**Interface**

An interface looks like a class, but has no implementation. The only thing it contains are declarations of methods, properties, indexers & events. A class that implements the interface must implement the members of the interface that are specified in the interface definition.

Interfaces can contain methods, properties, events, indexers, or any combination of those four member types. An interface can't contain constants, fields, operators, instance constructors, finalizers, or types. Interface members are automatically public, and they can't include any access modifiers. Members also can't be [static](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static).

To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.

The Interface in C# is a **Fully Unimplemented Class** used for declaring a set of operations/methods of an object. It is used to achieve multiple inheritances, which the class can’t achieve. It is used to achieve full abstraction because it cannot have a method body.

While working with Interface, we need to remember some Rules. Let us discuss those rules one by one with Examples.

**Point 1**: The first point that you need to remember is that the default scope for an interface’s members is public, whereas it is private in the case of a class.

**Point 2**: The second point that you need to remember is by default, every member of an interface is abstract, so we aren’t required to use the abstract modifier on it again, just like we do in the case of an abstract class. For a better understanding, please have a look at the below example. By default, the Add method is going to be public and abstract.

interface ITestInterface

{

//By default, the following method is public and abstract

void Add(int num1, int num2);

}

**Point 3**: You need to remember that we cannot declare fields/variables, constructors, and destructors in an interface in C#.

**Point 4:** The fourth point you need to remember is that an interface can inherit from another interface in C# just like a class inherits from another.

interface ITestInterface1

{

void Add(int num1, int num2);

}

interface ITestInterface2 : ITestInterface1

{

void Sub(int num1, int num2);

}

**Point 5:** The fifth point that you need to remember is every member of an interface should be implemented under the child class without fail (mandatory), but while implementing, we aren’t required to use the override modifier just like we have done in the case of an abstract class.

using System;

namespace AbstractClassMethods

{

interface ITestInterface1

{

void Add(int num1, int num2);

}

interface ITestInterface2 : ITestInterface1

{

void Sub(int num1, int num2);

}

public class ImplementationClass1 : ITestInterface1

{

//Implement only the Add method

public void Add(int num1, int num2)

{

Console.WriteLine($"Sum of {num1} and {num2} is {num1 + num2}");

}

}

public class ImplementationClass2 : ITestInterface2

{

//Implement Both Add and Sub method

public void Add(int num1, int num2)

{

Console.WriteLine($"Sum of {num1} and {num2} is {num1 + num2}");

}

public void Sub(int num1, int num2)

{

Console.WriteLine($"Divison of {num1} and {num2} is {num1 - num2}");

}

}

class Program

{

static void Main()

{

ImplementationClass1 obj1 = new ImplementationClass1();

//Using obj1 we can only call Add method

obj1.Add(10, 20);

//We cannot call Sub method

//obj1.Sub(100, 20);

ImplementationClass2 obj2 = new ImplementationClass2();

//Using obj2 we can call both Add and Sub method

obj2.Add(10, 20);

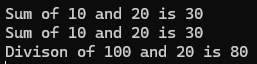
obj2.Sub(100, 20);

Console.ReadKey();

}

}

}



**Point 6:**We cannot create an instance of an interface, but we can create a reference of an interface. The interface reference is going to hold the child class instance. We can only invoke the methods declared in the interface using the interface reference.

using System;

namespace AbstractClassMethods

{

interface ITestInterface1

{

void Add(int num1, int num2);

}

public class ImplementationClass : ITestInterface1

{

//Interface Method Implementation

public void Add(int num1, int num2)

{

Console.WriteLine($"Sum of {num1} and {num2} is {num1 + num2}");

}

}

class Program

{

static void Main()

{

//Creating Reference of an Interface point to the

//child class instance

ITestInterface1 obj = new ImplementationClass();

//Add method signature declared in ITestInterface1, so we can

//Invoke the Add method

obj.Add(10, 20);

Console.ReadKey();

}

}

}



**Abstract Class**

The [abstract](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract) keyword enables you to create classes and [class](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/class) members that are incomplete and must be implemented in a derived class.

abstract class ShapesClass

{

}

An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share.

**Is Abstract Class Containing Only Abstract Methods in C#?**

Don’t think an abstract class can contain only abstract methods. It can also contain non-abstract methods. You need to remember that if a class is non-abstract, it contains only non-abstract methods, but if a class is abstract, it contains both abstract and non-abstract methods in C#.

**Note:** To define a method as abstract or class as abstract, we need to use the abstract keyword.

**Note:** Abstract methods are commonly used within abstract classes. An abstract class cannot be instantiated; it serves as a blueprint for other classes. Abstract methods within an abstract class define a contract that any derived (sub) class must implement. An abstract class is declared using the abstract keyword.

Example:

using System;

namespace AbstractApplication

{

abstract class ShapesClass

{

abstract public int Area();

}

class Square : ShapesClass

{

int side = 0;

public Square(int n)

{

side = n;

}

public override int Area()

{

return side \* side;

}

}

class Program

{

static void Main()

{

Square sq = new Square(12);

Console.WriteLine("Area of the square = {0}", sq.Area());

Console.ReadKey();

}

}

}

Output:

Area of the square = 144

**Sealed Class in C#**

A class from which it is not possible to create/derive a new class is known as a sealed class. In simple words, we can say that when we define the class using the sealed modifier, then it is known as a sealed class and a sealed class cannot be inherited by any other classes.

**What is Partial Class in C#?**

Partial Class is the new feature that has been added in C# 2.0 which allows us to define a class on multiple files i.e. we can physically split the content of the class into different files but even physically they are divided but logically it is one single unit only. A class in which code can be written in two or more files is known as a partial class. To make any class partial we need to use the keyword partial.

PartialEmployeeOne.cs

using System;

namespace PartialClassDemo

{

public partial class PartialEmployee

{

public string FirstName { get; set; }

public string LastName { get; set; }

public string Gender { get; set; }

public double Salary { get; set; }

}

}

PartialEmployeeTwo.cs

using System;

namespace PartialClassDemo

{

public partial class PartialEmployee

{

public void DisplayEmployeeDetails()

{

Console.WriteLine("Employee Details : ");

Console.WriteLine($"First Name : {FirstName}");

Console.WriteLine($"Last Name : {LastName}");

Console.WriteLine($"Gender : {Gender}");

Console.WriteLine($"Salary : {Salary}");

}

}

}

Program.cs

using System;

namespace PartialClassDemo

{

class Program

{

static void Main(string[] args)

{

PartialEmployee emp = new PartialEmployee()

{

FirstName = "Pranaya",

LastName = "Rout",

Salary = 100000,

Gender = "Male"

};

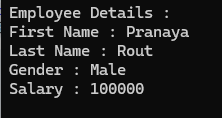
emp.DisplayEmployeeDetails();

Console.ReadKey();

}

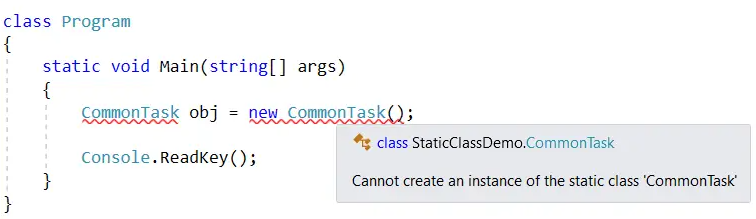
}

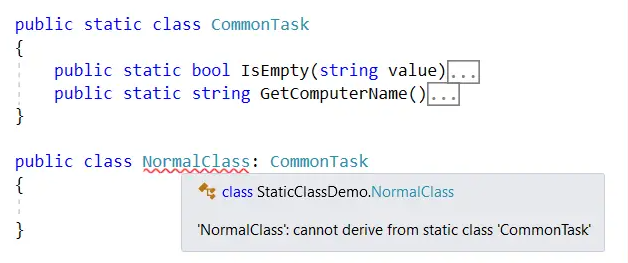
}



**Static Class in C#**

The class which is created by using the static modifier is called a static class in C#. A static class can contain only static members. It is not possible to create an instance of a static class. This is because it contains only static members. And we know we can access the static members of a class by using the class name.





**Introduction to Object Oriented Programming Concepts (OOP)**

It is a methodology to write the program where we specify the code in form of classes and objects.

The main advantages and goals of OOP are to make complex software faster to develop and easier to maintain. OOP enables the easy reuse of code by applying simple and widely accepted rules (principles).

**Class**

A class is a group of related methods and variables.

public class student

{

}

**Object**

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 behavior of that class.

student objstudent = new student();

According to the above sample, we can say that Student object, named objstudent, has created out of the student class.

**Methods**

A method is an action that an object can perform.

class SampleClass

{

public int sampleMethod(string sampleParam)

{

// Insert code here

}

}

**Constructor**

Constructor is a special method of a class, which will invoke automatically whenever instance or object of class is created. Constructors are responsible for object initialization and memory allocation of its class. If we create any class without constructor, the compiler will automatically create one default constructor for that class. There is always at least one constructor in every class.

Here you need to remember that a class can have any number of constructors, constructors don’t have any return type, not even void, and within a class we can create only one static constructor.

Generally, constructor name should be same as class name. If we want to create constructor in a class, we need to create a constructor method name same as class name.

**Some of the key points regarding the Constructor are:**

* A class can have any number of constructors.
* A constructor doesn't have any return type, not even void.
* A static constructor cannot be a parametrized constructor.
* Within a class you can create only one static constructor.

**Constructors can be divided into 5 types:**

* Default Constructor
* Parametrized Constructor
* Copy Constructor
* Static Constructor
* Private Constructor

**Default Constructor**

A constructor without any parameters is called a default constructor; in other words, this type of constructor does not take parameters. The drawback of a default constructor is that every instance of the class will be initialized to the same values.

The default constructor initializes:

1. All numeric fields in the class to zero.
2. All string and object fields to null.
3. All Boolean fields to false.

namespace ConstructorDemo

{

class Employee

{

public int Id, Age;

public string Address, Name;

public bool IsPermanent;

}

class Test

{

static void Main(string[] args)

{

Employee e1 = new Employee();

Console.WriteLine("Employee Id is: " + e1.Id);

Console.WriteLine("Employee Name is: " + e1.Name);

Console.WriteLine("Employee Age is: " + e1.Age);

Console.WriteLine("Employee Address is: " + e1.Address);

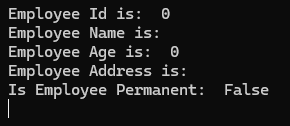
Console.WriteLine("Is Employee Permanent: " + e1.IsPermanent);

Console.ReadKey();

}

}

}



**Parameterized Constructor**

A constructor with at least one parameter is called a parametrized constructor. The advantage of a parametrized constructor is that you can initialize each instance of the class to different values.

using System;

namespace Constructor

{

class Sample

{

public int a, b;

// decalaring Paremetrized Constructor with ing x,y parameter

public Sample(int x, int y)

{

a = x;

b = y;

}

}

class Program

{

static void Main()

{

// Creating object of Parameterized Constructor and ing values

Sample obj = new Sample(100, 175);

Console.WriteLine("Value of a=" + obj.a);

Console.WriteLine("Value of b=" + obj.b);

Console.Read();

}

}

}



**Constructor Overloading**

In C #, we can overload constructor by creating another constructor with same method name and different parameters like as shown below

using System;

namespace ConsoleApplication

{

class Sample

{

public string param1, param2;

public Sample() // Default Constructor

{

param1 = "Hi";

param2 = "I am Default Constructor";

}

public Sample(string x, string y) // Declaring Parameterized constructor with Parameters

{

param1 = x;

param2 = y;

}

}

class Program

{

static void Main(string[] args)

{

Sample obj = new Sample(); // Default Constructor will Called

Sample obj1 = new Sample("Welcome", "Kamal"); // Parameterized Constructor will Called

Console.WriteLine(obj.param1 + ", " + obj.param2);

Console.WriteLine(obj1.param1 + ", " + obj1.param2);

Console.ReadLine();

}

}

}



**Copy Constructor**

A parameterized constructor that contains a parameter of same class type is called as copy constructor. Main purpose of copy constructor is to initialize new instance to the values of an existing instance.

using System;

namespace ConsoleApplication

{

class Sample

{

public string param1, param2;

public Sample(string x, string y)

{

param1 = x;

param2 = y;

}

public Sample(Sample obj) // Copy Constructor

{

param1 = obj.param1;

param2 = obj.param2;

}

}

class Program

{

static void Main(string[] args)

{

Sample obj = new Sample("Welcome", "Kamal"); // Create instance to class Sample

Sample obj1 = new Sample(obj); // Here obj details will copied to obj1

Console.WriteLine(obj1.param1 + ", " + obj1.param2);

Console.ReadLine();

}

}

}

**Output**

Welcome, Kamal

**Static Constructor**

When a constructor is created as static, it will be invoked only once for all of instances of the class and it is invoked during the creation of the first instance of the class or the first reference to a static member in the class. A static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.

Some key points of a static constructor is:

* A static constructor does not take access modifiers or have parameters.
* A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.
* A static constructor cannot be called directly.
* The user has no control on when the static constructor is executed in the program.
* A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.

using System;

namespace ConsoleApplication

{

class Sample

{

public string param1, param2;

static Sample()

{

Console.WriteLine("Static Constructor");

}

public Sample()

{

param1 = "Sample";

param2 = "Instance Constructor";

}

}

class Program

{

static void Main(string[] args)

{

// Here Both Static and instance constructors are invoked for first instance

Sample obj = new Sample();

Console.WriteLine(obj.param1 + " " + obj.param2);

// Here only instance constructor will be invoked

Sample obj1 = new Sample();

Console.WriteLine(obj1.param1 + " " + obj1.param2);

Console.ReadLine();

}

}

}

**Output**

Static Constructor

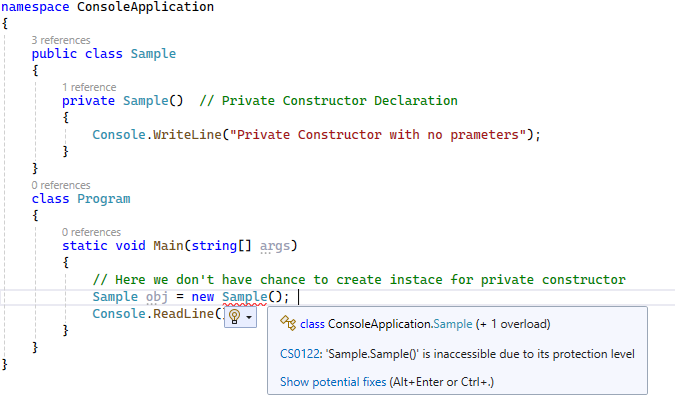
Sample Instance Constructor

Sample Instance Constructor

**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 classes that contain static members only. Some key points of a private constructor are:

1. One use of a private constructor is when we have only static members.
2. If we want to create object of class even if we have private constructors, then we need to have public constructor along with private.
3. Once we provide a constructor that is either private or public or any, the compiler will not allow us to add public constructor without parameters to the class.

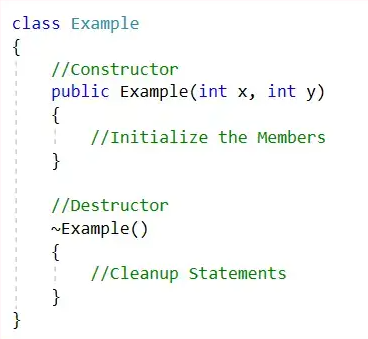


**Finalizers/Destructors**

Finalizers are used to destruct instances of classes. In the .NET Framework, the garbage collector automatically manages the allocation and release of memory for the managed objects in your application. However, you may still need finalizers to clean up any unmanaged resources that your application creates. There can be only one finalizer for a class.

The Destructor is also a special type of method present in a class, just like a constructor, having the same name as the class name but prefixed with **~** tilde. The Constructor in C# is Explicitly called when the object of the class is created. On the other hand, the Destructor in C# is Implicitly Called when the object of the class is destroyed.

**Note:** The most important point that you need to keep in mind is that a destructor method cannot have any parameters as well as cannot be applied with any modifiers. As the destructor is not parameterized, so we cannot overload the destructor.



When is a Destructor method Called in C#?

A destructor method gets called automatically by the garbage collector when the object of the class is destroyed. So, the point that you need to remember is that the destructor methods are automatically called by the garbage collector.

When will the Object of a Class get Destroyed in C#?

The object of a class in C# will be destroyed by the garbage collector in any of the following cases

* At the end of the program execution, each and every object that is associated with the program will be destroyed by the garbage collector.
* The Implicit calling of the garbage collector occurs sometime in the middle of the program execution provided the memory is full so the garbage collector will identify unused objects of the program and destroys them.
* The Explicit calling of the garbage collector can also be done in the middle of program execution by the programmer with the help of the “**Collect()**” statement so that if there are any unused objects associated with the program will be destroyed in the middle of the program execution.

using System;

namespace DestructorExample

{

class DestructorDemo

{

public DestructorDemo()

{

Console.WriteLine("Constructor Object Created");

}

~DestructorDemo()

{

string type = GetType().Name;

Console.WriteLine($"Object {type} is Destroyed");

}

}

class Program

{

static void Main(string[] args)

{

DestructorDemo obj1 = new DestructorDemo();

DestructorDemo obj2 = new DestructorDemo();

obj1 = null;

obj2 = null;

Console.ReadKey();

}

}

}



You can see the statements written inside the destructor are not printed in the output. The point that you need to remember is that the destructor method is implicitly called by the garbage collector and we cannot predict when it calls the destructor method. And hence you cannot see those print statements in the output. Once, the object is unused i.e. it ready for garbage collection, then it is the responsibility of the garbage collector to destroy that unused object and free the memory from the heap.

If you want, then you can also make an explicit call to the garbage collector in the middle of the application execution to destroy the unused object. To do so, we need to call the **GC.Collect** method as shown in the below example.

using System;

namespace DestructorExample

{

class DestructorDemo

{

public DestructorDemo()

{

Console.WriteLine("Constructor Object Created");

}

~DestructorDemo()

{

string type = GetType().Name;

Console.WriteLine($"Object {type} is Destroyed");

}

}

class Program

{

static void Main(string[] args)

{

DestructorDemo obj1 = new DestructorDemo();

DestructorDemo obj2 = new DestructorDemo();

//Making obj1 and obj2 ready for Garbage Collection

obj1 = null;

obj2 = null;

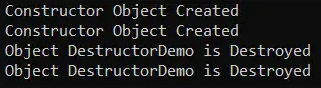
GC.Collect();

Console.ReadKey();

}

}

}



**Access Modifiers and Access Levels**

All classes and class members can specify what access level they provide to other classes by using access modifiers. Access Specifiers defines the scope of a class member.

The following access modifiers are available:

**public** :- The type or member can be accessed by any other code in the same assembly or another assembly that references it.

**private** :- The type or member can only be accessed by code in the same class.

**protected** :- The type or member can only be accessed by code in the same class or in a derived class.

**internal** :- The type or member can be accessed by any code in the same assembly, but not from another assembly.

**protected internal** :- The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.

**What is Garbage Collection in .NET Framework?**

When a dot net application runs, lots of objects are created. At a given point in time, it is possible that the application does not use some of those objects. The Garbage Collector in the .NET Framework is nothing but a Small Routine, or you can say it’s a Background Process Thread that runs periodically and tries to identify what objects are not being used currently by the application and de-allocates the memory of those objects.

So, Garbage Collector is nothing but a feature provided by CLR that helps us clean or destroy unused managed objects. Cleaning or destroying those unused managed objects basically reclaims the memory.

Garbage Collection (GC) in the .NET Framework is an automatic memory management system that helps manage the allocation and release of memory in your applications. In .NET, when we create an object using the new keyword, it automatically allocates memory on the managed heap.

**Note:** The Garbage Collector will destroy only the unused managed objects. It does not clean unmanaged objects.

**Managed Objects:**

Managed objects are allocated on the managed heap and controlled by the .NET Garbage Collector (GC). These objects are typically instances of classes and structures defined in .NET. The GC automatically manages the memory for managed objects. It allocates and releases memory for these objects and handles memory optimizations like compacting.

**Unmanaged Objects:**

Unmanaged objects are objects whose memory is not managed by the .NET GC. These are typically objects allocated using native code, like calls to Windows API or using languages such as C or C++. The developer is responsible for allocating and freeing the memory for unmanaged objects.

**Encapsulation**-- **Information hiding**.

Encapsulation is a process of binding data members (variables, properties) and member functions (methods) into a single unit.

**Encapsulation Summary**

* Through encapsulation, a class can hide the internal details of how an object does something. Encapsulation solves the problem at the implementation level.
* A class or structure can specify how accessible each of its members (variables, properties, and methods) is to code outside of the class or structure. Encapsulation simplifies the interaction between objects. An object can use another object without knowing all its data or how its data is maintained.
* With the help of encapsulation, a class can change the internal implementation without hurting the overall functionality of the system.
* Encapsulation protects abstraction.

namespace EncapsulationDemo

{

class Bank

{

public long AccountNumber;

public string Name;

public int Balance;

public void GetBalance()

{

}

public void WithdrawAmount()

{

}

public void Deposit()

{

}

}

}

**Abstraction**-- **Implementation hiding**.

Abstraction is a process of hiding the implementation details and displaying the essential features.

Abstraction lets you focus on what the object does instead of how it does it.

Abstraction provides you a generalized view of your classes or objects by providing relevant information.  
   
Abstraction is the process of hiding the working style of an object, and showing the information of an object in an understandable manner.

Real-Time Example of Abstraction

To take a real-time example, when we log in to any social networking site like Facebook, Twitter, LinkedIn, etc., we enter our user ID and password, and then we get logged in. Here, we don’t know how they are processing the data or what logic or algorithm they are using for login. This information is abstracted/hidden from us since they are not essential to us. This is basically what abstraction is.

using System;

namespace GarbageCollectionDemo

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Transaction doing SBI Bank");

SBI sbi = new SBI();

sbi.DepositMoney();

sbi.CheckBalanace();

sbi.WithdrawMoney();

sbi.MiniStatement();

Console.WriteLine("\nTransaction doing AXIX Bank");

AXIX AXIX = new AXIX();

AXIX.DepositMoney();

AXIX.CheckBalanace();

AXIX.WithdrawMoney();

AXIX.MiniStatement();

Console.Read();

}

}

public class SBI

{

public void DepositMoney()

{

Console.WriteLine("AXIS Bank Deposit Money");

}

public void CheckBalanace()

{

Console.WriteLine("AXIX Bank Check Balanace");

}

public void WithdrawMoney()

{

Console.WriteLine("AXIX Bank Withdraw Money");

}

public void MiniStatement()

{

Console.WriteLine("AXIX Bank Mini Statement");

}

}

public class AXIX

{

public void DepositMoney()

{

Console.WriteLine("AXIS Bank Deposit Money");

}

public void CheckBalanace()

{

Console.WriteLine("AXIX Bank Check Balanace");

}

public void WithdrawMoney()

{

Console.WriteLine("AXIX Bank Withdraw Money");

}

public void MiniStatement()

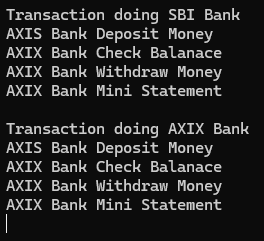
{

Console.WriteLine("AXIX Bank Mini Statement");

}

}

}



Then what is the problem with the above implementation? The problem is the user of our application accesses the SBI and AXIX classes directly. Directly means they can go to the class definition and see the implementation details of the methods. That is, the user will come to know how the services or methods are implemented. This might cause security issues. We should not expose our implementation details to the outside.

**How to Implement Abstraction Principle in C#?**

In C#, we can implement the abstraction OOPs principle in two ways. They are as follows:

* **Using Interface**
* **Using Abstract Classes and Abstract Methods**

using System;

namespace GarbageCollectionDemo

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Transaction doing SBI Bank");

IBank sbi = new SBI();

sbi.DepositMoney();

sbi.CheckBalanace();

sbi.WithdrawMoney();

sbi.MiniStatement();

Console.WriteLine("\nTransaction doing AXIX Bank");

IBank AXIX = new AXIX();

AXIX.DepositMoney();

AXIX.CheckBalanace();

AXIX.WithdrawMoney();

AXIX.MiniStatement();

Console.Read();

}

}

public interface IBank

{

void DepositMoney();

void CheckBalanace();

void WithdrawMoney();

void MiniStatement();

}

public class SBI: IBank

{

public void DepositMoney()

{

Console.WriteLine("AXIS Bank Deposit Money");

}

public void CheckBalanace()

{

Console.WriteLine("AXIX Bank Check Balanace");

}

public void WithdrawMoney()

{

Console.WriteLine("AXIX Bank Withdraw Money");

}

public void MiniStatement()

{

Console.WriteLine("AXIX Bank Mini Statement");

}

}

public class AXIX : IBank

{

public void DepositMoney()

{

Console.WriteLine("AXIS Bank Deposit Money");

}

public void CheckBalanace()

{

Console.WriteLine("AXIX Bank Check Balanace");

}

public void WithdrawMoney()

{

Console.WriteLine("AXIX Bank Withdraw Money");

}

public void MiniStatement()

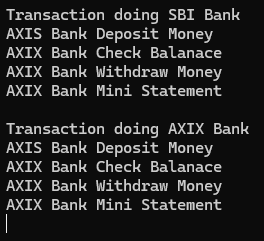
{

Console.WriteLine("AXIX Bank Mini Statement");

}

}

}



In the above example, I am using an interface to achieve the abstraction principle in C#. Using the interface, we can achieve 100% abstraction. Now, the user will only know the services that are defined in the interface, but how the services are implemented, the user will never know. This is how we can implement abstraction in C# by hiding the implementation details from the user. Here, the user will only know about IBank, but the user will not know about the SBI and AXIX Classes.

**Polymorphism**

The word polymorphism is derived from two Greek words: poly and morphs. The word “Poly” means many, and “morphs” means forms. Therefore, polymorphism means “many forms” or we can say that the word polymorphism means the ability to take more than one form.

In Polymorphism, we have two different types those are

- Compile Time Polymorphism (Called as Early Binding or Overloading or static binding)

- Run Time Polymorphism (Called as Late Binding or Overriding or dynamic binding)

**Compile Time Polymorphism**

Compile time polymorphism means we will declare methods with same name but different signatures because of this we will perform different tasks with same method name. This compile time polymorphism also called as early binding or method overloading.

Method Overloading or compile time polymorphism means same method names with different signatures (different parameters)

Example:

using System;

namespace PolymorphismApplication

{

class Printdata

{

void print(int i)

{

Console.WriteLine("Printing int: {0}", i);

}

void print(double f)

{

Console.WriteLine("Printing float: {0}", f);

}

void print(string s)

{

Console.WriteLine("Printing string: {0}", s);

}

static void Main(string[] args)

{

Printdata p = new Printdata();

// Call print to print integer

p.print(5);

// Call print to print float

p.print(500.263);

// Call print to print string

p.print("Hello C++");

Console.ReadKey();

}

}

}

Output:

Printing int: 5

Printing float: 500.263

Printing string: Hello C++

In above class we have three methods with same name but having different input parameters this is called method overloading or compile time polymorphism or early binding.

**Run time polymorphism**

Run time polymorphism or method overriding means same method names with same signatures. Run time polymorphism also called as late binding or method overriding or dynamic polymorphism.

In this run time polymorphism or method overriding we can override a method in base class by creating similar function in derived class this can be achieved by using inheritance principle and using “virtual & override” keywords.

In base class if we declare methods with virtual keyword, then only we can override those methods in derived class using override keyword

Example:

using System;

namespace PolymorphismApplication

{

//Base Class

public class Bclass

{

public virtual void Sample()

{

Console.WriteLine("Base Class");

}

}

// Derived Class

public class DClass : Bclass

{

public override void Sample()

{

Console.WriteLine("Derived Class");

}

}

// Using base and derived class

class Program

{

static void Main(string[] args)

{

// calling the overriden method

DClass objDc = new DClass();

objDc.Sample();

// calling the base class method

Bclass objBc = new DClass();

objBc.Sample();

Console.ReadKey();

}

}

}

Output:

Derived Class

Derived Class

**Inheritance**

Inheritance is a process of deriving the new class from already existing class.

C# is a complete object oriented programming language. Inheritance is one of the primary concepts of object-oriented programming. It allows you to reuse existing code. Through effective use of inheritance, you can save lot of time in your programming and reduce errors, which in turn will increase the quality of work and productivity.

Consider a base class Shape and its derived class Rectangle:

Example:

using System;

namespace InheritanceApplication

{

class Shape

{

protected int width;

protected int height;

public void setWidth(int w)

{

width = w;

}

public void setHeight(int h)

{

height = h;

}

}

// Derived class

class Rectangle : Shape

{

public int getArea()

{

return (width \* height);

}

}

class RectangleTester

{

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.ReadKey();

}

}

}

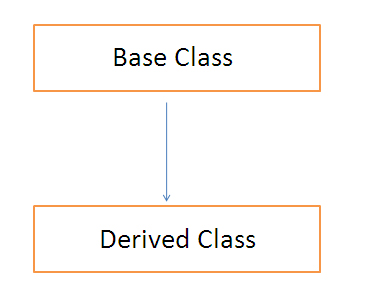
Output:

Total area: 35

**Types of inheritance in C#**

**Single Inheritance**

When a single base is been, implemented to single derived class is called as Single Inheritance. Means we have only one parent class and one child class.



using System;

namespace singleinheritance

{

class Company

{

public void CompanyName()

{

Console.WriteLine("Name of the Company");

}

public void CompanyAddress()

{

Console.WriteLine("Address of the Company");

}

}

class Employee : Company

{

public void NameofEmployee()

{

Console.WriteLine("Name of the Employee");

}

public void Salary()

{

Console.WriteLine("Salary of the Employee");

}

}

class output

{

static void Main(string[] args)

{

Employee emp = new Employee();

emp.CompanyName();

emp.CompanyAddress();

emp.NameofEmployee();

emp.Salary();

Console.ReadKey();

}

}

}

Output:

Name of the Company

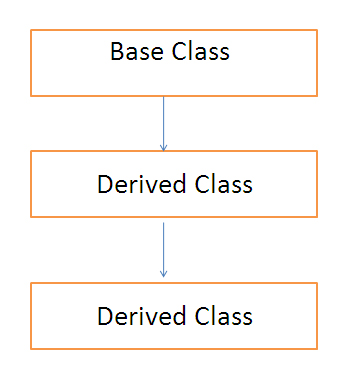
Address of the Company

Name of the Employee

Salary of the Employee

**Multilevel Inheritance**

When a derived class is created from another derived class or let me put it in a way that a class is created by using another derived class and this type of implementation is called as multilevel Inheritance.



using System;

namespace multilevelinheritance

{

class HeadOffice

{

public void HeadOfficeAddress()

{

Console.WriteLine("Head Office Address");

}

}

class BranchOffice : HeadOffice

{

public void BranchOfficeAddress()

{

Console.WriteLine("Branch Office Address");

}

}

class Employee : BranchOffice

{

public void NameofEmployee()

{

Console.WriteLine("Name of the Employee");

}

public void Salary()

{

Console.WriteLine("Salary of the Employee");

}

}

class output

{

static void Main(string[] args)

{

Employee emp = new Employee();

emp.HeadOfficeAddress();

emp.BranchOfficeAddress();

emp.NameofEmployee();

emp.Salary();

Console.ReadKey();

}

}

}

Output:

Head Office Address

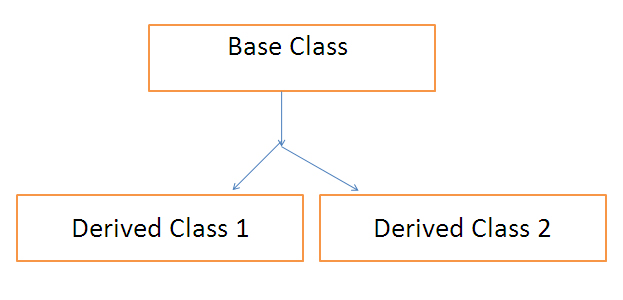
Branch Office Address

Name of the Employee

Salary of the Employee

**Hierarchical Inheritance**

When more than one derived classes are implemented from a same parent class or base class then that type of implementation is known as hierarchical inheritance.



using System;

namespace multilevelinheritance

{

class HeadOffice

{

public void HeadOfficeAddress()

{

Console.WriteLine("Head Office Address");

}

}

class BranchOffice1 : HeadOffice

{

public void BranchOfficeAddress()

{

Console.WriteLine("Branch Office Address");

}

}

class BranchOffice2 : HeadOffice

{

public void BranchOfficeAddress()

{

Console.WriteLine("Branch Office Address");

}

}

class output

{

static void Main(string[] args)

{

BranchOffice1 emp1 = new BranchOffice1();

emp1.HeadOfficeAddress();

emp1.BranchOfficeAddress();

BranchOffice2 emp2 = new BranchOffice2();

emp2.HeadOfficeAddress();

emp2.BranchOfficeAddress();

Console.ReadKey();

}

}

}

**Multiple Inheritance**

Due to the complexity of a code multiple inheritance is not been supported in C# or in DOT.NET but DOT.NET or C# supports multiple interfaces.

using System;

namespace AbstractClassMethods

{

interface ITestInterface1

{

void Add(int num1, int num2);

}

interface ITestInterface2

{

void Sub(int num1, int num2);

}

public class ImplementationClass : ITestInterface1, ITestInterface2

{

//Implement Both Add and Sub method

public void Add(int num1, int num2)

{

Console.WriteLine($"Sum of {num1} and {num2} is {num1 + num2}");

}

public void Sub(int num1, int num2)

{

Console.WriteLine($"Divison of {num1} and {num2} is {num1 - num2}");

}

}

class Program

{

static void Main()

{

ImplementationClass obj = new ImplementationClass();

//Using obj2 we can call both Add and Sub method

obj.Add(10, 20);

obj.Sub(100, 20);

Console.ReadKey();

}

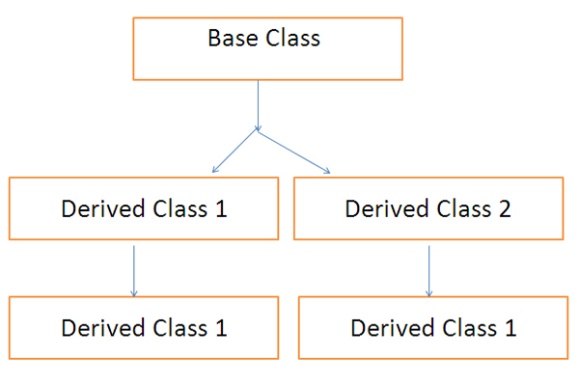
}

}



**Hybrid Inheritance**

This is a special type of inheritance and can be achieved from any combination of single, hierarchical and multi-level inheritance known as hybrid inheritance.



using System;

namespace multilevelinheritance

{

//This part of code is related to hierarchical inheritance

class HeadOffice

{

public void HeadOfficeAddress()

{

Console.WriteLine("Head Office Address");

}

}

class BranchOffice1 : HeadOffice

{

public void BranchOfficeAddress()

{

Console.WriteLine("Branch Office Address");

}

}

class BranchOffice2 : HeadOffice

{

public void BranchOfficeAddress()

{

Console.WriteLine("Branch Office Address");

}

}

////This part of code is related to combination of hierarchical inheritance and multi level inheritance

class Employee : BranchOffice2

{

public void NameofEmployee()

{

Console.WriteLine("Name of the Employee");

}

public void Salary()

{

Console.WriteLine("Salary of the Employee");

}

}

class output

{

static void Main(string[] args)

{

Employee emp1 = new Employee();

emp1.HeadOfficeAddress();

emp1.BranchOfficeAddress();

emp1.NameofEmployee();

emp1.Salary();

BranchOffice1 emp2 = new BranchOffice1();

emp2.HeadOfficeAddress();

emp2.BranchOfficeAddress();

Console.ReadKey();

}

}

}