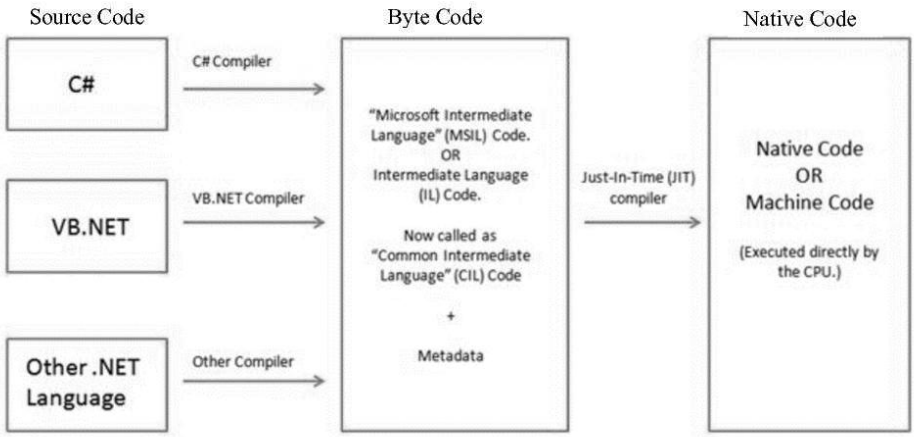
Unit-1 Language Preliminaries

Introduction to .Net framework:

.Net framework is not a programming language but it is a framework tools which make the environment for run the Program. .NET framework is an open source software framework developed by Microsoft for building different types of applications i.e. console applications, windows application, web applications and mobile applications etc. The first version of .Net framework was 1.0 which came in the year 2002.

Compilation and execution of .Net applications:



Compiler time process: The .Net framework has one or more language compilers, such as Visual Basic, C#, Visual C++ and other .NET language compiler. Anyone of the compilers translates your source code into Microsoft Intermediate Language (MSIL) code. Currently "Microsoft Intermediate Language" (MSIL) code is also known as the "Intermediate Language" (IL) Code **or** "Common Intermediate Language" (CIL) Code.

SOURCE CODE → .NET COMLIPER→ BYTE CODE (MSIL + META DATA)

Runtime process: The Common Language Runtime (CLR) includes a JIT compiler for converting MSIL to native code. The JIT Compiler in CLR converts the MSIL code into native machine code that is then executed by the OS. During the runtime of a program, the "Just in Time" (JIT) compiler of the Common Language Runtime (CLR) uses the Metadata and converts Microsoft Intermediate Language (MSIL) into native code. The CLR also provides other services related to automatic garbage collection, exception handling, and resource management.

BYTE CODE (MSIL + META DATA) →Just-In-Time (JIT) compiler→ NATIVE CODE

.NET Framework Features / Characteristics:

- NET stands for Network enabled technology.
- Easy development of different types of applications.
- Multi-Language support with asynchronous programming.
- Object oriented programming (OOPS) support.
- Easy and rich debugging support.
- Memory management and security.

Main Components of .NET Framework:

- **Common Language Runtime (CLR)**: CLR is the basic and Virtual Machine component of the .NET Framework. It is the run-time environment in the .NET Framework that runs the codes and helps in making the development process easier by providing the various services such as remoting, thread management, type-safety, memory management, robustness, etc.. Basically, it is responsible for managing the execution of .NET programs regardless of any .NET programming language.
- **Framework Class Library (FCL):** It is the collection of reusable, object-oriented class libraries and methods, etc that can be integrated with CLR. Also called the Assemblies. It is just like the header files in C/C++ and packages in the java. Installing .NET framework basically is the installation of CLR and FCL into the system. Below is the overview of .NET Framework

CLR: CLR provides an environment to execute .NET applications on target machines. CLR is also a common runtime environment for all .NET code irrespective of their programming language, as the compilers of respective language in .NET Framework convert every source code into a common language known as MSIL or IL (Intermediate Language).CLR also provides various services to execute processes, such as memory management service and security services. CLR performs various tasks to manage the execution process of .NET applications

Main components of CLR:

- Common Language Specification (CLS)
- Common Type System (CTS)
- Garbage Collection (GC)
- Just In – Time Compiler (JIT)

Common Language Specification (CLS):
It is responsible for converting the different .NET programming language syntactical rules and regulations into CLR understandable format. Basically, it provides the Language Interoperability. Language Interoperability means to provide the execution support to other programming languages also in .NET framework.

Language Interoperability can be achieved in two ways:

1. **Managed Code:** The MSIL code which is managed by the CLR is known as the Managed Code. Managed code CLR provides **three** .NET facilities:

- CAS(Code Access Security)
- Exception Handling
- Automatic Memory Management.

2. **Unmanaged Code:** Before .NET development the programming language like .COM Components & Win32 API do not generate the MSIL code. So these are not managed by CLR rather managed by Operating System.

Common Type System (CTS) :
Every programming language has its own data type system, so CTS is responsible for understanding all the data type systems of .NET programming languages and converting them into CLR understandable format which will be a common format.
There are 2 Types of CTS that every .NET programming language have:

1. **Value Types:** Value Types will store the value directly into the **memory location**. These types work with **stack** mechanism only. CLR allows memory for these at Compile Time.
2. **Reference Types:** Reference Types will contain a **memory address** of value because the reference types won't store the variable value directly in memory. These types work with **Heap** mechanism. CLR allots memory for these at Runtime.

Garbage Collector:

It is used to provide the Automatic Memory Management feature. If there was no garbage collector, programmers would have to write the memory management codes which will be a kind of overhead on programmers.

JIT (Just In Time Compiler):

It is responsible for converting the CIL (Common Intermediate Language) into machine code or native code using the Common Language Runtime environment.

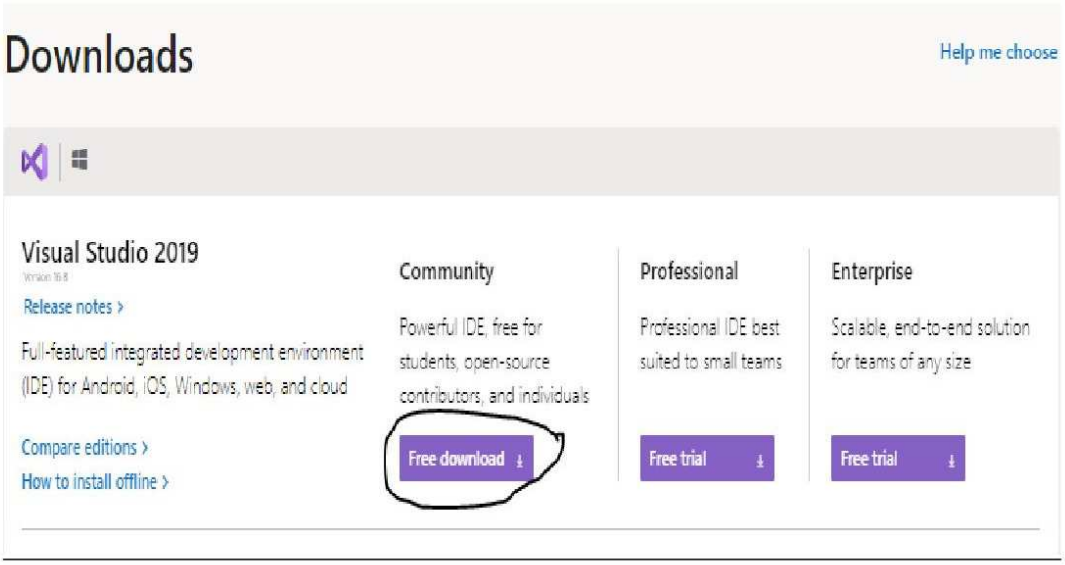
Benefits of CLR:

- It improves the performance by providing a rich interact between programs at run time.
- Enhance portability by removing the need of recompiling a program on any operating system that supports it.
- Security also increases as it analyzes the MSIL instructions whether they are safe or unsafe. Also, the use of delegates in place of function pointers enhance the type safety and security.
- Support automatic memory management with the help of Garbage Collector.
- Provides cross-language integration because CTS inside CLR provides a common standard that activates the different languages to extend and share each other's libraries.

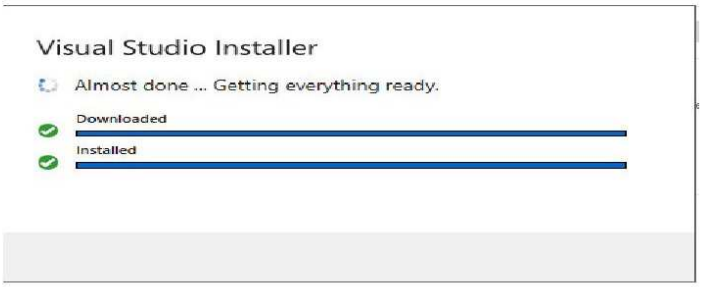
- Provides support to use the components that developed in other .NET programming languages.
- Provide language, platform, and architecture independence.
- It allows easy creation of scalable and multithreaded applications, as the developer has no need to think about the memory management and security issues.

Installing Visual Studio 2019 Community:

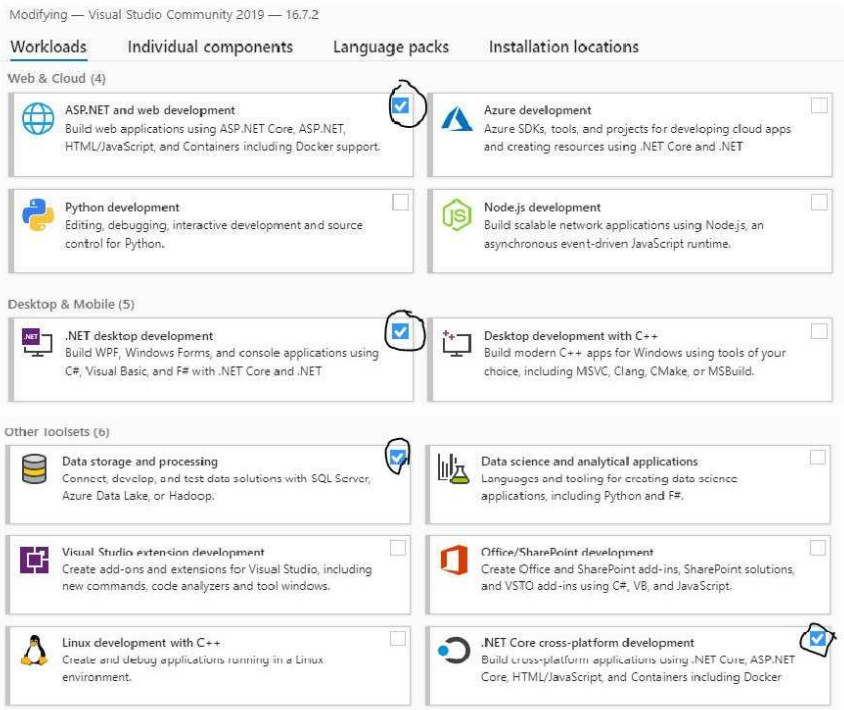
1. Goto: https://visualstudio.microsoft.com/downloads



2. Click to Community Edition:
3. Click Setup and install.



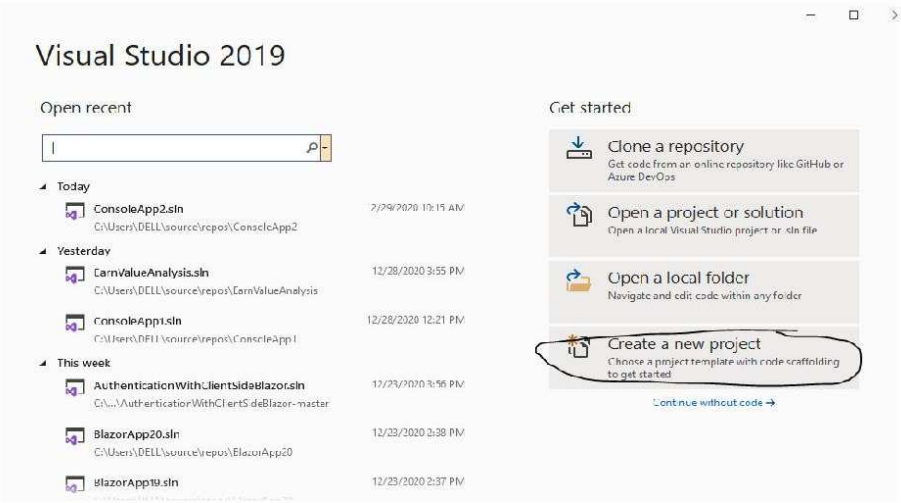
4. Check Following option



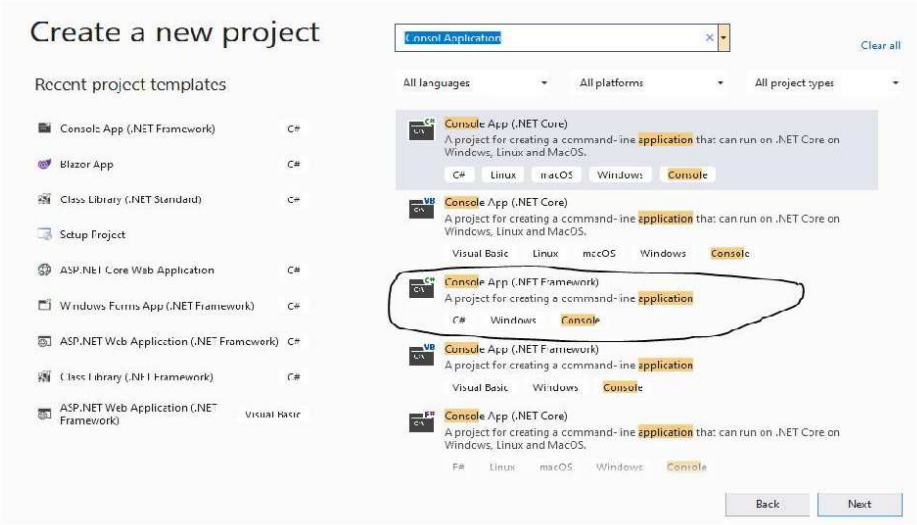
5. Click to Install.

How to Open project in Visual Studio 2019:

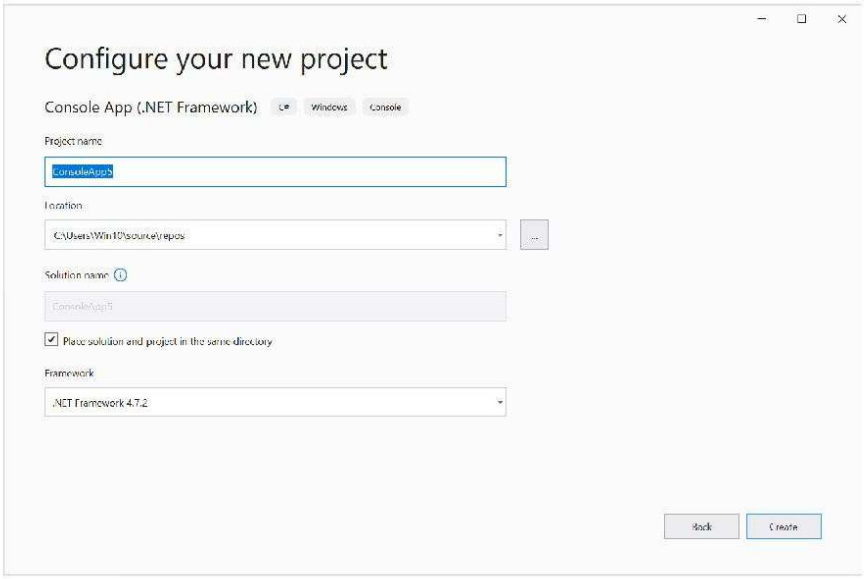
1. Open Visual Studio from program menu
2. Click Create a new project as highlighted below.



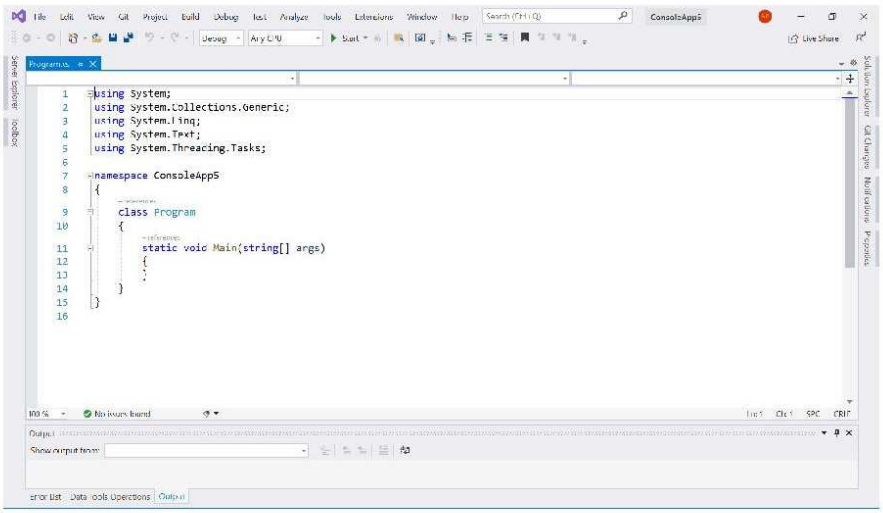
3. After Clicking window appears. Search Console application in search bar and click next.



4. Click create.



5. After clicking create.



Basic Languages constructs:

This simple one-class console "Hello world" program demonstrates many fundamental concepts throughout this article and several future articles.

```
using System;
namespace ConsoleApp5
{
   class Program
   {
      //Entry point of the program
      static void Main(string[] args)
      {
         //print Hello World!
         Console.WriteLine("Hello World!");
      }
   }
}
```

using System : The using System line means that you are using the System library in your project. Which gives you some useful classes like Console or functions/methods.

namespace: C# namespaces allow you to group related classes, interfaces, structs, enums, and delegates into a single logical unit. Namespaces also help you avoid naming conflict issues. For example

class: is a keyword which is used to define class.

Program: is the class name. A class is a blueprint or template from which objects are created. It can have data members and methods. Here, it has only Main method.

static: is a keyword which means object is not required to access static members. So it saves memory.

void: is the return type of the method. It doesn't return any value. In such case, return statement is not required.

Main: is the method name. It is the entry point for any C# program. Whenever we run the C# program, **Main()** method is invoked first before any other method. It represents startup of the program.

string[] args: is used for command line arguments in C#. While running the C# program, we can pass values. These values are known as arguments which we can use in the program.

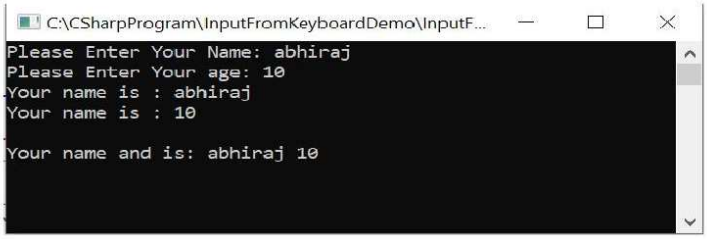
Console.Write (): This function displays text, values on the output device and does not insert a new line after the message.

Console.WriteLine(): This function displays text, values on the output device and inserts a new line after the message.

System.Console.WriteLine("Hello World!"): Here, System is the namespace. Console is the class defined in System namespace. The WriteLine() is the static method of Console class which is used to write the text on the console.

```
using System;
namespace InputFromKeyboardDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Please Enter Your Name: ");
            string str=Console.ReadLine();

            Console.Write("Please Enter Your age: ");
            int age = int.Parse(Console.ReadLine());

            Console.WriteLine("Your name is : " + str);//concatenation syntax
            Console.WriteLine("Your name is : " + age);
            Console.WriteLine();

            Console.WriteLine("Your name and is: {0} {1}",str,age);//placeholder syntax
            Console.ReadLine();
        }
    }
}
```

Output:



Constructor and Destructor:

Constructors:

Constructors are special methods in C# that are automatically called when an object of a class is created to initialize all the class data members. If there are no explicitly defined constructors in the class, the compiler creates a default constructor automatically. A constructor name must be the same as a class name. A constructor can be public, private, or protected. The constructor cannot return any value so cannot have a return type. A class can have multiple constructors with different parameters but can only have one parameter less constructor. A constructor is never inherited or overridden as it's not an instance method.

Syntax to constructor:

```
className (parameter-list){

 code-statements

}
```

Different types of constructors:

- No-argument constructor.
- Constructor with access modifiers
- Default no-argument constructor (supplied by the compiler).
- Parameterized constructor.

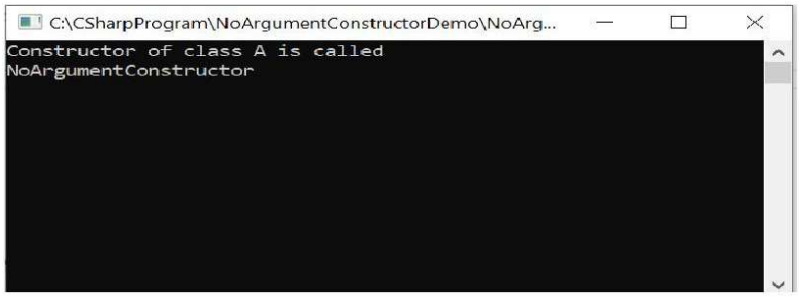
No-argument constructor:

To create an object of a class, we can create a constructor that accepts no arguments/parameters and because this constructor accepts no argument, such constructor is called a no-argument constructor.

Example of No-argument constructor:

```
using System;

namespace NoArgumentConstructorDemo
{
   public class NoArgumentConstructor
   {
      public NoArgumentConstructor()
      {
         Console.WriteLine("Constructor of class A is called");
      }
      public void display()
      {
         Console.WriteLine("NoArgumentConstructor");
         Console.ReadLine();
      }
      static void Main(string[] args)
      {
         NoArgumentConstructor noArgumentConstructor = new NoArgumentConstructor();
         noArgumentConstructor.display();
      }
   }
}
```

Output:



Constructor with access modifiers:

A constructor may have access modifier like - **public**, **protected** and yes, even **private**. The rules of access modifiers that apply to instance variables, methods of a class, apply to a constructor as well.

Example of Constructor with access modifiers:

Output:


```
using System;

namespace ConstructorAccessModifiersDemo
{

   public class ConstructorAccessModifiers
   {
      private ConstructorAccessModifiers()
      {
         Console.WriteLine("Constructor of ConstructorAccessModifiers is called");
         Console.ReadLine();
      }
      static void Main(string[] args)
      {
         ConstructorAccessModifiers constructorAccessModifiers = new ConstructorAccessModifiers();
      }
   }
}
```

Another Example of Constructor with access modifiers:

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

namespace ConstructorAccessModifiersDemo
{
   public class AccessModifiers
   {
      private AccessModifiers()
      {
         Console.WriteLine("Constructor of AccessModifiers is called");
      }
   }
   public class ConstructorAccessModifiers
   {
      static void Main(string[] args)
      {
         AccessModifiers accessModifiers = new AccessModifiers();

      }
   }
}
```

Default no-argument constructor (supplied by the compiler):

C# runtime system i.e. common runtime library (CRL) adds a default no-argument constructor to your class when it is created, only when you don't type any constructor in your class. If in case, you have provided a constructor in your class, then you are not supplied the default no-argument constructor.

Example Default no-argument constructor:

```
using System;

namespace DefaultNoArgumentConstructorDemo
{
   public class DefaultNoArgumentConstructor
   {
      public void display()
      {
         Console.WriteLine("Default no-argument constructor");
         Console.ReadLine();
      }
      static void Main(string[] args)
      {
       DefaultNoArgumentConstructor defaultNoArgumentConstructor = new DefaultNoArgumentConstructor();
         defaultNoArgumentConstructor.display();
      }
   }
}
```

Output:

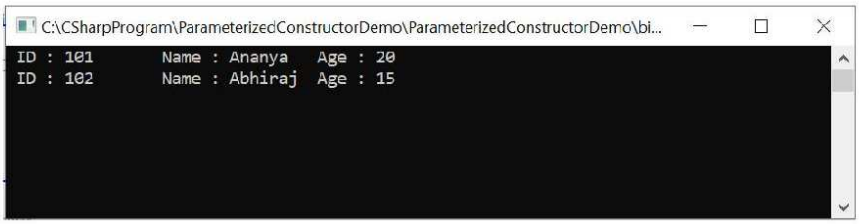
Parameterized constructor:

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

Example of Parameterized constructor:

```
using System;

namespace ParameterizedConstructorDemo
{
   public class Student
   {
      public int id;
      public String name;
      public int age;
      public Student(int i, String n, int a)
      {
         id = i;
         name = n;
         age = a;
      }
      public void display()
      {
         Console.WriteLine(" ID : "+id + "\t"+"Name : " + name + "\t"+"Age : " + age);

      }
   }
   public class ParameterizedConstructor
   {
      static void Main(string[] args)
      {
         Student student = new Student(101, "Ananya", 20);
         Student student1 = new Student(102, "Abhiraj", 15);
         student.display();
         student1.display();
         Console.ReadLine();
      }
   }
}
```

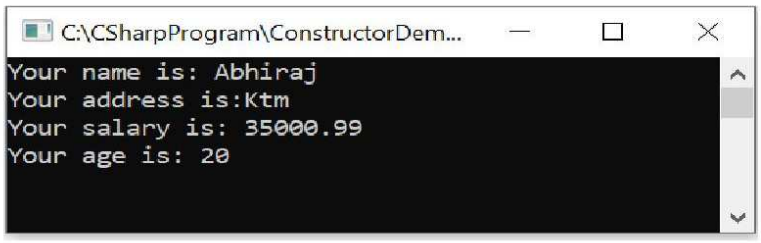
Output:



Example of Constructor:

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

namespace ConstructorDemo
{
   public class Program
   {
      public Program(int age)
      {
         Console.WriteLine("Your age is: "+age);
      }
      public Program(string name, string address)
      {
         Console.WriteLine("Your name is: "+name+"\n"+"Your address is:"+address);
      }
      public Program(double salary)
      {
         Console.WriteLine("Your salary is: " + salary);
      }

      static void Main(string[] args)
      {
         Program obj2 = new Program("Abhiraj","Ktm");
         Program obj3 = new Program(35000.99);
         Program obj1 = new Program(20);
         Console.ReadLine();
      }
   }
}
```

Output:



Destructor:

In C#, destructor (finalizer) is used to destroy objects of class when the scope of an object ends. It has the same name as the class and starts with a tilde ~. A destructor cannot have a return type. A destructor cannot be defined with arguments i.e. it has no-arguments.

Example of Destructor:

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

namespace DestructorDemo
{
   public class Destructor
   {
        int n;
      Destructor(int a)
        {
          n = a;
          Console.WriteLine("Creating an object" + n);
        }
        ~Destructor()
        {
          Console.WriteLine("Destructing an object" + n);
        }
      static void Main(string[] args)
      {
        for(int i = 0; i <= 5; i++)
        {
          Destructor destructor = new Destructor(i);
        }
        Console.ReadLine();
      }
   }
}
```

Output:

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***. This enables data to be accessed easily and helps to promote the flexibility and safety of methods. Encapsulation and hiding of information can also be achieved using properties. It uses pre-defined methods which are "**get**" and "**set**" methods which help to access and modify the properties.

Types of Properties:

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

- **Read and Write Properties:** When property contains both get and set methods.
- **Read-Only Properties:** When property contains only get method.
- **Write Only Properties:** When property contains only set method.
- **Auto Implemented Properties:** When there is no additional logic in the property.

Syntax of Properties:

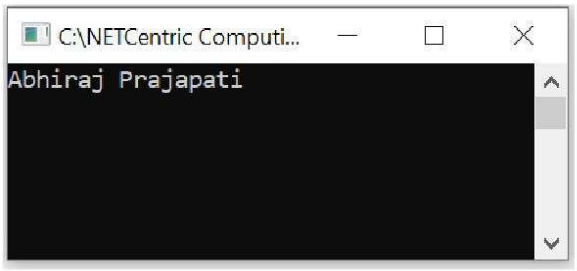
```
<access_modifier> <return_type> <property_name>
{
   get
   {
      // return the property value
   }
   set
   {
      // set a new value
   }
}
```

Where, <access_modifier> can be public, private, protected or internal. <return_type> can be any valid C# type. <property_name> can be user-defined.Properties can be different access modifiers like public, private, protected, internal. Access modifiers define how users of the class can access the property. The get and set accessors for the same property may have different access modifiers. A property may be declared as a **static property** by using the static keyword or may be marked as a **virtual property** by using the virtual keyword.

Example of Read only Properties:

```
using System;
namespace ReadonlyProperties
{
   public class Student
   {
      public string Name
      {
         get
         {
            return "Abhiraj Prajapati";
         }
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         Student st = new Student();
         Console.WriteLine(st.Name);
         Console.ReadLine();
      }
   }
}
```

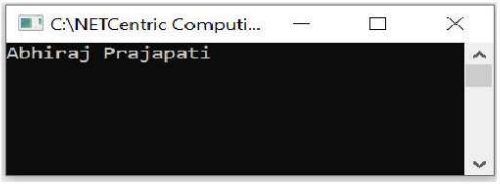
Output:



Example of Read Write only Properties:

```
using System;
namespace ReadWriteonlyProperties
{
   public class Student
   {
      private string name = "";
      public string Name
      {
         get { return name; }
         set { name = value; }
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         Student st = new Student();
         //writting into property
         st.Name = "Abhiraj Prajapati";
         //reading value from property
         Console.WriteLine(st.Name);
         Console.ReadLine();
      }}}
```

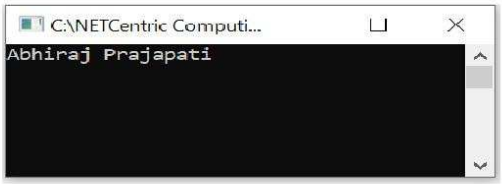
Output:



Example of Write only Properties:

```
using System;
namespace WriteonlyProperties
{
   public class Student
   {
      private string name = "";
      public string Name
      {
         set { name = value; }
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         Student st = new Student();
         //writting into property
         st.Name = "Abhiraj Prajapati";
         Console.ReadLine();
      }}}
```

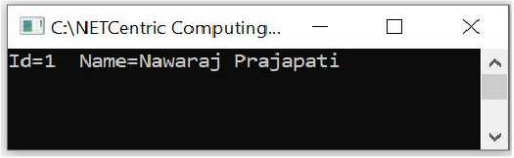
Output:



Example of Automatic Properties:

```
using System;
namespace AutomaticProperties
{
   public class Student
   {
      public int Id { get; set; }
      public string Name { get; set; }
   }
   class Program
   {
      static void Main(string[] args)
      {
         Student st = new Student();
         st.Id = 1;
         st.Name = "Nawaraj Prajapati";
         Console.WriteLine("Id={0}  Name={1}", st.Id, st.Name);
         Console.ReadLine();
      }
   }
}
```

Output:



Arrays and String:

An array is a collection of the same type variable. Whereas a string is a sequence of Unicode characters or array of characters.

An array is used to store multiple values in a single variable. The elements of the array share the same variable name but each element has its own unique index number. The index value of an array starts with '0'. An array can be of any type, For example, int, float, char, etc. The size must be an integer value but not a byte, short, or long. We use the following syntax to create an array.

Syntax of Array Declaration:

datatype[] arrayName;;

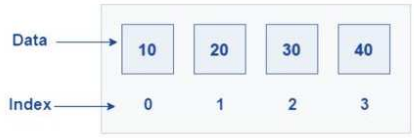
datatype[] variable_name = new datatype[array_size];

Syntax of Array Initialization:

int [] numbers = {10, 20, 30, 40};

string[] name={"Abhiraj","Rohit","Ram","Sita"}

int[] num = new int[4];



For integer:

Row	Column [0]	[1]	[2]
[0]	0	1	2
[1]	1	2	3
[2]	2	3	4

For String:

str[0][0]="Amit";

str[0][1]="Rahul";

str[1][2]="Ram";

str[1][3]="Gopal"

Simple example of array:

```
using System;

namespace SimpleExampleOfArray
{
   internal class Program
   {
      static void Main(string[] args)
      {
         int []a = new int[5];      //declaration and instantiation
         a[0] = 10;                 //initialization
         a[1] = 20;
         a[2] = 70;
         a[3] = 40;
         a[4] = 50;
         for (int i = 0; i < a.Length; i++)     //length is the property of array
            Console.WriteLine(a[i]);
            Console.ReadLine();

      }
   }
}
```

Output:

Advantages of array:

- Helps in optimizing the code.
- Multiple data items of the same type can be stored in a single variable name.
- Accessing an element is very easy by using the index number.
- 2D array are used to represent matrices. Type of array.
- Arrays allocate memory in contiguous memory locations.
- Stacks, queues, trees etc. other data structures can be implemented using array.

Disadvantages of array:

- Number of elements to be stored in an array should be known in advance.
- It has the static structure (fixed size). Once you declared the size of the array can't be modified.
- It stores only the homogeneous type of elements or data type.
- Allocating more memory than the requirement leads to wastage of memory space and less allocation of memory also leads to a problem.
- Insertion and deletion are quite difficult in an array as the elements are stored in consecutive memory location and the shifting operation is costly (time complexity will be increased).

Types of Array:

There are two type of array.

1. Single Dimensional Array
2. Multidimensional array
3. Jagged Arrays

Single Dimensional Array or One Dimensional Array:

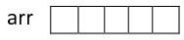
One dimensional array represents one row or one column of array elements with the same label but different index values.

Syntax of One Dimensional Array:

datatype []variable_name = new datatype[size];

Example of One Dimensional Array:

int arr[]=new int[5];



Size of the array = 5

Index of the array = 0 1 2 3 4 5

First index =0

Last index =4

arr[0]=10;

arr[1]=20;

arr[2]=30;

arr[3]=40;

arr[4]=50;

```
using System;
namespace SimpleExampleOfArray
{
   internal class Program
   {
      static void Main(string[] args)
      {
         int []a = new int[5];     //declaration and instantiation
         a[0] = 10;                //initialization
         a[1] = 20;
         a[2] = 70;
         a[3] = 40;
         a[4] = 50;
         for (int i = 0; i < a.Length; i++)    //length is the property of array
            Console.WriteLine(a[i]);
            Console.ReadLine();

      }
   }
}
```

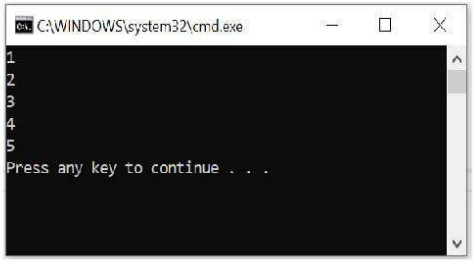
Output:

foreach loop:

To traverse the array elements we can use the foreach loop that returns all the elements of an array one by one. For example

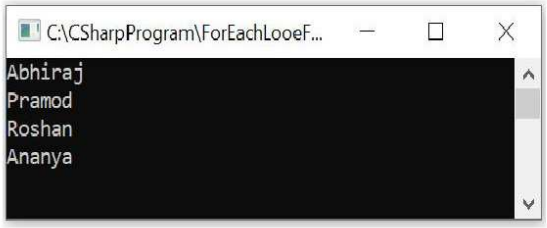
```
using System;

namespace ForeachArrayDemo
{
   internal class Program
   {
      static void Main(string[] args)
      {
         //creating and initializing array
         int[] arr = { 1, 2, 3, 4, 5 };

         //traversing array
         foreach (int i in arr)
         {
            Console.WriteLine(i);

         }
      }
   }
}
```

Output:




```
using System;
namespace ForEachLooeForString
{
   public class Program
   {
      static void Main(string[] args)
      {
         string[] name = { "Abhiraj", "Pramod", "Roshan", "Ananya" };

         foreach (string str in name)
         {
            Console.WriteLine("{0}", str);
         }
         Console.ReadLine();
      }
   }
}
```

Output:



Multidimensional array:

If there are multiple rows and multiple columns, then it is called as Multidimensional array.

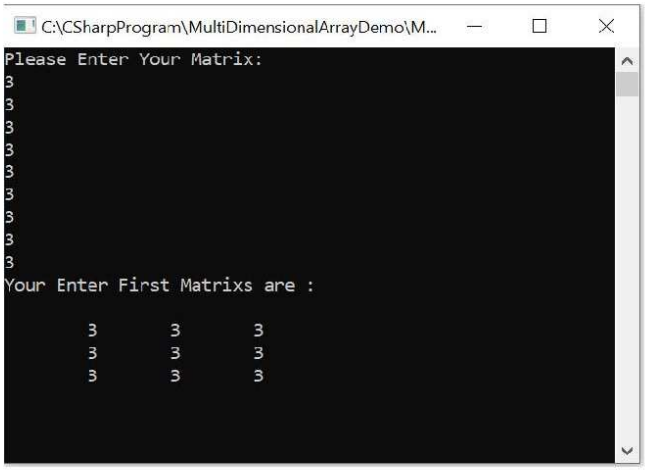
Syntax of Multidimensional array:

If we want to declare an array having 3 rows and 3 columns.

```
int[,]twoDimensionalArray= new int[3,3];
```

```
using System;
namespace MultiDimensionalArrayDemo
{
  internal class Program
  {
    static void Main(string[] args)
    {
      int i, j;
      //Declaring multi dimensional array
       int[,] first = new int[3, 3];
      Console.Write("Please Enter Your Matrix: \n");
      for (i = 0; i < 3; i++)
      {
        for (j = 0; j < 3; j++)
        {
       first[i, j] = int.Parse(Console.ReadLine());
        }
      }
      Console.WriteLine("Your Enter First Matrixs are :\n");
      for (i = 0; i < 3; i++)
      {
        for (j = 0; j < 3; j++)
        {
          Console.Write("\t"+first[i, j]);
        }
        Console.Write("\n");
      }
      Console.ReadLine();
    }
  }
}
```

Output:

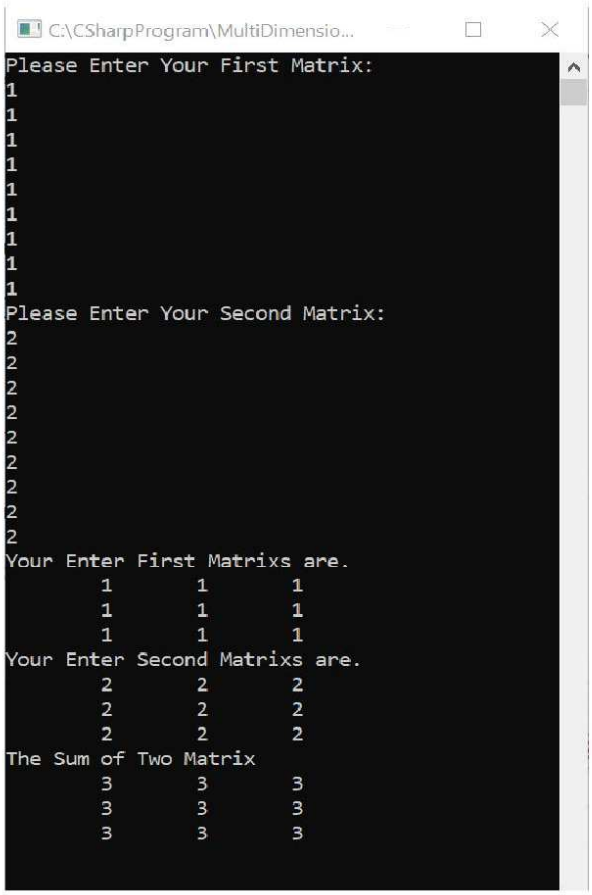


Sum of two matrix:

```
using System;
namespace MultiDimensionalArrayDemo
{
   internal class Program
   {
      static void Main(string[] args)
      {
         //Declaring multi dimensional array
         int[,] first = new int[3, 3];
         int[,] second = new int[3, 3];
         int[,] sum = new int[3, 3];

         Console.Write("Please Enter Your First Matrix: \n");
         for (int i – 0; i < 3; i++)
         {
            for (int j = 0; j < 3; j++)
            {
               first[i, j] = int.Parse(Console.ReadLine());
            }
         }
         Console.Write("Please Enter Your Second Matrix: \n");
         for (int i = 0; i < 3; i++)
         {
            for (int j = 0; j < 3; j++)
            {
               second[i, j] = Convert.ToInt32(Console.ReadLine());
            }
         }
         Console.WriteLine("Your Enter First Matrixs are.");
         for (int i = 0; i < 3; i++)
         {
            for (int j = 0; j < 3; j++)
            {
               Console.Write("\t" + first[i, j]);
            }
            Console.Write("\n");
         }
         Console.WriteLine("Your Enter Second Matrixs are.");
         for (int i = 0; i < 3; i++)
         {
            for (int j = 0; j < 3; j++)
            {
               Console.Write("\t" + second[i, j]);
            }
            Console.Write("\n");
         }

         for (int i = 0; i < 3; i++)
         {
```

```
            for (int j = 0; j < 3; j++)
            {
                sum[i,j] = first[i, j] + second[i, j];
            }

        }
        Console.WriteLine("The Sum of Two Matrix");
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                Console.Write("\t" + sum[i, j]);
            }
            Console.Write("\n");
        }
        Console.ReadLine();
    }
  }
}
```

Output:



Jagged Arrays:

In C#, jagged array is also known as "array of arrays" because its elements are arrays. The element size of jagged array can be different.

A jagged array is initialized with two square brackets [][]. The first bracket specifies the size of an array, and the second bracket specifies the dimensions of the array which is going to be stored.

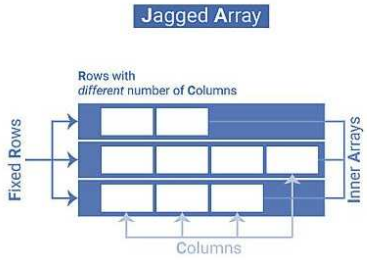
Syntax of Multidimensional array:

dataType[ ][ ] nameOfArray = new dataType[no. of rows][ ];

Initialization of Jagged array

int[][] jArray1 = new int[2][]; // can include two single-dimensional arrays

int[][,] jArray2 = new int[3][,]; // can include three two-dimensional arrays

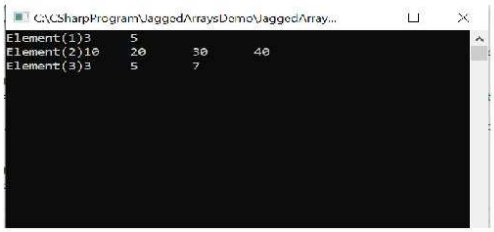



```
// Create an array of 3 int arrays.
 int[][] intArray = new int[3][];
// Create each array that is an element of the jagged
intArray[0] = new int[] { 1, 2 };
intArray[1] = new int[] { 3, 4, 5, 6 };
intArray[2] = new int[] { 7, 8, 9 };
```

Example of Jagged Arrays:

```
using System;
namespace JaggedArraysDemo
{
  internal class Program
  {
    static void Main(string[] args)
    {
      int[][] jaggedArr = new int[3][]; //array declaration

      jaggedArr[0] = new int[2] { 3, 5 }; //jagged array initialization and population
      jaggedArr[1] = new int[4] { 10, 20, 30, 40 };
      jaggedArr[2] = new int[3] { 3, 5, 7 };
      for (int i = 0; i < jaggedArr.Length; i++) // outer for loop to traverse through each elemment of array
      {
        Console.Write("Element({0})", i + 1);
        for (int j = 0; j < jaggedArr[i].Length; j++) // inner for loop to ouput the contents of jagged array
        {
          Console.Write(jaggedArr[i][j] + "\t"); // printing results of jagged array
        }
        Console.WriteLine();
        Console.ReadLine(); }}}}
```

Output:



C# Array Properties:

Property	Description
IsFixedSize	It is used to get a value indicating whether the Array has a fixed size or not.
IsReadOnly	It is used to check that the Array is read-only or not.
IsSynchronized	It is used to check that access to the Array is synchronized or not.
Length	It is used to get the total number of elements in all the dimensions of the Array.
LongLength	It is used to get a 64-bit integer that represents the total number of elements in all the dimensions of the Array.
Rank	It is used to get the rank (number of dimensions) of the Array.
SyncRoot	It is used to get an object that can be used to synchronize access to the Array.

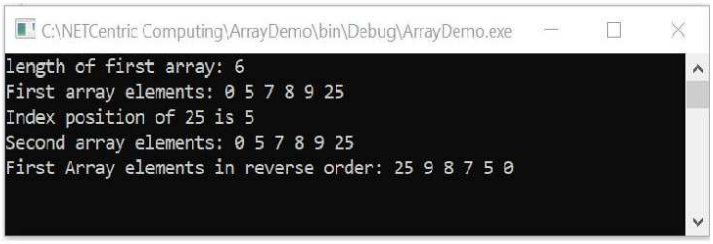
C# Array Methods:

Method	Description
AsReadOnly<T>(T[])	It returns a read-only wrapper for the specified array.
BinarySearch(Array,Int32,Int32,Object)	It is used to search a range of elements in a one-dimensional sorted array for a value.
BinarySearch(Array,Object)	It is used to search an entire one-dimensional sorted array for a specific element.
Clear(Array,Int32,Int32)	It is used to set a range of elements in an array to the default value.
Clone()	It is used to create a shallow copy of the Array.
Copy(Array,Array,Int32)	It is used to copy elements of an array into another array by specifying starting index.
CopyTo(Array,Int32)	It copies all the elements of the current one-dimensional array to the specified one-dimensional array starting at the specified destination array index
CreateInstance(Type,Int32)	It is used to create a one-dimensional Array of the specified Type and length.
Empty<T>()	It is used to return an empty array.
Finalize()	It is used to free resources and perform cleanup operations.
Find<T>(T[],Predicate<T>)	It is used to search for an element that matches the conditions defined by the specified predicate.
IndexOf(Array,Object)	It is used to search for the specified object and returns the index of its first occurrence in a one-dimensional array.
Initialize()	It is used to initialize every element of the value-type Array by calling the default constructor of the value type.
Reverse(Array)	It is used to reverse the sequence of the elements in the entire one-dimensional Array.
Sort(Array)	It is used to sort the elements in an entire one-dimensional Array.
ToString()	It is used to return a string that represents the current object.

Example of Array:

```
namespace ArrayDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         // Creating an array
         int[] arr = new int[6] { 5, 8, 9, 25, 0, 7 };

         // Creating an empty array
         int[] arr2 = new int[6];

         // Displaying length of array
         Console.WriteLine("length of first array: " + arr.Length);

         // Sorting array
         Array.Sort(arr);

         Console.Write("First array elements: ");
         // Displaying sorted array
         PrintArray(arr);

         // Finding index of an array element
         Console.WriteLine("\nIndex position of 25 is " + Array.IndexOf(arr, 25));

         // Coping first array to empty array
         Array.Copy(arr, arr2, arr.Length);
         Console.Write("Second array elements: ");

         // Displaying second array
         PrintArray(arr2);
         Array.Reverse(arr);
         Console.Write("\nFirst Array elements in reverse order: ");
         PrintArray(arr);
         Console.ReadLine();
      }
      // User defined method for iterating array elements
      static void PrintArray(int[] arr)
      {
         foreach (Object elem in arr)
         {
            Console.Write(elem + " ");
         }
      }

   }
}
```

Output:



C# Strings:

In C#, string is an object of **System.String** class that represent sequence of characters. We can perform many operations on strings such as concatenation, comparision, getting substring, search, trim, replacement etc.

C# String methods:

Method Name	Description
Clone()	It is used to return a reference to this instance of String.
Compare(String, String)	It is used to compares two specified String objects. It returns an integer that indicates their relative position in the sort order.
CompareTo(String)	It is used to compare this instance with a specified String object. It indicates whether this instance precedes, follows, or appears in the same position in the sort order as the specified string.
Concat(String, String)	It is used to concatenate two specified instances of String.
Contains(String)	It is used to return a value indicating whether a specified substring occurs within this string.
Copy(String)	It is used to create a new instance of String with the same value as a specified String.
EndsWith(String)	It is used to check that the end of this string instance matches the specified string.
Equals(String, String)	It is used to determine that two specified String objects have the same value.
Format(String, Object)	It is used to replace one or more format items in a specified string with the string representation of a specified object.
IndexOf(String)	It is used to report the zero-based index of the first occurrence of the specified string in this instance.
IsNullOrEmpty(String)	It is used to indicate that the specified string is **null** or an Empty string.
Join(String, String[])	It is used to concatenate all the elements of a string array, using the specified separator between each element.
LastIndexOf(Char)	It is used to report the zero-based index position of the last occurrence of a specified character within String.
PadLeft(Int32)	It is used to return a new string that right-aligns the characters in this instance by padding them with spaces on the left.
PadRight(Int32)	It is used to return a new string that left-aligns the characters in this string by padding them with spaces on the right.

Remove(Int32)	It is used to return a new string in which all the characters in the current instance, beginning at a specified position and continuing through the last position, have been deleted.
Replace(String, String)	It is used to return a new string in which all occurrences of a specified string in the current instance are replaced with another specified string.
Split(Char[])	It is used to split a string into substrings that are based on the characters in an array.
StartsWith(String)	It is used to check whether the beginning of this string instance matches the specified string.
Substring(Int32)	It is used to retrieve a substring from this instance. The substring starts at a specified character position and continues to the end of the string.
ToCharArray()	It is used to copy the characters in this instance to a Unicode character array.
ToLower()	It is used to convert String into lowercase.
ToLowerInvariant()	It is used to return convert String into lowercase using the casing rules of the invariant culture.
ToString()	It is used to return instance of String.
ToUpper()	It is used to convert String into uppercase.
Trim()	It is used to remove all leading and trailing white-space characters from the current String object.

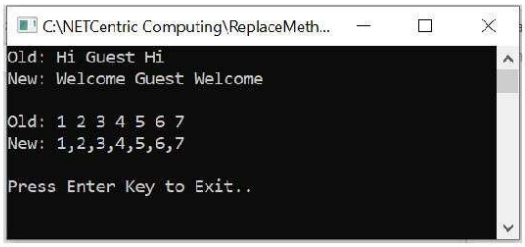
Example of Split Method

```
using System;
namespace Splitmethod
{
   class Program
   {
      static void Main(string[] args)
      {
         string msg = "Nawaraj,Abhiraj,Pooja";
         string[] strarr = msg.Split(',');
         for (int i = 0; i < strarr.Length; i++)
         {
            Console.WriteLine(strarr[i]);
         }
         Console.ReadLine();
      }
   }
}
```

Example of Replace Method:

```
using System;
namespace ReplaceMethod
{
  class Program
  {
    static void Main(string[] args)
    {
      string str1 = "Hi Guest Hi";
      string str2 = str1.Replace("Hi", "Welcome");
      Console.WriteLine("Old: {0}", str1);
      Console.WriteLine("New: {0} \n", str2);

      string str3 = "1 2 3 4 5 6 7";
      string str4 – str3.Replace(" ", ",");
      Console.WriteLine("Old: {0}", str3);
      Console.WriteLine("New: {0}", str4);

      Console.WriteLine("\nPress Enter Key to Exit..");
      Console.ReadLine();
    }
  }
}
```

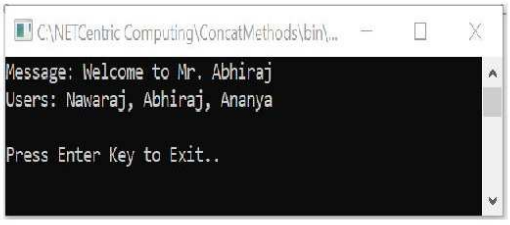
Output:



Example of Concat Method:

```
using System;
namespace ConcatMethods
{
  class Program
  {
    static void Main(string[] args)
    {
      string str1 = "Welcome to";
      string str2 = " " + "Mr. Abhiraj";
      Console.WriteLine("Message: {0}", string.Concat(str1, str2));

      string name1 = "Nawaraj";
      string name2 = ", " + "Abhiraj";
      string name3 = ", " + "Ananya";
      Console.WriteLine("Users: {0}", string.Concat(string.Concat(name1, name2), name3));

      Console.WriteLine("\nPress Enter Key to Exit..");
      Console.ReadLine();

    }
  }
}
```

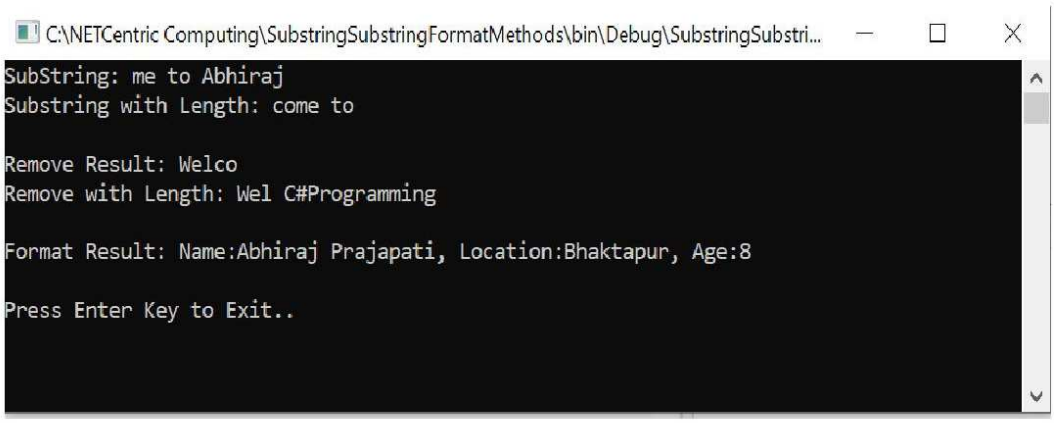
Output:



Substring, Substring and Format Methods:

```
using System;
namespace SubstringSubstringFormatMethods
{
  class Program
  {
    static void Main(string[] args)
    {
      string str1 = "Welcome to Abhiraj";
      Console.WriteLine("SubString: {0}", str1.Substring(5));
      Console.WriteLine("Substring with Length: {0} \n", str1.Substring(3, 7));

      string str2 = "Welcome to C#Programming";
      Console.WriteLine("Remove Result: {0}", str2.Remove(5));
      Console.WriteLine("Remove with Length: {0} \n", str2.Remove(3, 7));

      string s = "Name:{0} {1}, Location:{2}, Age:{3}";
      string msg = string.Format(s, "Abhiraj", "Prajapati", "Bhaktapur", 08);
      Console.WriteLine("Format Result: {0}", msg);
      Console.WriteLine("\nPress Enter Key to Exit..");

      Console.ReadLine();
    }
  }
}
```

Output:



Indexers:

C# indexers are usually known as **smart arrays**. A C# indexer is a class **property** that allows you to access a member variable of a class or struct using the features of an array. In C#, indexers are created using **this** keyword. Indexers in C# are applicable on both classes and structs. Indexer must have at least one parameter else a complier time error will be generated. Indexer define using "**this**" keyword and square[] brackets. **Indexers** are almost similar to the **Properties**.

Syntax of indexers:

```
<access_modifier> <return_type> this[parameters]
{
   get
   {
      // return the value specified by index
   }
   set
   {
      // set a new value
   }
}
```

access_modifier: It can be **public**, **private**, **protected** or **internal**.

return_type: It can be any valid C# data type

this: It is the keyword which points to the object of the current class

parameters: This specifies the parameter list of the indexer

get and **set**: These are the accessors.

Example:

```
using System;
namespace IndexerDemo
{
   class Student
   {
      private int[] id = new int[5];
      public int this[int index]
      {
         set {
            id[index] = value;
         }
         get {
            return id[index];
         }
      }
   }
   public class Program
   {
      static void Main(string[] args)
      {
```

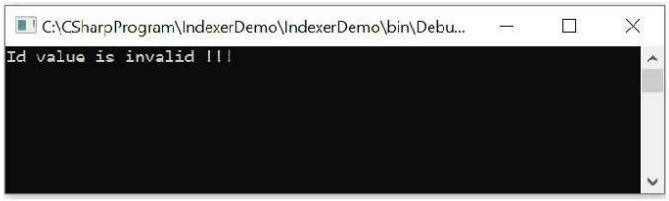
Output:

```
        Student student = new Student();
        student[0] = 101;
        Console.WriteLine(student[0]);
        Console.ReadLine();
} }}
```

Another Example of Indexers:

```
using System;
namespace IndexerDemo
{
  class Student
  {
    private int[] id = new int[5];
    public int this[int index]
    {
      set {
        if(index>=0 && index < id.Length)
        {
        if (value > 0)
        {
          id[index] = value;
        }
        else
        {
          Console.WriteLine("Id value is invalid !!!");
          Console.ReadLine();
        }
        }
        else
        {
          Console.WriteLine("Id is invalid index !!!");
          Console.ReadLine();
        }
      }
      get {
        return id[index];
      }
    }
  }
  public class Program
  {
    static void Main(string[] args)
    {
      Student student = new Student();
      student[3] = -101;
      Console.WriteLine(student[6]);
      Console.ReadLine();
    }
  }
}
```

Output:

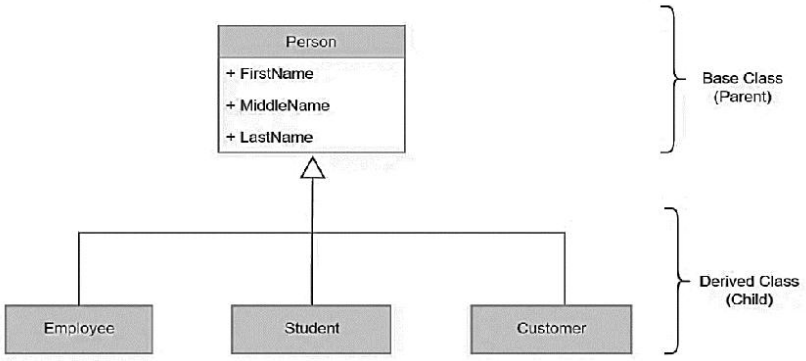


Difference between Indexers and Properties:

Indexers	Properties
Indexers are created with this keyword.	Properties don't require this keyword.
Indexers are identified by signature.	Properties are identified by their names.
Indexers are accessed using indexes.	Properties are accessed by their names.
Indexer are instance member, so can't be static.	Properties can be static as well as instance members.
A get accessor of an indexer has the same formal parameter list as the indexer.	A get accessor of a property has no parameters.
A set accessor of an indexer has the same formal parameter list as the indexer, in addition to the value parameter.	A set accessor of a property contains the implicit value parameter.

Inheritance:

Inheritance is a mechanism in which one class acquires the property of another class. For example, a child inherits the traits of his/her parents. With inheritance, we can **reuse** the fields and methods of the existing class. Hence, inheritance facilitates Reusability and is an important concept of OOPs.



Inheritance

What is the use of inheritance and why it is necessary?

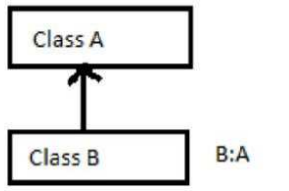
- We can reuse code from the base classes
- We can't access private members of class through inheritance.
- A sub class contains all the features of Super class so we should create the object of sub class.
- Method overriding only possible through inheritance.

Type of inheritance:

1. Single inheritance
2. Multi-level inheritance
3. Multiple inheritance
4. Hierarchical inheritance

Single inheritance:

Single inheritance is nothing but which contain only one Super class and only one Sub class is called single inheritance.



Single or Simple Inheritance

Example of Single Inheritance:

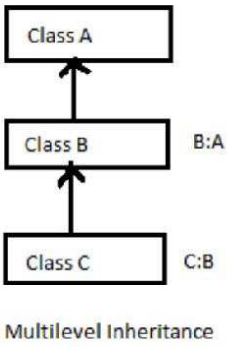
```
using System;

namespace SingleinheritanceExample
{
  public class SuperClass
  {
    public int emp_id;
    public string emp_name;
  }
  public class Program : SuperClass
  {
     void displayMethod()
     {
        emp_id = 101;
        emp_name = "Abhiraj";
        Console.WriteLine("Emp ID : " + emp_id + "\nEmp Name : " + emp_name);
        Console.ReadLine();
     }

     static void Main(string[] args)
     {
        Program program = new Program();
        program.displayMethod();
     }
  }
}
```

Output:

Multi-level inheritance:

In multi-level inheritance we have only one super class and multiple sub classes are called multi-level inheritance.

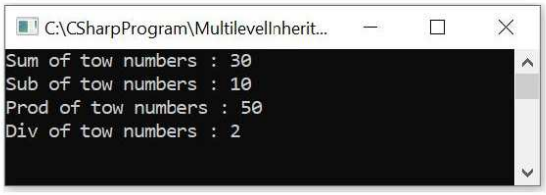
Example of Multi-level Inheritance:



Multilevel Inheritance

```
using System;
namespace MultilevelInheritanceExample
{
   public class SuperClasss
   {
      public int num1, num2, result;
      public void sum()
      {
         num1 = 10;
         num2 = 20;
         result = num1 + num2;
         Console.WriteLine("Sum of tow numbers : " + result);
      }
      public void sub()
      {
         num1 = 30;
         num2 = 20;
         result = num1 - num2;
         Console.WriteLine("Sub of tow numbers : " + result);
      }
   }
   public class SubClass : SuperClasss
   {
      public void prod()
      {
         num1 = 5;
         num2 = 10;
         result = num1 * num2;
         Console.WriteLine("Prod of tow numbers : " + result);
      }
   }
   public class SubClass1 : SubClass
   {
      public void div()
      {
         num1 = 10;
         num2 = 5;
         result = num1 / num2;
         Console.WriteLine("Div of tow numbers : " + result);
      }
   }
   public class Program
   {
      static void Main(string[] args)
      {
         SubClass1 subClass1 = new SubClass1();
         subClass1.sum();
         subClass1.sub();
         subClass1.prod();
         subClass1.div();
         Console.ReadLine();
      }
   }
}
```

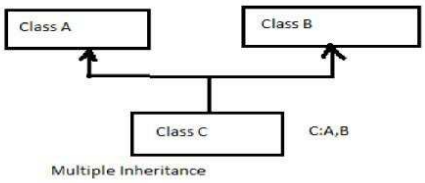
Output:



Multiple inheritance:

C# doesn't support multiple inheritance but we can overcome this problem using interface. Because a sub class wants to inherit the property of two or more super classes that have same method. But if you want to achieve it, then it can be achieved with the help of interfaces only.

Example of Multiple inheritance:



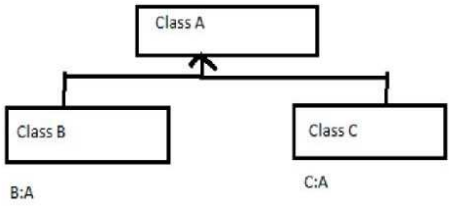
Multiple Inheritance

Hierarchical inheritance:

A hierarchical inheritance which contain only one Super class and multiple Sub class and all Sub class directly extend Super class is called hierarchical inheritance.

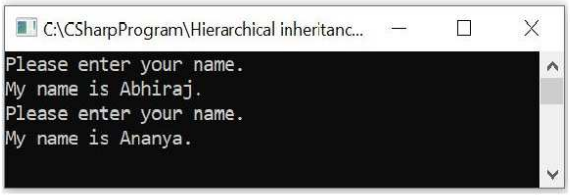
Example of Hierarchical inheritance:

```
using System;
namespace HierarchicalInheritanceExample
{
  public class SuperClass
  {
    public void inputMethod()
    {
      Console.WriteLine("Please enter your name.");
    }
  }
  public class SubClass : SuperClass
  {
    public void showMethod()
    {
      Console.WriteLine("My name is Abhiraj.");
    }
  }
  public class SubClass1 : SuperClass
  {
    public void displayMethod()
    {
      Console.WriteLine("My name is Ananya.");
    }
  }
  public class Program
  {
    static void Main(string[] args)
    {
      SubClass subClass = new SubClass();
      SubClass1 subClass2 = new SubClass1();

      subClass.inputMethod();
      subClass.showMethod();
      subClass2.inputMethod();
      subClass2.displayMethod();

      Console.ReadLine();
    }}}
```



Hierarchial Inheritance

Output:



Polymorphism:

The name should be same, but the behaviour /methodology is different are called polymorphism. For example Girl play a role of daughter at home and a manager at office.

Types of Polymorphism:

There are two types of polymorphism in C#

1. Compile-time Polymorphism
2. Run-time Polymorphism

Compile-time Polymorphism (Method Overloading):

Overloading deals with multiple methods in the same class with the same name but different method signatures.

- All the Methods should have Same Name.
- All the Methods Should be in Same Class.
- But Methods should have Different parameters.

Syntax of Method Overloading:

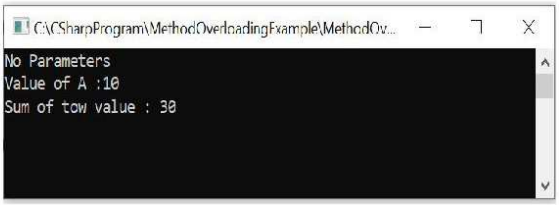
```
Class MethodOverloading{

    void display(int a){
    ……..
    ……..
    }
    void display(int a, int b){
    ……
    ……
    }
```

Both the above methods have the same method names but different method signatures, which means the method are overloading

Example of method overloading:

```
using System;

namespace MethodOverloadingExample
{
   public class MethodOverloading
   {
      void display()
      {
         Console.WriteLine("No Parameters");
      }
      void display(int a)
      {
         Console.WriteLine("Value of A :{0}", a);
      }
      void display(int a, int b)
      {
         Console.WriteLine("Sum of tow value : {0}", (a + b));
      }
      static void Main(string[] args)
      {
         MethodOverloading methodOverloading = new MethodOverloading();

         methodOverloading.display();

         methodOverloading.display(10);

         methodOverloading.display(10, 20);

         Console.ReadLine();
      }
   }
}
```

Output:



Method Overriding:

Overriding deals with method, one in the parent class and other on in the child class and has the same name and signatures.

Syntax of Method Overriding:

```
class MethodOverriding
{
        void display ()
        {
        }
}
class MethodOverridingImp: MethodOverriding
{
        void display ()
        {
        }
}
```

All the Methods should have Same Name.

All the Methods should be in Different Class.

But Methods should have same parameters.

Same Type Of parameters.

Same Numbers Of parameters.

Same Sequence Of parameters.

There should be Inheritance between Classes

Example of method Overriding

```
using System;

namespace MethodOverridingExample
{
  class MethodOverridingDemo
  {
    public void display()
    {
      Console.WriteLine("Parent Class Show Method");
    }
  }
  class MethodOverridingImp : MethodOverridingDemo
  {
    public void Display()
    {
      Console.WriteLine("Child Class Show Method");
    }
  }
  public class Program
  {
    static void Main(string[] args)
    {

      MethodOverridingImp methodOverridingImp = new MethodOverridingImp();
      methodOverridingImp.Display();
      Console.ReadLine();
    }
  }
}
```

Output:

Difference between Method Overloading and Method Overriding

	Method Overloading		Method Overriding
1.	Method overloading means methods with the same name but different signature (number and type of parameters) in the same scope.	1.	Method overriding means methods with the same name and same signature but in a different scope.
2.	It is performed within a class, and inheritance is not involved.	2.	It requires two classes, and inheritance is involved.
3.	The return type may be the same or different.	3.	The return type must be the same.
4.	It is an example of compile-time polymorphism.	4.	It is an example of run-time polymorphism.
5.	Static methods can be overloaded.	5.	Static methods can't be overridden.

Method hiding and overriding

Virtual Methods: By declaring a base class function as virtual, you allow the function to be overridden in any derived class. The idea behind a virtual function is to redefine the implementation of the base class method in the derived class as required. If a method is virtual in the base class then we have to provide the override keyword in the derived class.

Example

```
using System;

namespace VirtualAndOverrideExample
{
   public class BaseClass
   {
      public virtual int display(int a, int b)
      {
         return (a + b);
      }
   }
   public class DriveClass : BaseClass
   {
      public override int display(int a, int b)
      {
         return (a + b) * 2;
      }
   }
   internal class Program
   {
      static void Main(string[] args)
      {
         DriveClass driveClass = new DriveClass();
         Console.WriteLine(driveClass.display(2, 5));
         Console.ReadLine();
      }
   }
}
```

Output:

Hiding Methods:

This method is also known as Method Shadowing. The implementation of the methods of a base class can be hidden from the derived class in method hiding using the new keyword. Or in other words, the base class method can be redefined in the derived class by using the new keyword.

Example

```
using System;

namespace MethodHidingExample
{
   class Base
   {
      public void display()
      {
         Console.WriteLine("Base Class...");
      }
   }

   class Derived : Base
   {
      new public void display()
      {
         Console.WriteLine("Derived Class...");
      }
   }

   class Program
   {
      static void Main(string[] args)
      {
         Derived derived = new Derived();
         derived.display();
         Console.ReadLine();
      }
 }
}
```

Output:

base Keyword:

The base keyword in C# is used to access the members of a base class from within a derived class. It is used when we want to call a method, property, or field of a base class from a derived class. For example invoking constructor, method and variables etc. of base class from derived class in inheritance relationship.

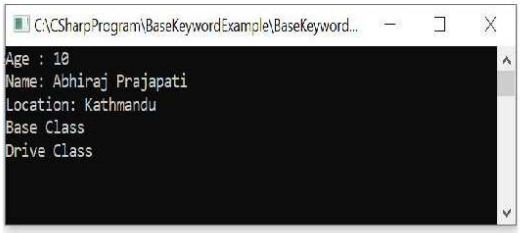
base keyword in c# programs can be used for following concepts.

- Calling base class constructor from derived class.
- Calling immediate parent class / base class method.
- Initialize a base class variable from derived class.

Example of Base Keyword:

```
using System;
namespace BaseKeywordExample
{
   public class BaseKeywordClass
   {
      public BaseKeywordClass(int age)
      {
         Console.WriteLine("Age : "+age);
      }
      public string name = "Abhiraj Prajapati";
      public string location = "Kathmandu";
      public void displayDetails()
      {
         Console.WriteLine("Name: {0}", name);
         Console.WriteLine("Location: {0}", location);
         Console.WriteLine("Base Class");
      }
   }
   public class DriveKeywordClass : BaseKeywordClass
   {
      public DriveKeywordClass(): base(10)
      {

      }
      public new void displayDetails()
      {
         base.displayDetails();
         Console.WriteLine("Drive Class");
      }
   }
   public class Program
   {
      static void Main(string[] args)
      {
         DriveKeywordClass driverKeywordClass = new DriveKeywordClass();
         driverKeywordClass.displayDetails();
         Console.ReadLine();
      }}}
```

Output:



Structs and Enums:

- A **struct** type is a value type.
- They are basically for light-weighted objects.
- Unlike class, **structs** in C# are value type than a reference type.
- It is almost similar to a class because both are user-defined data types and both hold a group of different data types.
- The structure can also include constants, fields, methods, properties, indexers, events, and etc.
- It cannot contain default constructors.

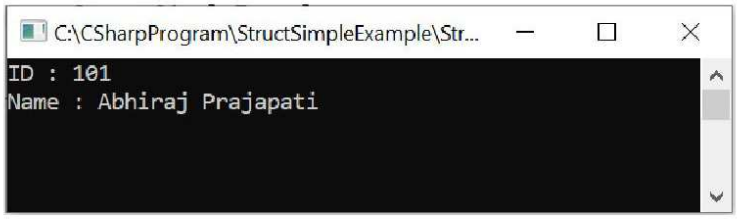
Syntax of struct:

```
struct StructureName {
    // fields
    //properties (constants/fields)
    //methods  etc
}
```

Example of struct:

```
using System;
namespace StructSimpleExample
{
  struct SimpleStructDemo
  {
    public int id;
    public string name;
  }
  public class Program
  {
    static void Main(string[] args)
    {
      SimpleStructDemo demo = new SimpleStructDemo();
      demo.id = 101;
      demo.name = "Abhiraj Prajapati";
      Console.WriteLine("ID : {0}", demo.id);
      Console.WriteLine("Name : {0}", demo.name);
      Console.ReadLine();
    }
  }
}
```

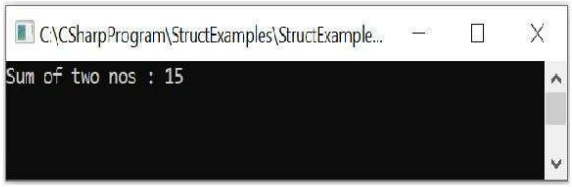
Output:



Example of struct Using Constructor and Method:

```
using System;
namespace StructExamples
{
   struct SumOfStructClass
   {
      // Fields
      public int x;
      public int y;

      // Constructor
      public SumOfStructClass(int x, int y)
      {
         this.x = x;
         this.y – y;
      }
      // Method
      public void display()
      {
         int sum;
         sum = x + y;
         Console.WriteLine("Sum of two nos : "+sum);
      }
   }
   internal class Program
   {
      static void Main(string[] args)
      {
         SumOfStructClass sumOfStructClassum = new SumOfStructClass(5,10);
         sumOfStructClassum.display();
         Console.ReadLine();
      }
   }
}
```

Output:



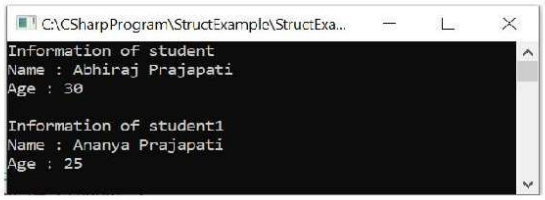
Difference between structure and classes:

	structure		classes
1.	The structure is a value type of data type.	1.	Class is a reference type data type.
2.	The structure can be defined using the 'struct' keyword.	2.	The class can be defined using the 'class' keyword.
3.	The structure variable is stored in stack	3.	The object of the class is stored in heap.
4	Used in the small program	4.	commonly used in large programs
5.	Struct has limited functionality.	5.	The class has unlimited functionality

Example of struct Using Constructor and Method:

```
using System;

namespace StructExample
{
   public struct Student
    {
      // Struct fields to store information about the Student.
      public string Name;
      public int Age;

     // Parameterized constructor for the Student struct.
      public Student(string name, int age)
      {
         this.Name – name;
         this.Age = age;
      }

      // Method to display information about the Student.
      public void displayMethod()
      {
         Console.WriteLine("Name : "+Name);
         Console.WriteLine("Age : "+Age);
      }
   }
   class Program
   {
      static void Main()
      {
         // Create an instance of the 'Student' struct and initialize its values.
         Student student = new Student("Abhiraj Prajapati", 30);
         Console.WriteLine("Information of student");
         student.displayMethod();
         // Create another instance of the 'Student' struct using the parameterized constructor.
         Student student1 = new Student("Ananya Prajapati", 25);
         Console.WriteLine("\nInformation of student1");
         student1.displayMethod();
         Console.ReadLine();
      }
   }
}
```

Output:



Enums:

The *enum* is a keyword that is used to declare an enumeration. Enumeration is a type that consists of a set of named constant called enumerators. By default the first enumerator has the value of "0" and the value of each successive enumerator is increased by one.
Like:
enum days {Mon, Tue, …}

It is not necessary to follow the general index number to be given to the enums. We can provide different values to the enums too.

Enum Day { Monday, Tuesday = 4, Wednesday…..}

In the above example "Monday" will have index value as 0 whereas "Tuesday" will have index value as 4 and then everything following this will have one value increased index value, which means "Wednesday" will have value 5 for index eventually.

Note: Now, if the data member of the enum member has not been initialized, then its value is set according to rules.

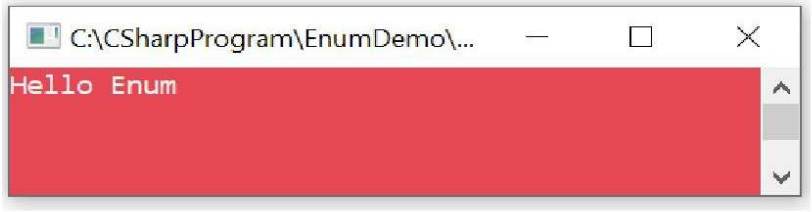
Syntax of enum:

```
enum NameOfEnum
{
    // The enumerator list
};
```

Example of enum:

```
using System;
namespace EnumDemo
{
  internal class Program
  {
    static void Main(string[] args)
    {
      Console.BackgroundColor = ConsoleColor.Red;
      Console.ForegroundColor = ConsoleColor.Black;
      Console.WriteLine("Hello Enum");
      Console.ReadLine();
    }
  }
}
```

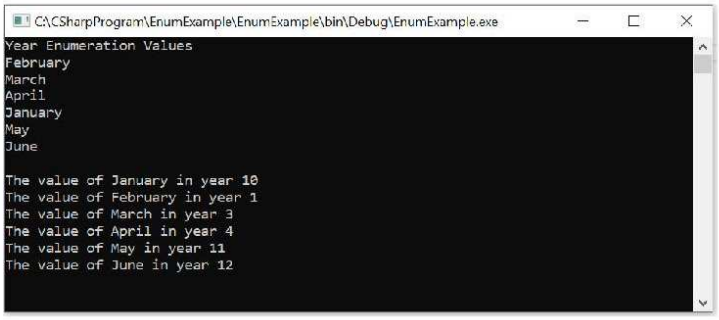
Output:



Example of enum:

```
using System;
namespace EnumExample
{
  enum year
  {
    January=10,
    February=1,
    March=3,
    April,
    May=11,
    June=12,
  }
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Year Enumeration Values");
      foreach (string strYear in Enum.GetNames(typeof(year)))
      {
        Console.WriteLine(strYear);
      }
      Console.WriteLine();
      Console.WriteLine("The value of January in year " + (int)year.January);
      Console.WriteLine("The value of February in year "+ (int)year.February);
      Console.WriteLine("The value of March in year " + (int)year.March);
      Console.WriteLine("The value of April in year " + (int)year.April);
      Console.WriteLine("The value of May in year " + (int)year.May);
      Console.WriteLine("The value of June in year " + (int)year.June);
      Console.ReadLine();
    }
  }
}
```

Output:



Difference between struct and enum:

	Struct	Enum
1	The "struct" keyword is used to declare a structure	The "enum" keyword is used to declare enum.
2	The structure is a user-defined data type that is a collection of dissimilar data types.	Enum is to define a collection of options available.
3	A struct can contain both data variables and methods.	Enum can only contain data types.
4	A struct supports a private but not protected access specifier.	Enum does not have private and protected access specifier.
5	The struct cannot be inherited.	Enum also does not support Inheritance.
6	Structure supports encapsulation.	Enum doesn't support encapsulation.
7	When the structure is declared, the values of its objects can be modified.	Once the enum is declared, its value cannot be changed, otherwise, the compiler will throw an error.
8	Struct only contains parameterized constructors and no destructors.	Enum does not contain constructors and destructors.
9	The values allocated to the structure are stored in stack memory.	The memory to enum data types is allocated in the stack.
10	Value Type	Reference Type

Abstract class Sealed class:

Abstract Classes:
An abstract class which contains the abstract keyword. An abstract class must be extended/ Sub class. A class having abstract method (i.e. not having a definition/ body) is termed. Abstract class and they can't be instantiated or simply we can't create object of this class.

Syntax of abstract class:

```
abstract class <ClassName>{
   ...
}
```

Abstract method:

A method which is declared abstract and has no body is called abstract method. It can be declared inside the abstract class only. Its implementation must be provided by derived classes (inherited from).

Syntax of Abstract method:

```
abstract return_type function_name ();
```

Example of Abstract class:

```
using AbstractExample;

namespace AbstractExample
{
   abstract class AbstractDemo
   {
      public abstract void add();
      public abstract void sub(int a, int b);
   }

    class AbstractImpl : AbstractDemo
    {
      static void Main(string[] args)
      {
         AbstractImpl abstractImpl = new AbstractImpl();
         abstractImpl.add();
         abstractImpl.sub(20, 10);
         Console.ReadLine();
      }

      public override void add()
      {
         Console.WriteLine(" Sum of (10 + 20) = " + (10 + 20));
      }

      public override void sub(int a, int b)
      {
         Console.WriteLine(" Sub of (20 -10) = " + (a - b));
      }
   }
}
```

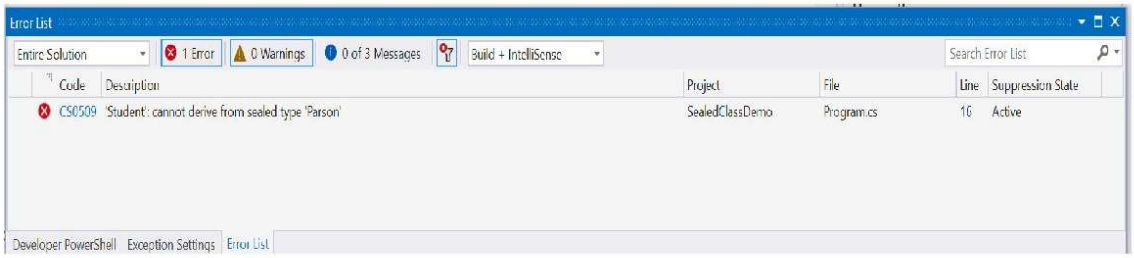
Output:

Sealed Class:

Sealed classes are used to restrict the inheritance feature of object oriented programming. Once a class is defined as a **sealed class,** this class cannot be inherited.
In C#, the sealed modifier is used to declare a class as **sealed**. In Visual Basic .NET, **Not Inheritable** keyword serves the purpose of sealed. If a class is derived from a sealed class, compiler throws an error.

Example of sealed class:

```
using System;
namespace SealedClassDemo
{
   sealed public class Parson
   {
      public void display()
      {
         Console.WriteLine("This is Sealed Class");
      }
   }
   public class SealedClass : Parson
   {
      public void print()
      {
         Console.WriteLine("This is Class");
      }
   }
   internal class Program
   {
      static void Main(string[] args)
      {
         SealedClass obj=new SealedClass();
         obj.display();
         Console.ReadLine();

      }
   }
}
```



Interfaces:

In C#, an interface is similar to abstract class. All the methods of interface should be body empty. The purpose of interface, we can use the multiple inheritance. Otherwise we can't use the multiple inheritance in C#.

Syntax of Interface:

```
interface <interface_name>{
        // method without body
        void display();
    }
```

Notes on Interfaces:

- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default abstract and public
- Interface attributes are by default public, static and final
- An interface cannot contain a constructor (as it cannot be used to create objects)

Example of Interface:

```
using System;
namespace InterfaceDemos
{
  interface InterfaceExample
  {
     void accept();
     void display();
  }
   class InterfaceImp : InterfaceExample
   {
     int rollNo;
     string name;
     string coures;
     public void accept()
     {
        Console.WriteLine("Student Information.....");
        rollNo = 101;
        name = "Abhiraj";
        coures = "BCA"; }
     public void display()
     {
        Console.WriteLine("RollNo : "+ rollNo+"\t"+"Name : "+ name+"\t"+"Coures : "+ coures);
     }}
  public class InterfaceDemo
   {
     static void Main(string[] args)
{
        InterfaceImp interfaceImp = new InterfaceImp();
        interfaceImp.accept();
        interfaceImp.display();
```

Output:

```
        Console.ReadLine();
         }
         }
}
```

Difference between Abstract Class and Interface

ABSTRACT CLASS	INTERFACE CLASS
It contains both declaration and definition part.	It contains only a declaration part.
Multiple inheritance is not achieved by abstract class.	Multiple inheritance is achieved by interface.
It contain constructor.	It does not contain constructor.
It can contain static members.	It does not contain static members.
It can contain different types of access modifiers like public, private, protected etc.	It only contains public access modifier because everything in the interface is public.
The performance of an abstract class is fast.	The performance of interface is slow because it requires time to search actual method in the corresponding class.
It is used to implement the core identity of class.	It is used to implement peripheral abilities of class.
A class can only use one abstract class.	A class can use multiple interface.
If many implementations are of the same kind and use common behavior, then it is superior to use abstract class.	If many implementations only share methods, then it is superior to use Interface.
Abstract class can contain methods, fields, constants, etc.	Interface can only contain methods .
It can be fully, partially or not implemented.	It should be fully implemented.

Delegates and Events:

Delegates:

In C# delegates are used as a function pointer to refer a method. It is specifically an object that refers to a method that is assigned to it. The same delegate can be used to refer different methods, as it is capable of holding the reference of different methods but, one at a time. Which method will be invoked by the delegate is determined at the runtime. The syntax of declaring a delegate is as follow:

```
delegate return_type delegate_name(parameter_list);
```

There are three steps involved while working with delegates:

1) Declare a delegate
   [<modifier>] delegate void|type(return_type) <delegate_name>([<parameter list>]);
2) Create an instance and reference a method
   <delegate_name> object_name = new <delegate_name>(<method_name>);
3) Invoke a delegate
   <delegate_object>.Invoke(<parameters>);

Events:
Events are the action performed which changes the state of an object. Events are declared using delegates, without the presence of delegates you cannot declare events. You can say that an event provides encapsulation to the delegates.

```
modifier event delegate_name MyEvent;
```

Defining an event a two steps.

1. First you need to define a delegate type that will hold the list of methods to be called when the events is fired.
2. Next you declare an event using the event keyword.

```
modifier event delegate_name MyEvent;
```

Two types of Delegates are:

1. Single Cast Delegates(Point a single Method)
2. Multicast Delegates(Point Multiple Method)

Single Cast Delegates:

```
using System;
namespace DelegatesOfSingleCast
{
   public delegate int SumOperation(int x, int y);
   internal class Program
   {
      static int Addition(int a, int b)
      {
         return a + b;
      }
      static void Main(string[] args)
      {
         // Delegate instantiation
         SumOperation obj – new SumOperation(Program.Addition);
         // output
         Console.WriteLine(obj(23, 27));
         Console.ReadLine();
      }
   }
}
```

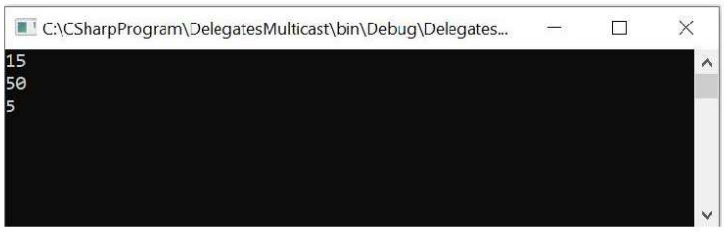
Delegate

- It can be declared using the 'delegate' keyword.
- It is a function pointer.
- It holds the reference to one or more methods during runtime.
- It is an independent keyword.
- It doesn't depend on events.
- It contains the Combine() and Remove() methods that help add methods to the list of invocation.
- It can be passed as a parameter to a method.
- The '=' operator can be used to assign a single method.
- The '+=' operator can be used to assign multiple methods to a delegate.

Multicast Delegates:

```
using System;
namespace DelegatesMulticast
{
   public delegate void operation(int x, int y);
   internal class Program
   {
            static void Main(string[] args)
            {
               operation obj = Calculation.Add;
               obj += Calculation.Multiple;
               obj += Calculation.Sub;
               obj(10, 5);
               Console.ReadLine();
            }
   }
   public class Calculation
   {
            public static void Add(int a, int b)
            {
               Console.WriteLine(a + b);
            }
            public static void Sub(int a, int b)
            {
               Console.WriteLine(a - b);
            }
            public static void Multiple(int a, int b)
            {
               Console.WriteLine(a * b);
            }
   }

}
```

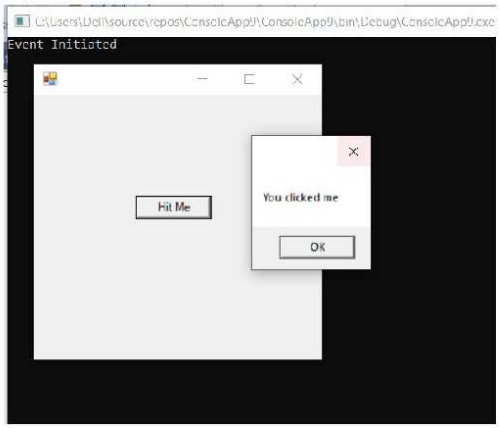
Output:



Event example using delegates

```
namespace ConsoleApp9
{
   public delegate void DelEventHandler();
   internal class Program:Form
   {
      public event DelEventHandler add;
      public Program()
      {
         // desing a button over form
         Button btn = new Button();
         btn.Parent = this;
         btn.Text = "Hit Me";
         btn.Location = new Point(100, 100);

         btn.Click += new EventHandler(onClcik);
         add += new DelEventHandler(Initiate);
         //invoke the event
         add();
      }

      private void Initiate()
      {
        Console.WriteLine("Event Initiated");
      }

      private void onClcik(object sender, EventArgs e)
      {
         MessageBox.Show("You clicked me");
      }

      static void Main(string[] args)
      {
         Application.Run(new Program());
         Console.ReadLine();

      }
   }
}
```

Output:



Partial class:

A partial class splits the definition of a class over two or more source (.cs) files. You can create a class definition in multiple physical files but it will be compiled as one class when the classes are compiled.

Suppose you have a "Person" class. That definition is divided into two physical source files, Person1.cs and Person2.cs. Then these two files have a class that is a partial class. You compile the source code and then create it in a single class.

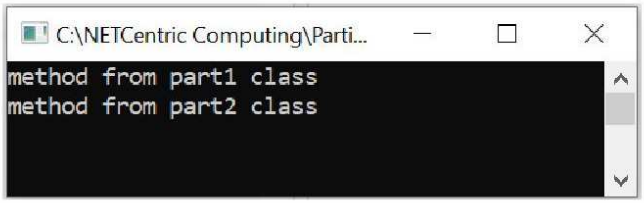
Partial Class Syntax:

```
//File1.cs
public partial class Products
{
   // Implementation Here
}

//File2.cs
public partial class Products
{
   // Implementation Here
}
```

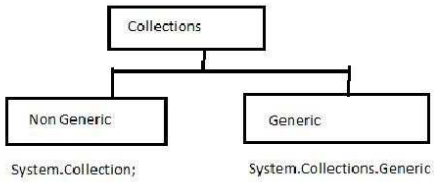
Example:

```
using System;
namespace PartialClassDemo
{
 public partial class Products
  {
    public void method1()
    {
      Console.WriteLine("method from part1 class");
    }
  }
  public partial class Products
  {
    public void method2()
    {
      Console.WriteLine("method from part2 class");
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      Products obj = new Products();
      obj.method1();
      obj.method2();
      Console.ReadLine();
    }
  }}
```

Output:



Collection:
Collections standardize the way of which the objects are handled by your program. In other words, it contains a set of classes to contain elements in a generalized manner. With the help of collections, the user can perform several operations on objects like the store, update, delete, retrieve, search, sort etc.



Generic collection in C# is defined in System.Collection.Generic namespace. It provides a generic implementation of standard data structure like linked lists, stacks, queues, and dictionaries. These collections are type-safe because they are generic means only those items that are type-compatible with the type of the collection can be stored in a generic collection, it eliminates accidental type mismatches. Generic collections are defined by the set of interfaces and classes. Below table contains the frequently used classes of the System.Collections.Generic namespace:

CLASS NAME	DESCRIPTION
Dictionary<TKey,TValue>	It stores key/value pairs and provides functionality similar to that found in the non-generic Hashtable class.
List<T>	It is a dynamic array that provides functionality similar to that found in the non-generic ArrayList class.
Queue<T>	A first-in, first-out list and provides functionality similar to that found in the non-generic Queue class.
SortedList<TKey,TValue>	It is a sorted list of key/value pairs and provides functionality similar to that found in the non-generic SortedList class.
Stack<T>	It is a first-in, last-out list and provides functionality similar to that found in the non-generic Stack class.
HashSet<T>	It is an unordered collection of the unique elements. It prevent duplicates from being inserted in the collection.
LinkedList<T>	It allows fast inserting and removing of elements. It implements a classic linked list.

Non-Generic collection in C# is defined in System.Collections namespace. It is a general-purpose data structure that works on object references, so it can handle any type of object, but not in a safe-type manner. Non-generic collections are defined by the set of interfaces and classes. Below table contains the frequently used classes of the System.Collections namespace:

CLASS NAME	DESCRIPTION
ArrayList	It is a dynamic array means the size of the array is not fixed, it can increase and decrease at runtime.
Hashtable	It represents a collection of key-and-value pairs that are organized based on the hash code of the key.
Queue	It represents a first-in, first out collection of objects. It is used when you need a first-in, first-out access of items.
Stack	It is a linear data structure. It follows LIFO(Last In, First Out) pattern for Input/output.

Generic Collections:

Generic Collections work on the specific type that is specified in the program whereas non-generic collections work on the object type.

- Specific type
- Array Size is not fixed
- Elements can be added / removed at runtime.

List Example:

```
class Program
  {
    static void Main(string[] args)
    {
      List<int> lst = new List<int>();
      lst.Add(1);
      lst.Add(2);
      lst.Add(3);
      foreach (int item in lst)
      {
        Console.WriteLine(item);
      }
      Console.ReadLine();
    }
  }
```

Dictionary Example:(store value in key-value pair)

```
class Program
  {
    static void Main(string[] args)
    {
      Dictionary<int, string> dct = new Dictionary<int, string>();
      dct.Add(1, "cs.net");
      dct.Add(2, "vb.net");
      dct.Add(3, "vb.net");
      dct.Add(4, "vb.net");
      foreach (KeyValuePair<int, string> kvp in dct)
      {
        Console.WriteLine(kvp.Key + " " + kvp.Value);

      }
      Console.ReadLine();
    }
  }
```

SortedList Example(similar to Dictonary)

```
class Program
  {
    static void Main(string[] args)
    {
      SortedList<string, string> sl = new SortedList<string, string>();
      sl.Add("ora", "oracle");
      sl.Add("vb", "vb.net");
      sl.Add("cs", "cs.net");
      sl.Add("asp", "asp.net");

      foreach (KeyValuePair<string, string> kvp in sl)
      {
        Console.WriteLine(kvp.Key + " " + kvp.Value);
      }
      Console.ReadLine();
    }
  }
```

Stack Example:

```
class Program
  {
    static void Main(string[] args)
    {
      Stack<string> stk = new Stack<string>();
      stk.Push("cs.net");
      stk.Push("vb.net");
      stk.Push("asp.net");
      stk.Push("sqlserver");

      foreach (string s in stk)
      {
        Console.WriteLine(s);
      }
      Console.ReadLine();
    }
  }
```

Queue Example:

```
class Program
   {
      static void Main(string[] args)
      {
         Queue<string> q = new Queue<string>();

         q.Enqueue("cs.net");
         q.Enqueue("vb.net");
         q.Enqueue("asp.net");
         q.Enqueue("sqlserver");

         foreach (string s in q)
         {
            Console.WriteLine(s);
         }
         Console.ReadLine();
      }
   }
```

Non Generic Collection Examples:

ArrayList Example:

```
using System.Collections;

namespace ConsoleApp5
{
   class Program
   {
      static void Main(string[] args)
      {
         ArrayList al = new ArrayList();
         string str = "Nawaraj Prajapati";
         int x = 7;
         DateTime d = DateTime.Parse("8-oct-1985");
         al.Add(str);
         al.Add(x);
         al.Add(d);

         foreach (object o in al)
         {
            Console.WriteLine(o);
         }
         Console.ReadLine();
      }
   }
}
```

Hashtable Example

```
class Program
{
   static void Main(string[] args)
   {
      Hashtable ht = new Hashtable();
      ht.Add("ora", "oracle");
      ht.Add("vb", "vb.net");
      ht.Add("cs", "cs.net");
      ht.Add("asp", "asp.net");

      foreach (DictionaryEntry d in ht)
      {
         Console.WriteLine(d.Key + " " + d.Value);
      }
      Console.ReadLine();
   }
}
```

Sorted List Example:

```
class Program
   {
      static void Main(string[] args)
      {
         SortedList sl = new SortedList();
         sl.Add("ora", "oracle");
         sl.Add("vb", "vb.net");
         sl.Add("cs", "cs.net");
         sl.Add("asp", "asp.net");

         foreach (DictionaryEntry d in sl)
         {
           Console.WriteLine(d.Key + " " + d.Value);

         }
         Console.ReadLine();
      }
   }
```

Stack

```
class Program
  {
    static void Main(string[] args)
    {
      Stack stk = new Stack();
      stk.Push("cs.net");
      stk.Push("vb.net");
      stk.Push("asp.net");
      stk.Push("sqlserver");

      foreach (object o in stk)
      {
        Console.WriteLine(o);
      }
      Console.ReadLine();
    }
  }
```

Queue Example:

```
class Program
  {
    static void Main(string[] args)
    {
      Queue q = new Queue();
      q.Enqueue("cs.net");
      q.Enqueue("vb.net");
      q.Enqueue("asp.net");
      q.Enqueue("sqlserver");

      foreach (object o in q)
      {
        Console.WriteLine(o);
      }
      Console.ReadLine();
    }
  }
```

File IO

Generally, the file is used to store the data. The term File Handling refers to the various operations like creating the file, reading from the file, writing to the file, appending the file, etc. There are two basic operation which is mostly used in file handling is reading and writing of the file. The file becomes stream when we open the file for writing and reading. A stream is a sequence of bytes which is used for communication. Two stream can be formed from file one is input stream which is used to read the file and another is output stream is used to write in the file. In C#, System.IO namespace contains classes which handle input and output streams and provide information about file and directory structure.

Class Types	Description
Directory/ DirectoryInfo	These classes support the manipulation of the system directory structure.
DriveInfo	<p" style="outline: 0px;">This class provides detailed information regarding the drives that a given machine has. </p">
FileStream	This gets you random file access with data represented as a stream of bytes.
File/FileInfo	These sets of classes manipulate a computer's files.
Path	It performs operations on System.String types that contain file or directory path information in a platform-neutral manner.
BinaryReader/ BinaryWriter	These classes allow you to store and retrieve primitive data types as binary values.
StreamReader/StreamWriter	Used to store textual information to a file.
StringReader/StringWriter	These classes also work with textual information. However, the underlying storage is a string buffer rather than a physical file.
BufferedStream	This class provides temp storage for a stream of bytes that you can commit to storage at a later time.

Example:(using System.IO)

```
Using System.IO
class Program
   {
      static void Main(string[] args)
      {
         DriveInfo[] di = DriveInfo.GetDrives();

         Console.WriteLine("Total Partitions");
         Console.WriteLine("----------------------");
         foreach (DriveInfo items in di)
         {
            Console.WriteLine(items.Name);
         }
         Console.Write("\nEnter the Partition::");
         string ch = Console.ReadLine();

         DriveInfo dInfo = new DriveInfo(ch);

         Console.WriteLine("\n");
```

```
            Console.WriteLine("Drive Name::{0}", dInfo.Name);
            Console.WriteLine("Total Space::{0}", dInfo.TotalSize);
            Console.WriteLine("Free Space::{0}", dInfo.TotalFreeSpace);
            Console.WriteLine("Drive Format::{0}", dInfo.DriveFormat);
            Console.WriteLine("Volume Label::{0}", dInfo.VolumeLabel);
            Console.WriteLine("Drive Type::{0}", dInfo.DriveType);
            Console.WriteLine("Root dir::{0}", dInfo.RootDirectory);
            Console.WriteLine("Ready::{0}", dInfo.IsReady);
            Console.ReadLine();
        }
    }
```

Example Create,Copy,Delete,Read,Write File using System.IO

Creating File:

```
class Program
    {
        static void Main(string[] args)
        {
            string fileLoc = @"C:\filename.txt";
            FileStream fs = null;
            if (!File.Exists(fileLoc))
            {
                using (fs = File.Create(fileLoc))
                {

                }
            }
            Console.WriteLine("File Created");
            Console.ReadLine();
        }
    }
```

Writing in File

```
class Program
    {
        static void Main(string[] args)
        {
            string fileLoc = @"C:\filename.txt";
            if (File.Exists(fileLoc))
            {
                using (StreamWriter sw = new StreamWriter(fileLoc))
                {
                    sw.Write("Some sample text for the file");
                }
            }
            Console.WriteLine("Write is Success");
            Console.ReadLine();
        }
    }
```

Reading From File:

```
class Program
   {
      static void Main(string[] args)
      {
         string fileLoc = @"C:\filename.txt";
         if (File.Exists(fileLoc))
         {
            using (TextReader tr = new StreamReader(fileLoc))
            {
               Console.WriteLine(tr.ReadLine());
            }
         }
         Console.ReadLine();
      }
   }
```

Copy a Text File:

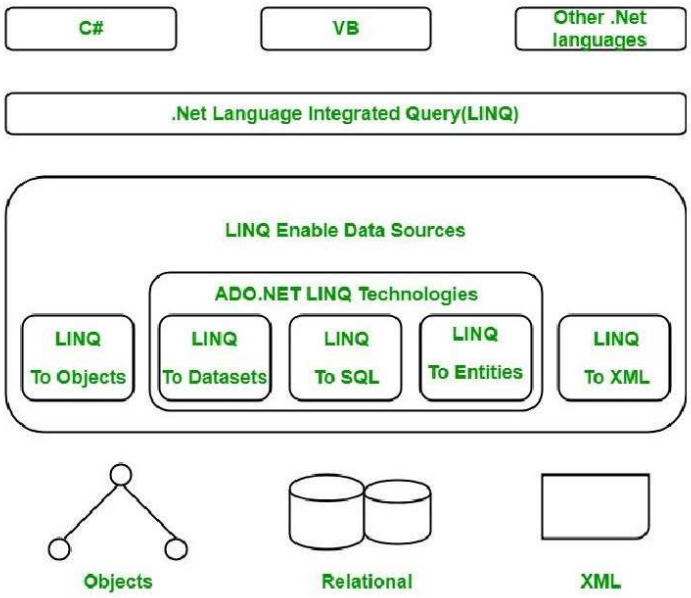
```
class Program
   {
      static void Main(string[] args)
      {
         string fileLoc = @"C:\filename.txt";
         string fileLocCopy = @"D:\filename.txt";
         if (File.Exists(fileLoc))
         {
            // If file already exists in destination, delete it.
            if (File.Exists(fileLocCopy))
               File.Delete(fileLocCopy);
            File.Copy(fileLoc, fileLocCopy);
         }
         Console.WriteLine("File Copied");
         Console.ReadLine();
      }
   }
```

Delete a Text File

```
class Program
   {
      static void Main(string[] args)
      {
         string fileLoc = @"C:\filename.txt";

         if (File.Exists(fileLoc))
         {
            File.Delete(fileLoc);
         }
         Console.WriteLine("File Deleted");
         Console.ReadLine();}}
```

LINQ (Language Integrated Query):

LINQ in C# is used to work with data access from sources such as objects, data sets, SQL Server, and XML. LINQ stands for Language Integrated Query. LINQ is a data querying API with SQL like query syntaxes. LINQ provides functions to query cached data from all kinds of data sources. The data source could be a collection of objects, database or XML files. We can easily retrieve data from any object that implements the IEnumerable<T> interface.



Advantages of LINQ

User does not need to learn new query languages for a different type of data source or data format.

- It increase the readability of the code.
- Query can be reused.
- It gives type checking of the object at compile time.
- It provides IntelliSense for generic collections.
- It can be used with array or collections.
- LINQ supports filtering, sorting, ordering, grouping.
- It makes easy debugging because it is integrated with C# language.

- It provides easy transformation means you can easily convert one data type into another data type like transforming SQL data into XML data.

Lambda expressions:

Lambda expressions are anonymous functions that contain expressions or sequence of operators. All lambda expressions use the lambda operator =>, that can be read as "goes to" or "becomes". The left side of the lambda operator specifies the input parameters and the right side holds an expression or a code block that works with the entry parameters. Usually lambda expressions are used as predicates or instead of delegates (a type that references a method).

Expression Lambdas:

Parameter => expression

Parameter-list => expression

Count => count + 2;

Sum => sum + 2;

n => n % 2 == 0

The lambda operator => divides a lambda expression into two parts. The left side is the input parameter and the right side is the lambda body.

Example1

```
class Program
  {
    static void Main(string[] args)
    {
      List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 };
      List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);

      foreach (var num in evenNumbers)
      {
        Console.Write("{0} ", num);
      }
      Console.WriteLine();
      Console.ReadLine();
    }
  }
```

Example2

```
class Dog
   {
      public string Name { get; set; }
      public int Age { get; set; }
   }
   class Program
   {
      static void Main(string[] args)
      {
                  List<Dog> dogs = new List<Dog>()
        {
         new Dog { Name = "Rex", Age = 4 },
         new Dog { Name = "Sean", Age = 0 },
         new Dog { Name = "Stacy", Age = 3 }
        };          var names = dogs.Select(x => x.Name);
        foreach (var name in names)
        {
           Console.WriteLine(name);

        }
        Console.ReadLine();
      }
   }
```

Example: 3(Lambda Expressions with Anonymous Types)

```
class Dog
   {
      public string Name { get; set; }
      public int Age { get; set; }
   }
   class Program
   {
      static void Main(string[] args)
      {
                  List<Dog> dogs = new List<Dog>()
        {
         new Dog { Name = "Rex", Age = 4 },
         new Dog { Name = "Sean", Age = 0 },
         new Dog { Name = "Stacy", Age = 3 }
        };
         var newDogsList = dogs.Select(x => new { Age = x.Age, FirstLetter = x.Name[0] });
        foreach (var item in newDogsList)
        {
           Console.WriteLine(item);
        }
        Console.ReadLine();
      }}
```

Sorting using a lambda expression:

```
class Dog
  {
    public string Name { get; set; }
    public int Age { get; set; }
  }
  class Program
  {
    static void Main(string[] args)
    {
      List<Dog> dogs = new List<Dog>()
      {
       new Dog { Name = "Rex", Age = 4 },
       new Dog { Name = "Sean", Age = 0 },
       new Dog { Name = "Stacy", Age = 3 }
      };
      var sortedDogs = dogs.OrderByDescending(x => x.Age);
      foreach (var dog in sortedDogs)
      {
        Console.WriteLine(string.Format("Dog {0} is {1} years old.", dog.Name, dog.Age));
      }
      Console.ReadLine();
    }
  }
```

Lambda expression in async:

```
public class AsyncClass
  {
    public async Task<string> Hello()
    {
      return await Task<string>.Run(() => {
        return "Return From Hello";
      });
    }
    public async void fun()
    {
      Console.WriteLine(await Hello());
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      new AsyncClass().fun();
      Console.ReadLine();

    }
  }
```

Example:

```
class Program
  {
    static void Main(string[] args)
    {
      List<string> countries = new List<string>();
      countries.Add("Nepal");
      countries.Add("China");
      countries.Add("US");
      countries.Add("Australia");
      countries.Add("Russia");
      IEnumerable<string> result = countries.Select(x => x);
      foreach (var item in result)
      {
        Console.WriteLine(item);
      }
      Console.ReadLine();
    }
  }
```

Lambda expression fit nice with collection:

```
class person
    {
        public string name { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            int[] data = { 1, 2, 4, 5, 6, 10 };
            //find all even number from array
            int[] even = data.Where(fn => fn % 2 == 0).ToArray();
            foreach (var item in even)
            {
                Console.WriteLine(item);
            }
            //find all odd number from array
            int[] odd = data.Where(fn => fn % 2 != 0).ToArray();
            foreach (var item in odd)
            {
                Console.WriteLine(item);
            }
            List<person> persons = new List<person> {
            new person {name = "Sourav"},
            new person {name = "Sudip"},
            new person {name = "Ram"}
        };
            //List of person whose name starts with "S"
          List<person> nameWithS = persons.Where(fn => fn.name.StartsWith("S")).ToList();
            foreach (var item in nameWithS)
            {
                Console.WriteLine("{0}", item);
            }
            Console.ReadLine(); }}
```

Try statements and Exceptions:

try-catch statement is useful to handle **unexpected or runtime** exceptions which will occur during the execution of the program. The **try-catch** statement will contain a **try** block followed by one or more **catch** blocks to handle different exceptions.

In c#, whenever an exception occurred in the **try** block, then the **CLR** (common language runtime) will look for the **catch** block that handles an exception. In case, if the currently executing method does not contain such a **catch** block, then the CLR will display an unhandled exception message to the user and stops the execution of the program.

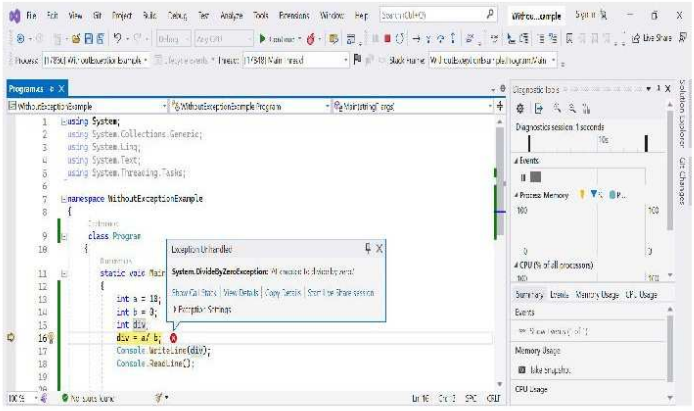
Syntax:

```
    try
{
  // put the code here that may raise exceptions
}
Catch(Excetion ex)
{
  // handle exception here
        Throw ex;
}
finally
{
  // final cleanup code
}
```

Keyword	Definition
try	Used to define a try block. This block holds the code that may throw an exception.
catch	Used to define a catch block. This block catches the exception thrown by the try block.
finally	Used to define the finally block. This block holds the default code.
throw	Used to throw an exception manually.

Example:

```
using System;
namespace WithoutExceptionExample
{
   class Program
   {
      static void Main(string[] args)
      {
         int a = 10;
         int b = 0;
         int div;
         div = a/ b;
         Console.WriteLine(div);
         Console.ReadLine();
      }}}
```



Example: (DivideByZeroException)

```
using System;
namespace WithoutExceptionExample
{
   class Program
   {
      static void Main(string[] args)
      {
         try
         {
            int a = 10;
            int b = 0;
            int div;
            div = a / b;
            Console.WriteLine("This will not be printed.");
         }
         catch (DivideByZeroException ex)
         {
            Console.WriteLine("Division by zero.");
         }
         finally
         {
            Console.WriteLine("This is Finally block.");
         }
         Console.ReadLine();
      }
   }
}
```

Output:

Exception Classes in C#:

All the exception classes in C# are derived from the System.Exception class. Some of the common exception classes present in C#:

- **System. DivideByZeroException**: handles the error generated by dividing a number with zero.
- **System. NullReferenceException**: handles the error generated by referencing the null object.
- **System.InvalidCastException**: handles the error generated by invalid typecasting.
- **System.IO.IOException**: handles the Input Output errors.
- **System. FieldAccessException**: handles the error generated by invalid private or protected field access.
- **System.OutOfMemoryException**: Handles the errors related to memory allocation.
- **System.IndexOutOfRangeException**: Handles the errors of accessing an array element, which is out of bound.

Custom Exception:

For Implementing Custom Exception Handling, **we need to derive the class CustomException from the system Base Class ApplicationException.**

Steps to Create Custom Exception

- Implement error handling in the user interface.
- Create and implement custom error messages.
- Create and implement custom error handlers.
- Raise and handle errors.

Example (Custom Exception)

```
using System;
namespace ExceptionCustomDemo
{
   class MyException : ApplicationException
   {
      public  MyException(string str):base(str)
      {

      }
   }
   class Program
   {
      static int chackage()
      {
         Console.WriteLine("Enter Age :");
         int age = Convert.ToInt32(Console.ReadLine());
         if (age < 18)
         {
            throw new MyException("Age is not valid");
         }
         return age;
      }
      static void Main(string[] args)
      {
         try
         {
            int result = chackage();
            Console.WriteLine("Your Age is : "+result);
            Console.ReadLine();
         }
         catch (MyException mye)
         {
            Console.WriteLine(mye.Message);
         }
         Console.ReadLine();
      }} }
```