
3: Multiplication
4: Division
2
The result is = -1
Do you want to Continue (Y/N): y

Enter value of A : 3
Enter value of B : 3

Enter The OPeration
1: Addition
2: Substraction
3: Multiplication
4: Division
3
The result is = 9
Do you want to Continue (Y/N): y

Enter value of A : 6
Enter value of B : 4

Enter The OPeration
1: Addition
2: Substraction
3: Multiplication
4: Division
4
The result is = 1.5
Do you want to Continue (Y/N): n
```

Virtual And Override Keywords

The **virtual** keyword is used to modify a method, property, indexer or event declaration, and allow it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it:

```
public virtual double Area()
{
    return x * y;
}
```

- The implementation of a virtual member can be changed by an overriding member in a derived class.
- When a virtual method is invoked, the run-time type of the object is checked for an overriding member.
- The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.
- By default, methods are non-virtual.
- You cannot override a non-virtual method.
- You cannot use the **virtual** modifier with the **static**, **abstract, private** or **override** modifiers.

Virtual properties behave like abstract methods, except for the differences in declaration and invocation syntax.

- It is an error to use the **virtual** modifier on a static property.
- A virtual inherited property can be overridden in a derived class by including a property declaration that uses the **override** modifier.

The **override** modifier is required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event.

- An **override** method provides a new implementation of a member inherited from a base class.
- The method overridden by an **override** declaration is known as the overridden base method.
- The overridden base method must have the same signature as the **override** method.
- For information on inheritance.
- You cannot override a non-virtual or static method.
- The overridden base method must be **virtual**, **abstract**, or **override**.
- An **override** declaration cannot change the accessibility of the **virtual** method.
- Both the **override** method and the **virtual** method must have the same access level modifier.
- You cannot use the modifiers **new**, **static**, **virtual**, or **abstract** to modify an **override** method.
- An overriding property declaration must specify the exact same access modifier, type, and name as the inherited property, and the overridden property must be **virtual**, **abstract**, or **override**.

Program 13: **Demonstrate use of Virtual and override key words in C# with a simple program.**

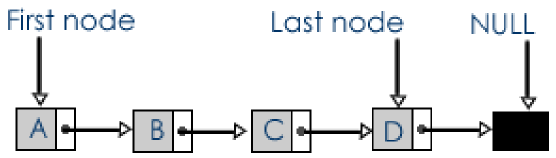
```
using System;
using System.Collections.Generic;
using System.Text;
namespace VirtualPgm
{
   public class Customer
   {
      public virtual void CustomerType()
      {

         Console.WriteLine("I am customer");
      }
   }
   public class CorporateCustomer : Customer
   {
      public override void CustomerType()
      {
         Console.WriteLine("I am Corporate Customer");
      }
   }
   public class PersonalCustomer : Customer
   {
      public override void CustomerType()
      {
         Console.WriteLine("I am personal customer");
      }
   }
   class VirtualPgm
   {
      static void Main(string[] args)
      {
         Customer[] c = new Customer[3];
         c[0] = new CorporateCustomer();
         c[1] = new CorporateCustomer();
         c[2] = new Customer();
         foreach (Customer customerobject in c)
         {
            customerobject.CustomerType();
         }
         Console.ReadLine();
      }
   }
}
```

OUTPUT:

```
I am Corporate Customer
I am Corporate Customer
I am customer
```

LINKED LIST

It is a linear collection of Self-Referential class objects, called nodes, connected by reference links.



In our Linked List we will implement how to insert new Item at front & back of the linked list, as well the Removal of those Items from both sides. In Addition we will implement insertion at specific place in the Linked list, in this case we will conseder our list as Zero-Based Index.

InsertAtFront:

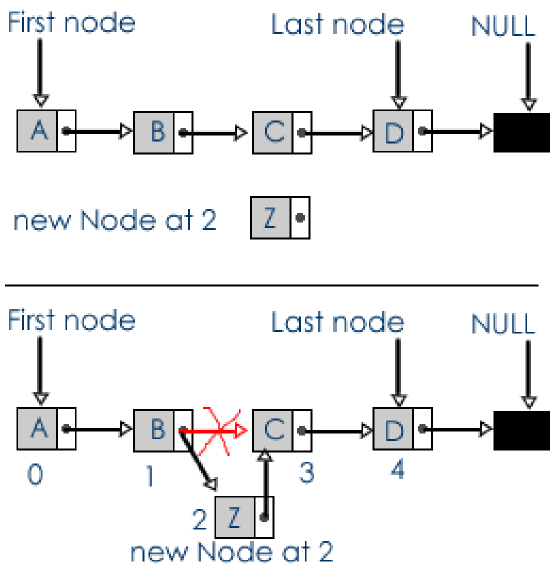
1-Call IsEmpty Property to determine whether the list is empty

2-If the list is empty, then the first node and last node will refer to same new ListNode and set its value to ObjItem. But if the list is not Empty, a new node will be added into the list by setting m_FirstNode to refer to a new ListNode, the 2nd Argument in the ListNode Constructor is set to be previuos m_FirstNode.

InsertAtBack:

Well it perform the same functionality of InsertAtFront if the list is Empty, but if it is not, then the m_LastNode will be set to new ListNode and the Node before it will link to it and the list node will link to null.

Diagram for *InsertAt*:



Program 14: **Implement Linked List in C# using the existing collections namespace.**

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

namespace LinkedList
{
   class LinkedList
   {
      static void Main(string[] args)
      {
         int ch = 0, item, ch2, pos;
         LinkedList<int> lst = new LinkedList<int>();
         LinkedListNode<Int32> lno;
         do
         {
            sceen();
            ch = read();
            switch (ch)
            {
               case 1: Console.WriteLine("\nenter the value:");
                  item = read();
                  Console.WriteLine("\n1.first\n2.last\n3.position");
                  Console.WriteLine();
                  ch2 = read();
                  switch (ch2)
                  {
                     case 1: lst.AddFirst(item);
                        break;
                     case 2: lst.AddLast(item);
                        break;
                     case 3: Console.WriteLine("\n enter the position:");
                        pos = read();
                        if (pos < 0 || pos > lst.Count)
                        {
                           Console.WriteLine("\n position out of range");
                           Console.WriteLine("\n press any key to continue");
                           Console.ReadKey();
                           break;
                        }
                        lno = lst.First;
                        for (int i = 1; i < lst.Count; i++)
                        {
                           if (pos == i)
                           {
                              lst.AddAfter(lno, item);
                              break;
                           }
                           lno = lno.Next;
```

```
                    }
                    break;
                default: Console.WriteLine("\n invalid entry");
                    Console.WriteLine("\n press any key to continue: ");
                    Console.ReadKey();
                    break;
            }
            break;
        case 2: if (lst.Count == 0)
            {
                display(lst);
                break;
            }
            Console.WriteLine("\n 1.first\n2.second\n3.position\n");
            ch2 = read();
            switch (ch2)
            {
                case 1: lst.RemoveFirst();
                    break;
                case 2: lst.RemoveLast();
                    break;
                case 3: Console.WriteLine("\n enter the position");
                    pos = read();
                    if (pos > lst.Count)
                    {
                        Console.WriteLine("\n position out of range");
                        break;
                    }
                    lno = lst.First;
                    for (int i = 0; i < lst.Count; i++)
                    {
                        if (pos == i)
                        {
                            lst.Remove(lno.Value);
                            break;
                        }
                        lno = lno.Next;
                    }
                    break;
                default: Console.WriteLine("\n press any key to continue");
                    Console.ReadKey();
                    break;
            }
            break;
        case 3: display(lst);
            break;
        case 4: break;
        default: Console.WriteLine("\n invalid entry");
            break;
    }
} while (ch != 4);
```

```
}
public static int read()
{
    Boolean b = true;
    int i = 0;
    do
    {
        try
        {
            i = Convert.ToInt32(Console.ReadLine());
            b = true;
        }
        catch
        {
            Console.WriteLine("\n enter int value:");
            b = false;
        }
    } while (b != true);
    return i;
}
public static void sceen()
{
    Console.Clear();
    Console.WriteLine("\nlinked list");
    Console.WriteLine("\n1.insert \n2.remove \n3.display \n4.exit");
}
public static void display(LinkedList<Int32> lst)
{
    LinkedListNode<int> lno = new LinkedListNode<int>(0);
    {
        if (lst.Count == 0)
        {
            Console.WriteLine("\nlist empty");
        }
        else
        {
            lno = lst.First;
            for (int k = 0; k < lst.Count; k++)
            {
                Console.WriteLine("\n node" + k);
                if (lno.Previous == null)
                    Console.WriteLine("left->null");
                else
                    Console.WriteLine("left->{0}", lno.Previous.Value);
                Console.WriteLine("\t\tinfo->{0}", lno.Value);
                if (lno.Next == null)
                    Console.WriteLine("right->null");
                else
                    Console.WriteLine("right->{0}", lno.Next.Value);
                Console.WriteLine("\n");
                lno = lno.Next;
                    }
                }
            }
            Console.WriteLine("\n press any key to continue");
            Console.ReadKey();
        }
    }
}
```

OUTPUT:

```
linked list

1.insert
2.remove
3.display
4.exit
1
enter the value:
1

1.first
2.last
3.position

1

linked list

1.insert
2.remove
3.display
4.exit
3

 node0
left->null
                info->1
right->null

 press any key to continue
n
```

Abstract class and Abstract Method

Abstract class: Abstract class is a class that has no direct instances, but whose descendants may have direct instances.

- There are case i which it is useful to define classes for which the programmer never intends to instantiate any objects; because such classes normally are used as bae-classes in inheritance hierarchies.
- we call such classes **abstract classes** These classes cannot be used to instantiate objects; because abstract classes are incomplete.
- Derived classes called **concrete classes**must define the missing pieces.
- Abstract classes normally contain one or more *abstract methods or abstract properties*, such methods or properties do not provide implementations, but our derived classes must override inherited abstract methods or properties to enable obejcts ot those derived classes to be instantiated, not to override those methods or properties in derived classes is syntax error, unless the derived class also is an abstract class.
- A class is made abstract by declaring it with Keyword ***abstract***.

Abstract methods: When an instance method declaration includes an `abstract` modifier, that method is said to be an abstract method.

- Although an abstract method is implicitly also a virtual method, it cannot have the modifier `virtual`.
- An abstract method declaration introduces a new virtual method but does not provide an implementation of that method.
- Instead, non-abstract derived classes are required to provide their own implementation by overriding that method.
- Because an abstract method provides no actual implementation, the *method-body* of an abstract method simply consists of a semicolon.
- Abstract method declarations are only permitted in abstract classes.
- An abstract method declaration is permitted to override a virtual method.
- This allows an abstract class to force re-implementation of the method in derived classes, and makes the original implementation of the method unavailable.

Program 15: **Write a Program to demonstrate abstract class and abstract methods in C#.**

```
using System;

namespace AbstractClass
{
   public abstract class Vehicle
   {
      public string Name;
      public int Wheels;
      public double Amount;
      public abstract void Calculate();
   }
   public class Motorcycle : Vehicle
   {
      public Motorcycle()
      {
         this.Wheels = 2;
      }
      public Motorcycle(string s)
      {
         this.Wheels = 2;
         this.Name = s;
      }
      public override void Calculate()
      {
         this.Amount = 100000 + (500 * this.Wheels);
         Console.WriteLine("This motor cycle name is " + this.Name + " and its price is " +
this.Amount);
      }
   }
   public class Car : Vehicle
   {
      private string EngineType;
      public Car()
      {
         this.Wheels = 4;
      }
      public Car(string s, string t)
      {
         this.Wheels = 4;
         this.Name = s;
         this.EngineType = t;
      }
      public override void Calculate()
      {
         this.Amount = 100000 + (500 * this.Wheels) + 8000;
         Console.WriteLine("This car name is " + this.Name + " has engine type " + this.EngineType + "
and price " + this.Amount);
      }
```

```
    }
    public class AbstractClass
    {
        public static void Main(string[] args)
        {
            Vehicle v;
            Motorcycle m = new Motorcycle("Pulsar");
            Car c = new Car("Jazz", "Petrol");
            //m.Calculate();
            //c.Calculate();
            v = m;
            v.Calculate();
            v = c;
            v.Calculate();
            Console.ReadLine();
        }
    }
}
```

OUTPUT:

```
This motor cycle name is Pulsar and its price is 101000

This car name is Jazz has engine type Petrol and price 110000
```

Interface Class

- Interfaces are defined by using the interface keyword, as shown in the following example:

```
interface IEquatable<T>
{
  bool Equals(T obj);
}
```

- Interfaces describe a group of related functionalities that can belong to any class or struct.
- When a class or struct is said to inherit an interface, it means that the class or struct provides an implementation for all of the members defined by the interface.
- The interface itself provides no functionality that a class or struct can inherit in the way that base class functionality can be inherited.
- However, if a base class implements an interface, the derived class inherits that implementation.
- Classes and structs can inherit from interfaces in a manner similar to how classes can inherit a base class or struct, with two exceptions:

    1. A class or struct can inherit more than one interface.
    2. When a class or struct inherits an interface, it inherits only the method names and signatures, because the interface itself contains no implementations.
- An interface contains only the signatures of methods, delegates or events.
- The implementation of the methods is done in the class that implements the interface.
- An interface cannot contain fields.
- Interfaces members are automatically public.
- An interface can be a member of a namespace or a class and can contain signatures of the following members:

    1. Methods
    2. .Properties
    3. Indexers
    4. Events

- An interface can inherit from one or more base interfaces.
- When a base type list contains a base class and interfaces, the base class must come first in the list.
- A class that implements an interface can explicitly implement members of that interface.
- An explicitly implemented member cannot be accessed through a class instance, but only through an instance of the interface.

Program 16: **Write a Program in C# to build a class which implements an interface which is already existing.**

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

namespace InterfaceClass
{
   interface Addition
   {
      int Add();
   }
   interface Multiplication
   {
      int Multiply();
   }
   class Compute : Addition, Multiplication
   {
      int x, y;
      public Compute(int a, int b)
      {
         this.x = a;
         this.y = b;
      }
      public int Add()
      {
         return (x + y);
      }
      public int Multiply()
      {
         return (x * y);
      }
   }
   class InterfaceClass
   {
      static void Main(string[] args)
      {
         int a, b;
         Console.Write("Enter 2 nos:");
         a = Convert.ToInt32(Console.ReadLine());
         b = Convert.ToInt32(Console.ReadLine());
         Compute ob1 = new Compute(a, b);
         Console.WriteLine("Addition is:" + ob1.Add());
         Console.WriteLine("Multiplication is:" + ob1.Multiply());
         Console.ReadLine();
      }
   }
}
```

OUTPUT:

```
Enter 2 nos:
4
 5

Addition is:9

Multiplication is:20
```

Properties

- Properties are members that provide a flexible mechanism to read, write, or compute the values of private fields.
- Properties can be used as though they are public data members, but they are actually special methods called *accessors*.
- This enables data to be accessed easily while still providing the safety and flexibility of methods.

Properties Overview:

- Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
- **get** property accessor is used to return the property value, and a **set** accessor is used to assign a new value.
- These accessors can have different access levels.
- The **value** keyword is used to define the value being assigned by the **set** indexer.
- Properties that do not implement a **set** method are read only.
- The code block for the **get** accessor is executed when the property is read; the code block for the **set** accessor is executed when the property is assigned a new value.
- A property without a **set** accessor is considered read-only. A property without a **get** accessor is considered write-only. A property with both accessors is read-write.
- Unlike fields, properties are not classified as variables. Therefore, it is not possible to pass a property as a ref (C# Reference) or out (C# Reference) parameter.

Properties have many uses:

- They can validate data before allowing a change;
- They can transparently expose data on a class where that data is actually retrieved from some other source, such as a database;
- They can take an action when data is changed, such as raising an event, or changing the value of other fields.
- Properties are declared within the class block by specifying the access level of the field, followed by the type of the property, followed by the name of the property, then a code block declaring a **get**-accessor and/or a **set** accessor. For example:

```
public class Date{
    private int month = 7;  //"backing store"
    public int Month
    {
        get
        {
            return month;
        }
        set
        {
            if ((value > 0) && (value < 13))
            {
                month = value;
            }
        }
    }
}
```

Program 17: Write a Program to illustrate the use of different Properties in C#.

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

namespace properties
{
    public class Student
    {

        private int usn;
        private string percentile;
        private string fullName;

        public int ID
        {
            get { return usn; }
            set
            {

                usn = value;
            }
        }

        public string Name
        {
            get { return fullName; }
            set { fullName = value; }
        }

        public string marks
        {
            get { return percentile; }
            set { percentile = value; }
        }
    }
   class Properties
    {
        static void Main(string[] args)
        {
            Student e = new Student();
            e.ID = 81;
            e.Name = "SCIENTIST THE RESEARCHIST";
            e.marks = "58%";
            Console.WriteLine("ID= {0},NAME={1},marks percentile={2}", e.ID, e.Name, e.marks);
            Console.ReadLine();
         }
    }
}
```

OUTPUT:

```
ID= 81,

NAME=SCIENTIST THE RESEARCHIST,

marks percentile=58%
```

Program 18: **Demonstrate arrays of interface types with a C# program.**

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

namespace InterfaceType
{
   public interface IShape
   {
      void Calculate();
      void Display();
   }
   public class Rectangle : IShape
   {
      private double Area;
      private double Length;
      private double Breadth;
      public Rectangle()
      {
         this.Length = 0;
         this.Breadth = 0;
      }
      public Rectangle(double l, double b)
      {
         this.Length = l;
         this.Breadth = b;
      }
      public void Calculate()
      {
         this.Area = this.Length * this.Breadth;
      }
      public void Display()
      {
         Console.WriteLine("Area of Rectangle is : " + this.Area);
      }
   }
   public class Square : IShape
   {
      private double Area;
      private double Side;
      public Square()
      {
         this.Side = 0;
      }
      public Square(double s)
      {
         this.Side = s;
      }
      public void Calculate()
```

```
        {
           this.Area = this.Side * this.Side;
        }
        public void Display()
        {
           Console.WriteLine("Area of Square is : " + this.Area);
        }
    }
    public class Circle : IShape
    {
        private double Area;
        private double Radius;
        public Circle()
        {
           this.Radius = 0;
        }
        public Circle(double s)
        {
           this.Radius = s;
        }
        public void Calculate()
        {
           this.Area = 3.1416 * this.Radius * this.Radius;
        }
        public void Display()
        {
           Console.WriteLine("Area of Circle is : " + this.Area);
        }
    }
    public class InterfaceType
    {
        public static void Main(string[] args)
        {
           IShape[] s = { new Rectangle(10, 20), new Square(30), new Circle(40) };
           for (int i = 0; i < s.Length; i++)
           {
              s[i].Calculate();
              s[i].Display();
           }
           Console.ReadLine();
        }
    }
}
```

OUTPUT:

```
Area of Rectangle is : 200

Area of Square is : 900

Area of Circle is : 5026.56
```