

**C# (C Sharp) Tutorial**

Provided by

**Muhammad Farhan**

Contact No: +92316-2859445

Email: [mohammadfarhan44500@gmail.com](mailto:mohammadfarhan44500@gmail.com)

GitHub: <https://github.com/muhammadfarhandevelper>

**OOP (Object Oriented Programming) C#**

**Class and Objects:**

* C# program are composed of classes that represents the entities of the program which also include code to instantiate the classes as objects.
* When the program runs, objects are created for the classes and they many interact with each other to provide the functionalities of the program.
* An object is a tangible entity such as a car, a table, or a briefcase.
* Every object has some characteristics and is capable of performing certain actions.
* The concept of objects in the real world can also be extended to the programming world. An object in a programming language has a unique identity, state and behavior.
* An object has various features that can describe it which could be the company name, model, price, mileage, and so on.
* An object stores its identity and state in fields (also called variables) and exposes its behavior though methods.

**Example:**

class student

{

public int id;

public string name;

public int age;

}

class Program

{

static void Main(string[] args)

{

student st1 = new student();

st1.id = 1;

st1.name = "Farhan Ali";

st1.age = 23;

Console.WriteLine(st1.id + " Name : " + st1.name + " Age : " + st1.age);

} }

**Another Example:**

class student

{

public int id;

public string name;

public int age;

public student(int id, string name, int age)

{

this.id = id;

this.name = name;

this.age = age;

}

}

class Program

{

static void Main(string[] args)

{

student st1 = new student(1,"Ali rehman",23);

Console.WriteLine(st1.id + " Name : " + st1.name + " Age : " + st1.age);

}

}

**Another Example:**

class student

{

public int id;

public string name;

public void setid(int id)

{

this.id = id;

}

public int getid()

{

return this.id;

}

}

class Program

{

static void Main(string[] args)

{

student st1 = new student();

st1.setid(12);

Console.WriteLine(st1.getid());

}

}

**Note: You can set me values in one method.**

**Constructor:**

* A class constructor is a special member function of a class that is executed whenever we create new objects of that class.
* A constructor has exactly the same name as that of class and it does not have any return type.

**Default Constructor:**

A constructor which has not defined any parameters or we can say without any parameters is called default constructor.

**Parametrized Constructor:**

A constructor which has at least one parameter is called parametrized constructor. Using this type of constructor, we can initialize each instance of the class to different values.

**Example:**

class student

{

public int id;

public string name;

public student(int id , string name)

{

this.id = id;

this.name = name;

}

}

class Program

{

static void Main(string[] args)

{

student st1 = new student(1,"Ali Ahmed");

Console.WriteLine("Your Id : " + st1.id);

Console.WriteLine("Your name : " + st1.name);

Console.WriteLine("========================");

student st2 = new student(1, "Ali Ahmed");

Console.WriteLine("Your Id : " + st2.id);

Console.WriteLine("Your name: " + st2.name);

}

}

**Constructor Overloading:**

* Constructor overloading in Java refers to the use of more than one constructor in an instance class.
* each overloaded constructor must have different signatures.

**Example:**

class student

{

public student()

{

Console.WriteLine("this is parameter less constructor !!! ");

}

public student(int a , int b)

{

Console.WriteLine("Your sum is : " + (a+b));

}

public student(string name)

{

Console.WriteLine("Your Name is : " + name);

}

}

static void Main(string[] args)

{

student st1 = new student();

student st1 = new student("Muhammad Farhan");

}

**Static and Instance Members:**

**Instance Members:**

* Instance member have a separate copy for each and every object of the class.
* Instance member belongs to the objects of the class.
* When no static keyword is present the class member is called non-static or instance members.
* Instance or non-static members are invoked using objects of the class.

**Example:**

class student

{

public int id;

public string name;

public student(int id, string name)

{

this.id = id;

this.name = name;

}

}

static void Main(string[] args){

student st1 = new student(22,"Muhammad Farhan");

Console.WriteLine(st1.id);

Console.WriteLine(st1.name);

student st2 = new student(33,"Ahmed Raza");

Console.WriteLine(st2.id);

Console.WriteLine(st2.name);

}

**Static Members:**

* Static members belongs to the class.
* When we declare a member of a class as static, it means no matter how many objects of the class are created, there is only one copy of the static member.
* Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it.
* You can also declare a member function as static.
* Static members are invoked or call using class name.

**Example:**

class student

{

public int id;

public string name;

public static string companyname = "Farhan Company";

public student(int id, string name)

{

this.id = id;

this.name = name;

}

}

static void Main(string[] args)

{

student st1 = new student(22,"Muhammad Farhan");

Console.WriteLine(st1.id);

Console.WriteLine(st1.name);

Console.WriteLine(student.companyname);

student st2 = new student(33, "Ali Raza");

Console.WriteLine(st2.id);

Console.WriteLine(st2.name);

Console.WriteLine(student.companyname);

}

**Static Constructor:**

* A static constructor is used to initialize static variables of the class and to perform a particular action only once.
* Static constructor is called only once, no matter how many objects you create.
* Static constructor is called before instance (default or parameterized) constructor.
* A static constructor does not take any parameters and does not use any access modifiers.
* Only one static constructor can be created in the class.
* It is called automatically before the first instance of the class created.

**Example:**

class student

{

public static string companyname;

static student()

{

companyname = "Farhan Company";

Console.WriteLine("Static Constructor ! ");

}

}

static void Main(string[] args)

{

student st1 = new student();

Console.WriteLine(student.companyname);

}

**Copy Constructor:**

The constructor which creates an object by copying variables from another object is called a copy constructor. The purpose of a copy constructor is to initialize a new instance to the value of an existing instance.

In C#, copy constructor is a parametrized constructor which contains a parameter of same class type. The copy constructor in C# is useful whenever we want to initialize a new instance to the values of an existing instance.

In simple words, we can say copy constructor is a constructor which copies a data of one object into another object.

**Example:**

class student

{

public int id;

public string name;

public student(int id, string name)

{

this.id = id;

this.name = name;

}

public void getdata()

{

Console.WriteLine("Id : " + this.id);

Console.WriteLine("Name : " + this.name);

}

public student(student st) // copy constructor

{

this.id = st.id;

this.name = st.name;

}

}

static void Main(string[] args)

{

student st1 = new student(22,"Farhan");

st1.getdata();

student st2 = new student(st1);

st2.getdata();

}

**Private Constructor:**

When a constructor is created with a private specifier, it is not possible for other classes to derive from this class, neither is it possible to create an instance of this class. They are usually used in class that contain static members only.

* One use of a private constructor is when we have only static members.
* Once we provide a constructor that is either private or public or any, the compiler will not add the parameter less public constructor to the class.
* In the presence of parameter less private constructor you cannot create a default constructor.

**Example:**

class student

{

private student()

{

}

public static void myfun()

{

Console.WriteLine("this is funcion but static 1 ");

Console.WriteLine("this is funcion but static 2 ");

}

}

static void Main(string[] args)

{

student.myfun();

}

**Static Class:**

* Classes that cannot be instantiated or inherited are known as classes and the static keyword is used before the class name that consists of static data members and static method.
* It is not possible to create an instance of a static class using the new keyword. The main features:
  + The can only contain static members.
  + The cannot be instantiated or inherited and cannot contain instance constructors. However, the developer can create static constructor to initialize the static members.

**Example:**

static class student

{

public static int id;

public static string name;

static student()

{

id = 2;

name = "Ali Ahmed";

}

public static void myfun()

{

Console.WriteLine("student iD : " + id);

Console.WriteLine("Name is : " + name);

}

}

static void Main(string[] args)

{

student.myfun();

}

**Destructors:**

A destructor is a special method which has the same name as the class but starts with the character ~ before the class name and immediately de-allocates memory of objects that are no longer required.

Following are the features of destructors:

* Destructors cannot be overloaded or inherited.
* Destructors cannot be explicitly invoked.
* Destructors cannot specify access modifiers and cannot take parameters.

**Example:**

~student()

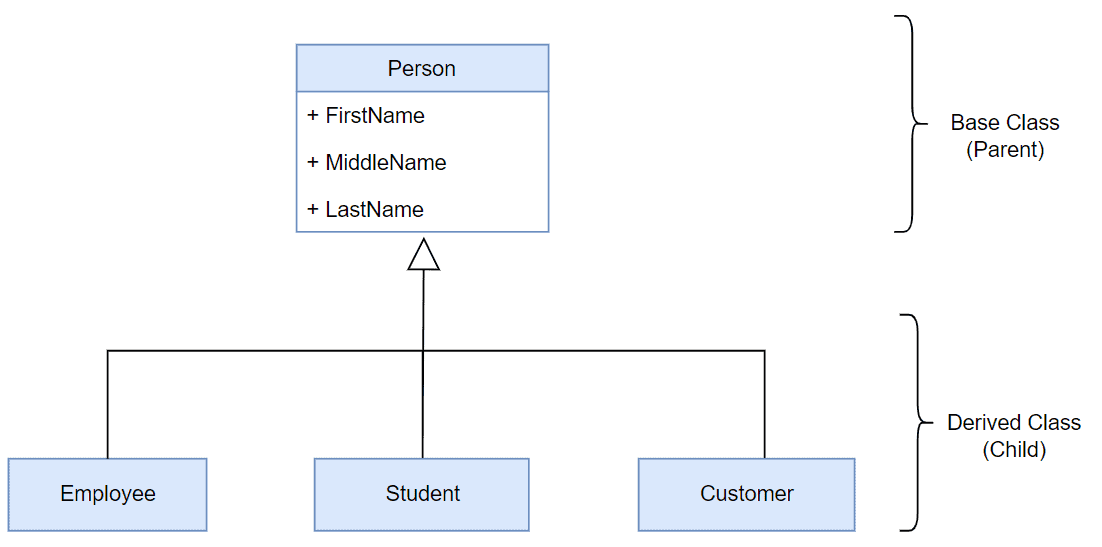
{

Console.WriteLine("this is destructor !");

}

**INHERITANCE IN C# ­**

* The similarity in physical features of a child to that of its parents is due to the child having inherited these features from its parents.
* Similarity, in C#, inheritance allows you to create a class by deriving the common attributes and methods of an existing class.
* Inheritance provides reusability by allowing us to extend an existing class.
* The reason stbehind OOP programming is to promote the reusability of code and to reduce complexity in code and it is possible by using inheritance.



**Example:**

class person

{

public int id;

public string name;

}

class student : person

{

public int rollnum;

public string standard;

}

class teacher : person

{

public int salary;

}

static void Main(string[] args)

{

student st = new student();

st.id = 1;

st.name = "Ali";

st.rollnum = 23333;

st.standard = "10th";

Console.WriteLine(st.id);

Console.WriteLine(st.name);

Console.WriteLine(st.rollnum);

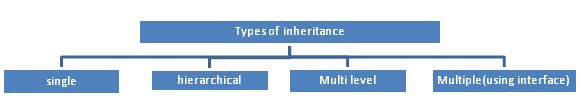
Console.WriteLine(st.standard);

}

The inheritance concept is based on a base class and derived class. Let us see the definition of a base and derived class.

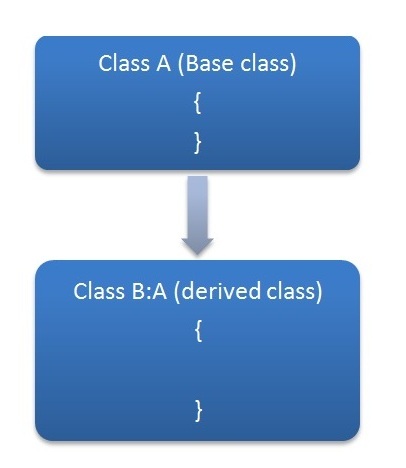
**Base Class:** itis the class from which features are to be inherited into another class.

**Derived Class:** it is the class in which the base class features are inherited.



**Single Inheritance:**

It is the type of inheritance in which there is one base class and one derived class.



**Example:**

class person{

public void myfun1()

{

Console.WriteLine("This is method by parent class : ");

}

}

class student : person

{

public void myfun2()

{

Console.WriteLine("This is method by Child class : ");

}

}

static void Main(string[] args){

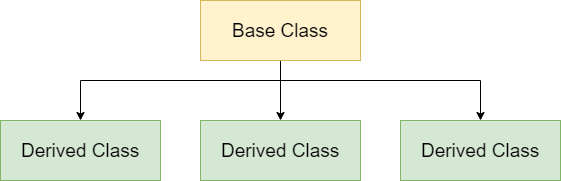
student st = new student();

st.myfun1();

}

**HIERARCHICAL INHERITANCE:**

* This is the type of inheritance in which there are multiple classes derived from one base class.
* This type of inheritance is used when there is a requirement of one class feature that is needed in multiple classes.



**Example:**

class person

{

public int id;

public string name;

}

class student : person

{

public int rollnum;

public string standard;

}

class teacher : person{

public int salary;

}

static void Main(string[] args)

{

student st = new student();

st.id = 1;

st.name = "Ali";

st.rollnum = 23333;

st.standard = "10th";

Console.WriteLine(st.id);

Console.WriteLine(st.name);

Console.WriteLine(st.rollnum);

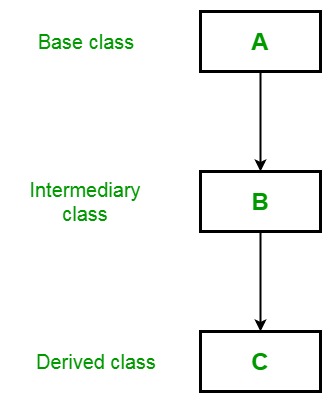
Console.WriteLine(st.standard);

}

­­

**MULTILEVEL INHERITANCE:**

* When one class is derived from another derived class then this type of inheritance is called multilevel inheritance.



**Example:**

class A

{

public void method1()

{

Console.WriteLine("This is method 1 by parent class ");

}

}

class B: A

{

public void method2()

{

Console.WriteLine("This is method 2 by Intermediate class ");

}

}

class C : B

{

public void method3()

{

Console.WriteLine("This is method 3 by Child class ");

}

}

static void Main(string[] args){

C c1 = new C();

c1.method1();

}

**MULTIPLE INHERITANCE:**

Multiple inheritance perform using interface.

**CONSTRUCTOR IN INHERITANCE:**

* A constructor is a method with the same name as the class name and is invoked automatically when a new instance of a class is created.
* Constructors of both classes must be executed when the object of child class is created.
* Derived Class’s constructor invokes constructor of Base class.
* Explicit call to the super class constructor from sub class’s can be made using **base()**.
* If u don’t write **base()** explicitly then C# compiler implicitly write the **base()**.
* If base Class have parameterized constructor then you can add parameters in base().

**Base Keyword**

**The base keyword allows you to do the following:**

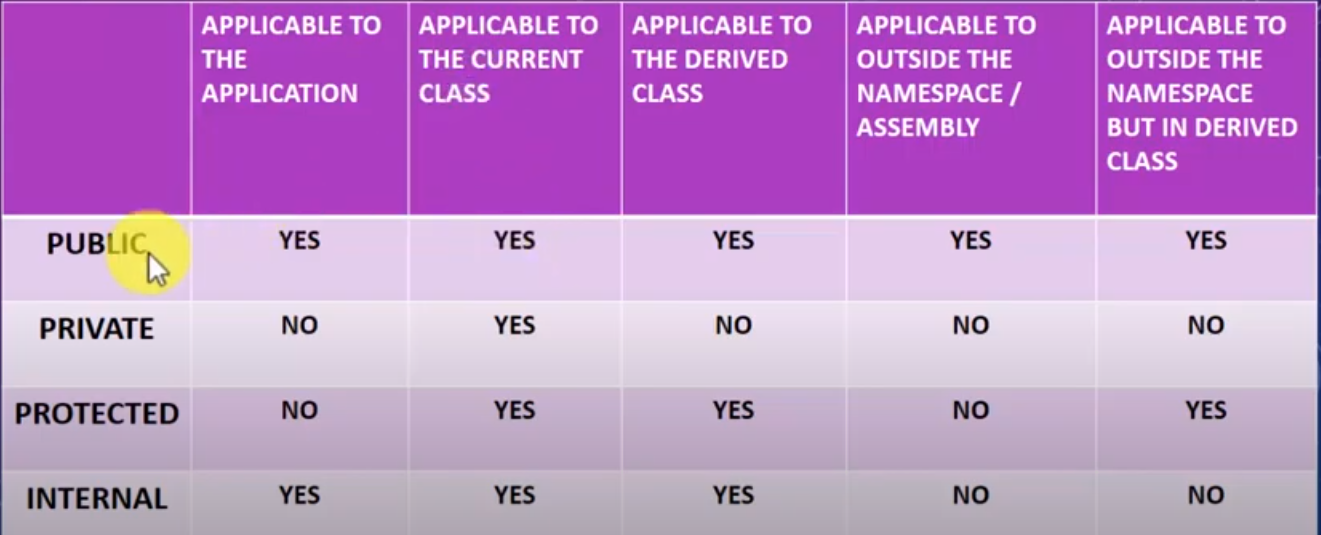
* Access the variables and methods of the base class from the derived class.
* Re-declare the methods and variables defined in the base class.
* Invoke the derived class data members.
* Access the base class members using the base keyword.

**Access Modifiers:**

C# provides you with access modifier that allow you to specify which classes can access the data members of a particular class.

In C#, there are four commonly used access modifiers.

* Public
* Private
* Protected
* Internal



**Example:**

Let’s practical.

**Encapsulation:**

* Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism and abstraction.
* Encapsulation in C# is a mechanism of wrapping the data (variables) and code acting on the data (methods or properties) together as a single unit.
* In encapsulation, the variables of a class will be hidden from other classes, and can be access only through the methods or properties of their current class. Therefore, it is also known as data hiding.
* In a different way, encapsulation is a protective shield that prevents the data from being accessed by the code outside this shield.
* Encapsulation is the procedure of encapsulation data and functions into a single unit **(called class)**.

To achieve encapsulation in C#:

* Declare the variable of a class as private.
* Provide public setter and getter methods or properties to modify and view the variable values.

**Advantages of Encapsulation:**

* The fields of a class can be made **read-only and write-only.**
* A class can have total control over what is stored in its fields.

**Example:**

class student

{

private int id;

private string name;

public void setid(int id)

{

if(id > 0)

{

this.id = id;

}

}

public int getid()

{

return this.id;

}

public void setname(string name)

{

if (string.IsNullOrEmpty(name) == true)

{

Console.WriteLine("this is invalid string ");

}

else

{

this.name = name;

}

}

public string getname()

{

return this.name;

}

}

class Program

{

public static void Main(string[] args)

{

student st = new student();

st.setid(23);

st.setname("farhan");

Console.WriteLine(st.getid());

Console.WriteLine(st.getname());

}

}

**Properties in C#:**

* Properties allow you to control the accessibility of a class variables, and are the recommended way to access variables from the outside in C#.
* A property is much like a combination of a variable and a method. It can’t take any parameters, but you are able to process the value before it’s assigned to our returned.
* Properties are like data fields (variables), but have login behind them.
* From the outside, they look like any other member variable but act like a function.
* Defend like a field, with GET and SET **accessors** code added.
* Properties are also used for encapsulation.

**Types of Properties:**

* Read / Write Properties
* Read Only Properties
* Write Properties
* Auto Implemented Properties.

**Example:**

class student

{

private int \_id;

private string \_name;

public int id

{

set {

if(value > 0)

{

this.\_id = value;

}

}

get {

return this.\_id;

}

}

public string name

{

set

{

if(string.IsNullOrEmpty(value) == true) {

Console.WriteLine("string is wrong ");

}

else

{

this.\_name = value;

}

}

get

{

return this.\_name;

}

}

}

class Program

{

public static void Main(string[] args)

{

student st = new student();

st.id = 1;

st.name = "Farahan";

Console.WriteLine(st.id);

Console.WriteLine(st.name);

}

}

**Automatic Properties:**

public int id{ get; set; }

public string name{ get; set; }

**Static Property:**

* The static property is used to access and manipulate static fields of a class in a safe manner.
* The static property declared by using the static keyword.
* The static property accessed using the class name and thus, belongs to the class rather than just an instance of the class.
* The static property called by a programmer without creating an instance of the class.
* We cannot initialize instance fields within static property.

**Example:**

class student

{

private static string \_uniname;

private static string \_depname;

public static string uniname{

set

{

\_uniname = value;

}

get

{

return \_uniname;

}

}

public static string depname

{

set

{

\_depname = value;

}

get

{

return \_depname;

}

} }

class Program

{

public static void Main(string[] args)

{

student.uniname = "Farhan University";

student.depname = "Computer Science";

Console.WriteLine(student.uniname);

Console.WriteLine(student.depname);

} }

**Polymorphism:**

* Polymorphism is one of the four pillars of object oriented Programming.
* Polymorphism in C# is a concept by which we can perform a single action by different ways.
* Polymorphism is derived from 2 Greek words: Poly and Morphs.
* The word “poly” means many and “morhps” means forms.
* So Polymorphism means many forms.



**There are two types of polymorphism:**

1. Static Polymorphism (compile time polymorphism) or early binding or method overloading
2. Dynamic Polymorphism (Run time Polymorphism) or late binding or method overriding

**Static Polymorphism (Compile Time Polymorphism):**

* The mechanism of linking a function with an object during compile time is called static polymorphism or early binding.
* It is also called static binding.
* Static Polymorphism is **Method or Function Overloading.**

**Method Overloading:**

* You can have multiple definitions for the same function name in the same scope.
* The definition of the function must differ from each other by the types or the number of arguments in the argument list.
* You cannot overload function declarations that differ only by return type.

**Example:**

class calculation

{

public void sum()

{

int a = 6;

int b = 4;

int c = a + b;

Console.WriteLine("sum is : " + c);

}

public void sum(int a , int b)

{

int c = a + b;

Console.WriteLine("Sum is : " + c);

}

public void sum(double a , double b)

{

double c = a + b;

Console.WriteLine("sum is " + c);

}

}

class Program

{

public static void Main(string[] args)

{

calculation cal = new calculation();

cal.sum();

cal.sum(10, 20);

}

}

**Method Hiding:**

* Method hiding occurs in inheritance relationship when base class and derived class both have a method with same name and same signature.
* When we create the object of derived class it will hide the base class method and will call its own method and this is called method hiding or name hiding in C# inheritance.
* We use **“new”** keyword in derived function name to show that implementation of the function in derived class in intentional and derived class no longer want to use base class method.
* The new keyword is not compulsory.

**Example:**

class parent

{

public void sum()

{

Console.WriteLine("this is parent method ! ");

}

}

class child : parent

{

public new void sum()

{

Console.WriteLine("this is child method ! ");

}

}

class Program

{

public static void Main(string[] args)

{

child ch = new child();

ch.sum();

((parent)ch).sum();

}

}

**Dynamic or Runtime Polymorphism:**

* Run time polymorphism is achieved by method overriding.
* Method overriding allows us to have virtual and abstract methods in the base using derived classes with the same name and the same parameter.

**Method Overriding:**

* If derived class defines same method as defined in its base class, it is known as method overriding.
* It is used to achieve runtime polymorphism.
* It enables you to provide specific implementation of the method in child class which is already provided by its base class.
* To perform method overriding in C#, you need to use **virtual** keyword with base class method and **override** keyword with derived class method.
* In the derived class, you need to declare the inherited virtual method using the override keyword.
* In the derived class, you need to declare the inherited virtual method using the override keyword which is mandatory for any virtual method that is inherited in the derived class.

**Example:**

class parent

{

public virtual void sum()

{

Console.WriteLine("this is parent method ! ");

}

}

class child : parent

{

public override void sum()

{

Console.WriteLine("this is child method ! ");

}

}

class Program

{

public static void Main(string[] args)

{

child ch = new child();

ch.sum();

}

}

|  |  |
| --- | --- |
| **Method Overriding** | **Method Hiding** |
| In method overriding, you need to define the method of a parent class as a virtual method using virtual keyword and the method of child class as an overridden method using override keyword. | In method hiding, you just simply create a method in a parent class and in child class you need to define that method using new keyword. |
| It only redefines the implementation of the method. | In method hiding, you can completely redefine the method. |
| overriding is object-type specific. | New is reference-type specific |
| **Example:**  class parent  {  public virtual void sum()  {  Console.WriteLine("this is parent method ! ");  }  }  class child : parent  {  public override void sum()  {  Console.WriteLine("this is child method ! ");  }  }    class Program  {  public static void Main(string[] args)  {  Parent ch = new child();  ch.sum();    }  } | class parent  {  public void sum()  {  Console.WriteLine("this is parent method ! ");  }  }  class child : parent  {  public new void sum()  {  Console.WriteLine("this is child method ! ");  }  }    class Program  {  public static void Main(string[] args)  {  parent ch = new child();  ch.sum();    }  } |
| **Output:** |  |

**Sealed Class:**

A sealed class is a class that prevents inheritance.

The features of a sealed class are as follows:

* A sealed class can be declared by preceding the class keyword with the **sealed** keyword.
* The sealed keyword prevents a class from being inherited by any other class.
* The sealed class cannot be a base class as it cannot be inherited by any other class. If a class tries to derive a sealed class, the C# compiler generates an error.

**Purpose of Sealed Class:**

* Consider a class named **SystemInformation** that consists of critical methods that affect the working of the operating system.
* You might not want any third party to inherit the class **SystemInfomation** and override its methods, thus, causing security and copyright issues.

**Example:**

sealed class parent

{

}

class child : parent

{

}

**Sealed Methods:**

* When the derived class override a base class method, variable, property or event, then the new method, variable, property, or event can be declared as sealed.
* Sealing the new method prevents the method from further overriding.
* An overridden method can be sealed by preceding the override keyword with the sealed keyword.

**Steps to remember for sealed Methods:**

* Sealed method is always an override method of child class.
* We cannot again override the sealed method.
* Sealed method is only available with method overriding.
* Sealed keyword is not available with the method hiding.
* Sealed is used together with override method.
* We cannot make normal methods as sealed.

**Example:**

class parent

{

public virtual void method1()

{

Console.WriteLine("this is method");

}

}

class child : parent

{

public sealed override void method1()

{

Console.WriteLine("this is method");

}

}

class grandchild : child

{

public override void method1()

{

Console.WriteLine("this is method");

}

}

**Abstraction:**

Abstraction is one of the principle of object oriented programming. It is used to display only necessary and essential features of an object to outside the world. Means displaying what is necessary and encapsulate the unnecessary things to outside the world. Hiding can be achieved using **“private”** access modifiers.

Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user.

In other words, the user will have the information on **what** the object does instead of **how** it does it.

**Note:** Outside the world means when we use reference of object then it will show only necessary methods and properties **(Abstraction)** and hide methods which are not necessary **(Encapsulation).**

**Abstraction:**

**Indexers:**

* Indexers allow our objects to be used just like an array, or we can say we can index the object using[] brackets just like arrays.
* We can say indexers are special type of properties which adds logic that how can array or list store the values.
* Syntax of indexers resembles to properties.
* We can all access modifiers with indexers and also have return types.
* Indexers is created with the help of this keyword.
* In C# introduce new concepts is indexer. This is very useful for some situations. Let as discuss something about indexers.
  + Indexers concept is object act as an array.
  + Indexers an object to be indexed in the same way as an array.
  + Indexer modifiers can be private, public, protected or internal.

**Example:**

namespace batch11goop

{

class Student

{

private int[] age = new int[3];

public int this[int index]

{

set

{

if(value > 0)

{

age[index] = value;

}

else

{

Console.WriteLine("Invalid Input");

}

}

get {

return age[index];

}

}

}

class Program

{

public static void Main(string[] args)

{

Student st = new Student();

st[0] = 5;

Console.WriteLine(st[0]);

}

}

}

**What is Delegate?**

**Delegate Meaning:** A person sent or authorized to represent others, in particular an elected representative sent to a conference.

* Delegate is a type which holds a methods reference in an object.
* It is also called function pointer.
* Delegate is a reference type.
* Delegate Signature should be as same as the method signature referencing by a delegate.
* Delegate can point to a parametrized method or non-parameterized method.
* Delegate has no implementation means no body with {}.
* We can use invoke() method with delegates.
* Delegates are used to encapsulate methods.
* In the .NET framework, A delegate points to one or more methods. Once you instantiate the delegate, the corresponding methods invoke.
* Delegates are objects that contain references to methods that need to be invoked instead of containing the actual method names.

**Types of Delegates:**

* Multiple Delegates
* Single Cast Delegates
* Multi Cast Delegates

**Multiple Delegates:**

In C#, a user can invoke multiple delegates within a single program. Depending on the delegate name or the type of parameters passed to the delegate the appropriate delegate is invoked.

**Single Cast Delegates:**

Single delegate point to single method at a time. In this the delegate is assigned to a single method at a time. They are derived from System.Delegate class.

**Example:**

namespace batch11goop

{

public delegate void mysum(int a, int b);

class Program

{

public void sum(int a, int b)

{

int c = a + b;

Console.WriteLine("your sum is " + c);

}

public static void Main(string[] args)

{

Program pa = new Program();

mysum obj = new mysum(pa.sum);

obj.Invoke(4,8);

obj(10, 20);

}

} }

**Multi Cast Delegate:**

When a delegate is wrapped with more than one method that is known as a multicast delegate.

In C#, delegates are multicast, which means that they can point to more than one function at a time. They are derived from System.MulticastDelegate class.

We can use += and -= assignment operators to implement multi cast delegates.

**Example:**

namespace batch11goop

{

public delegate void calculation(int a, int b);

class Program

{

public void sum(int a, int b)

{

int c = a + b;

Console.WriteLine("your sum is " + c);

}

public void sub(int a, int b)

{

int c = a - b;

Console.WriteLine("your Subtraction is " + c);

}

public static void Main(string[] args)

{

Program pa = new Program();

calculation obj = new calculation(pa.sum);

obj += pa.sub;

obj(20, 10);

}

}

}

**Anonymous Function:**

* We discussed that delegates are used to reference any methods that has the same signature as that of the delegate.
* As the name suggests, an **anonymous method** is a method without a name just the body.
* Anonymous methods in C# can be defined using the delegate keyword.
* Anonymous method can be assigned to a variable of delegate type.
* Anonymous method must be assigned to a delegate.

**Advantage of Anonymous Functions:**

* Lesser typing work because we don’t require to write access modifiers, return type and name of the function.

**Example:**

namespace cnewpracticeoop{

public delegate void mydel(int a);

class Program

{

static void Main(string[] args)

{

Program pa = new Program();

mydel dl = delegate (int a)

{

a += 10;

Console.WriteLine(a);

};

dl.Invoke(5);

}

}

}

**Lambda Expression:**

* It is also work like an anonymous methods.
* The difference that in Lambda expressions you don’t need to specify the type of the value that you input thus making it more flexible to use.
* It means lambda expressions simplifies the anonymous functions or you can say that lambda expression is a shorthand for an anonymous functions.
* The **=>** is the lambda operator which is used in all lambda expressions.
* The lambda expressions can be of two types:
  1. **Statement lambda:** consists of the input and a set of statements to be executed.
  2. **Expressions Lambda:** consist of the input and the expression.

**Example:**

namespace cnewpracticeoop{

public delegate void mydel(int a);

class Program

{

static void Main(string[] args)

{

mydel dl = (a) =>

{

a += 10;

Console.WriteLine(a);

};

dl.Invoke(5);

}

}

}

**Exception and Exception Handling:**

* An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program’s instructions.
* In general, when a C# code encounters a situation that it cannot cope with, it raise an exception.
* An exception is a C# object that represents an error.
* When a C# code raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
* When an exception occur 3 things happen.
  + Program terminates or program crashed.
  + Ugly kind of error message is displayed that user can never ever understand.
  + Statements after exception will not be executed.
* The exception handling in C# is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

**Example of Exception:**

namespace cnewpracticeoop{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter First Number :");

int a = int.Parse(Console.ReadLine());

Console.WriteLine("Enter Second Number :");

int b = int.Parse(Console.ReadLine());

int c = a / b;

Console.WriteLine("Divsiion is : " + c);

Console.WriteLine("after division line 1 ");

Console.WriteLine("after division line 2");

Console.WriteLine("after division line 3 ");

Console.WriteLine("after division line 4 ");

}

}

}

**Generics:**

**Concrete Class:**

A concrete class is a class that allows creating an instance or an object using the new keyword. A concrete method is a method that can be called directly.

**Dependency Injection:**

Dependency Injection (DI) is a software design pattern that helps developers build better software. It allows us to develop loosely-coupled code that is easy to maintain. Dependency Injection reduces the hard-coded dependencies among your classes by injecting those dependencies at run time instead of design time technically.

**What is GUID:**

Guid obj = Guid.NewGuid();

Console.WriteLine("New Guid is " + obj.ToString());