C# is an object-oriented, type-safe, and managed language that is compiled by .Net framework to generate Microsoft Intermediate Language.

**An object** is an instance of a class through which we access the methods of that class. "New" keyword is used to create an object. A class that creates an object in memory will contain the information about the methods, variables, and behaviour of that class.

The two most fundamental core concepts on which OO has been built upon in C# are **this** pointer and **Dynamic Dispatch.**

**Internal** is the default Modifier of Class if no modifier is specified.

**By default a Method is Private** is no modifier is specified.

For variables of class **types** and other reference **types**, this **default** value is null . However, since **structs** are value **types** that cannot be null , the **default** value of a **struct** is the value produced by setting all value **type** fields to their **default** value and all reference **type** fields to null .

A **structure** type **can**'t inherit from other **class** or **structure** type and it **can**'t be the base of a **class**. However, a **structure** type **can** implement interfaces. **You can**'t **declare** a finalizer within a **structure** type.

Default value of bool is false.

**Static class**

* A static class can have only static member and static methods.
* Static class can have only one constructor which is static constructor.
* Static class is sealed by default you cannot put the sealed keyword with static it will give compile time error.
* Static class cannot implement interface.
* We cannot create instance of static class.

**Static Constructor**

* In any class we can have only one static constructor.
* Static constructor must be parameter less.
* Access modifiers are NOT allowed to static constructor.
* It is used to initialize static members of a class.
* It is called by CLR, so it is not certain when it is called.
* It cannot be explicitly called by code.

**Private Constructor**

* Private Constructor is used to restrict a class from being inherited.
* We can have both private and public constructor in a class.
* If a class have public constructor then that class can be inherited regardless of private constructor.

**Sealed class**

* We cannot inherit from the sealed class.
* A sealed class can have multiple constructor.
* We can create object of sealed class.

**Sealed class Constructor**

* Sealed class constructors can be parameterized or parameter less.

**Sealed Keyword**

* To prevent a method from being further override in derived classes.

**Basic:**

Pillars of OOPs (E-API)

1. Encapsulation
2. Abstraction
3. Polymorphism
4. Inheritance

- We cannot explicitly define a class as private directly in namespace it can be a subclass and private class cannot be inherited.

- By default a method is private, private method can be used only inside the same class.

- If a class has private constructor, it cannot be inherited.

**Method Overloading:**

* Deals with the methods inside the same class.
* Method with Same Name but Different types of arguments or number of arguments or order of arguments.
* Overloading has nothing to do with the return type of method.

**Method Overriding:**

* Deals with the methods in parent and child class.
* Method in base class is virtual and in child class it is decorated with override keyword.
* Methods must have same signature and return type in both parent and child classes.

public class AA

{

public virtual void GetName()

{

Console.WriteLine("Hello Base");

}

public virtual void GetName(string name) // Overloading GetName()

{

Console.WriteLine("From Base :" + name);

}

}

public class BB : AA

{

//Overriding the base class method GetName()

public override void GetName() //public new void GetName()

{

Console.WriteLine("Hello Child");

}

}

public class mainClass

{

static void Main(string[] args)

{

AA ob = new AA();

ob.GetName();

AA obj1 = new BB();

obj1.GetName();

BB obj = new BB();

obj.GetName();

//BB obj2 = new AA(); //Child class cannot have the object of parent class.

Console.ReadLine();

}

}

**Output :**

1. **If GetName() method in BB is decorated with override**

**Hello Base**

**Hello Child**

**Hello Child**

1. **If GetName() method in BB is decorated with new**

**Hello Base**

**Hello Base**

**Hello Child**

**Static Constructor**

In c#, Static Constructor **is useful to perform a particular action only once throughout the application**. If we declare a constructor as static, then **it will be invoked only once** irrespective of the number of class instances and **it will be called automatically before the first instance is created**.

Generally, **in c# the static constructor will not accept any access modifiers and parameters**. In simple words, we can say it is parameterless.

The following are the properties of static constructor in c# programming language.

* Static constructor in c# won’t accept any parameters and access modifiers.
* The static constructor will invoke automatically, whenever we create the first instance of a class.
* The static constructor will be invoked by CLR so we don’t have a control on static constructor execution order in c#.
* In c#, only one static constructor is allowed to create in a class.

using System;

namespace Tutlane

{

    class User

    {

        // Static Constructor

        static User()

        {

            Console.WriteLine("I am Static Constructor");

        }

// Default Constructor

        public User()

        {

            Console.WriteLine("I am Default Constructor");

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            // Both Static and Default constructors will invoke for first instance

            User user = new User();

            // Only Default constructor will invoke

            User user1 = new User();

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

            Console.ReadLine();

        }

    }

}

**Output:**

I am Static Constructor

I am Default Constructor

I am Default Constructor

Press Enter Key to Exit.

**Abstract Class:**

* It is best to use when we have some common methods/functionality for the child classes.
* Put all the common functionality in simple methods and all the methods whose implementation is different but name is same, make them Abstract method.
* An abstract class should be a base class.
* An abstract class can inherit abstract/non-abstract class.
* It prevents user to create object of base class.
* It is mandatory to implement all abstract methods.
* Abstract class can be partially created.
* An abstract class cannot be a sealed or static class because we need to inherit abstract class.
* Declaration of abstract methods is only allowed in abstract classes.
* The access modifier of the abstract method must be public and must be same in both the abstract class and its derived class.
* Can have abstract method and concrete methods.
  1. These methods can be public, protected or private.
  2. Private methods in abstract class can be called only from abstract class constructor.
  3. Public void method of abstract class can only be used by inherited class while public void method of normal class can be access directly by creating class object.
* Can have constructor, but cannot create object of abstract class.
  1. Abstract class constructor can be invoked from the child class.

**Interface:**

* It is best to use when we have only unique methods/functionality for the child classes.
* Provides the feature of multiple inheritance and avoids the Diamond problem that we have in c++.
* All members are by default public, static and final.
* Interface can also be partially created.
* Interface cannot have instance field. For e.g int a; is not allowed but int a {get;set;} is allowed.
* The class which implements the interface will be responsible for implementing all of its methods.
* Cannot create constructor of interface, so we cannot create object of it.

public abstract partial class Test

{

public abstract void Display();

}

public abstract partial class Test

{

public abstract void Show();

}

public partial interface ITest

{

void Hello();

}

public partial interface ITest

{

void Hi();

}

public class CheckPartial : Test, ITest

{

public override void Display()

{

throw new NotImplementedException();

}

public void Hello()

{

throw new NotImplementedException();

}

public void Hi()

{

throw new NotImplementedException();

}

public override void Show()

{

throw new NotImplementedException();

}

}

**Difference between Abstract Class and Virtual Class**

* A virtual class can be instantiated directly, whereas an abstract class cannot.
* Both virtual and abstract classes can contain virtual methods (virtual methods can have a default implementation that is inherited by child classes, whereas abstract methods can only be signatures, and must be implemented in child classes)
* Only abstract classes can contain abstract methods

**Difference between Abstract method and Virtual method**

* Virtual methods have an implementation and provide the derived classes with the option of overriding it.
* Abstract methods do not provide an implementation and force the derived classes to override the method. So, abstract methods have no actual code in them, and subclasses HAVE TO override the method.

**Delegate:**

**Delegate is a class**, which is used to create and invoke delegates at runtime. A delegate in C# is similar to a function pointer in C or C++. It's a new type of object in C#. It is a reference type variable.

It is a special type of user-defined variable that is declared globally like a class.

A delegate provides a template for a method, likes an interface provides a template for a class.

A delegate provides necessary information for a method which includes a return type, no argument or one or more argument.

It is a method template which used to implement the concept of function pointer.

**Types of Delegate:**

1. **Singlecast Delegate:** Delegates that **represent only a single function** are known as single cast delegate.

namespace TestLogics

{

delegate int Operation(int x, int y);

public class SingleDelegate

{

public static int Add(int a,int b)

{

return a + b;

}

public static int Sub(int a, int b)

{

return a - b;

}

}

class TestDelegate

{

public static void Main(string[] args)

{

Operation opr1 = new Operation(SingleDelegate.Add);

Operation opr2 = new Operation(SingleDelegate.Sub);

int ans1 = opr1(200, 100);

int ans2 = opr2(200, 100);

Console.WriteLine($"Addition of 200 and 100 is : {ans1}");

Console.WriteLine($"Substraction of 200 and 100 is : {ans2}");

Console.ReadLine();

}

}

}

1. **Multicast Delegate:** delegates that **represent more than one function** are called Multicast delegate.

public delegate void MathDelegate(int No1, int No2);

public class Program

{

public static void Add(int x, int y)

{

Console.WriteLine("THE SUM IS : " + (x + y));

}

public static void Sub(int x, int y)

{

Console.WriteLine("THE SUB IS : " + (x - y));

}

public void Mul(int x, int y)

{

Console.WriteLine("THE MUL IS : " + (x \* y));

}

public void Div(int x, int y)

{

Console.WriteLine("THE DIV IS : " + (x / y));

}

static void Main(string[] args)

{

Program p = new Program();

MathDelegate del1 = new MathDelegate(Add);

MathDelegate del2 = new MathDelegate(Program.Sub);

MathDelegate del3 = p.Mul;

MathDelegate del4 = new MathDelegate(p.Div); ;

//In this example del5 is a multicast delegate. We can use +(plus)

// operator to chain delegates together and -(minus) operator to remove.

MathDelegate del5 = del1 + del2 + del3 + del4;

del5.Invoke(20, 5);

Console.WriteLine();

del5 **-**= del2;

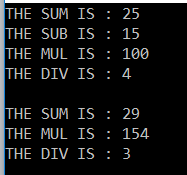
del5(22, 7);

Console.ReadKey();

}

}

**OUTPUT:**



* A multicast delegate invokes the methods in the invocation list, in the same order in which they are added.
* If the delegate has a return type other than void and if the delegate is a multicast delegate then only the value of the last invoked method will be returned.

namespace MulticastDelegateDemo

{

**// Deletegate's return type is int**

public delegate int SampleDelegate();

public class Program

{

static void Main()

{

SampleDelegate del = new SampleDelegate(MethodOne);

del += MethodTwo;

// The ValueReturnedByDelegate will be 2, returned by the MethodTwo(),

// as it is the last method in the invocation list.

int ValueReturnedByDelegate = del();

Console.WriteLine("Returned Value = {0}", ValueReturnedByDelegate);

Console.ReadKey();

}

// This method returns one

public static int MethodOne()

{

return 1;

}

// This method returns two

public static int MethodTwo()

{

return 2;

}

}

}

**Output:**

**Returned Value = 2**

* Along the same lines, if the delegate has an out parameter, the value of the output parameter will be the value assigned by the last method.

namespace MulticastDelegateDemo

{

**// Deletegate has an int output parameter**

public delegate void SampleDelegate(out int Integer);

public class Program

{

static void Main()

{

SampleDelegate del = new SampleDelegate(MethodOne);

del += MethodTwo;

// The ValueFromOutPutParameter will be 2, initialized by MethodTwo(),

// as it is the last method in the invocation list.

int ValueFromOutPutParameter = -1;

del(out ValueFromOutPutParameter);

Console.WriteLine("Returned Value = {0}", ValueFromOutPutParameter);

Console.ReadKey();

}

// This method sets ouput parameter Number to 1

public static void MethodOne(out int Number)

{

Number = 1;

}

// This method sets ouput parameter Number to 2

public static void MethodTwo(out int Number)

{

Number = 2;

}

}

}

**Output:**

**Returned Value = 2**

**What will be the output of following program.**

class A

{

public virtual void Func1()

{

Console.WriteLine("A.Func1");

}

}

class B : A

{

public override void Func1()

{

Console.WriteLine("B.Func1");

}

}

class C : B

{

public void Func1()

{

Console.WriteLine("C.Func1");

}

}

class D : C

{

public void Func1()

{

Console.WriteLine("D.Func1");

}

}

A ab = new B();

A ac = new C();

C cb = new B(); // Error

A ad = new D();

C cd = new D();

ab.Func1(); //B.Func1

ac.Func1(); //B.Func1

cb.Func1(); // Error

ad.Func1(); //B.Func1

cd.Func1(); //C.Func1

**What is the use of Using block**

* The C# using statement defines a boundary for the object outside of which, the object is automatically destroyed.
* Provides a convenient syntax that ensures the correct use of IDisposable objects.
* The "using" statement allows you to specify multiple resources in a single statement.

using (A aObj = new A(), aObj1 = new A())//Objects should be of same type

{

}

* The object could also be created outside the "using" statement. The objects specified within the using block must implement the IDisposable interface.

Employee emp = new Employee();

using (emp)

{

emp.Use(); // Use emp ojbect

}//The compiler will dispose the emp object now

using (Employee emp = new Employee)

{

emp.Use(); // Use emp ojbect

}//The compiler will dispose the emp object now

* The objects specified within the using block must implement the IDisposable interface. The framework invokes the Dispose method of objects specified within the "using" statement when the block is exited.

**C# Language Preprocessor Directives**

The preprocessor directives give instruction to the compiler to preprocess the information before actual compilation starts.

All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not statements, so they do not end with a semicolon (;).

**Main use of directives are**

**Conditional compilation:** Using special preprocessing directives, you can include or exclude parts of the program according to various conditions.

**e.g. -1**

#define TEST

using System;

#if (TEST) // or #if TEST

Console.WriteLine("TEST is defined");

Console.WriteLine("TEST is defined1");

#else

Console.WriteLine("TEST is not defined");

#endif

**e.g. -2**

#define DEBUG

#define VC\_V6

using System;

#if (DEBUG && !VC\_V6)

Console.WriteLine("DEBUG is defined");

#elif (!DEBUG && VC\_V6)

Console.WriteLine("VC\_V6 is defined");

#elif (DEBUG && VC\_V6)

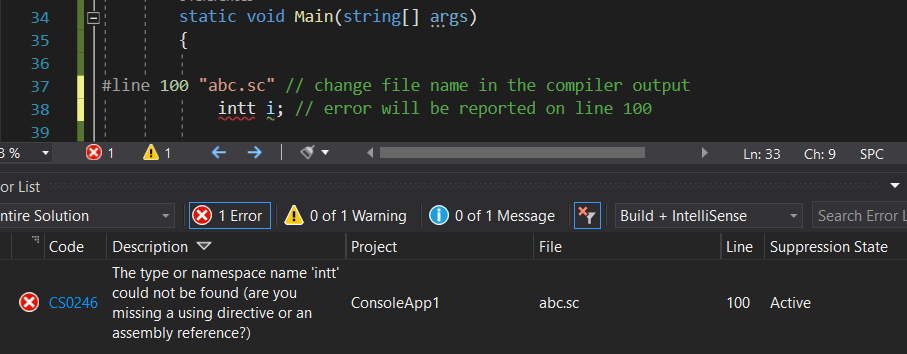
Console.WriteLine("DEBUG and VC\_V6 are defined");

#else

Console.WriteLine("DEBUG and VC\_V6 are not defined");

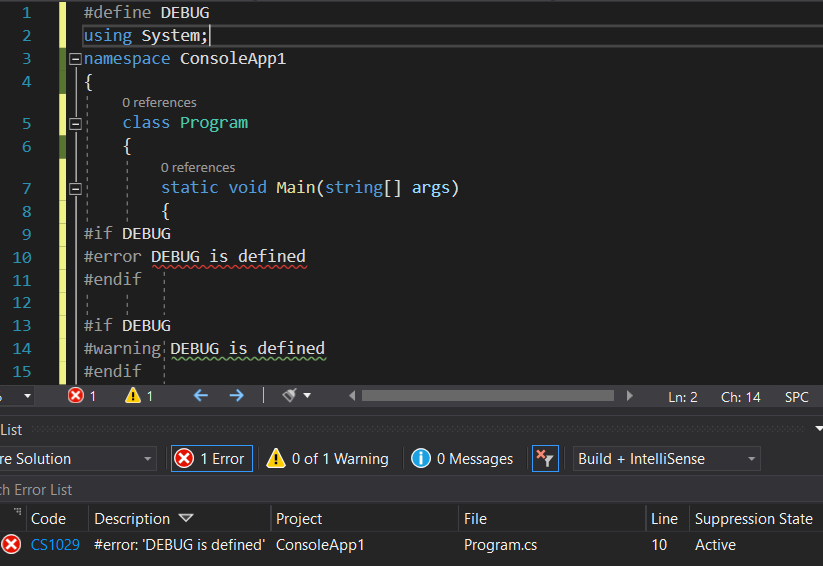
#endif

**Line control:** If you use a program to combine or rearrange source files into an intermediate file, which is then compiled, you can use line control to inform the compiler of where each source line originally came from.

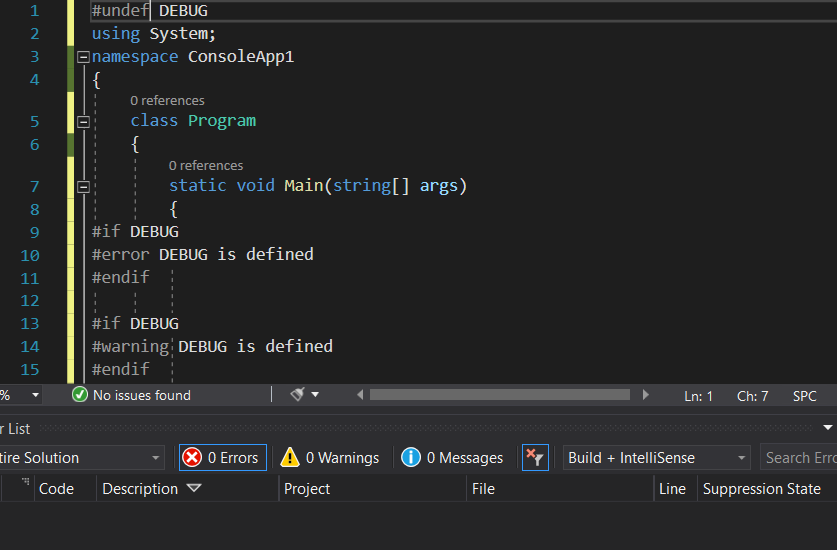


**Error and Warning reporting:** The directive '#error' causes the preprocessor to report a fatal error and the directive '#warning' is like the directive '#error', but causes the preprocessor to issue a warning and continue preprocessing.

**With #define**



**With #undef**



The C# language's preprocessor directives are as follows

|  |  |
| --- | --- |
| **Sr.No.** | **Preprocessor Directive & Description** |
| 1 | **#define**  It defines a sequence of characters, called symbol. |
| 2 | **#undef**  It allows you to undefine a symbol. |
| 3 | **#if**  It allows testing a symbol or symbols to see if they evaluate to true. |
| 4 | **#else**  It allows to create a compound conditional directive, along with #if. |
| 5 | **#elif**  It allows creating a compound conditional directive. |
| 6 | **#endif**  Specifies the end of a conditional directive. |
| 7 | **#line**  It lets you modify the compiler's line number and (optionally) the file name output for errors and warnings. |
| 8 | **#error**  It allows generating an error from a specific location in your code. |
| 9 | **#warning**  It allows generating a level one warning from a specific location in your code. |
| 10 | **#region**  It lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor. |
| 11 | **#endregion**  It marks the end of a #region block. |

CI/CD - Continuous Integration/Continuous Delivery

C4 Model

- Level 1: System **Context** diagram : Review the software system

- Level 2: **Container** diagram : on which platfrom you can create this software

- Level 3: **Component** diagram : components of software like modules

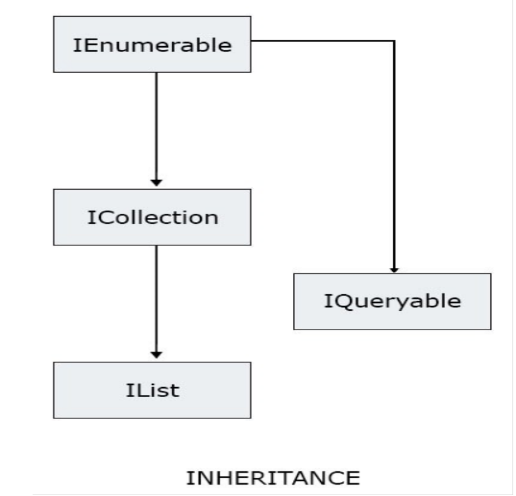
- Level 4: **Code** : implementation of components

**Difference between Struct and Class**

* A structure is a value type so it is stored on the stack, but a class is a reference type and is stored on the heap.
* A structure doesn't support inheritance, and polymorphism, but a class supports both.
* By default, all the struct members are public but class members are by default private in nature.
* As a structure is a value type, we can't assign null to a struct object, but it is not the case for a class.

**Difference between IEnumerable and IList:**

System.Collections

**Extern Keyword in c#**

* Using the extern modifier means that the method is implemented outside the C# code, whereas using the abstract modifier means that the method implementation is not provided in the class.
* External methods are implemented externally, typically using a language other than C#.
* Because an external method declaration provides no actual implementation, the method-body of an external method simply consists of a semicolon.
* An external method may not be generic.
* The extern modifier is typically used in conjunction with a DllImport attribute, allowing external methods to be implemented by DLLs (Dynamic Link Libraries).

class ExternTest

{

[DllImport("User32.dll", CharSet=CharSet.Unicode)]

public static extern int MessageBox(IntPtr h, string m, string c, int type);

static int Main()

{

string myString;

Console.Write("Enter your message: ");

myString = Console.ReadLine();

return MessageBox((IntPtr)0, myString, "My Message Box", 0);

}

}

**Cohesion:** Within a Module

**Coupling:** Between Moudule

**Serilialization**

When we want to transport an object through a network, then we have to convert the object into a stream of bytes. The process of converting an object into a stream of bytes is called Serialization. For an object to be serializable, it should implement ISerialize Interface.

**De-serialization** is the reverse process of creating an object from a stream of bytes.

**Difference between “throw” and “throw ex”**

throw : If we use "throw" statement, it preserve original error stack information. ... throw ex : If we use "throw ex" statement, stack trace of exception will be replaced with a stack trace starting at the re-throw point. It is used to intentionally hide stack trace information.

Default Interface implementation is available in C# v8.0 or above.

#### Hashtable Vs Dictionary

|  |  |
| --- | --- |
| Hashtable | Dictionary |
| A Hashtable is a non-generic collection. | A Dictionary is a generic collection. |
| Hashtable is defined under System.Collections namespace. | Dictionary is defined under System.Collections.Generic namespace. |
| In Hashtable, you can store key/value pairs of the same type or of the different type. | In Dictionary, you can store key/value pairs of same type. |
| In Hashtable, there is no need to specify the type of the key and value. | In Dictionary, you must specify the type of key and value. |
| The data retrieval is slower than Dictionary due to boxing/ unboxing. | The data retrieval is faster than Hashtable due to no boxing/ unboxing. |
| In Hashtable, if you try to access a key that doesn’t present in the given Hashtable, then it will give null values. | In Dictionary, if you try to access a key that doesn’t present in the given Dictionary, then it will give error. |
| It is thread safe. | It is also thread safe but only for public static members. |
| It doesn’t maintain the order of stored values. | It always maintain the order of stored values. |

**What will be the output of following program:**

public class A

{

static A()//2

{

Console.WriteLine("Static A");

}

public A()//3

{

Console.WriteLine("A");

}

}

public class B : A

{

static B() //1

{

Console.WriteLine("Static B");

}

public B()//4

{

Console.WriteLine("B");

}

}

public class C : B

{

public static void Main()

{

C obj = new C();

Console.Read();

}

}

**Output:**

Static B

Static A

A

B