Net Centric Computing:

Chapter 1:

1. Introduction to .Net Framework:

.NET is a software framework which is designed and developed by Microsoft. The first version of the .Net framework was 1.0 which came in the year 2002. In easy words, it is a virtual machine for compiling and executing programs written in different languages like C#,VB. Net etc. It is used to develop Form-based applications, Web-based applications, and Web services. There is a variety of programming languages available on the .Net platform, VB.Net and C# being the most common ones. It is used to build applications for Windows, phone, web, etc. It provides a lot of functionalities and also supports industry standards.

1.1 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

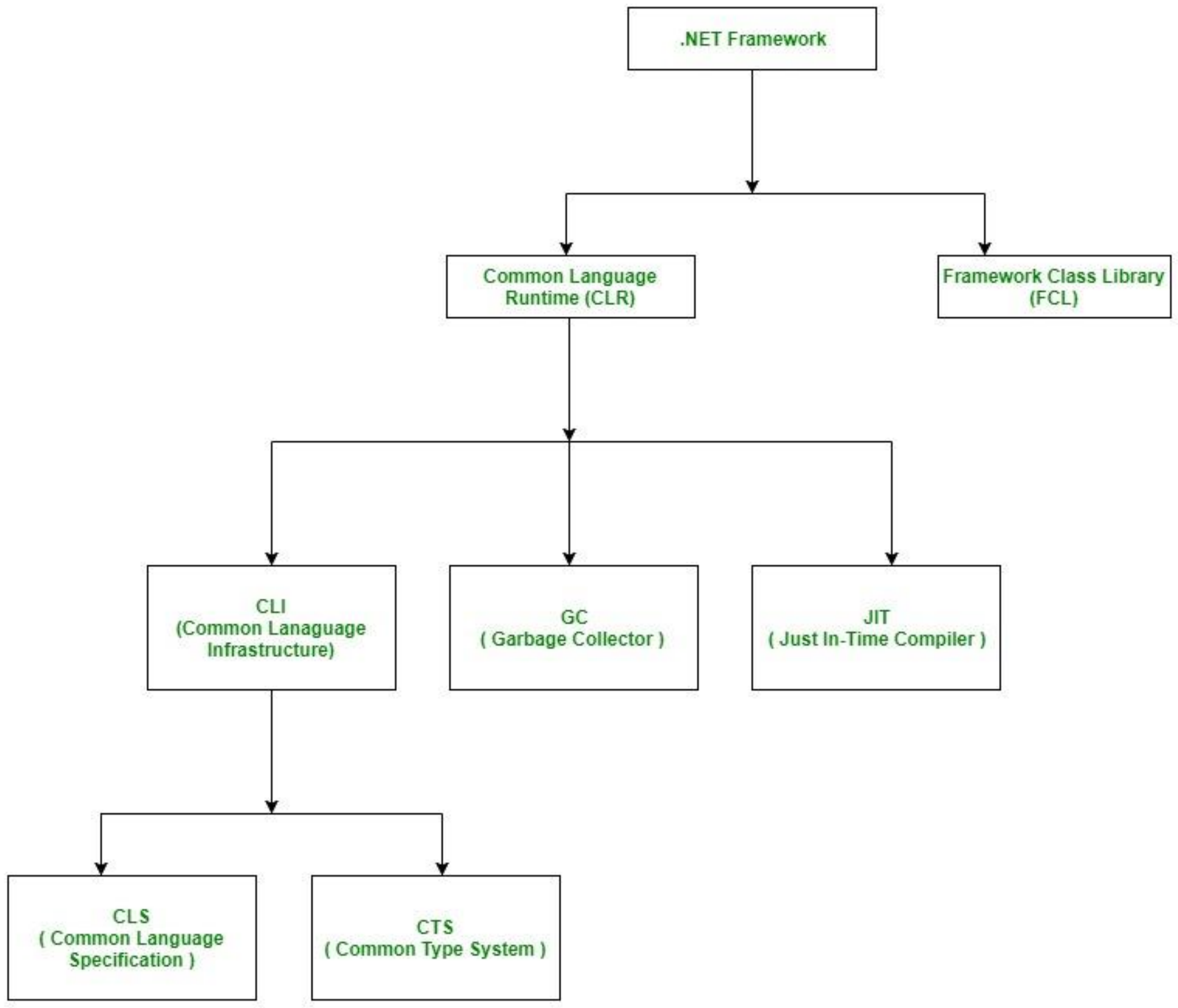

.NET Framework
Common Language Runtime (CLR)
Framework Class Library (FCL)
CLI (Common Lanaguage Infrastructure)
GC (Garbage Collector)
JIT (Just In-Time Compiler)
CLS (Common Language Specification)
CTS (Common Type System)

Figure 1: .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 proc ess of .NET applications

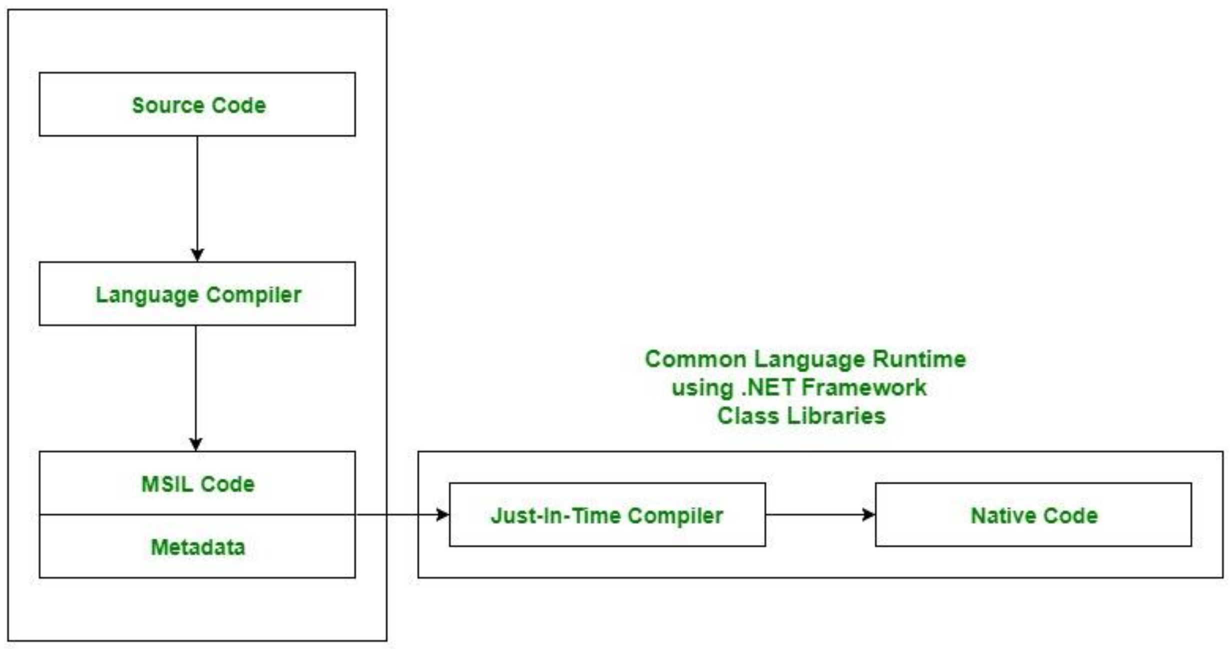


Figure 2: Common Language Runtime

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:

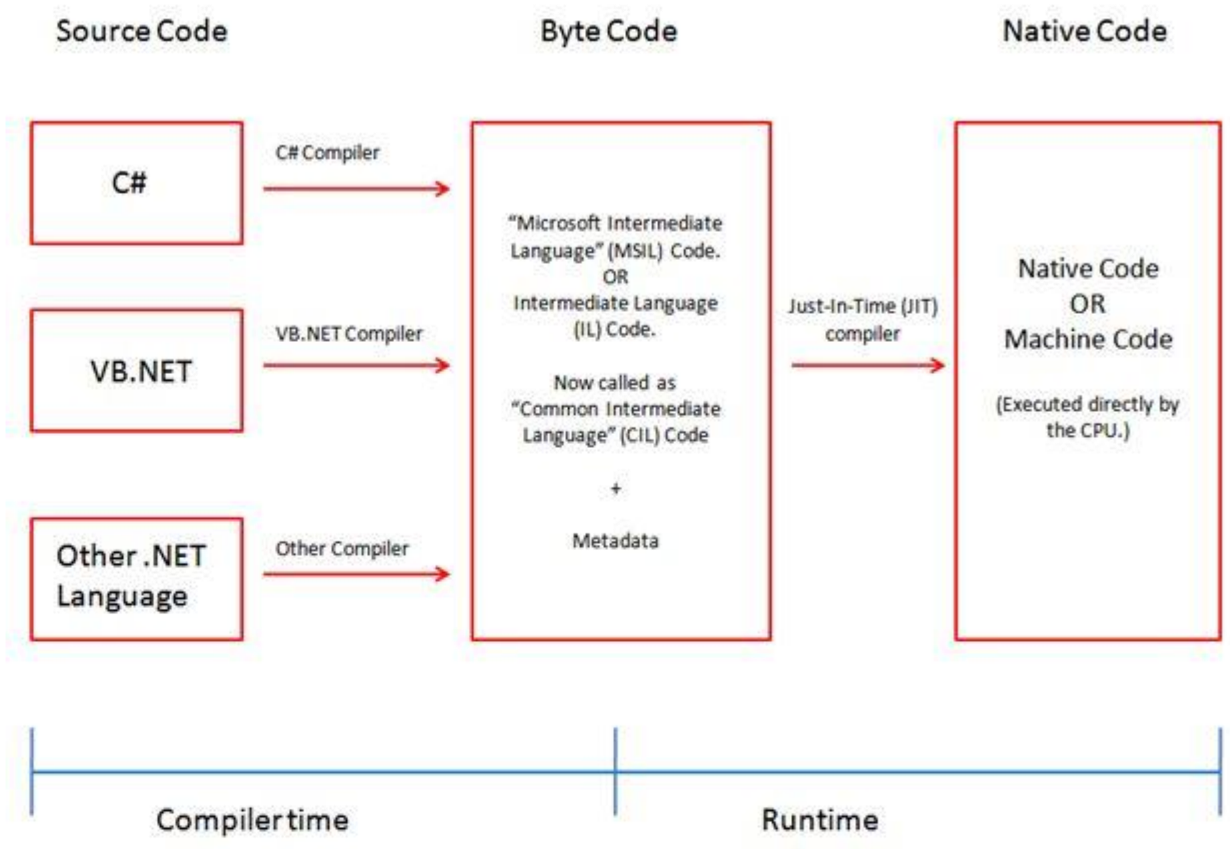
- It improves the performance by providing a rich interact between programs at run time.
- Enhance portability by removing the need of recompiling a program on any operating system that supports it.
- Security also increases as it analyzes the MSIL instructions whether they are safe or unsafe. Also, 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.

Compilation and execution of .Net applications:

The Code Execution Process involves the following two stages:

1. Compiler time process.
2. Runtime process.



Source Code
Byte Code
Native Code
C#
VB.NET
Other .NET Language
C# Compiler
VB.NET Compiler
Other Compiler
"Microsoft Intermediate Language" (MSIL) Code.
OR
Intermediate Language (IL) Code.
Now called as "Common Intermediate Language" (CIL) Code
+
Metadata
Just-In-Time (JIT) compiler
Native Code
OR
Machine Code
(Executed directly by the CPU.)
Compilertime
Runtime

Figure 3: Code Execution Process in .net Framework

1. Compiler time process

1. The .Net framework has one or more language compilers, such as Visual Basic, C#, Visual C++, JScript, or one of many third-party compilers such as an Eiffel, Perl, or COBOL compiler.
2. Anyone of the compilers translates your source code into Microsoft Intermediate Language (MSIL) code.
3. For example, if you are using the C# programming language to develop an application, when you compile the application, the C# language compiler will convert your source code into Microsoft Intermediate Language (MSIL) code.
4. In short, VB.NET, C#, and other language compilers generate MSIL code. (In other words, compiling translates your source code into MSIL and generates the required metadata.)
5. 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)

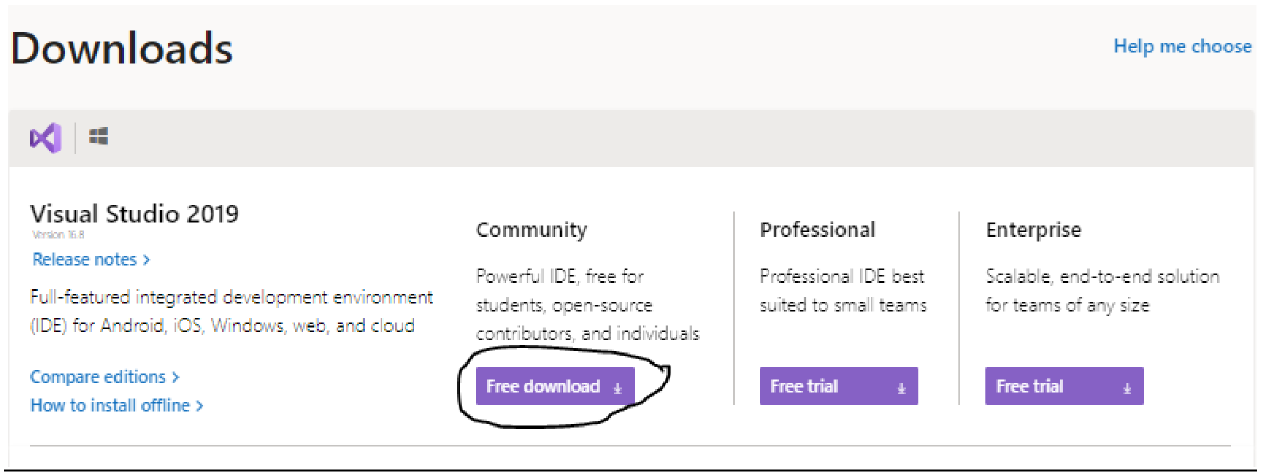
2. Runtime process.

1. The Common Language Runtime (CLR) includes a JIT compiler for converting MSIL to native code.
2. The JIT Compiler in CLR converts the MSIL code into native machine code that is then executed by the OS.
3. 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.

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

Installing Visual Studio 2019 Community:

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



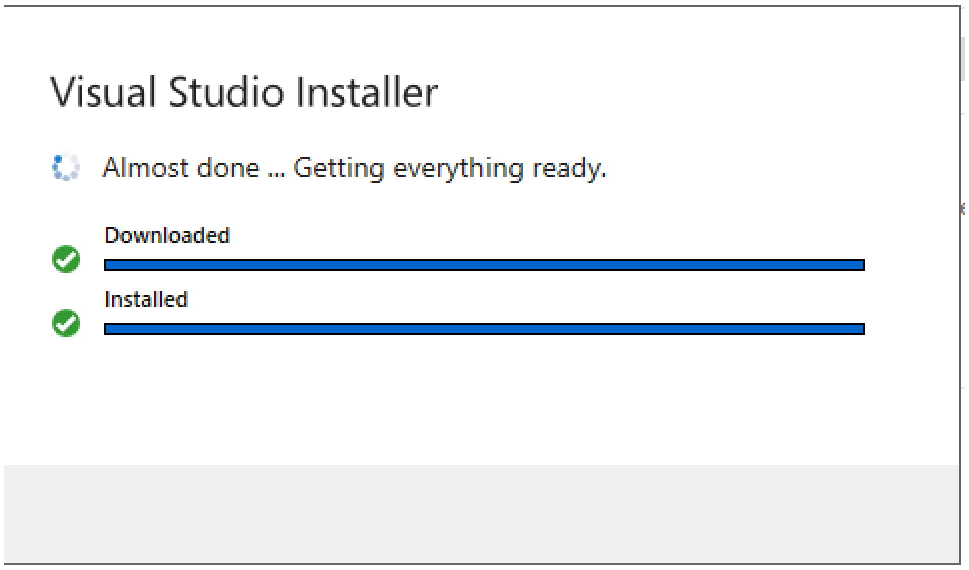
Downloads

Help me choose

Visual Studio 2019
Version 16.8
Release notes >
Full-featured integrated development environment (IDE) for Android, iOS, Windows, web, and cloud
Compare editions >
How to install offline >

Community	Professional	Enterprise
Powerful IDE, free for students, open-source contributors, and individuals	Professional IDE best suited to small teams	Scalable, end-to-end solution for teams of any size
Free download	Free trial	Free trial

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



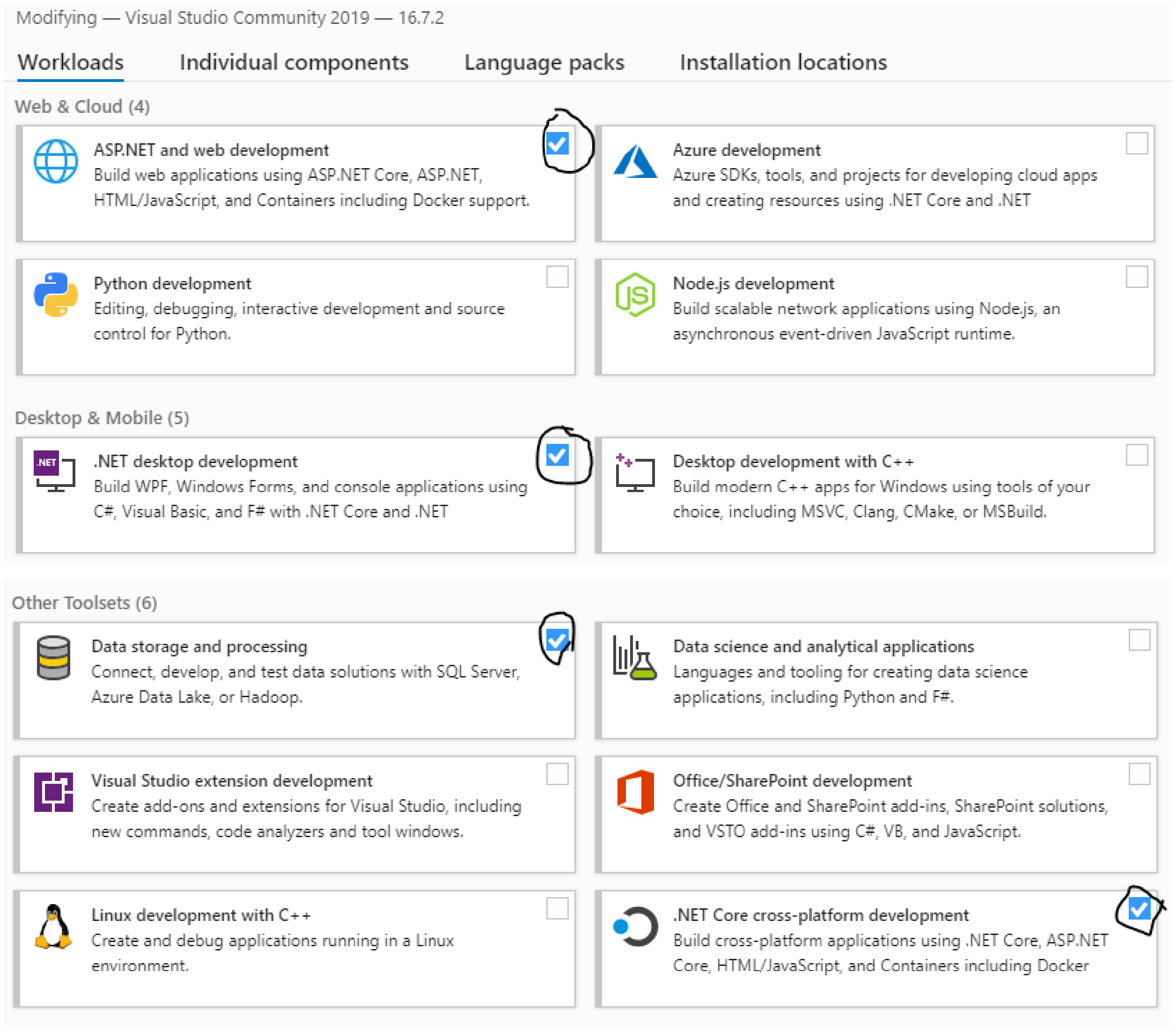
Visual Studio Installer

Almost done ... Getting everything ready.

Downloaded

Installed

4. Check Following option



Modifying — Visual Studio Community 2019 — 16.7.2

Workloads | Individual components | Language packs | Installation locations

Web & Cloud (4)

[✓] ASP.NET and web development
Build web applications using ASP.NET Core, ASP.NET, HTML/JavaScript, and Containers including Docker support.

[ ] Azure development
Azure SDKs, tools, and projects for developing cloud apps and creating resources using .NET Core and .NET

[ ] Python development
Editing, debugging, interactive development and source control for Python.

[ ] Node.js development
Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

Desktop & Mobile (5)

[✓] .NET desktop development
Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET Core and .NET

[ ] Desktop development with C++
Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild.

Other Toolsets (6)

[✓] Data storage and processing
Connect, develop, and test data solutions with SQL Server, Azure Data Lake, or Hadoop.

[ ] Data science and analytical applications
Languages and tooling for creating data science applications, including Python and F#.

[ ] Visual Studio extension development
Create add-ons and extensions for Visual Studio, including new commands, code analyzers and tool windows.

[ ] Office/SharePoint development
Create Office and SharePoint add-ins, SharePoint solutions, and VSTO add-ins using C#, VB, and JavaScript.

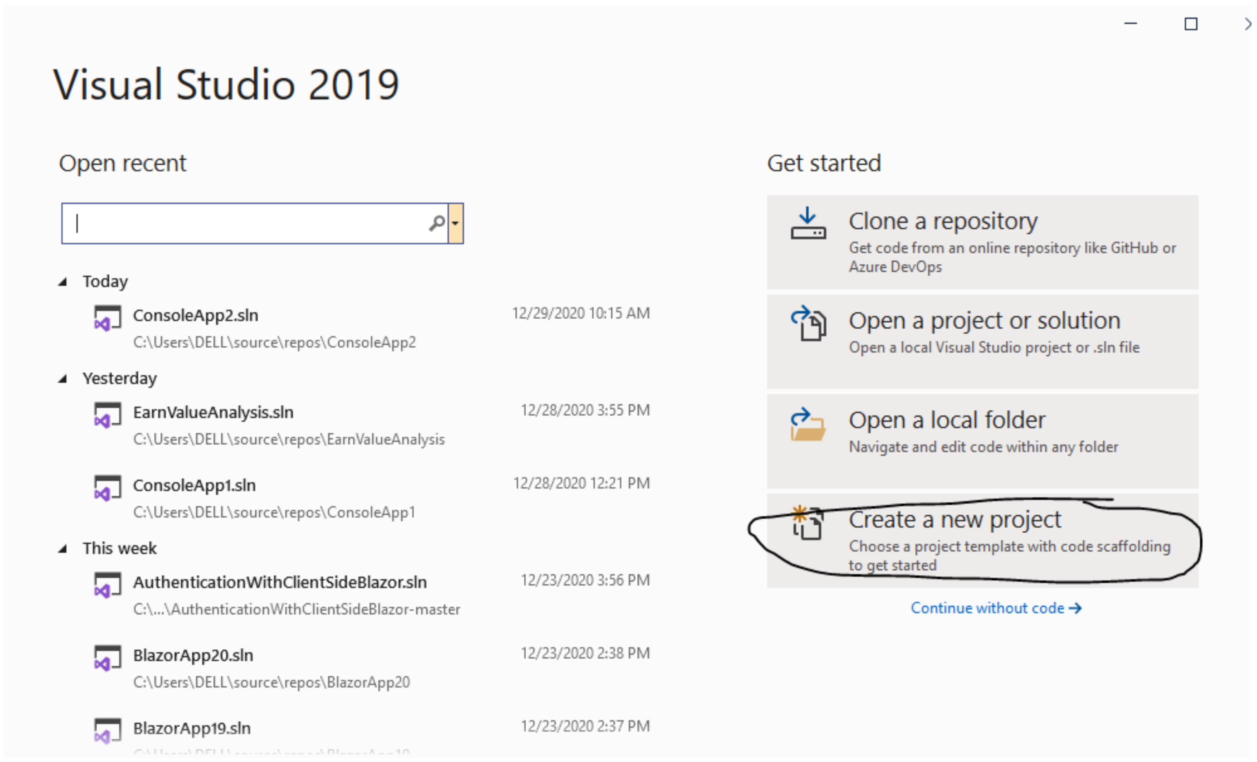
[ ] Linux development with C++
Create and debug applications running in a Linux environment.

[✓] .NET Core cross-platform development
Build cross-platform applications using .NET Core, ASP.NET Core, HTML/JavaScript, and Containers including Docker

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.



Visual Studio 2019

Open recent

Today

ConsoleApp2.sln 12/29/2020 10:15 AM
C:\Users\DELL\source\repos\ConsoleApp2

Yesterday

EarnValueAnalysis.sln 12/28/2020 3:55 PM
C:\Users\DELL\source\repos\EarnValueAnalysis

ConsoleApp1.sln 12/28/2020 12:21 PM
C:\Users\DELL\source\repos\ConsoleApp1

This week

AuthenticationWithClientSideBlazor.sln 12/23/2020 3:56 PM
C:\...\AuthenticationWithClientSideBlazor-master

BlazorApp20.sln 12/23/2020 2:38 PM
C:\Users\DELL\source\repos\BlazorApp20

BlazorApp19.sln 12/23/2020 2:37 PM

Get started

Clone a repository
Get code from an online repository like GitHub or Azure DevOps

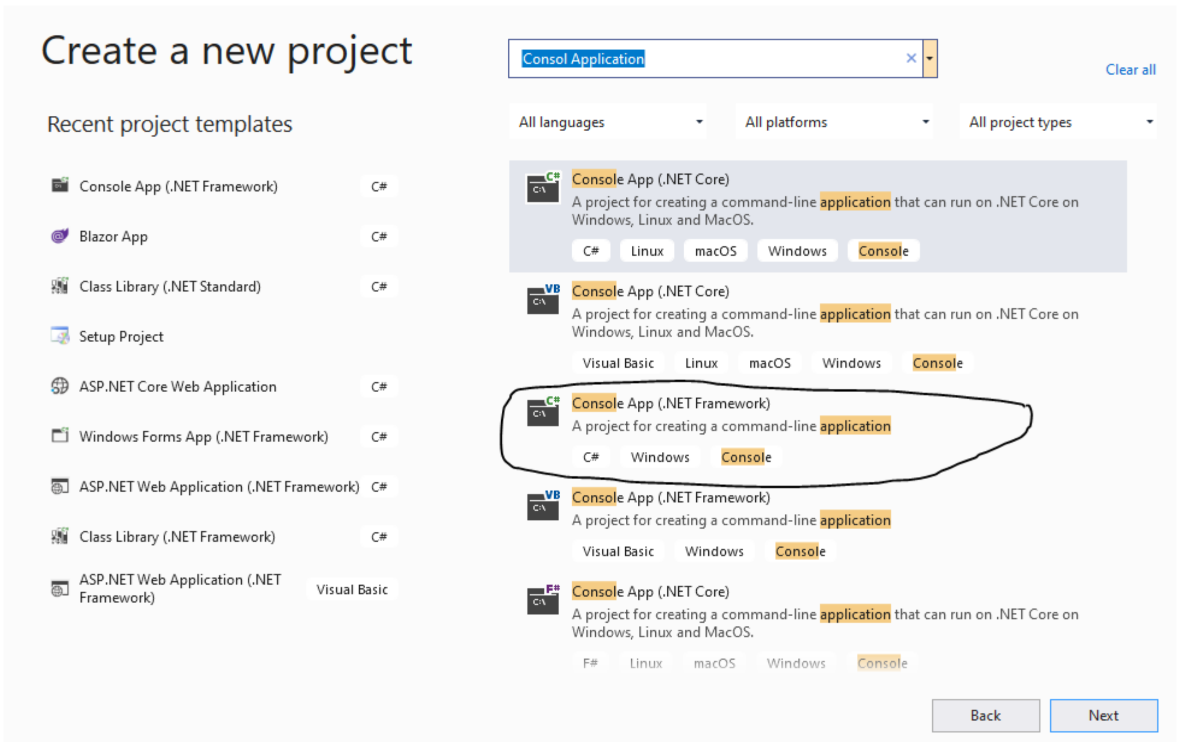
Open a project or solution
Open a local Visual Studio project or .sln file

Open a local folder
Navigate and edit code within any folder

Create a new project
Choose a project template with code scaffolding to get started

Continue without code →

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



Create a new project

Consol Application

Clear all

Recent project templates

Console App (.NET Framework)	C#
Blazor App	C#
Class Library (.NET Standard)	C#
Setup Project	
ASP.NET Core Web Application	C#
Windows Forms App (.NET Framework)	C#
ASP.NET Web Application (.NET Framework)	C#
Class Library (.NET Framework)	C#
ASP.NET Web Application (.NET Framework)	Visual Basic

All languages | All platforms | All project types

Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
C# Linux macOS Windows Console

Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
Visual Basic Linux macOS Windows Console

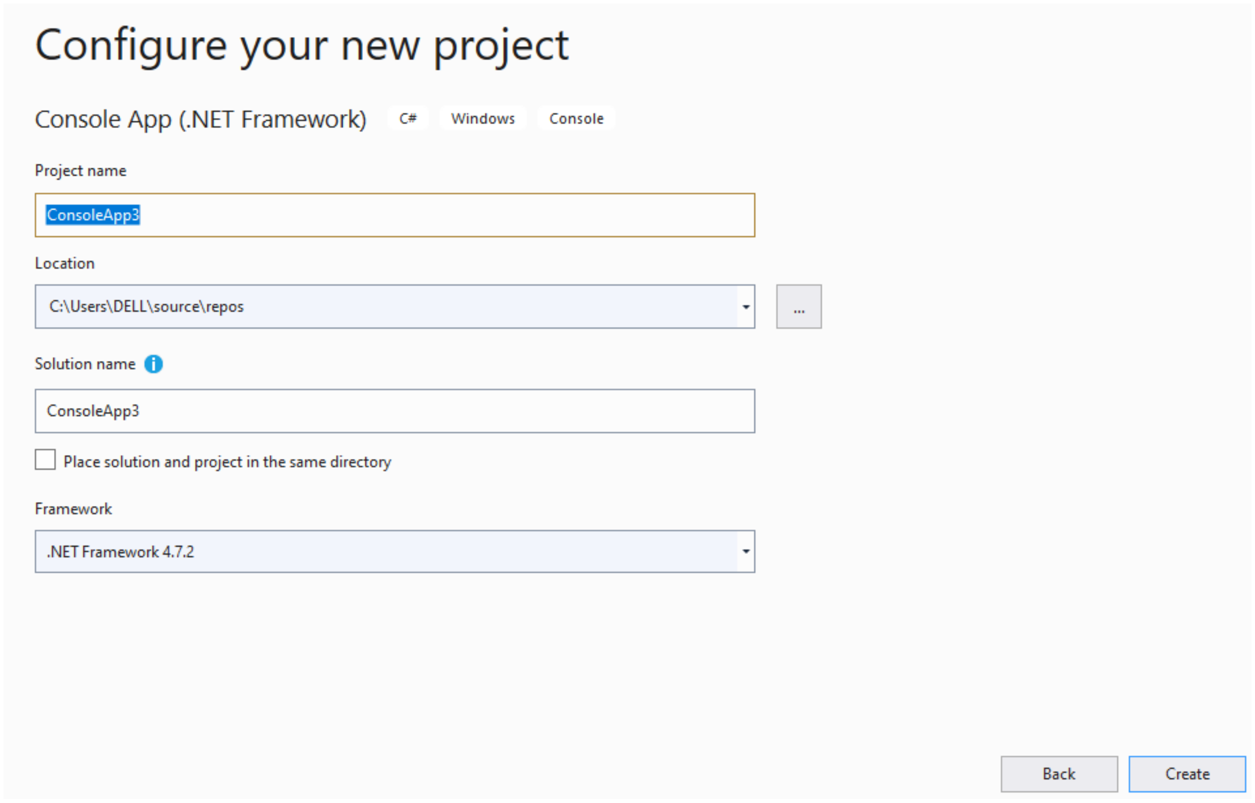
Console App (.NET Framework)
A project for creating a command-line application
C# Windows Console

Console App (.NET Framework)
A project for creating a command-line application
Visual Basic Windows Console

Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
F# Linux macOS Windows Console

Back Next

4. Click create.



Configure your new project

Console App (.NET Framework) C# Windows Console

Project name: ConsoleApp3

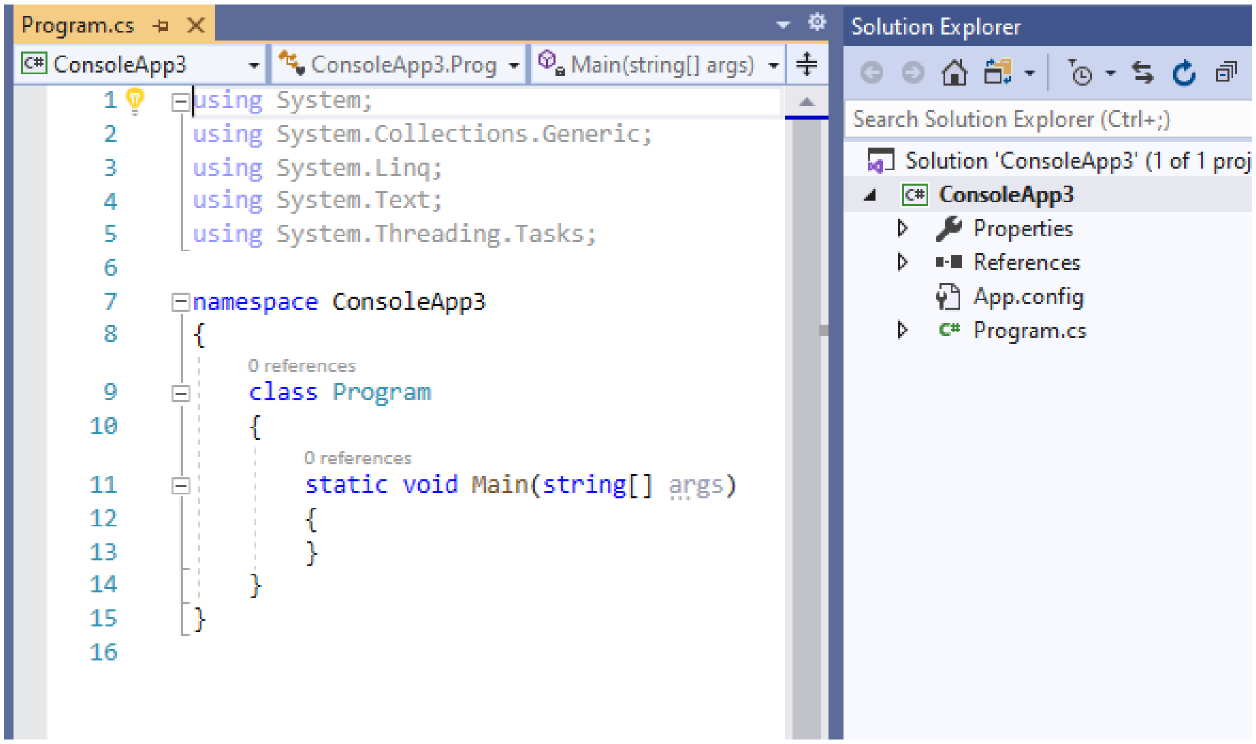
Location: C:\Users\DELL\source\repos

Solution name: ConsoleApp3

Place solution and project in the same directory

Framework: .NET Framework 4.7.2

5. After clicking create.



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

namespace ConsoleApp3
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
        }
    }
}
```

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 ConsoleApp3
{
    class Program
    {
        //Entry point of the program
        static void Main(string[] args)
        {
            //print Hello world"
            Console.WriteLine("Hello World!");
        }
    }
}
```

Constructor and Destructor:

A constructor is a specialized function that is used to initialize fields. A constructor has the same name as the class. Instance constructors are invoked with the new operator and can't be called in the same manner as other member functions. There are some important rules pertaining to constructors as in the following;

Classes with no constructor have an implicit constructor called the default constructor, that is parameter less. The default constructor assigns default values to fields.

- A public constructor allows an object to be created in the current assembly or referencing assembly.
- Only the extern modifier is permitted on the constructor.
- A constructor returns void but does not have an explicitly declared return type.
- A constructor can have zero or more parameters.
- Classes can have multiple constructors in the form of default, parameter or both.

The following example shows one constructor for a customer class.

```
using System;

namespace ConsoleApp3
{
    class Program
    {
        // Member Variables
        public string Name;

        //constuctor for initializing fields
        public Program(string fname, string lname)
        {
            Name = fname + " " + lname;
        }
        //method for displaying customer records
        public void AppendData()
        {
            Console.WriteLine(Name);
        }
        //Entry point
        static void Main(string[] args)
        {
            /// object instantiation
            Program obj = new Program("Barack", "Obama");

            //Method calling
            obj.AppendData();
        }
    }
}
```

Destructors:

The purpose of the destructor method is to remove unused objects and resources. Destructors are not called directly in the source code but during garbage collection. Garbage collection is nondeterministic. A destructor is invoked at an undetermined moment. More precisely a programmer can't control its execution; rather it is called by the Finalize () method. Like a constructor, the destructor has the same name as the class except a destructor is prefixed with a tilde (~). There are some limitations of destructors as in the following;

- Destructors are parameterless.
- A Destructor can't be overloaded.
- Destructors are not inherited.
- Destructors can cause performance and efficiency implications.

Destructor Example:

```
using System;
namespace ConsoleApp3
{
    class customer
    {
        // Member Variables
        public int x, y;
        //constuctor for  initializing fields
        customer()
        {
            Console.WriteLine("Fields inititalized");
            x = 10;
        }
        //method for get field
        public void getData()
        {
            y = x * x;
            Console.WriteLine(y);
        }
        //method to release resource explicitly
        public void Dispose()
        {
            Console.WriteLine("Fields cleaned");
            x = 0;
            y = 0;
        }
        //destructor
        ~customer()
        {
            Dispose();
        }
        //Entry point
        static void Main(string[] args)
        {
            //instance created
            customer obj = new customer();

            obj.getData();

        }
    }
}
```

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.

Accessors: The block of "set" and "get" is known as "Accessors". It is very essential to restrict the accessibility of property. There are two type of accessors i.e. **get accessors** and **set accessors**. There are different types of properties based on the "get" and set accessors:

- **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 accessors and it introduce in C# 3.0.

The syntax for Defining Properties:

```
<access_modifier> <return_type> <property_name>
{
        get { // body }
        set { // body }
}
```

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.

Read only Properties:

```
namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Student st = new Student();
            Console.WriteLine(st.Name);
            Console.ReadLine();
        }
    }
    public class Student
    {
        public string Name
        {
            get { return "Sunil Chaudhary"; }
        }
    }
}
```

Read Write only Properties:

```
namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Student st = new Student();
            //writting into property
            st.Name = "Sunil Chaudhary";
            //reading value from property
            Console.WriteLine(st.Name);
            Console.ReadLine();
        }
    }
    public class Student
    {
        private string name = "";
        public string Name
        {
            get { return name; }
            set { name = value; }
        }
    }
}
```

Write only Properties:

```
namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Student st = new Student();
            //writting into property
            st.Name = "Sunil Chaudhary";
            Console.ReadLine();
        }
    }
    public class Student
    {
        private string name = "";
        public string Name
        {

            set { name = value; }
        }
    }
}
```

Automatic Properties:

```
namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Student st = new Student();
            st.Id = 1;
            st.Name = "Sunil Chaudhary";
            Console.WriteLine("Id={0}  Name={1}", st.Id, st.Name);
            Console.ReadLine();
        }
    }
    public class Student
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }
}
```

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.

In C#, an array index starts at zero. That means the first item of an array starts at the 0th position. The position of the last item on an array will the total number of items - 1. So if an array has 10 items, the last 10th item is in 9th position.

C#, arrays can be declared as fixed-length or dynamic. A *fixed-length* array can store a predefined number of items. A *dynamic array* does not have a predefined size. The size of a *dynamic array* increases as you add new items to the array. You can declare an array of fixed length or dynamic.

Initializing Array :

```
// Initialize a fixed array
        int[] staticIntArray = new int[3] { 1, 3, 5 };
```

Alternative, we can also add array items one at a time as listed in the following code snippet.

```
// Initialize a fixed array one item at a time
            int[] staticIntArray = new int[3];
            staticIntArray[0] = 1;
            staticIntArray[1] = 3;
            staticIntArray[2] = 5;
```

The following code snippet declares a dynamic array with string values.

```
// Initialize a dynamic array items during declaration
            string[] strArray = new string[] { "Ramesh", "Suresh", "Hari", "Dinesh",
"Bhim" };
```

Accessing Array:

```
            // Initialize a fixed array one item at a time
            int[] staticIntArray = new int[3];
            staticIntArray[0] = 1;
            staticIntArray[1] = 3;
            staticIntArray[2] = 5;
            // Read array items one by one
            Console.WriteLine(staticIntArray[0]);
            Console.WriteLine(staticIntArray[1]);
            Console.WriteLine(staticIntArray[2]);
```

Accessing an array using a foreach Loop

```
// Initialize a dynamic array items during declaration
            string[] strArray = new string[] { "Ramesh", "Suresh", "Hari", "Dinesh", 
"Bhim" };
            // Read array items using foreach loop
            foreach (string str in strArray)
            {
                Console.WriteLine(str);
            }
```

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.

Array Example:

```
    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);
            }
            // User defined method for iterating array elements
            static void PrintArray(int[] arr)
            {
                foreach (Object elem in arr)
                {
                    Console.Write(elem + " ");
                }
            }
```

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 :1

```
            string msg = "Suresh,Rohini,Trishika";
               string[] strarr = msg.Split(',');
               for (int i = 0; i < strarr.Length; i++)
               {
                   Console.WriteLine(strarr[i]);
               }
               Console.WriteLine("\nPress Enter Key to Exit..");
               Console.ReadLine();
```

Example: 2

```
        static void Main(string[] args)
         {
             string msg = "Hi Guest Hi";
             string nmsg = msg.Replace("Hi", "Welcome");
             Console.WriteLine("Old: {0}", msg);
             Console.WriteLine("New: {0}", nmsg);

             string x = "aaaaa";
             string nx = x.Replace("a", "b").Replace("b", "c");
             Console.WriteLine("Old: {0}", x);
             Console.WriteLine("New: {0}", nx);

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

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

Example: 3

```
static void Main(string[] args)
        {
            string msg1 = "Welcome to";
            string msg2 = " " + "ACHS";
            Console.WriteLine("Message: {0}", string.Concat(msg1, msg2));

            string name1 = "Suresh";
            string name2 = ", " + "Rohini";
            string name3 = ", " + "Trishika";
            Console.WriteLine("Users: {0}", string.Concat(string.Concat(name1, name2), name3));

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

        }
```

Example: 5

```
            string msg = "Welcome to ACHS";
            Console.WriteLine("SubString: {0}", msg.Substring(5));
            Console.WriteLine("Substring with Length: {0}", msg.Substring(3, 7));
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
```

Example: 6

```
string msg = "Welcome to ACHS";
            Console.WriteLine("Remove Result: {0}", msg.Remove(5));
            Console.WriteLine("Remove with Length: {0}", msg.Remove(3, 7));
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
```

Example: 7

```
            string s = "Name:{0} {1}, Location:{2}, Age:{3}";
            string msg = string.Format(s, "Suresh", "Dasari", "Hyderabad", 32);
            Console.WriteLine("Format Result: {0}", msg);
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
```

Indexer:

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.

Array access operator i.e (**[ ]**) is used to access the instance of the class which uses an indexer. A user can retrieve or set the indexed value without pointing an instance or a type member. **Indexers** are almost similar to the **Properties**.

Syntax:

```
[access_modifier] [return_type] this [argument_list]
```

```
{
  get

  {
     // get block code
  }
  set
  {
    // set block code
  }

}
```

In the above syntax:

- **access_modifier:** It can be public, private, protected or internal.
- **return_type:** It can be any valid C# type.
- **this:** It is the keyword which points to the object of the current class.
- **argument_list:** This specifies the parameter list of the indexer.
- **get{ } and set { }:** These are the **accessors**.

Difference between Indexers and Properties:

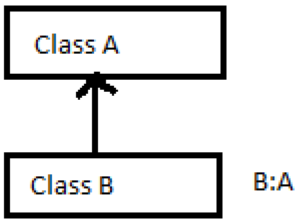
Indexers	Properties
Indexers are created with this keyword.	Properties don't require this keyword.
Indexers are identified by signature.	Properties are identified by their names.
Indexers are accessed using indexes.	Properties are accessed by their names.
Indexer are instance member, so can't be static.	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.

Example:

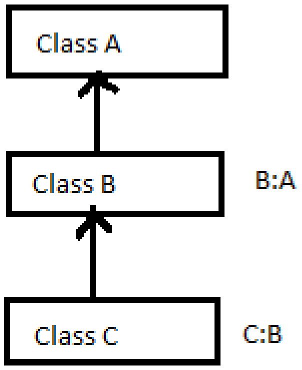
```
class Program
    {
        class IndexerClass
        {
            private string[] names = new string[10];
            public string this[int i]
            {
                get
                {
                    return names[i];
                }
                set
                {
                    names[i] = value;
                }
            }
        }
        static void Main(string[] args)
        {
            IndexerClass Team = new IndexerClass();
            Team[0] = "Ram";
            Team[1] = "Shyam";
            Team[2] = "Hari";
            Team[3] = "Gita";
            Team[4] = "Sita";
            Team[5] = "Hema";
            Team[6] = "Rita";
            Team[7] = "Mohan";
            Team[8] = "Bikash";
            Team[9] = "Bimal";
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine(Team[i]);
            }
            Console.ReadLine();
        }
    }
```

Inheritance

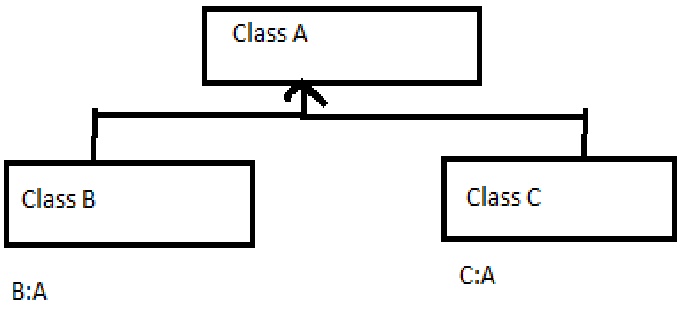
Inheritance is a mechanism in which one class acquires the property of another class. For example, a child inherits the traits of his/her parents. With inheritance, we can **reuse** the fields and methods of the existing class. Hence, inheritance facilitates Reusability and is an important concept of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



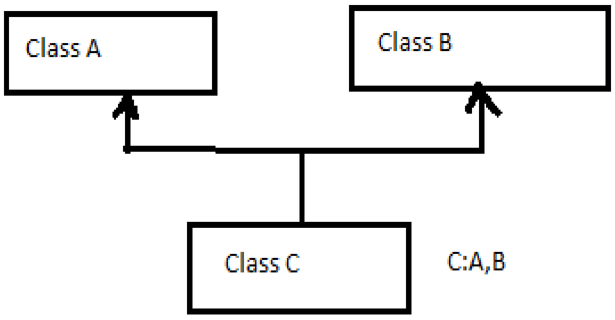
Single or Simple Inheritance



Multilevel Inheritance



Hierarchial Inheritance



Multiple Inheritance

Single or Simple Inheritance Example:

```
class Program
    {
        static void Main(string[] args)
        {
            B bee = new B();
            Console.WriteLine(bee.MethodA());
            Console.WriteLine(bee.MethodB());
        }
    }
    class A
    {
        public string MethodA()
        {
            return "MethodA";
        }
    }
    class B:A
    {
        public string MethodB()
        {
            return "MethodB";
        }
    }
```

Multilevel Inheritance Example:

```
  class Program
    {
        static void Main(string[] args)
        {
            C cee = new C();
            Console.WriteLine(cee.MethodA());
            Console.WriteLine(cee.MethodB());
            Console.WriteLine(cee.MethodC());
            Console.ReadLine();
        }
    }
    class A
    {
        public string MethodA()
        {
            return "MethodA";
        }
    }
    class B:A
    {
        public string MethodB()
        {
            return "MethodB";
        }
    }
    class C : B
    {
        public string MethodC()
        {
            return "MethodC";
        }
    }
```

Hierarchical Inheritance:

```
class Program
  {
      static void Main(string[] args)
      {
          B bee = new B();
          Console.WriteLine(bee.MethodA());
          Console.WriteLine(bee.MethodB());
          C cee = new C();
          Console.WriteLine(cee.MethodA());
          Console.WriteLine(cee.MethodC());
          Console.ReadLine();
      }
  }
  class A
  {
      public string MethodA()
      {
          return "MethodA";
      }
  }
  class B:A
  {
      public string MethodB()
      {
          return "MethodB";
      }
  }
  class C : A
  {
      public string MethodC()
      {
          return "MethodC";
      }
  }
```

Multiple Inheritance:

But C# does not support multiple class inheritance. To overcome this problem we use interfaces to achieve multiple class inheritance. With the help of the interface, class C(as shown in the above diagram) can get the features of class A and B.

Multiple Inheritance Example:

```
class Program
    {
        static void Main(string[] args)
        {
            C cee = new C();
            Console.WriteLine(cee.MethodA());
            Console.WriteLine(cee.MethodB());
            Console.ReadLine();
        }
    }
    interface IA
    {
        string MethodA();

    }
    interface IB
    {
        string MethodB();

    }
    class C : IA, IB
    {
        public string MethodA()
        {
            return "MethodA";
        }

        public string MethodB()
        {
            return "MethodB";
        }
    }
```

Real World Single Inheritance Example:1

```
class Program
    {
        static void Main(string[] args)
        {
            Rectangle Rect = new Rectangle();
            Rect.setWidth(5);
            Rect.setHeight(7);
            // Print the area of the object.
            Console.WriteLine("Total area: {0}", Rect.getArea());
            Console.ReadLine();
        }
    }
    class Shape
    {
        public void setWidth(int w)
        {
            width = w;
        }
        public void setHeight(int h)
        {
            height = h;
        }
        protected int width;
        protected int height;
    }
    // Derived class
    class Rectangle : Shape
    {
        public int getArea()
        {
            return (width * height);
        }
    }
```

Real World Multiple Inheritance Example:2

```
    class Program
     {
         static void Main(string[] args)
         {
             Rectangle Rect = new Rectangle();
             int area;
             Rect.setWidth(5);
             Rect.setHeight(7);
             area = Rect.getArea();

             // Print the area of the object.
             Console.WriteLine("Total area: {0}", Rect.getArea());
             Console.WriteLine("Total paint cost: ${0}", Rect.getCost(area));
             Console.ReadLine();
         }
     }
     class Shape
     {
         public void setWidth(int w)
         {
             width = w;
         }
         public void setHeight(int h)
         {
             height = h;
         }
         protected int width;
         protected int height;
     }

     // Base class PaintCost
     public interface PaintCost
     {
         int getCost(int area);
     }
     // Derived class
     class Rectangle : Shape, PaintCost
     {
         public int getArea()
         {
             return (width * height);
         }
         public int getCost(int area)
         {
             return area * 70;
         }
     }
```

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:

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

            // class instance
            new myDerived().VirtualMethod();
            Console.ReadLine();
        }
    }
    class myBase
        {
            //virtual function
            public virtual void VirtualMethod()
            {
                Console.WriteLine("virtual method defined in the base class");
            }
        }
    class myDerived : myBase
        {
            // redifing the implementation of base class method
            public override void VirtualMethod()
            {
                Console.WriteLine("virtual method defined in the Derive class");
            }
        }
```

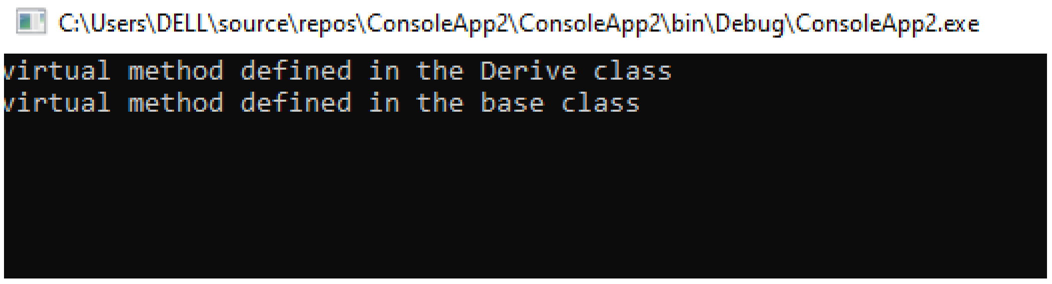
Hiding Methods:

If a method with the same signature is declared in both base and derived classes, but the methods are not declared as virtual and overriden respectively, then the derived class version is said to hide the base class version. In most cases, you would want to override methods rather than hide them. Otherwise .NET automatically generates a warning.

Example:

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

            // class instance
            new myDerived().VirtualMethod();
            Console.ReadLine();
        }
    }
    class myBase
    {
        //virtual function
        public virtual void VirtualMethod()
        {
            Console.WriteLine("virtual method defined in the base class");
        }
    }

    class myDerived : myBase
    {
        // hiding the implementation of base class method
        public new void VirtualMethod()
        {
            Console.WriteLine("virtual method defined in the Derive class");

            //still access the base class method
            base.VirtualMethod();
        }
    }
```

Output:



C:\Users\DELL\source\repos\ConsoleApp2\ConsoleApp2\bin\Debug\ConsoleApp2.exe

```
virtual method defined in the Derive class
virtual method defined in the base class
```

Polymorphism

Polymorphism is the ability to treat the various objects in the same manner. It is one of the significant benefits of inheritance. We can decide the correct call at runtime based on the derived type of the base reference. This is called late binding.

In the following example, instead of having a separate routine for the hrDepart, itDepart and financeDepart classes, we can write a generic algorithm that uses the base type functions. The method LeaderName() declared in the base abstract class is redefined as per our needs in 2 different classes. **Example:**

```
public abstract class Employee
 {
     public virtual void LeaderName()
     {
     }
 }

 public class hrDepart : Employee
 {
     public override void LeaderName()
     {
         Console.WriteLine("Mr. jone");
     }
 }
 public class itDepart : Employee
 {
     public override void LeaderName()
     {
         Console.WriteLine("Mr. Tom");
     }
 }

 public class financeDepart : Employee
 {
     public override void LeaderName()
     {
         Console.WriteLine("Mr. Linus");
     }
 }

 class Program
 {
     static void Main(string[] args)
     {
         hrDepart obj1 = new hrDepart();
         itDepart obj2 = new itDepart();
         financeDepart obj3 = new financeDepart();

         obj1.LeaderName();
         obj2.LeaderName();
         obj3.LeaderName();

         Console.ReadLine();
     }
 }
```

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);
```

Events:

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

```
event event_delegate event_name;
```

Below are some differences between the Delegates and Interfaces in C#:

DELEGATE	INTERFACE
It could be a method only.	It contains both methods and properties.
It can be applied to one method at a time.	If a class implements an interface, then it will implement all the methods related to that interface.
Delegates can me implemented any number of times.	Interface can be implemented only one time.
It is used to handling events.	It is not used for handling events.
It can access anonymous methods.	It can not access anonymous methods.
It does not support inheritance.	It supports inheritance.
It created at run time.	It created at compile time.

Delegates:

Two types of Delegates are:

1. Single Cast Delegates(Point a single Method)

2. Multicast Delgates(Point Multiple Method)

Example Single Cast:

```
public delegate void delmethod();

    public class DelegatesExample
    {
        public static void display()
        {
            Console.WriteLine("Hello!");
        }
        public static void show()
        {
            Console.WriteLine("Hi!");
        }

        public void print()
        {
            Console.WriteLine("Print");
        }

    }
    class Program
    {
        static void Main(string[] args)
        {
            // here we have assigned static method show() of class P to delegate
delmethod()
            delmethod del1 = P.show;
            // here we have assigned static method display() of class P to delegate
delmethod() using new operator
            // you can use both ways to assign the delagate
            delmethod del2 = new delmethod(DelegatesExample.display);
            DelegatesExample obj = new DelegatesExample();
            // here first we have create instance of class P and assigned the method
print() to the delegate i.e. delegate with class
            delmethod del3 = obj.print;
            del1();
            del2();
            del3();
            Console.ReadLine();
        }
    }
```

Multicast Delegates Example:

```
public delegate void delmethod(int x, int y);
    public class TestMultipleDelegate
    {
        public void plus_Method1(int x, int y)
        {
            Console.Write("You are in plus_Method");
            Console.WriteLine(x + y);
        }
        public void subtract_Method2(int x, int y)
        {
            Console.Write("You are in subtract_Method");
            Console.WriteLine(x - y);
        }

    }
    class Program
    {
        static void Main(string[] args)
        {
            TestMultipleDelegate obj = new TestMultipleDelegate();
            delmethod del = new delmethod(obj.plus_Method1);

            // Here we have multicast
            del += new delmethod(obj.subtract_Method2);
            // plus_Method1 and subtract_Method2 are called
            del(50, 10);
            Console.WriteLine();
            //Here again we have multicast
            del -= new delmethod(obj.plus_Method1);
            //Only subtract_Method2 is called
            del(20, 10);
            Console.ReadLine();
        }
    }
```

Event Example:

```
public delegate void MyDelegate(int a, int b);
    public class XX
    {
        public event MyDelegate MyEvent;
        public void RaiseEvent(int a, int b)
        {
            MyEvent(a, b);
            Console.WriteLine("Event Raised");
        }
        public void Add(int x, int y)
        {
            Console.WriteLine("Add Method {0}", x + y);
        }
        public void Subtract(int x, int y)
        {
            Console.WriteLine("Subtract Method {0}", x - y);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            XX obj = new XX();
            obj.MyEvent += new MyDelegate(obj.Add);
            obj.MyEvent += new MyDelegate(obj.Subtract);
            obj.RaiseEvent(20, 10);
            Console.ReadLine();
        }
    }
```

Structs and Enums

A **struct** type is a value type that is typically used to encapsulate small groups of related variables such as the coordinates of a rectangle and all. They are basically for light-weighted objects. Unlike class, **structs** in C# are value type than a reference type. It is useful if you have data that is not intended to be modified after the creation of the struct. **Structs** contain methods, properties, indexers, and so on. It cannot contain default constructors.

Eample:

```
public struct Rectangle
  {
      public int width, height;
  }
  class Program
  {
      static void Main(string[] args)
      {
          Rectangle r = new Rectangle();
          r.width = 4;
          r.height = 5;
          Console.WriteLine("Area of Rectangle is: " + (r.width * r.height));
          Console.ReadLine();
      }
  }
```

However, structs are always value types, while classes are always referenced types.Structs are quite similar to classes, but there is are a few important differences between them. Structs can not have a parameter less constructor and structs do not support inheritance. Structs are value types so in order to create an instance of struct we have to pass to it values for each of its properties.

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.

Example:

```
class Program
    {
        enum Level
        {
            Low,
            Medium,
            High
        }
        static void Main(string[] args)
        {
            Level myVar = Level.Medium;
            Console.WriteLine(myVar);
            Console.ReadLine();
        }
    }
```

Can be used inside the switch statement too.

Example:

```
class Program
    {
        enum Level
        {
            Low,
            Medium,
            High
        }
        static void Main(string[] args)
        {
            Level myVar = Level.Medium;
            switch (myVar)
            {
                case Level.Low:
                    Console.WriteLine("Low level");
                    break;
                case Level.Medium:
                    Console.WriteLine("Medium level");
                    break;
                case Level.High:
                    Console.WriteLine("High level");
                    break;
            }
        }
    }
```

Abstract class Sealed class

An abstract class is an incomplete class or special class we can't be instantiated. The purpose of an abstract class is to provide a blueprint for derived classes and set some rules what the derived classes must implement when they inherit an abstract class.

We can use an abstract class as a base class and all derived classes must implement abstract definitions. An abstract method must be implemented in all non-abstract classes using the override keyword. After overriding the abstract method is in the non-Abstract class. We can derive this class in another class and again we can override the same abstract method with it.

Example:

```
public abstract class Shape
    {
        public abstract void draw();
    }
    public class Rectangle : Shape
    {
        public override void draw()
        {
            Console.WriteLine("drawing rectangle...");
        }
    }
    public class Circle : Shape
    {
        public override void draw()
        {
            Console.WriteLine("drawing circle...");
        }
    }
    class Program
    {

        static void Main(string[] args)
        {
            Shape s;
            s = new Rectangle();
            s.draw();
            s = new Circle();
            s.draw();
        }
    }
```

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:

```
public sealed class Utilities
    {
        public int Add(int x, int y)
        {
            return x + y;
        }
    }
    public class Helper : Utilities//Gives Compilation Error
    {

    }
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
```

Interface:

Interface in C# is a blueprint of a class. It is like abstract class because all the methods which are declared inside the interface are abstract methods. It cannot have method body and cannot be instantiated.

- Interfaces specify what a class must do and not how.
- Interfaces can't have private members.
- By default all the members of Interface are public and abstract.
- The interface will always defined with the help of keyword '***interface***'.
- Interface cannot contain fields because they represent a particular implementation of data.
- *Multiple inheritance* is possible with the help of Interfaces but not with classes.

Example:

```
public interface xyz
    {
        void methodA();
        void methodB();
    }

    // interface method implementation
    class test : xyz
    {
        public void methodA()
        {
            Console.WriteLine("methodA");
        }
        public void methodB()
        {
            Console.WriteLine("methodB");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            test obj = new test();
            obj.methodA();
            obj.methodB();
        }
    }
```

Difference between Abstract Class and Interface

ABSTRACT CLASS	**INTERFACE**
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.

Partial Class

Typically, a class will reside entirely in a single file. However, in situations where multiple developers need access to the same class, then having the class in multiple files can be beneficial. The partial keywords allow a class to span multiple source files. When compiled, the elements of the partial types are combined into a single assembly.

There are some rules for defining a partial class as in the following;

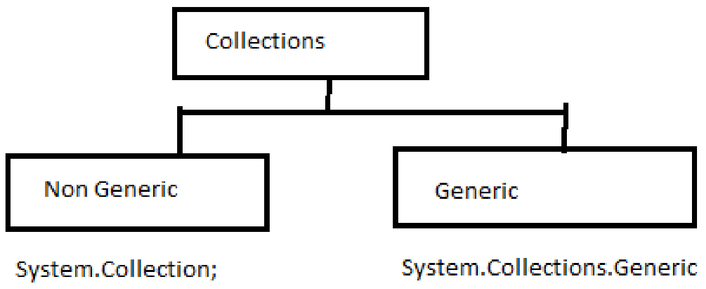
- A partial type must have the same accessibility.
- Each partial type is preceded with the "partial" keyword.
- If the partial type is sealed or abstract then the entire class will be sealed and abstract.

Example:

```
public partial class partialclassDemo
    {
        public void method1()
        {
            Console.WriteLine("method from part1 class");
        }
    }
    public partial class partialclassDemo

    {
        public void method2()
        {
            Console.WriteLine("method from part2 class");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            //partial class instance
            partialclassDemo obj = new partialclassDemo();
            obj.method1();
            obj.method2();
            Console.ReadLine();
        }
    }
```

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>();
            //or
            //we can also declare like below
            //List<int> lst=new List<int>(){1,2,3};
            lst.Add(1);
            lst.Add(2);
            lst.Add(3);
            foreach (var 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 = "Sunil Chaudhary";
            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 = @"E:\sample1.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 = @"E:\sample1.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 = @"E:\sample1.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 = @"E:\sample1.txt";
            string fileLocCopy = @"F:\sample1.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 = @"E:\sample1.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.

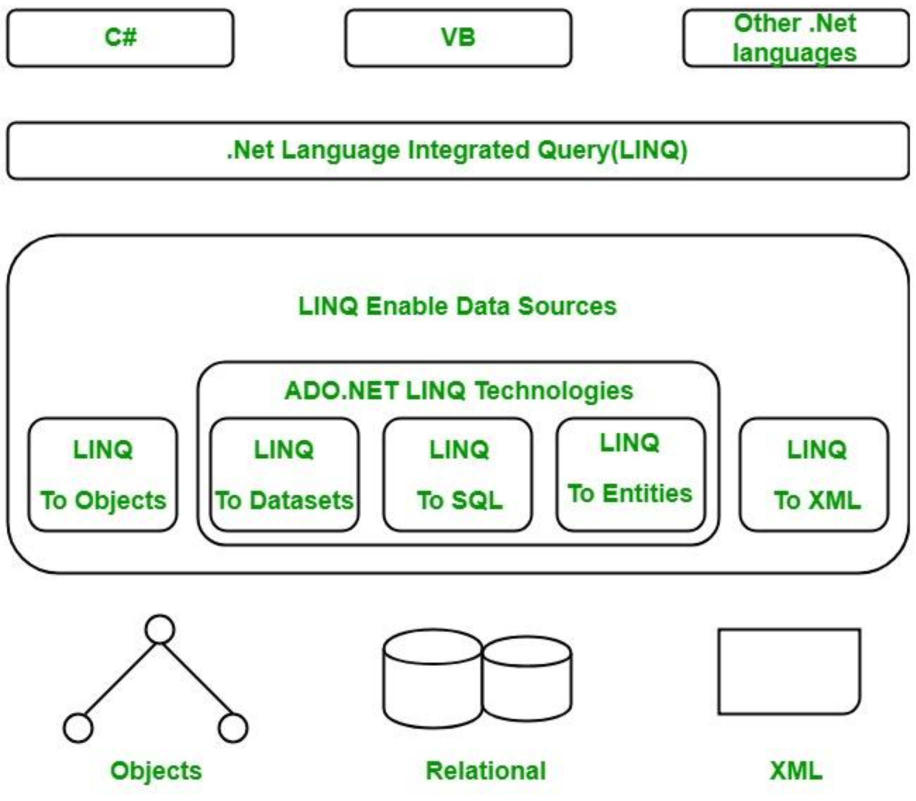


Figure 4: Architecture of LINQ

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.

Example: 1

```
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();
        }
    }
```

Example: 2

```
    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: (DivideByZeroException)

```
class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int i = 20;
                // Suspect code
                int result = i / 0;
            }
            catch (DivideByZeroException ex)
            {
                Console.WriteLine("Attempted divide by zero. {0}", ex.Message);
            }

        }

    }
```

Example: (Multiple Catch Block)

```
class Program
    {

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

                int i = 20; int j = 0;
                double result = i/ j;

                object obj = default;
                int i2 = (int)obj; // Suspect of casting error

            }
            catch (StackOverflowException ex)
            {
                Console.WriteLine("Overflow. {0}", ex.Message);
            }
            catch (DivideByZeroException ex)
            {
                Console.WriteLine("Attempted divide by zero. {0}", ex.Message);
            }
            catch (InvalidCastException ex)
            {
                Console.WriteLine("Invalid casting. {0}", ex.Message);
            }
Catch(Exception ex)
{
}
Finally
{}

            Console.ReadLine();

        }
```

Attributes

Attributes are used in C# to convey declarative information or metadata about various code elements such as methods, assemblies, properties, types, etc. Attributes are added to the code by using a declarative tag that is placed using square brackets ([ ]) on top of the required code element.

[[*target*:]? *attribute-name* (

positional-param+ |

[*named-param* = *expr*]+ |

positional-param+, [*named-param* = *expr*]+)?]

Attribute Classes

An attribute is defined by a class that inherits (directly or indirectly) from the abstract class `System.Attribute`. When specifying an attribute to an element, the attribute name is the name of the type. By convention, the derived type name ends in `Attribute`, although specifying the suffix is not required when specifying the attribute.

In this example, the `Foo` class is specified as serializable using the `Serializable` attribute:

```
[Serializable]
public class Test {...}
```

Named and Positional Parameters:

Attributes can take parameters, which can specify additional information on the code element beyond the mere presence of the attribute.

In this example, the class `Test` is specified as obsolete using the `Obsolete` attribute. This attribute allows parameters to be included to specify both a message and whether the compiler should treat the use of this class as an error:

```
[Obsolete("Use Bar class instead", IsError=true)]

public class Test {...}
```

Attribute parameters fall into one of two categories: positional and named. In the preceding example, `Use Bar class instead` is a positional parameter and `IsError=true` is a named parameter.

The positional parameters for an attribute correspond to the parameters passed to the attribute type's public constructors. The named parameters for an attribute correspond to the set of public read-write or write-only instance properties and fields of the attribute type.

Attribute Targets

Implicitly, the target of an attribute is the code element it immediately precedes, such as with the attributes we have covered so far. Sometimes it is necessary to explicitly specify that the attribute applies to particular target.

Here is an example of using the `CLSCompliant` attribute to specify the level of CLS compliance for an entire assembly:

[assembly:CLSCompliant(true)]

Specifying Multiple Attributes

Multiple attributes can be specified for a single code code element. Each attribute can be listed within the same pair of square brackets (separated by a comma), in separate pairs of square brackets, or in any combination of the two.

Consequently, the following three examples are semantically identical:

```
[Serializable,Obsolete, CLSCompliant(false)]
public class Test {...}
[Serializable]
[Obsolete]
[CLSCompliant(false)]
public class Test {...}
```

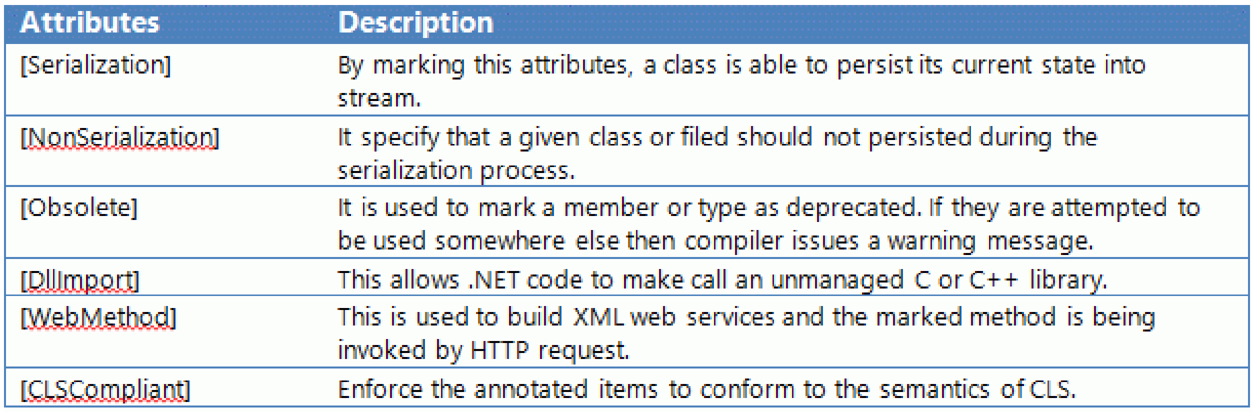
There are two types of Attributes implementations provided by the *.NET Framework* are:

1. Predefined Attributes
2 Custom Attributes

Properties of Attributes:

- Attributes can have arguments just like methods, properties, etc. can have arguments.
- Attributes can have zero or more parameters.
- Different code elements such as methods, assemblies, properties, types, etc. can have one or multiple attributes.
- Reflection can be used to obtain the metadata of the program by accessing the attributes at run-time.
- Attributes are generally derived from the **System.Attribute Class**.

Predefined attributes are those attributes that are a part of the .NET Framework Class Library and are supported by the C# compiler for a specific purpose. Some of the predefined attributes that are derived from the *System.Attribute* base class are given as follows:

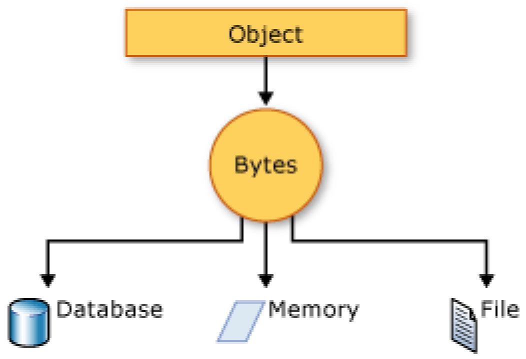


Attributes	Description
[Serialization]	By marking this attributes, a class is able to persist its current state into stream.
[NonSerialization]	It specify that a given class or filed should not persisted during the serialization process.
[Obsolete]	It is used to mark a member or type as deprecated. If they are attempted to be used somewhere else then compiler issues a warning message.
[DllImport]	This allows .NET code to make call an unmanaged C or C++ library.
[WebMethod]	This is used to build XML web services and the marked method is being invoked by HTTP request.
[CLSCompliant]	Enforce the annotated items to conform to the semantics of CLS.

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. . Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called **deserialization**.

How serialization works

This illustration shows the overall process of serialization:



The object is serialized to a stream that carries the data. The stream may also have information about the object's type, such as its version, culture, and assembly name. From that stream, the object can be stored in a database, a file, or memory

Uses for serialization

Serialization allows the developer to save the state of an object and re-create it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions such as:

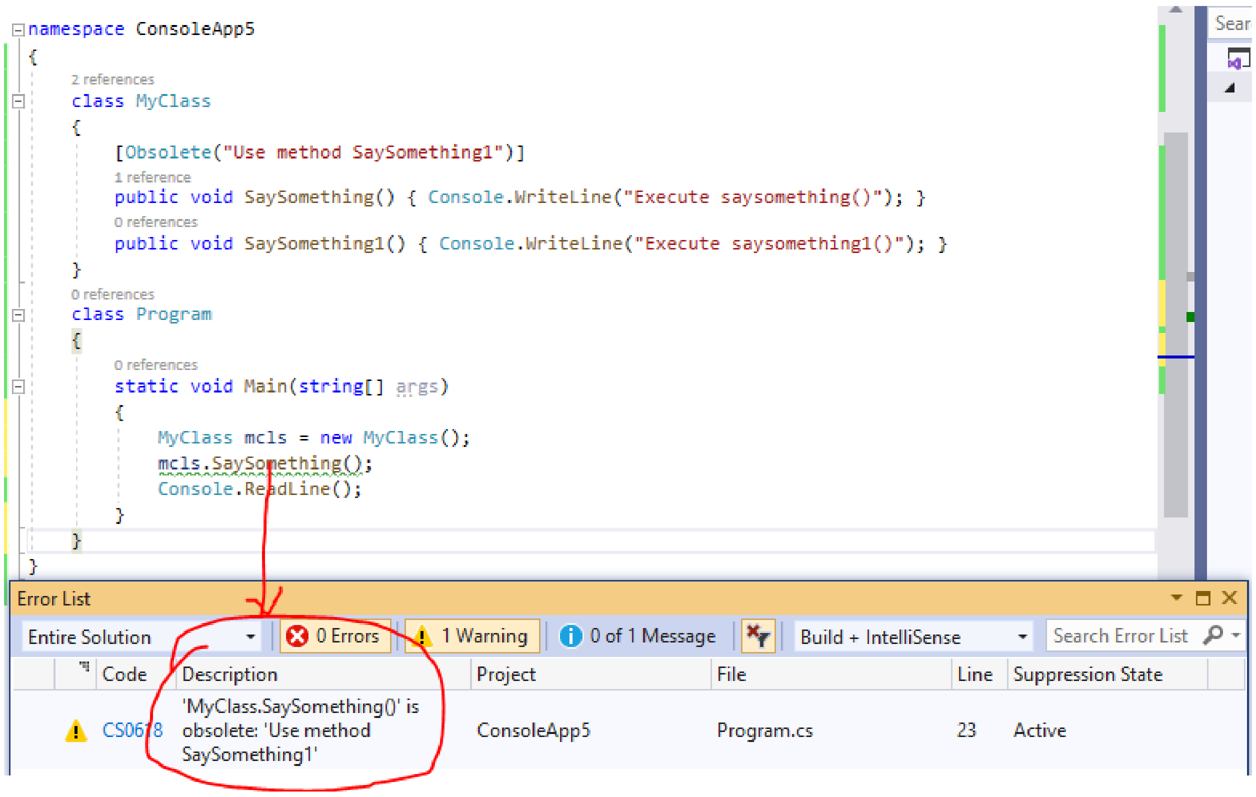
- Sending the object to a remote application by using a web service
- Passing an object from one domain to another
- Passing an object through a firewall as a JSON or XML string
- Maintaining security or user-specific information across applications

Example:

```
namespace ConsoleApp5
{
    [Serializable()]
    public class UserDetails
    {
        public int userId { get; set; }
        public string userName { get; set; }
        public string location { get; set; }
        public UserDetails(int id, string name, string place)
        {
            userId = id;
            userName = name;
            location = place;
        }
        public void GetDetails()
        {
            Console.WriteLine("UserId: {0}", userId);
            Console.WriteLine("UserName: {0}", userName);
            Console.WriteLine("Location: {0}", location);
        }

    }
    class Program
    {
        static void Main(string[] args)
        {
            UserDetails ud = new UserDetails(1, "Suresh", "Hyderabad");
            Console.WriteLine("Before serialization the object contains: ");
            ud.GetDetails();
            string fpath = @"E:\Test.txt";
            // Check if file exists
            if (File.Exists(fpath))
            {
                File.Delete(fpath);
            }
            //Opens a file and serializes the object into it in binary format.
            Stream stream = File.Open(fpath, FileMode.Create);
            BinaryFormatter bf = new BinaryFormatter();
            bf.Serialize(stream, ud);
            stream.Close();
            Console.WriteLine("\nSerialization Successful");
            Console.ReadLine();
        }

    }

}
```

Use of Obsolete:

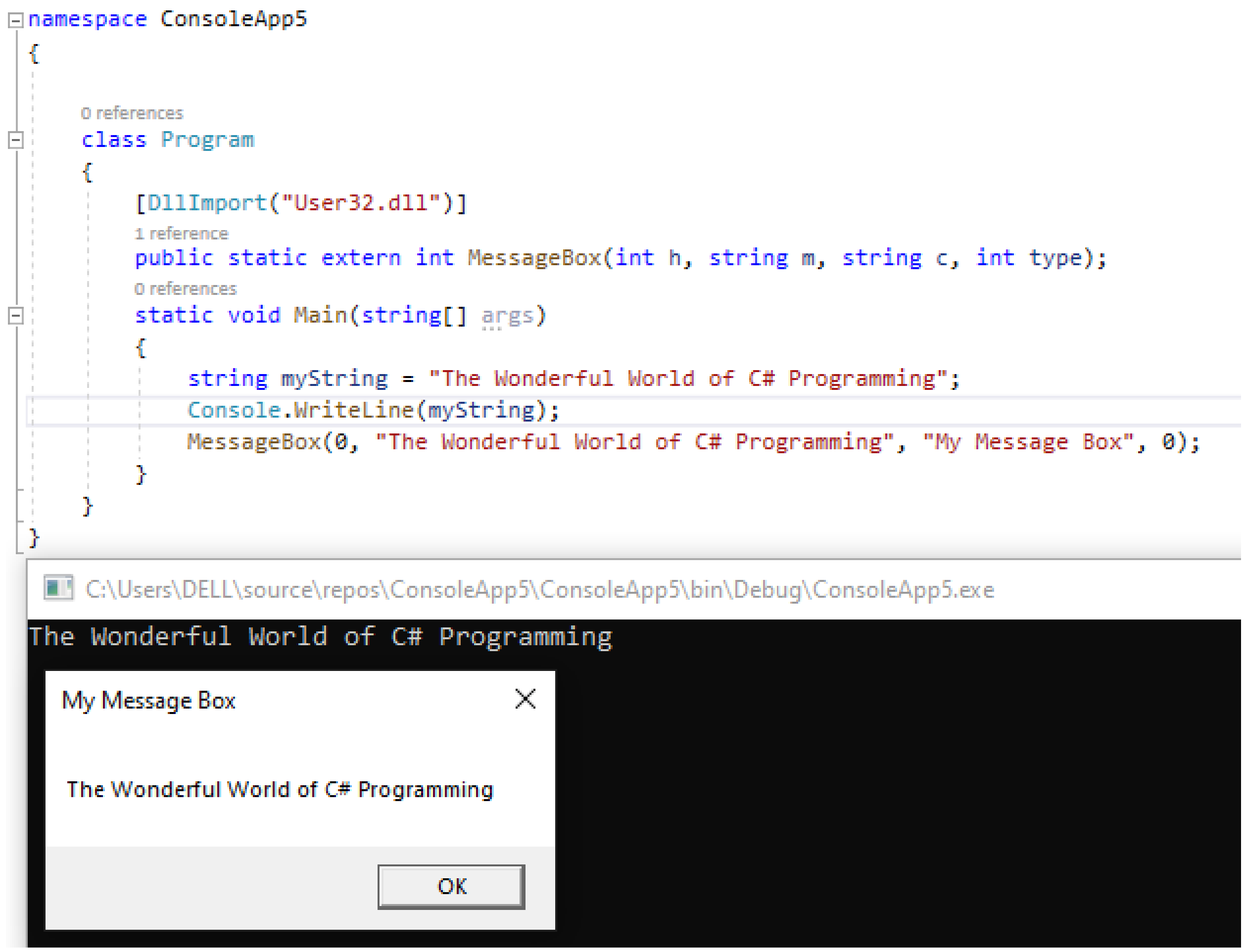
Suppose I made a class named MyClass having the method SaySomething() and you implemented this class and the method in the entire project. Then the requirement comes that there is no need to implement the class MyClass or the method called SaySomething. But you need to implement a method called SaySomething1(). In this scenario we use the Obsolete attribute.



```
namespace ConsoleApp5
{
    2 references
    class MyClass
    {
        [Obsolete("Use method SaySomething1")]
        1 reference
        public void SaySomething() { Console.WriteLine("Execute saysomething()"); }
        0 references
        public void SaySomething1() { Console.WriteLine("Execute saysomething1()"); }
    }
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            MyClass mcls = new MyClass();
            mcls.SaySomething();
            Console.ReadLine();
        }
    }
}
```

	Code	Description	Project	File	Line	Suppression State
⚠	CS0618	'MyClass.SaySomething()' is obsolete: 'Use method SaySomething1'	ConsoleApp5	Program.cs	23	Active

DLLImport(Here is an example of using DLLImport to call a Win32 function) dllexport enable interop with DLL files. We can use a C++ DLL dynamic link library, or a custom legacy DLL—even one we can't rewrite but have the ability to modify.



```
namespace ConsoleApp5
{
    0 references
    class Program
    {
        [DllImport("User32.dll")]
        1 reference
        public static extern int MessageBox(int h, string m, string c, int type);
        0 references
        static void Main(string[] args)
        {
            string myString = "The Wonderful World of C# Programming";
            Console.WriteLine(myString);
            MessageBox(0, "The Wonderful World of C# Programming", "My Message Box", 0);
        }
    }
}
```

```
C:\Users\DELL\source\repos\ConsoleApp5\ConsoleApp5\bin\Debug\ConsoleApp5.exe
The Wonderful World of C# Programming
```

My Message Box

The Wonderful World of C# Programming

Custom attributes can be created in C# for attaching declarative information to methods, assemblies, properties, types, etc. in any way required. This increases the extensibility of the .NET framework.

Steps for creating Custom Attributes:

- Define a custom attribute class that is derived from *System.Attribute* class.
- The custom attribute class name should have the suffix **Attribute**.
- Use the attribute AttributeUsage to specify the usage of the custom attribute class created.
- Create the constructor and the accessible properties of the custom attribute class.

CLS Compliant

If you are writing .Net classes, which will be used by other .Net classes irrespective of the language they are implemented, then your code should conform to the CLS [Common Language Specification]. This means that your class should only expose features that are common across all .Net languages. The following are the basic rules that should be followed when writing a CLS complaint C# code.

For marking an entire assembly as CLS compliant the following syntax is used

```
using System;
[assembly: CLSCompliant(true)]
```

For marking a particular method as CLS compliant the following syntax is used

```
[CLSCompliant(true)]
public void MyMethod()
```

Example:

```
[CLSCompliant(true)]
    public class Test
    {
        public void MyMethod()
        {
        }
        //error because methods differ only in their case
        public void MYMETHOD()
        {
        }
        static void Main()
        {
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
```

Compiling the above code will result in the following error:

'Test' cannot be marked as CLS compliant because the assembly is not marked as compliant

WebMethod:

The **WebMethod** attribute enables the method to be called through the web service.

This attribute is used to customize the behavior of your Web Service. This attribute holds up several properties like enable session, description, cacheduration, and bufferresponse.

Example:

```
[WebMethod(CacheDuration = 10)]
 public string GetTime()
 {
     return DateTime.Now.ToString("T");
 }
```

Asynchronous Programming

With visual studio 2012, now developers can use ‘await’ and ‘async’ keywords. Async/await methods are sequential in nature, but asynchronous when compiled and executed. The added benefit of using the ‘async’ keyword is that it provides a simpler way to perform potentially long-running operations without blocking the callers thread. The caller thread/method can continue its work without waiting for this asynchronous method to complete its job. This approach reduces additional code development effort. Each ‘async’ modifier/method should have at least one ‘await’.

The asynchronous model works best when:

- There is a large number of tasks, so there is probably always at least one task that can make progress;
- Tasks perform many I/O operations, which causes the synchronous program to spend a lot of time in blocking mode when other tasks can be performed;
- Tasks are largely independent of each other, so there is no need for intertask interaction (and therefore, no need to wait for one task from another).

Async

This keyword is used to qualify a function as an asynchronous function. In other words, if we specify the async keyword in front of a function then we can call this function asynchronously. Have a look at the syntax of the asynchronous method.

```
public static async void CallProcess()
 {
     await LongProcess();
    Console.WriteLine("Long Process finish");
 }
```

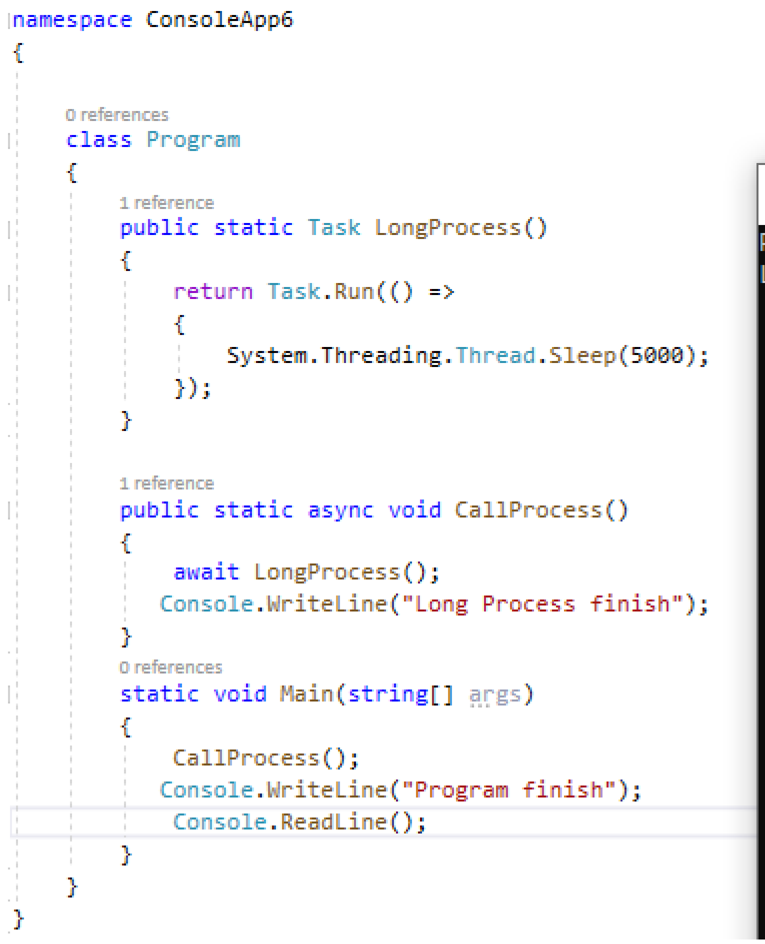
Await

This keyword is used when we want to call any function asynchronously. Have a look at the following example to understand how to use the await keyword.

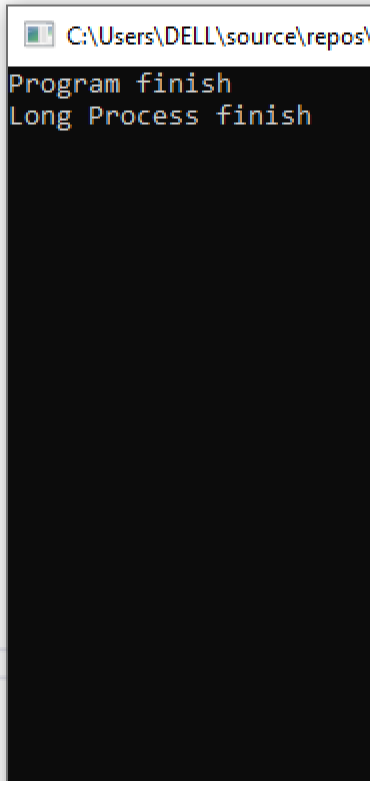
```
public static Task LongProcess()
```

```
        {
            return Task.Run(() =>
            {
                System.Threading.Thread.Sleep(5000);
            });
        }
```

Example:



```
namespace ConsoleApp6
{
    0 references
    class Program
    {
        1 reference
        public static Task LongProcess()
        {
            return Task.Run(() =>
            {
                System.Threading.Thread.Sleep(5000);
            });
        }

        1 reference
        public static async void CallProcess()
        {
            await LongProcess();
           Console.WriteLine("Long Process finish");
        }
        0 references
        static void Main(string[] args)
        {
            CallProcess();
           Console.WriteLine("Program finish");
            Console.ReadLine();
        }
    }
}
```



C:\Users\DELL\source\repos\

```
Program finish
Long Process finish
```