**Object Oriented Programming (OOPs) in C#**

**Object-Oriented Programming (OOPs) in C# | OOPs Concept in C#**

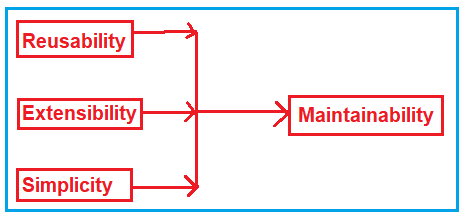
In this article, I am going to give an overview of **Object-Oriented Programming (OOPs)** in C# i.e. going to discuss the OOPs Concepts in C#. Object-Oriented Programming commonly known as OOPs is a technique, not technology. It means it doesn’t provide any syntaxes or APIs instead it provides suggestions to design and develop objects in programming languages. As part of this article, we are going to cover the following OOPs Concept in C#.

1. **What are the Problems of Functional Programming?**
2. **How we can overcome the Functional Programming Problem?**
3. **What Is Object-Oriented Programming in C#?**
4. **What are the OOPs Principles?**
5. **Why do we need Object Oriented Programming in C#?**
6. **Why do we need real-world objects in a Project?**
7. **What types of programming languages come under the OOP system?**

**What are the Problems of Functional Programming?**

Functional programming has the following problems.

1. Reusability
2. Extensibility
3. Simplicity
4. Maintainability



**Reusability:**

In Functional Programming, we need to write the same code or logic at multiple places which increases the code duplication. Later if we want to change the logic, then we need to change it at all places.

**Extensibility:**

It is not possible in functional programming to extend the features of a function. Suppose you have a function and you want to extend it with some additional features then it is not possible. You have to create a completely new function and then change the function as per your requirement.

**Simplicity:**

As extensibility and reusability are not possible in functional programming, usually we end up with lots of functions and lots of scattered code.

**Maintainability:**

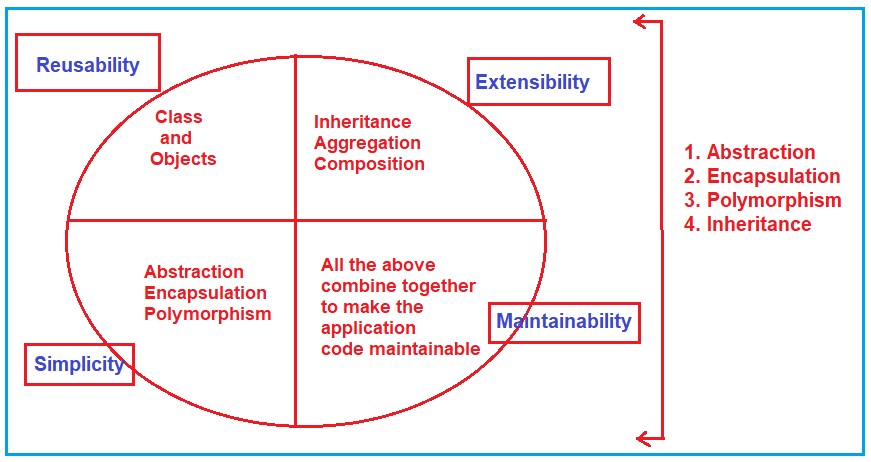
As we don’t have Reusability, Extensibility, and Simplicity in functional Programming, so it is very difficult to manage and maintain the application code.

**How we can overcome Functional Programming Problems?**

We can overcome the functional programming problems (Reusability, Extensibility, Simplicity, and Maintainability) using Object-Oriented Programming. OOPs provide some principles and using those principles we can overcome the functional programming problems.

**What Is Object-Oriented Programming?**

Let us understand Object-Oriented Programming i.e. OOPs Concepts using C#. Object-Oriented Programming (OOPs) in C# is a design approach where we think in terms of real-world objects rather than functions or methods. Unlike procedural programming language, here in oops, programs are organized around objects and data rather than action and logic. Please have a look at the following diagram to understand this better.



**Reusability:**

To address reusability, object-oriented programming provides something called Classes and Objects. So, rather than copy-pasting the same code again and again in different places what you can do here is, create a class and make an instance of the class which is called object, and reuses them whenever you want.

**Extensibility:**

Suppose you have a function and you want to extend it with some new features that were not possible with functional programming. You have to create a completely new function and then change the whole function whatever you want. In OOPs, this problem is addressed by using some concepts called Inheritance, Aggregation, and Composition. In our upcoming article, we will discuss all these concepts in detail.

**Simplicity:**

Because we don’t have extensibility and reusability in functional programming, we end up with lots of functions and lots of scattered code. In OOPs, this problem is addressed by using some concepts called Abstraction, Encapsulation, and Polymorphism.

**Maintainability:**

As OOPs address Reusability, Extensibility, and Simplicity, we have good maintainable code and clean code which increases the maintainability of the application.

**What are the OOPs Principles or OOPs Concepts in C#?**

OOPs, provide 4 principles. They are

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

Let’s understand the definitions of the OOPs Principle in this article. From the next article onwards we will discuss all these principles in detail using some real-time examples.

**What are Abstraction and Encapsulation?**

The process of representing the essential features without including the background details is called [**Abstraction**](https://dotnettutorials.net/lesson/abstraction-csharp-realtime-example/). In simple words, we can say that it is a process of defining a class by providing necessary details to call the object operations (i.e. methods) by hiding or removing its implementation details.

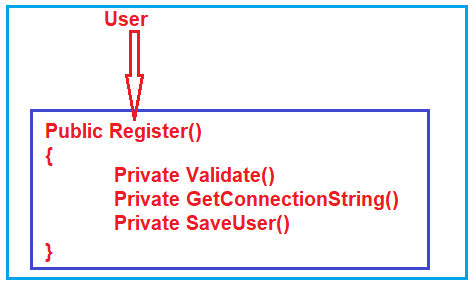
The process of binding the data and functions together into a single unit (i.e. class) is called [**Encapsulation**](https://dotnettutorials.net/lesson/encapsulation-csharp/). In simple words, we can say that it is a process of creating a class by hiding its internal data members from outside the class and accessing those internal data members only through publicly exposed methods or properties. Data encapsulation is also called data hiding because by using this principle we can hide the internal data from outside the class.

Abstraction and Encapsulation are related to each other. We can say that Abstraction is logical thinking whereas Encapsulation is its physical implementation.

**Understanding Abstraction and Encapsulation with an Example:**

Let us understand Abstraction and Encapsulation with an example. Suppose you want to design one class for providing the register functionality of a user. For that what you need to do is, first you need to get the data and validate the data, then you need to get the connection string for the database, finally, you need to save the data in the database. And for this, you have three methods i.e. Validate, GetConnectionString, and SaveUser. If you provide access to these three methods to the users of this class, then he may end up calling these methods in the wrong order or it may be possible that he may forget to call any of these methods.

Here, what you need to do is, you need to create one method called Register and as part of that method you need to call all these methods (Validate, GetConnectionString, and SaveUser). And finally, you need to give access to the Register method instead of the Validate, GetConnectionString, and SaveUser methods. This is what we discuss is nothing but abstraction. How you implement this is nothing but encapsulation. So, here, you need to create the Validate, GetConnectionString, and SaveUser methods with private access specifier so that the user cannot access these methods. And make the Register method as Public so that the user can access this method as shown below.



**What is Inheritance?**

The process by which the members of one class are transferred to another class is called [**inheritance**](https://dotnettutorials.net/lesson/inheritance-c-sharp/). The class from which the members are transferred is called the Parent/base class and the class which inherits the members of the Parent class is called the Derived/ child class. We can achieve code**extensibility** through inheritance.

**What is Polymorphism?**

The word [**Polymorphism**](https://dotnettutorials.net/lesson/polymorphism-csharp/)is derived from the Greek word, where Poly means many and morph means faces/ behaviors. So, the word polymorphism means the ability to take more than one form. Technically, we can say that when the same function/operator will show different behaviors by taking different types of values or with a different number of values called [**Polymorphism**](https://dotnettutorials.net/lesson/polymorphism-csharp/). There are two types of polymorphism

1. Static polymorphism/compile-time polymorphism/Early binding
2. Dynamic polymorphism/Run time polymorphism/Late binding

Static polymorphism is achieved by using **function overloading and operator overloading** whereas dynamic polymorphism is achieved by using **function overriding**.

**Why do we need Object-Oriented Programming (OOPs) in C#?**

If you want to represent the real-world objects in a programming language for automating the business by achieving **Reusability, Extensibility, Simplicity, and Maintainability**, then you need the concept OOPs. OOPs provide some principles and by using those principles we can develop real-world objects in a programming language by achieving Reusability, Extensibility, Simplicity, and Maintainability.

All living and non-living things are considered objects. So the real-world objects such as Person, Animal, Bike, Computer, etc. can be developed in object-oriented programming languages by using the OOPs concept.

**Why do we need real-world objects in a Project?**

We need real-world objects in a project because real-world objects are part of our business. As we are developing applications (software) for automating the business, we must have to create the business-related real-world objects in the project.

For example, to automate the Bank business we must create real-world objects like Customer, Manager, Clerk, Office Assistant, Marketing Executive, Computer, Printer, Chair, table, etc. So along with the Bank object, we must also have to create all the above objects because without all the above objects we cannot run a Bank business. Technically we call the above objects are business objects.

**What types of programming languages come under the OOP system?**

The programming languages which implement all the four principles provided by OOPs are called object-oriented programming languages. Examples: Java, .Net, C++, etc.

**Points to Remember:**

1. Object-Oriented Programming Principles or OOPs Concepts in C# are design principles that suggest how we should develop a program so that we can reuse it from other layers of the project effectively and with high scalability.
2. Scalability means we have to develop the project in such a way that it should accept future changes without doing major changes in the project, that small change also should be accepted from external files like properties files, XML files, etc. Scalability is achieved by developing classes by integrating them in a loosely coupled way.
3. We should develop the project with scalability as there will be a growth in business, according to the growth in business we must add required changes to the project with minimal modifications.
4. As a developer, we must remember that in the initial stage of business customer never makes a significant investment. As the business grows customers increase investment according to the growing new requirements are added to the projects. To add those new requirements we should not design the project entirely.
5. So we must design the project by following OOPs principles strictly even though they are not needed at the initial stage but for accepting future changes

**Class and Objects in C#**

**Class and Objects in C# with Examples**

In this article, I am going to discuss **Class and Objects in C#** with examples. Please read our previous article before proceeding to this article where we discussed the basics concepts of [**Object-Oriented Programming**](https://dotnettutorials.net/lesson/object-oriented-programming-csharp/). Understanding class and objects in C# is very important for you as a developer. This is because everything in C# is in the form of class and objects. As part of this article, we are going to discuss the following pointers in detail.

1. **Class and Objects from Layman Point of View.**
2. **Class and Objects from Programming Language Point of View.**
3. **How can we create a Class and Object in C#?**
4. **Difference between Class and Objects in C#**
5. **Types of classes in C#**

**Class and Objects in C#**

As we already discussed in our previous article, class, and objects addresses the reusability functionality. Again we discussed in Object-Oriented Programming, we need to think in terms of objects rather than functions. So, let us discuss what exactly classes and objects are from the Layman point of view as well as from the programming point of view.

**Class and Objects from Layman Point of View.**

From the layman’s point of view, we can define a class as a blueprint of a specific object. Every living and non-living thing is considered as objects such as cars, People, Places, etc. Again, each and every object has some color, shape, properties, and functionalities.

For example, consider the luxury car Ferrari. Here, **Ferrari** is an **object** of the luxury car type. The **luxury car** is a **class** that specifies some characteristics such as speed, color, shape, etc. So any car manufactures company that makes a car and if that car meets all those requirements, then it is an object of the luxury car type.

If you take the example BMW, Lamborghini, and Cadillac, then all these cars are an object of the ‘Luxury Car’ class. Here, ‘**Luxury Ca**r’ is a class and every single car (BMW, Lamborghini, and Cadillac) is an object of the luxury car class. Now, let us understand what exactly classes and objects are from the programming point of view.

**Class and Objects from Programming Language Point of View.**

Here we are going to understand the class and objects from the C# programming language point of view. But this is also applicable to any object-oriented programming language like java and c++.

**Class:**

A class is simply a user-defined data type that represents both state and behavior. The state represents the properties and **behavior** is the action that objects can perform.

In other words, we can say that a class is the blueprint/plan/template that describes the details of an object. A class is a blueprint from which the individual objects are created. In C#, a Class is composed of three things i.e. a name, attributes, and operations.

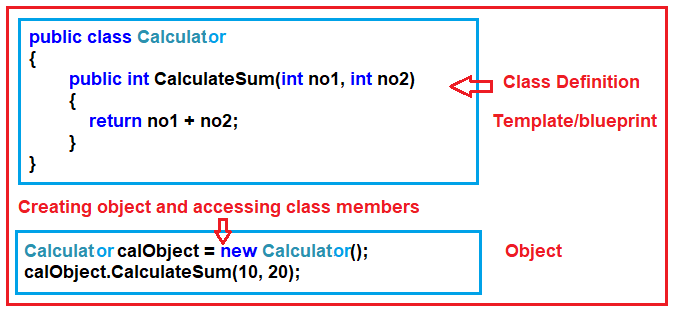
**Objects:**

It is an instance of a class. A class is brought live by creating objects. An object can be considered as a thing that can perform activities. The set of activities that the object performs defines the object’s behavior.

All the members of a class can be accessed through the object. To access the class members, we need to use the dot (.) operator. The dot operator links the name of an object with the name of a member of a class.

**How can we create a Class and Object in C#?**

Let us understand how to create class and object in C#. In order to understand this, please have a look at the following image. As you can see in the below image, a class definition starts with the keyword **class** followed by the class name (here the class name is Calculator), and the class body is enclosed by a pair of curly braces. As part of the class body, you define class members (properties, methods, variables, etc.). Here as part of the body, we define one method called CalculateSum. The class Calculator is just a template. In order to use this class or template, you need an object. As you can see in the second part of the image, we create an object of the class Calculator using the new keyword. And then store the object reference on the variable calObject which is of type Calculator. Now, using this calObject object we can access the class members using a dot.



So, the point that you need to remember is, to create a class you need to use the class keyword while if you want to create an object of a class then you need to use the new keyword. Once you create the object then you can access the class members using the object.

**The complete example code is given below.**

**using** *System;*

**namespace** *ClassObjectsDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Creating object

Calculator calObject = new Calculator**()**;

//Accessing Calculator class member using Calculator class object

**int** result = calObject.CalculateSum**(**10, 20**)**;

Console.WriteLine**(**result**)**;

Console.ReadKey**()**;

**}**

**}**

//Defining class or blueprint or template

**public** **class** Calculator

**{**

**public** **int** CalculateSum**(int** no1, **int** no2**)**

**{**

**return** no1 + no2;

**}**

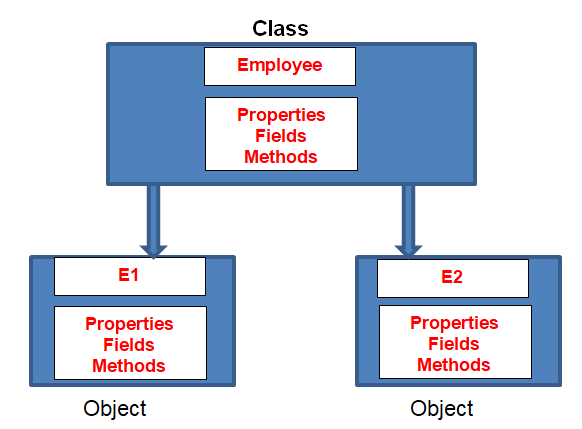
**}**

**}**

**Output**: 30

**Difference between Class and Objects in C#**

Many programmers or developers still get confused by the difference between class and object. As we already discussed, in object-oriented programming, a Class is a template or blueprint for creating Objects, and every Object in C# must belong to a Class. Please have a look at the following diagram to understand the difference between them.



As you can see in the above image, here we have one class called “Employee”. All the Employees having some properties such as employee id, name, salary, gender, and department, etc. These properties are nothing but the attributes (properties or fields) of the Employee class.

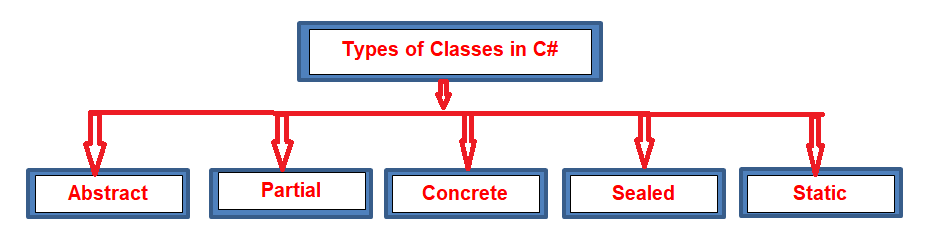
If required you can also add some methods (functions) that are common to all Employees such as InsertData and DisplayData to insert and display the Employee Data.

So, the idea is that the template or blueprint of the Employee is not going to change. Each and every Object is going to build from the same template (Class) and therefore contains the same set of methods and properties. Here, all Objects share the same template but maintain a separate copy of the member data (Properties or fields).

For example: if we create two employees, let’s say e1 and e2, then both e1 and e2 are Employees, so they can be classified as belonging to the Employee class. Both have the same methods (InsertData and DisplayData) but are different in models (properties or fields)

**Types of classes in C#:**

Please have a look at the following image.



In C# we have below types of classes

1. Abstract class
2. Concrete class
3. Sealed class
4. Partial Class
5. Static class

**Constructors in C#**

**Constructors in C# with Examples**

In this article, I am going to discuss the **Constructors in C#** with examples. Please read our previous article before proceeding to this article where we discussed [**how to create classes and objects in C#**](https://dotnettutorials.net/lesson/class-and-objects-csharp/) with examples. As part of this article, we are going to discuss the following pointers in detail with are related to C# constructors.

1. **What is a Constructor in C#?**
2. **Rules to follow while creating Constructors in C#.**
3. **What a Constructor can have in C#?**
4. **Can we define a method with the same class name in C#?**
5. **How many types of constructors are there in C#.Net?**
6. **What is Default Constructor?**
7. **When do we need to provide the constructor explicitly?**
8. **What is a User-Defined Default Constructor?**
9. **When should we define a parameterized constructor in a class?**
10. **What is a Parameterized Constructor?**
11. **How many constructors can be defined in a class?**
12. **What is Copy Constructor?**
13. **Understanding Static Constructor.**
14. **Can we initialize non-static data members within a static constructor?**
15. **Is it possible to initialize static data fields within a non-static constructor?**
16. **Can we initialize static data fields in both static and non-static constructors?**
17. **What is a Private Constructor?**
18. **Understanding constructor overloading?**

**What is a Constructor in C#?**

In simple words, we can define the constructors in C# are the special types of methods of a class that are automatically executed whenever we create an instance (object) of that class. The Constructors are responsible for two things. One is the object initialization and the other one is memory allocation. The role of the new keyword is to create the object.

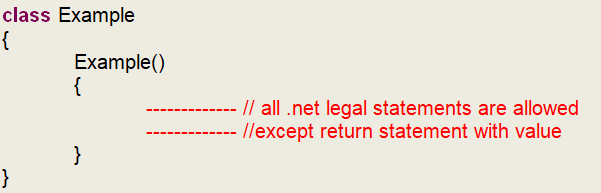
**Rules to follow while creating the C# Constructors:**

1. The constructor name should be the same as the class name.
2. It should not contain return type even void also.
3. The constructor should not contain modifiers.
4. As part of the constructor body return statement with value is not allowed.

**What does a Constructor have in C#?**

1. It can have all five accessibility modifiers.
2. The constructor can have parameters.
3. It can have a throws clause it means we can throw an exception from the constructor.
4. The constructor can have logic, as part of logic it can have all C#.NET legal statements except return statements with value.
5. We can place a return; in the constructor.

**Syntax:**



**Example: Program to show the use of the Constructor.**

**namespace** *ConstructorDemo*

**{**

**class** Program

**{**

Program**()**

**{**

Console.WriteLine**(**"This is Constructor"**)**;

**}**

**public** **static** **void** Main**(**string**[]** args**)**

**{**

Program p = new Program**()**;

Console.WriteLine**(**"Main method"**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**OUTPUT:**

C# Constructors with Examples

**Can we define a method with the same class name in C#?**

No, it is not allowed to define a method with the same class name in C#. It will give you a compile-time error.

**How many types of constructors are there in C#.net?**

There are five types of constructors available in C#, they are as follows

1. **Default Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor**
4. **Static Constructor**
5. **Private Constructor**

Let’s discuss each of these constructors in detail with examples.

**Default Constructor in C#:**

The Constructor without parameter is called a default constructor. Again the default constructor is classified into two types.

1. System-defined default constructor
2. User-defined default constructor

**What is System Defined Default Constructor in C#?**

As a programmer, if you are not defined any constructor explicitly in your program, then by default the system will provide one constructor at the time of compilation. That constructor is called a default constructor. The default constructor will assign default values to the data members (non-static variables). As this constructor is created by the system this is also called a system-defined default constructor.

**Let us see an example for a better understanding of the system-defined default constructor.**

**namespace** *ConstructorDemo*

**{**

**class** Employee

**{**

**int** eid, eage;

String eaddress, ename;

**public** **void** Display**()**

**{**

Console.WriteLine**(**"\nemployee id is: " + eid**)**;

Console.WriteLine**(**"employee name is: " + this.ename**)**;

Console.WriteLine**(**"employee age is: " + this.eage**)**;

Console.WriteLine**(**"employee address is: " + eaddress**)**;

**}**

**}**

**class** Test

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Employee e1 = new Employee**()**;

Employee e2 = new Employee**()**;

e1.Display**()**;

e2.Display**()**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Note:** The point that you need to keep in mind is that the System will only provide the default constructor if as a programmer you are not defined any constructor explicitly.

**When do we need to provide the constructor explicitly?**

If you want to execute some logic at the time of object creation, that logic may be object initialization logic or some other useful logic, then as a developer, we must provide the constructor explicitly.

**What is a User-Defined Default Constructor in C#?**

The constructor which is defined by the user without any parameter is called a user-defined default constructor. This constructor does not accept any argument but as part of the constructor body, you can write your own logic.

**Let’s understand the user-defined default constructor in C# with an example:**

**namespace** *ConstructorDemo*

**{**

**class** Employee

**{**

**int** eid, eage;

string eaddress, ename;

**public** Employee**()**

**{**

this.eid = 100;

eage = 30;

this.ename = "Pranaya";

eaddress = "MUMBAI";

**}**

**public** **void** Display**()**

**{**

Console.WriteLine**(**"employee id is: " + eid**)**;

Console.WriteLine**(**"employee name is: " + this.ename**)**;

Console.WriteLine**(**"employee age is: " + this.eage**)**;

Console.WriteLine**(**"employee address is: " + eaddress**)**;

**}**

**}**

**class** Test

**{**

**public** **static** **void** Main**(**string**[]** args**)**

**{**

Employee e1 = new Employee**()**;

Employee e2 = new Employee**()**;

e1.Display**()**;

e2.Display**()**;

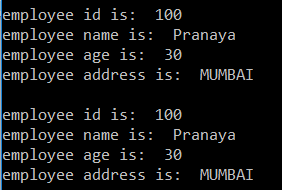
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



The drawback of the above user-defined default constructor is every instance (i.e. object) of the class will be initialized (assigned) with the same values. That means it is not possible to initialize each instance of the class with different values.

**When should we define a parameterized constructor in a class?**

If you want to initialize the object dynamically with the user-given values then you need to use the parameterized constructor. The advantage is that you can initialize each object with different values.

**What is Parameterized Constructor in C#?**

The developer given constructor with parameters is called the parameterized constructor in C#. With the help of a Parameterized constructor, we can initialize each instance of the class with different values. That means using parameterized constructor we can store a different set of values into different objects created to the class.

**Let us understand the parameterized constructor in C# with one example.**

**namespace** *ConstructorDemo*

**{**

**class** Employee

**{**

**int** eid, eage;

String eaddress, ename;

**public** Employee**(int** id, **int** age, string name, string address**)**

**{**

this.eid = id;

this.eage = age;

this.ename = name;

this.eaddress = address;

**}**

**public** **void** Display**()**

**{**

Console.WriteLine**(**"employee id is: " + eid**)**;

Console.WriteLine**(**"employee name is: " + this.ename**)**;

Console.WriteLine**(**"employee age is: " + this.eage**)**;

Console.WriteLine**(**"employee address is: " + eaddress**)**;

**}**

**}**

**class** Test

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Employee e1 = new Employee**(**101, 30, "Pranaya", "Mumbai"**)**;

Employee e2 = new Employee**(**101, 28, "Rout", "BBSR"**)**;

e1.Display**()**;

e2.Display**()**;

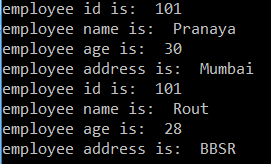
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**How many constructors can be defined in a class in C#?**

In C#, within a class, we can define any number of constructors. But the most important point that you need to remember is that each and every constructor must have a different signature. A different signature means the number, type, and parameter order should be different. So in a class, we can define **one no-argument** constructor plus **‘n’ number of parameterized** constructors in C#.

**What is Copy Constructor in C#?**

The constructor which takes a parameter of the class type is called a copy constructor. This constructor is used to copy one object’s data into another object. The main purpose of the copy constructor is to initialize a new object (instance) with the values of an existing object (instance).

**Let us understand the copy constructor in C# with one example:**

**namespace** *ConstructorDemo*

**{**

**class** Employee

**{**

**int** eid, age;

string address, name;

**public** Employee**()**

**{**

Console.WriteLine**(**"ENTER EMPLOYEE DETAILS"**)**;

Console.WriteLine**(**"Enter the employee id"**)**;

this.eid = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"Enter the employee age"**)**;

this.age = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"Enter the employee name"**)**;

this.name = Console.ReadLine**()**;

Console.WriteLine**(**"Enter the employee address:"**)**;

this.address = Console.ReadLine**()**;

**}**

**public** Employee**(**Employee tempobj**)**

**{**

this.eid = tempobj.eid;

this.age = tempobj.age;

this.name = tempobj.name;

this.address = tempobj.address;

**}**

**public** **void** Display**()**

**{**

Console.WriteLine**()**;

Console.WriteLine**(**"Employee id is: " + this.eid**)**;

Console.WriteLine**(**"Employee name is: " + this.name**)**;

Console.WriteLine**(**"Employee age is: " + this.age**)**;

Console.WriteLine**(**"Employee address is: " + this.address**)**;

**}**

**}**

**class** Test

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Employee e1 = new Employee**()**;

Employee e2 = new Employee**(**e1**)**;

e1.Display**()**;

e2.Display**()**;

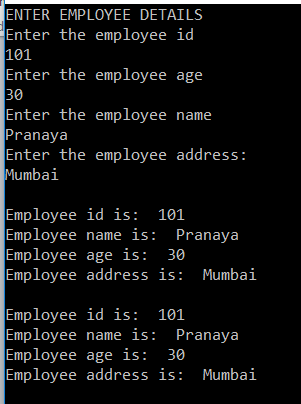
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Understanding Static Constructor in C#:**

In C#, it is also possible to create a constructor as static and when we do so, it is called Static Constructor. The static Constructor in C# will be invoked only once. There is no matter how many numbers instances (objects) of the class are created, it is going to be invoked only once and that is when the class is load for the first time.

The static constructor is used to initialize the static fields of the class. You can also write some code inside the static constructor which is going to be executed only once. The static data members in C# are created only once even though we created any number of objects.

**Points to Remember while creating Static Constructor in C#:**

1. There can be only one static constructor in a class.
2. The static constructor should be without any parameters.
3. It can only access the static members of the class.
4. There should not be any access modifier in the static constructor definition.
5. If a class is static then we cannot create the object for the static class.
6. Static constructor will be invoked only once i.e. at the time of first object creation of the class, from 2nd object creation onwards static constructor will not be called.

**Let us understand Static Constructor with an example.**

**namespace** *ConstructorDemo*

**{**

**class** Example

**{**

**int** i;

**static** **int** j;

**public** Example**()**

**{**

i = 100;

**}**

**static** Example**()**

**{**

j = 100;

**}**

**public** **void** Display**()**

**{**

Console.WriteLine**(**"value of i : " + i**)**;

i++;

Console.WriteLine**(**"value of j : " + j**)**;

j++;

**}**

**}**

**class** Test

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Example e1 = new Example**()**;

e1.Display**()**;

e1.Display**()**;

Example e2 = new Example**()**;

e2.Display**()**;

e2.Display**()**;

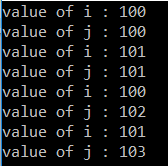
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Can we initialize non-static data members within a static constructor in C#?**

It is not possible to initialize non-static data members within a static constructor, it raises a compilation error. Have a look at the following example.

**static** Example**()**

**{**

**int** = 101;//not allowed

j = 100;

**}**

**Can we initialize static data fields within a non-static constructor in C#?**

Yes, you can initialize static data members within a non-static constructor but after then they lose their static nature. Consider the following example:

**public** Example**()**

**{**

i = 100;

j = 100; //allows but j lose its static nature

**}**

**Can we initialize static data fields in both static and non-static constructors?**

Yes, we can initialize static data fields in both static and non-static constructors but static data fields lose their static nature. For example

**public** Example**()**

**{**

i = 100;

j = 100; //allows but j looses its static nature

**}**

**static** Example**()**

**{**

j = 105; //allowed

**}**

**What is Private Constructor in C#?**

In C#, it is also possible to create a constructor as private. The constructor whose accessibility is private is known as a private constructor. When a class contains a private constructor then we cannot create an object for the class outside of the class. So, private constructors are used to creating an object for the class within the same class. Generally, private constructors are used in the Remoting concept.

**Let us see an example for understanding private constructor**

**namespace** *ConstructorDemo*

**{**

**class** Program

**{**

**private** Program**()**

**{**

Console.WriteLine**(**"this is private constructor"**)**;

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

Program p = new Program**()**;

Console.WriteLine**(**"main method"**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**What is constructor overloading?**

When we define multiple constructors within a class with different parameter types, numbers and orders then it is called constructor overloading.

**Let us see an example for a better understanding of the Constructor Overloading in C#.**

**namespace** *ConstructorDemo*

**{**

**class** ConstructorOverloading

**{**

**int** x;

**public** ConstructorOverloading**()**

**{**

this.x = 10;

**}**

**public** ConstructorOverloading**(int** x**)**

**{**

this.x = x;

**}**

**public** **void** Display**()**

**{**

Console.WriteLine**(**"the value of x:{0}", x**)**;

**}**

**}**

**class** Test

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

ConstructorOverloading obj1 = new ConstructorOverloading**()**;

ConstructorOverloading obj2 = new ConstructorOverloading**(**20**)**;

obj1.Display**()**;

obj2.Display**()**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Constructors Overloading in C#

**Destructor in C#**

**Destructor in C# with Example**

In this article, I am going to discuss the **Destructor in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**Constructors in C#**](https://dotnettutorials.net/lesson/constructors-csharp/)with examples. As part of this article, we are going to discuss the following pointers related to Destructor.

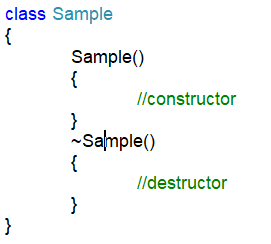
1. **What is Destructor in C#**
2. **When is the destructor called in c#?**
3. **When will be the object of a class get destroyed in C#?**

**What is Destructor in C#?**

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 prefix with ~ tilde. The constructor in C# is called when the object of the class is created. On the other hand, the destructor in C# is gets executed when the object of the class is destroyed.

The Constructor and destructor methods will exactly have the same name as the class to which they belong. So to differentiate between these two a tilde (~) operator is used before the destructor method.

**For example:**



**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.

**When is a destructor method called in c#?**

A destructor method gets called when the object of the class is destroyed.

**When will be 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

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

**Let us see an example for a better understanding of Destructor in C#**

**namespace** *DestructorExample*

**{**

**class** DestructorDemo

**{**

**public** DestructorDemo**()**

**{**

Console.WriteLine**(**"constructor object created"**)**;

**}**

~DestructorDemo**()**

**{**

Console.WriteLine**(**"object is destroyed"**)**;

**}**

**}**

**class** Test

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

DestructorDemo obj1 = new DestructorDemo**()**;

DestructorDemo obj2 = new DestructorDemo**()**;

obj1 = **null**;

obj2 = **null**;

GC.Collect**()**;

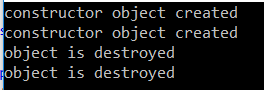
Console.ReadKey**()**;

**}**

**}**

**}**

**OUTPUT:**



**Access Specifiers in C#**

**Access Specifiers in C# with Examples**

In this article, I am going to discuss the **Access Specifiers in C#** with Examples. Please read our previous article before proceeding to this article where we discussed the [**Destructor in C#**](https://dotnettutorials.net/lesson/destructor-csharp/) with an example. As part of this article, we are going to discuss the following pointers which are related to the C# access specifiers.

1. **What are Access Specifiers in C#?**
2. **Types of Access Specifiers supported by dot net.**
3. **Understanding Type and Type members in C#.**
4. **Understand Private, Public, Protected, Internal, and ProtectedInternal access specifiers with examples.**

**What are Access Specifiers in C#?**

The Access Specifiers in C# are also called access modifiers which are used to define the scope of the type (class and interface) as well as the scope of their members (variables, properties, and methods). That is who can access them and who cannot access them are defined by the Access Specifiers.

**Types of Access Specifiers in C#:**

C# supports 5 types of access specifiers. They are as follows

1. **Private**
2. **Public**
3. **Protected**
4. **Internal**
5. **Protected Internal**

**Note:** Members that are defined in a type with any scope or specifiers are always accessible within that type; restriction comes into the picture only when they try to access them outside of the type.

**Understand Type and Type members in C#:**

Before going to understand Access Specifier in C#, let us first understand what are Types and Type Members are. In the below example, **Customer** is the **Type** and variables (**\_id, \_firstName, \_lastName**), Properties (**Id, FirstName, LastName**) and method **GetFullName()** are type members.

**namespace** *AccessSpecifierDemo*

**{**

**public** **class** Customer

**{**

#region Private Fields

**private** **int** \_id;

**private** string \_firstName;

**private** string \_lastName;

#endregion

#region Properties

**public** **int** Id

**{**

**get** **{** **return** \_id; **}**

**set** **{** \_id = **value**; **}**

**}**

**public** string FirstName

**{**

**get** **{** **return** \_firstName; **}**

**set** **{** \_firstName = **value**; **}**

**}**

**public** string LastName

**{**

**get** **{** **return** \_lastName; **}**

**set** **{** \_lastName = **value**; **}**

**}**

#endregion

#region Methods

**public** string GetFullName**()**

**{**

**return** this.\_firstName + " " + this.\_lastName;

**}**

#endregion

**}**

**}**

So in general **classes, structs, enums, interfaces, delegates** are called **types,**and **variables, properties, constructors, methods,**etc. that normally reside within a type are called **type members.**The**Type members** can have all the 5 access modifiers whereas **types** can have only 2 (**internal, public**) access modifiers

**Note:** The **customer** class makes use of regions. Using **regions** we can expand and collapse sections of our code either manually, or using visual studio **Edit** -> **Outlining** -> **Toggle All Outlining**

Let’s discuss each access specifiers in C# with some examples. For this create a new console application with the name **AccessSpecifierDemo**. First, we will discuss Access Specifiers with the Type Members and then we will discuss Access Specifiers with the Type.

**Private and Public Access Specifiers in C#:**

Private members are available only within the containing type whereas public members are available anywhere. There is no restriction for public members. Let understand Private and Public Members with an example:

**namespace** *AccessSpecifierDemo*

**{**

**public** **class** Customer

**{**

**private** **int** \_id;

**public** **int** Id

**{**

**get**

**{**

**return** \_id;

**}**

**set**

**{**

\_id = **value**;

**}**

**}**

**}**

**public** **class** MainClass

**{**

**private** **static** **void** Main**()**

**{**

Customer CustomerInstance = new Customer**()**;

CustomerInstance.Id = 101;

// Compiler Error: 'Customer.\_id' is inaccessible due to its protection level

// CustomerInstance.\_id = 101;

**}**

**}**

**}**

In the above example, \_id variable is private. So, this member is only available within the Customer class (Containing Type). It is a compile-time error to access \_id outside of the Customer Class. The following line in the MainClass will generate a compiler error stating, **‘Customer.\_id’ is inaccessible due to its protection leve**l.  
**CustomerInstance.\_id = 101;**

On the other hand, the Id property is a public member. So, we can access this member anywhere even outside of the Customer class. In fact, we invoke the Id property of the Customer class in the Main() method of MainClass as shown below and it will not give us any error.  
**CustomerInstance.Id = 101;**

**Protected Access Specifier in C#:**

Protected Members in C# are available within the containing type as well as to the types that are derived from the containing type. Let us understand this with an example.

**namespace** *AccessSpecifierDemo*

**{**

**public** **class** Customer

**{**

**protected** **int** ID = 101;

**public** **void** PrintID**()**

**{**

//Protected member ID is accessible with in Customer class

Console.WriteLine**(**this.ID**)**;

**}**

**}**

**public** **class** CorporateCustomer : Customer

**{**

**public** **void** PrintCustomerID**()**

**{**

CorporateCustomer corporateCustomerInstance = new CorporateCustomer**()**;

// Can access the base class protected instance member using the derived class object

Console.WriteLine**(**corporateCustomerInstance.ID**)**;

// Can access the base class protected instance member using this or base keyword

Console.WriteLine**(**this.ID**)**;

Console.WriteLine**(base**.ID**)**;

**}**

**}**

**public** **class** RetailCustomer

**{**

**public** **void** PrintCustomerID**()**

**{**

RetailCustomer retailCustomerInstance = new RetailCustomer**()**;

//RetailCustomer class is not deriving from Customer class, hence it is an error

//to access Customer class protected ID member, using the retailCustomerInstance

//Console.WriteLine(retailCustomerInstance.ID); //Error

//Both these below lines also produce the same Error

//Console.WriteLine(this.ID); // Error

//Console.WriteLine(base.ID); // Error

**}**

**}**

**}**

The customer class defines a protected member ID. CorporateCustomer class derives from the Customer class so protected ID member is accessible in the Customer class (Containing Type) and also from the CorporateCustomer class (Derived Type). Within the PrintID() method in the Customer class Protected member ID is accessible.  
**Console.WriteLine(this.ID);**

There are 3 ways to access the base class-protected member in the derived class as shown below.

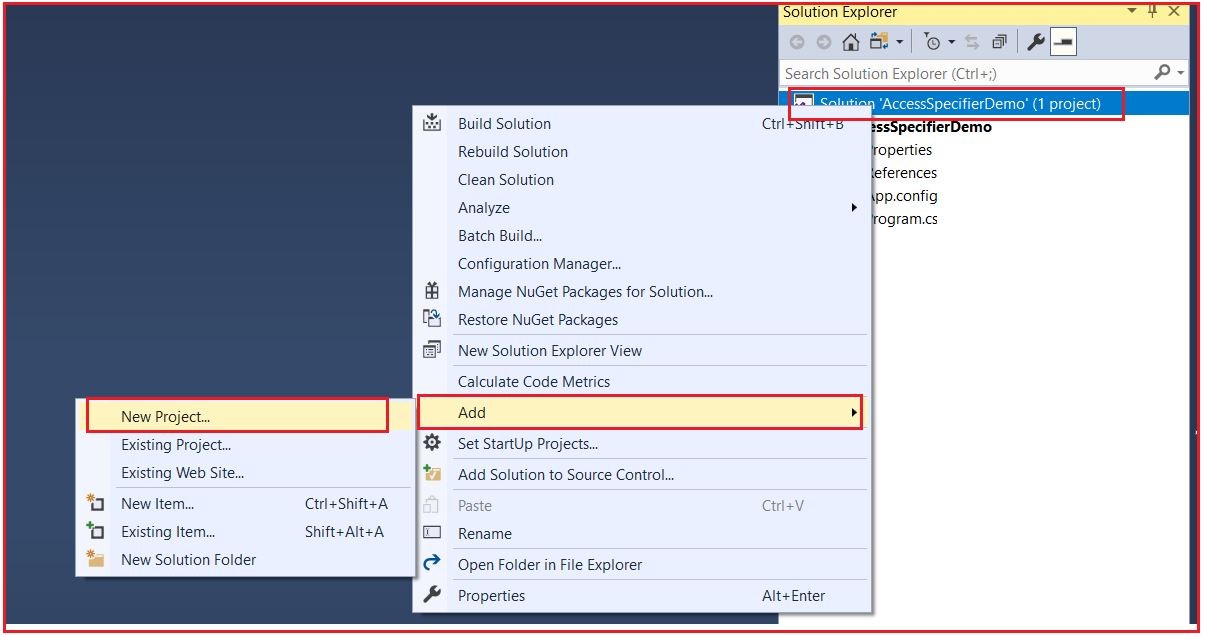
1. Using the derived class object: **Console.WriteLine(corporateCustomerInstance.ID);**
2. Using this keyword: **Console.WriteLine(this.ID);**
3. Using the base keyword: **Console.WriteLine(base.ID);**

On the other hand, the RetailCustomer class is not deriving from the Customer class hence it’s a compile-time error to access Customer class-protected ID member.

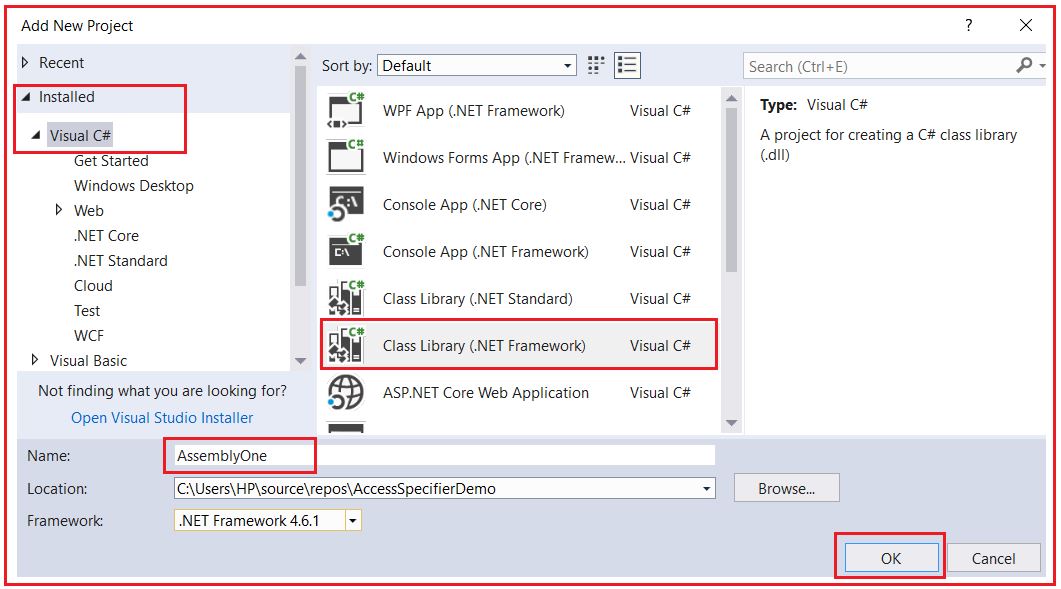
**Internal Access Specifier in C#:**

Whenever a member is declared with Internal Access Specifier in C#, then it is available anywhere within the containing assembly. It’s a compile-time error to access an internal member from outside the containing assembly. So, To understand the Internal Access Specifier in C#, we need 2 assemblies. To generate 2 assemblies, we need to follow the below steps.

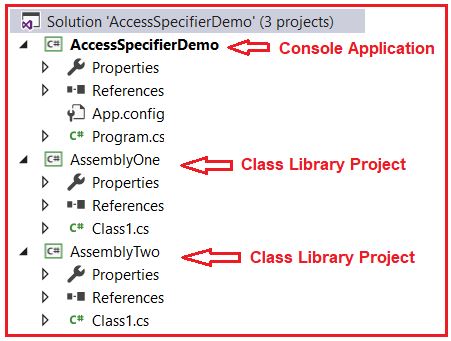
Right-click on the Solution Explorer and then select Add -> New Project option from the context menu as shown in the below image.



Once you click on the New Project, it will open the following Add New Project Dialog Box. Here, select **Visual C#** from the **Installed** section, and then select**Class Library** from the center pane, and provide the name as **AssemblyOne** and finally click on the OK button as shown in the below image.



Once you click on the OK button, it will add the Class Library Project with the name **AssemblyOne** to our solution. Again, repeat the process to create another class library project with the name AssemblyTwo to our solution. If you have followed the steps correctly, now we should have three projects in the solution explorer as shown in the below image.



Now if we build the solution, we should have 3 assemblies generated. Two DLLs and one exe. To locate the physical assembly follow these steps.

1. Right-click on the AssemblyOne project, in solution explorer and select Open Folder in Windows Explorer.
2. Open bin folder
3. Now open Debug folder
4. In the Debug folder, you should see AssemblyOne.dll, which is the physical assembly.

**Creating Class in AssemblyOne Project:**

Now, create a class file with the name **AssemblyOneClass.cs** within the AssemblyOne Project, and once you create the class file, then Copy and paste the following code into it.

**namespace** *AssemblyOne*

**{**

**public** **class** AssemblyOneClassI

**{**

**internal** **int** ID = 999;

**}**

**public** **class** AssemblyOneClassII

**{**

**public** **void** Test**()**

**{**

AssemblyOneClassI instance = new AssemblyOneClassI**()**;

// Can access inetrnal member ID as both AssemblyOneClassII and AssemblyOneClassI

// are present in the same assembly

Console.WriteLine**(**instance.ID**)**;

**}**

**}**

**}**

In this example, **AssemblyOneClassI**has an **internal member ID**. We can access this ID member from **AssemblyOneClassII** because this class is also present in the same assembly as **AssemblyOneClassI**. So, this proofs that the Internal Members access anywhere within the same assembly.

**Creating Class in AssemblyTwp Project:**

Now, create a class file with the name **AssemblyTwoClass.cs** within the AssemblyTwo Project, and once you create the class file, then Copy and paste the following code into it.

**using** *AssemblyOne;*

**namespace** *AssemblyTwo*

**{**

**public** **class** AssemblyTwoClassI

**{**

**public** **void** Test**()**

**{**

AssemblyOneClassI instance = new AssemblyOneClassI**()**;

//Console.WriteLine(instance.ID);

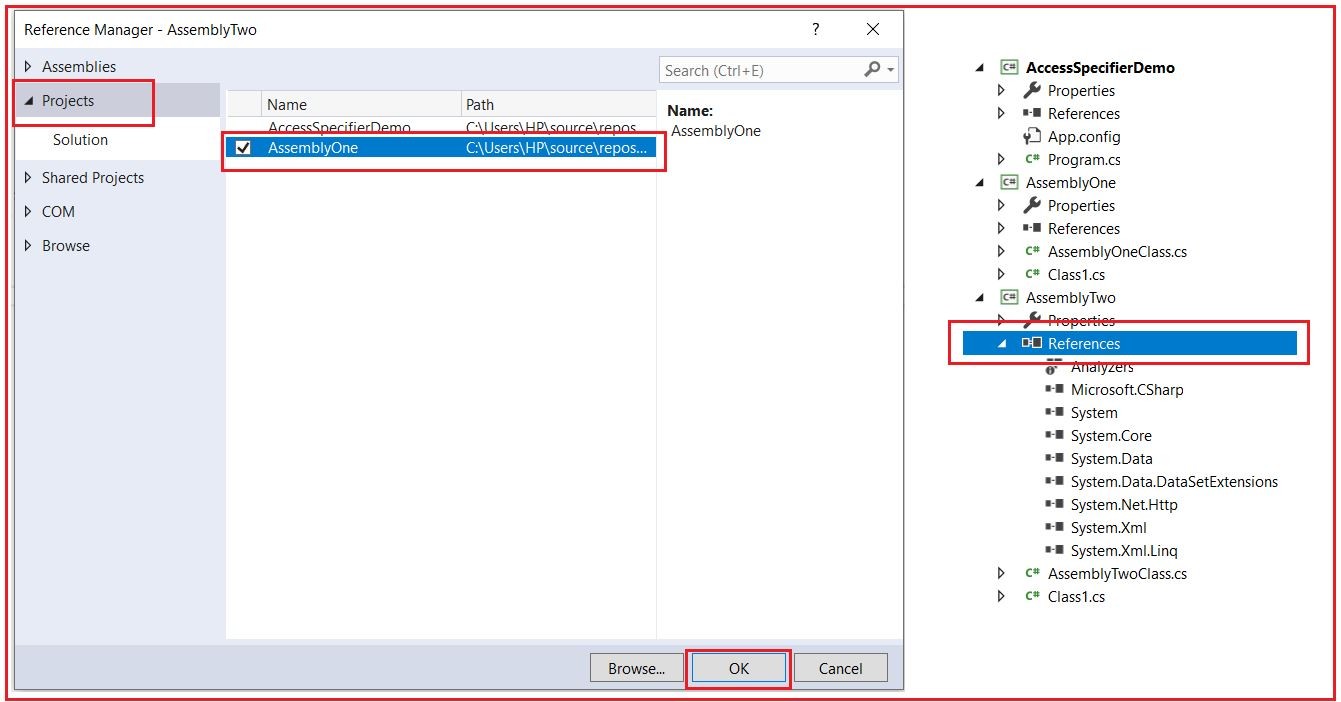
**}**

**}**

**}**

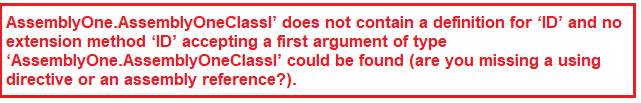
Notice, here, we got 2 compiler errors at this point. To solve this we need to add an assembly reference to AssemblyOne Project from AssemblyTwo Project. To do so, please follow the below steps.

1. Expand the References folder under the AssemblyTwo project, from Solution Explorer.
2. Right-click on the References folder and select Add Reference
3. From the Add Reference dialog box, select the Projects tab
4. From the list, select the AssemblyOne project and click on the OK button as shown in the below image.



At, this point all the compiler errors should have gone. Now, uncomment the following line from the AssemblyTwoClass.cs file from the AssemblyTwo project and rebuild the solution.  
**Console.WriteLine(instance.ID);**

Now, we will get a compiler error as shown in the below image.



This is because AssemblyTwoClassI is not present in AssemblyOne assembly and hence cannot access the internal ID member defined in AssemblyOne assembly. This proves that internal members are only accessible within the same assembly. Code outside of the containing assembly cannot access internal members.

**Protected Internal Access Specifier in C#:**

Protected Internal Members in C# can be accessed anywhere within the assembly in which it is declared or from within a derived class in another assembly. So, we can think, it is a combination of Protected and Internal. If you understood the Protected and Internal access specifiers in C#, then this should be very easy to follow. Now change the access modifier from internal to protected internal for ID member in AssemblyOneClassI of AssemblyOneClass.cs file in the AssemblyOne project.

**internal int ID = 999; to protected internal int ID = 999;**

And then, modify the code in the AssemblyTwoClass.cs file in the AssemblyTwo project as shown below. As you can see, now the AssemblyTwoClassI is derived from the AssemblyOneClassI.

**using** *AssemblyOne;*

**namespace** *AssemblyTwo*

**{**

// Make AssemblyTwoClassI inherit from AssemblyOneClassI

**public** **class** AssemblyTwoClassI : AssemblyOneClassI

**{**

**public** **void** Test**()**

**{**

AssemblyOneClassI instance = new AssemblyOneClassI**()**;

// Access the base class member using the base keyword

Console.WriteLine**(base**.ID**)**;

**}**

**}**

**}**

So this proofs that the protected internal ID member defined in AssemblyOne is accessible in AssemblyTwo. As of now, we have discussed how to use access specifiers with the type members. Let see how to use the access specifiers in C# with the type.

**Access Specifiers with Type in C#:**

We can use all 5 access specifiers with type members in C# but type allows only Internal and Public access specifiers. It is a compile-time error to use private, protected, and protected internal access specifiers with types. The following code will generate a compiler error stating Elements defined in a namespace cannot be explicitly declared as private, protected, or protected internal

**namespace** *AccessSpecifierDemo*

**{**

//Error: Cannot mark types with private, protected and protected internal access modifiers

**private** **class** MainClass

**{**

**public** **static** **void** Main**()**

**{**

Console.WriteLine**(**"This code will not compile"**)**;

**}**

**}**

**}**

We are also going to work with the same class library projects that we have created already. Copy and paste the following code in the AssemblyOneClass.cs file of the AssemblyOne project.

**namespace** *AssemblyOne*

**{**

//Class is marked internal. This class is available only within AssemblyOne

**internal** **class** AssemblyOneClass

**{**

**public** **void** Print**()**

**{**

Console.WriteLine**(**"Hello"**)**;

**}**

**}**

**}**

Now, copy and paste the following code in the AssemblyTwoClass.cs file of the AssemblyTwo project.

**using** *System;*

**using** *AssemblyOne;*

**namespace** *AssemblyTwo*

**{**

//Class is marked public. This class is available in any assembly

**public** **class** AssemblyTwoClass

**{**

**public** **void** Print**()**

**{**

AssemblyOneClass instance = new AssemblyOneClass**()**;

instance.Print**()**;

**}**

**}**

**}**

Now build the solution. You will notice the following 4 compiler errors.

1. **‘AssemblyOne.AssemblyOneClass’ is inaccessible due to its protection level**
2. **The type ‘AssemblyOne.AssemblyOneClass’ has no constructors defined**
3. **‘AssemblyOne.AssemblyOneClass’ is inaccessible due to its protection level**
4. **‘AssemblyOne.AssemblyOneClass’ does not contain a definition for ‘Print’ and no extension method ‘Print’ accepting a first argument of type ‘AssemblyOne.AssemblyOneClass’ could be found (are you missing a using directive or an assembly reference?)**

All these errors are in the AssemblyTwo project and are related to AssemblyOne.AssemblyOneClass being inaccessible due to its protection level. Now convert the access modifier of AssemblyOneClass from Internal to Public and rebuild the solution. Now we will not get any errors. This proofs that internal types are accessible only within the containing assembly.

Now just remove the public access modifier from AssemblyOneClass and rebuild the solution. Now again we get the same 4 errors that we got before. This is because if we don’t specify an access specifier for a type then by default the access modifier will be internal.

Note: If we don’t specify an access specifier in C# then for Types the default is internal and for type members it is private.

**Access Specifiers in C#**

**Access Specifiers in C# with Examples**

In this article, I am going to discuss the **Access Specifiers in C#** with Examples. Please read our previous article before proceeding to this article where we discussed the [**Destructor in C#**](https://dotnettutorials.net/lesson/destructor-csharp/) with an example. As part of this article, we are going to discuss the following pointers which are related to the C# access specifiers.

1. **What are Access Specifiers in C#?**
2. **Types of Access Specifiers supported by dot net.**
3. **Understanding Type and Type members in C#.**
4. **Understand Private, Public, Protected, Internal, and ProtectedInternal access specifiers with examples.**

**What are Access Specifiers in C#?**

The Access Specifiers in C# are also called access modifiers which are used to define the scope of the type (class and interface) as well as the scope of their members (variables, properties, and methods). That is who can access them and who cannot access them are defined by the Access Specifiers.

**Types of Access Specifiers in C#:**

C# supports 5 types of access specifiers. They are as follows

1. **Private**
2. **Public**
3. **Protected**
4. **Internal**
5. **Protected Internal**

**Note:** Members that are defined in a type with any scope or specifiers are always accessible within that type; restriction comes into the picture only when they try to access them outside of the type.

**Understand Type and Type members in C#:**

Before going to understand Access Specifier in C#, let us first understand what are Types and Type Members are. In the below example, **Customer** is the **Type** and variables (**\_id, \_firstName, \_lastName**), Properties (**Id, FirstName, LastName**) and method **GetFullName()** are type members.

**namespace** *AccessSpecifierDemo*

**{**

**public** **class** Customer

**{**

#region Private Fields

**private** **int** \_id;

**private** string \_firstName;

**private** string \_lastName;

#endregion

#region Properties

**public** **int** Id

**{**

**get** **{** **return** \_id; **}**

**set** **{** \_id = **value**; **}**

**}**

**public** string FirstName

**{**

**get** **{** **return** \_firstName; **}**

**set** **{** \_firstName = **value**; **}**

**}**

**public** string LastName

**{**

**get** **{** **return** \_lastName; **}**

**set** **{** \_lastName = **value**; **}**

**}**

#endregion

#region Methods

**public** string GetFullName**()**

**{**

**return** this.\_firstName + " " + this.\_lastName;

**}**

#endregion

**}**

**}**

So in general **classes, structs, enums, interfaces, delegates** are called **types,**and **variables, properties, constructors, methods,**etc. that normally reside within a type are called **type members.**The**Type members** can have all the 5 access modifiers whereas **types** can have only 2 (**internal, public**) access modifiers

**Note:** The **customer** class makes use of regions. Using **regions** we can expand and collapse sections of our code either manually, or using visual studio **Edit** -> **Outlining** -> **Toggle All Outlining**

Let’s discuss each access specifiers in C# with some examples. For this create a new console application with the name **AccessSpecifierDemo**. First, we will discuss Access Specifiers with the Type Members and then we will discuss Access Specifiers with the Type.

**Private and Public Access Specifiers in C#:**

Private members are available only within the containing type whereas public members are available anywhere. There is no restriction for public members. Let understand Private and Public Members with an example:

**namespace** *AccessSpecifierDemo*

**{**

**public** **class** Customer

**{**

**private** **int** \_id;

**public** **int** Id

**{**

**get**

**{**

**return** \_id;

**}**

**set**

**{**

\_id = **value**;

**}**

**}**

**}**

**public** **class** MainClass

**{**

**private** **static** **void** Main**()**

**{**

Customer CustomerInstance = new Customer**()**;

CustomerInstance.Id = 101;

// Compiler Error: 'Customer.\_id' is inaccessible due to its protection level

// CustomerInstance.\_id = 101;

**}**

**}**

**}**

In the above example, \_id variable is private. So, this member is only available within the Customer class (Containing Type). It is a compile-time error to access \_id outside of the Customer Class. The following line in the MainClass will generate a compiler error stating, **‘Customer.\_id’ is inaccessible due to its protection leve**l.  
**CustomerInstance.\_id = 101;**

On the other hand, the Id property is a public member. So, we can access this member anywhere even outside of the Customer class. In fact, we invoke the Id property of the Customer class in the Main() method of MainClass as shown below and it will not give us any error.  
**CustomerInstance.Id = 101;**

**Protected Access Specifier in C#:**

Protected Members in C# are available within the containing type as well as to the types that are derived from the containing type. Let us understand this with an example.

**namespace** *AccessSpecifierDemo*

**{**

**public** **class** Customer

**{**

**protected** **int** ID = 101;

**public** **void** PrintID**()**

**{**

//Protected member ID is accessible with in Customer class

Console.WriteLine**(**this.ID**)**;

**}**

**}**

**public** **class** CorporateCustomer : Customer

**{**

**public** **void** PrintCustomerID**()**

**{**

CorporateCustomer corporateCustomerInstance = new CorporateCustomer**()**;

// Can access the base class protected instance member using the derived class object

Console.WriteLine**(**corporateCustomerInstance.ID**)**;

// Can access the base class protected instance member using this or base keyword

Console.WriteLine**(**this.ID**)**;

Console.WriteLine**(base**.ID**)**;

**}**

**}**

**public** **class** RetailCustomer

**{**

**public** **void** PrintCustomerID**()**

**{**

RetailCustomer retailCustomerInstance = new RetailCustomer**()**;

//RetailCustomer class is not deriving from Customer class, hence it is an error

//to access Customer class protected ID member, using the retailCustomerInstance

//Console.WriteLine(retailCustomerInstance.ID); //Error

//Both these below lines also produce the same Error

//Console.WriteLine(this.ID); // Error

//Console.WriteLine(base.ID); // Error

**}**

**}**

**}**

The customer class defines a protected member ID. CorporateCustomer class derives from the Customer class so protected ID member is accessible in the Customer class (Containing Type) and also from the CorporateCustomer class (Derived Type). Within the PrintID() method in the Customer class Protected member ID is accessible.  
**Console.WriteLine(this.ID);**

There are 3 ways to access the base class-protected member in the derived class as shown below.

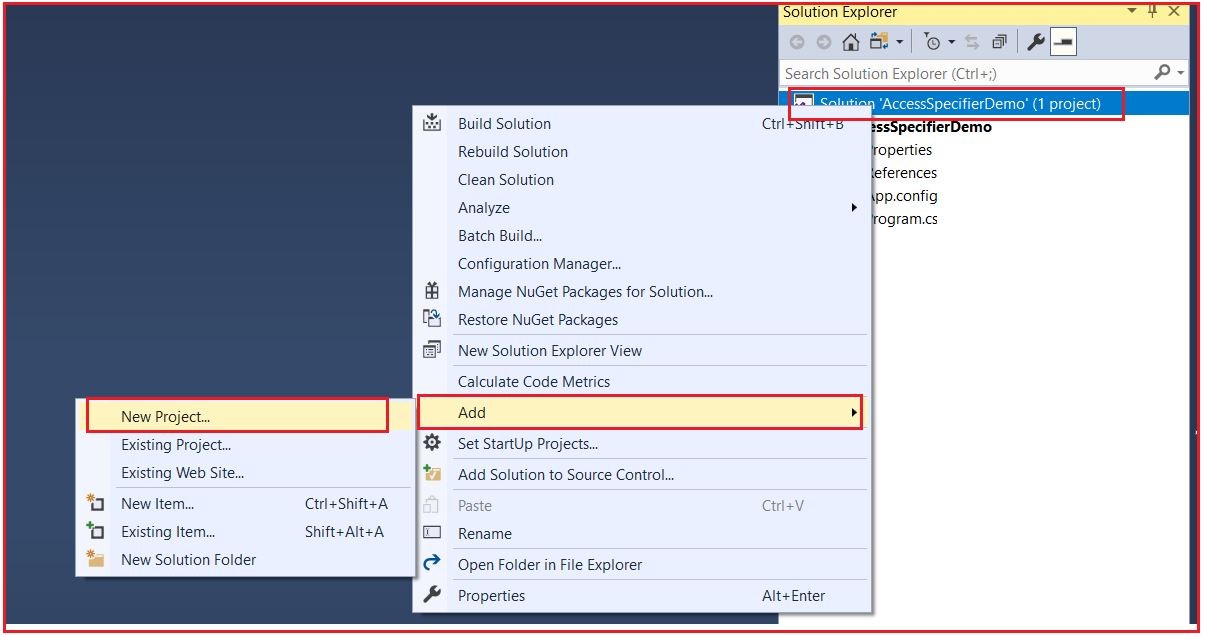
1. Using the derived class object: **Console.WriteLine(corporateCustomerInstance.ID);**
2. Using this keyword: **Console.WriteLine(this.ID);**
3. Using the base keyword: **Console.WriteLine(base.ID);**

On the other hand, the RetailCustomer class is not deriving from the Customer class hence it’s a compile-time error to access Customer class-protected ID member.

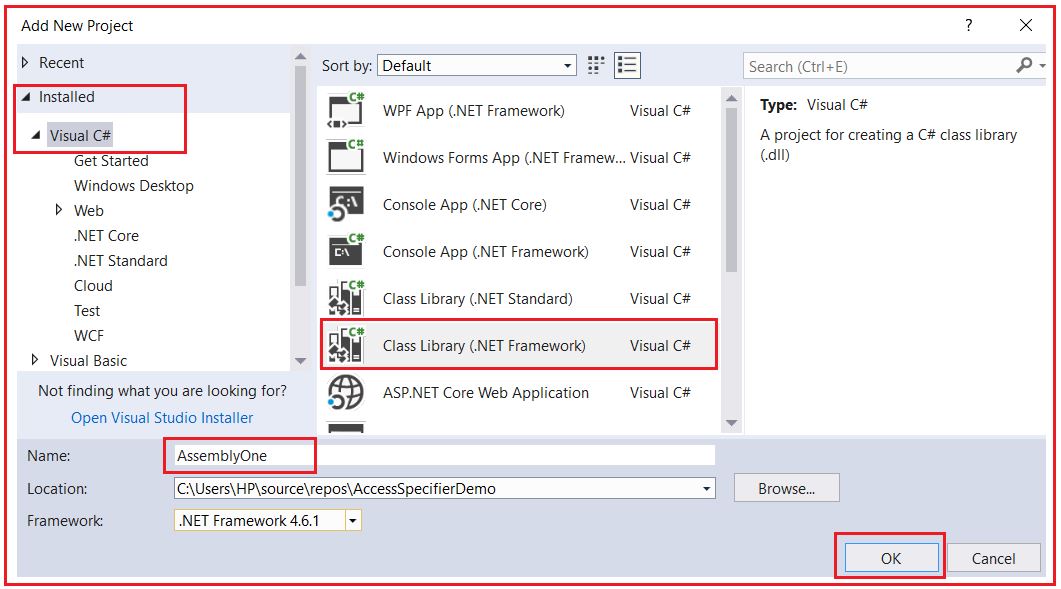
**Internal Access Specifier in C#:**

Whenever a member is declared with Internal Access Specifier in C#, then it is available anywhere within the containing assembly. It’s a compile-time error to access an internal member from outside the containing assembly. So, To understand the Internal Access Specifier in C#, we need 2 assemblies. To generate 2 assemblies, we need to follow the below steps.

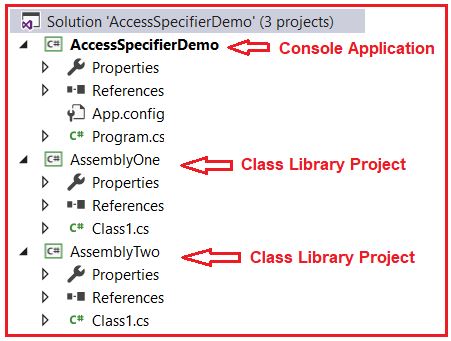
Right-click on the Solution Explorer and then select Add -> New Project option from the context menu as shown in the below image.



Once you click on the New Project, it will open the following Add New Project Dialog Box. Here, select **Visual C#** from the **Installed** section, and then select**Class Library** from the center pane, and provide the name as **AssemblyOne** and finally click on the OK button as shown in the below image.



Once you click on the OK button, it will add the Class Library Project with the name **AssemblyOne** to our solution. Again, repeat the process to create another class library project with the name AssemblyTwo to our solution. If you have followed the steps correctly, now we should have three projects in the solution explorer as shown in the below image.



Now if we build the solution, we should have 3 assemblies generated. Two DLLs and one exe. To locate the physical assembly follow these steps.

1. Right-click on the AssemblyOne project, in solution explorer and select Open Folder in Windows Explorer.
2. Open bin folder
3. Now open Debug folder
4. In the Debug folder, you should see AssemblyOne.dll, which is the physical assembly.

**Creating Class in AssemblyOne Project:**

Now, create a class file with the name **AssemblyOneClass.cs** within the AssemblyOne Project, and once you create the class file, then Copy and paste the following code into it.

**namespace** *AssemblyOne*

**{**

**public** **class** AssemblyOneClassI

**{**

**internal** **int** ID = 999;

**}**

**public** **class** AssemblyOneClassII

**{**

**public** **void** Test**()**

**{**

AssemblyOneClassI instance = new AssemblyOneClassI**()**;

// Can access inetrnal member ID as both AssemblyOneClassII and AssemblyOneClassI

// are present in the same assembly

Console.WriteLine**(**instance.ID**)**;

**}**

**}**

**}**

In this example, **AssemblyOneClassI**has an **internal member ID**. We can access this ID member from **AssemblyOneClassII** because this class is also present in the same assembly as **AssemblyOneClassI**. So, this proofs that the Internal Members access anywhere within the same assembly.

**Creating Class in AssemblyTwp Project:**

Now, create a class file with the name **AssemblyTwoClass.cs** within the AssemblyTwo Project, and once you create the class file, then Copy and paste the following code into it.

**using** *AssemblyOne;*

**namespace** *AssemblyTwo*

**{**

**public** **class** AssemblyTwoClassI

**{**

**public** **void** Test**()**

**{**

AssemblyOneClassI instance = new AssemblyOneClassI**()**;

//Console.WriteLine(instance.ID);

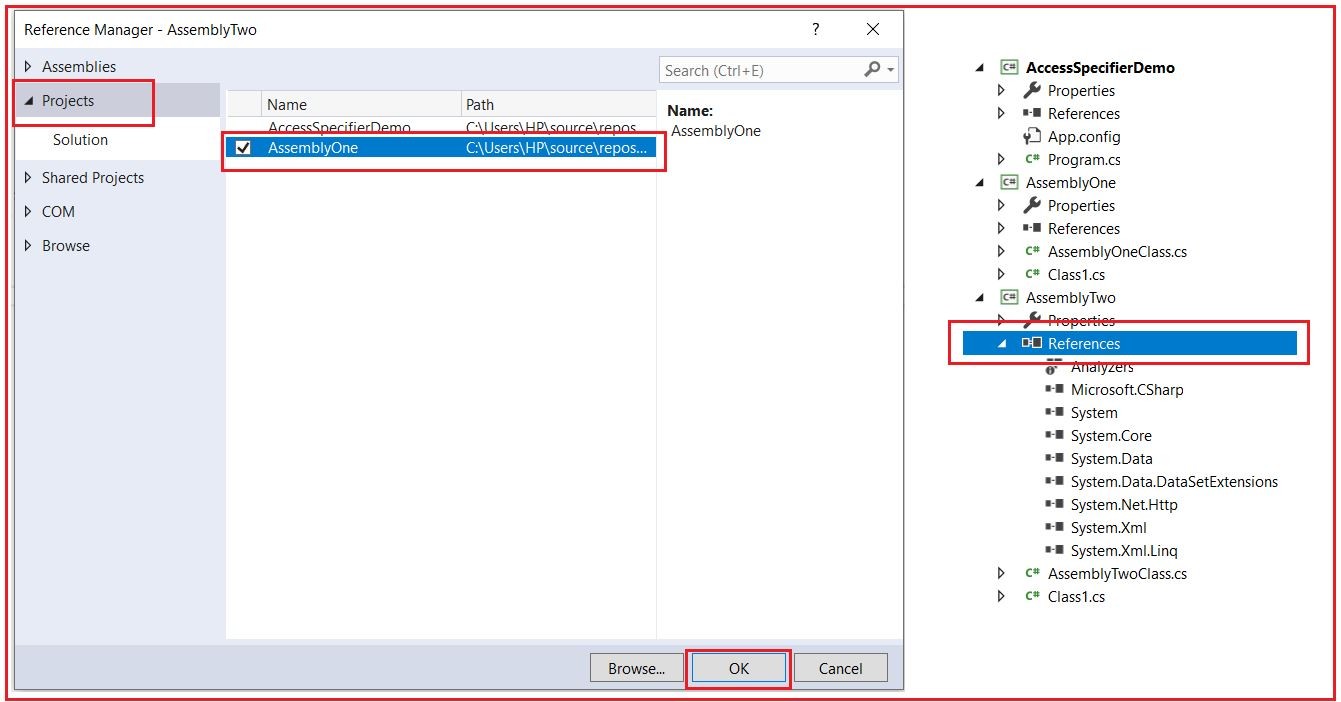
**}**

**}**

**}**

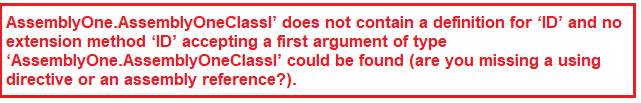
Notice, here, we got 2 compiler errors at this point. To solve this we need to add an assembly reference to AssemblyOne Project from AssemblyTwo Project. To do so, please follow the below steps.

1. Expand the References folder under the AssemblyTwo project, from Solution Explorer.
2. Right-click on the References folder and select Add Reference
3. From the Add Reference dialog box, select the Projects tab
4. From the list, select the AssemblyOne project and click on the OK button as shown in the below image.



At, this point all the compiler errors should have gone. Now, uncomment the following line from the AssemblyTwoClass.cs file from the AssemblyTwo project and rebuild the solution.  
**Console.WriteLine(instance.ID);**

Now, we will get a compiler error as shown in the below image.



This is because AssemblyTwoClassI is not present in AssemblyOne assembly and hence cannot access the internal ID member defined in AssemblyOne assembly. This proves that internal members are only accessible within the same assembly. Code outside of the containing assembly cannot access internal members.

**Protected Internal Access Specifier in C#:**

Protected Internal Members in C# can be accessed anywhere within the assembly in which it is declared or from within a derived class in another assembly. So, we can think, it is a combination of Protected and Internal. If you understood the Protected and Internal access specifiers in C#, then this should be very easy to follow. Now change the access modifier from internal to protected internal for ID member in AssemblyOneClassI of AssemblyOneClass.cs file in the AssemblyOne project.

**internal int ID = 999; to protected internal int ID = 999;**

And then, modify the code in the AssemblyTwoClass.cs file in the AssemblyTwo project as shown below. As you can see, now the AssemblyTwoClassI is derived from the AssemblyOneClassI.

**using** *AssemblyOne;*

**namespace** *AssemblyTwo*

**{**

// Make AssemblyTwoClassI inherit from AssemblyOneClassI

**public** **class** AssemblyTwoClassI : AssemblyOneClassI

**{**

**public** **void** Test**()**

**{**

AssemblyOneClassI instance = new AssemblyOneClassI**()**;

// Access the base class member using the base keyword

Console.WriteLine**(base**.ID**)**;

**}**

**}**

**}**

So this proofs that the protected internal ID member defined in AssemblyOne is accessible in AssemblyTwo. As of now, we have discussed how to use access specifiers with the type members. Let see how to use the access specifiers in C# with the type.

**Access Specifiers with Type in C#:**

We can use all 5 access specifiers with type members in C# but type allows only Internal and Public access specifiers. It is a compile-time error to use private, protected, and protected internal access specifiers with types. The following code will generate a compiler error stating Elements defined in a namespace cannot be explicitly declared as private, protected, or protected internal

**namespace** *AccessSpecifierDemo*

**{**

//Error: Cannot mark types with private, protected and protected internal access modifiers

**private** **class** MainClass

**{**

**public** **static** **void** Main**()**

**{**

Console.WriteLine**(**"This code will not compile"**)**;

**}**

**}**

**}**

We are also going to work with the same class library projects that we have created already. Copy and paste the following code in the AssemblyOneClass.cs file of the AssemblyOne project.

**namespace** *AssemblyOne*

**{**

//Class is marked internal. This class is available only within AssemblyOne

**internal** **class** AssemblyOneClass

**{**

**public** **void** Print**()**

**{**

Console.WriteLine**(**"Hello"**)**;

**}**

**}**

**}**

Now, copy and paste the following code in the AssemblyTwoClass.cs file of the AssemblyTwo project.

**using** *System;*

**using** *AssemblyOne;*

**namespace** *AssemblyTwo*

**{**

//Class is marked public. This class is available in any assembly

**public** **class** AssemblyTwoClass

**{**

**public** **void** Print**()**

**{**

AssemblyOneClass instance = new AssemblyOneClass**()**;

instance.Print**()**;

**}**

**}**

**}**

Now build the solution. You will notice the following 4 compiler errors.

1. **‘AssemblyOne.AssemblyOneClass’ is inaccessible due to its protection level**
2. **The type ‘AssemblyOne.AssemblyOneClass’ has no constructors defined**
3. **‘AssemblyOne.AssemblyOneClass’ is inaccessible due to its protection level**
4. **‘AssemblyOne.AssemblyOneClass’ does not contain a definition for ‘Print’ and no extension method ‘Print’ accepting a first argument of type ‘AssemblyOne.AssemblyOneClass’ could be found (are you missing a using directive or an assembly reference?)**

All these errors are in the AssemblyTwo project and are related to AssemblyOne.AssemblyOneClass being inaccessible due to its protection level. Now convert the access modifier of AssemblyOneClass from Internal to Public and rebuild the solution. Now we will not get any errors. This proofs that internal types are accessible only within the containing assembly.

Now just remove the public access modifier from AssemblyOneClass and rebuild the solution. Now again we get the same 4 errors that we got before. This is because if we don’t specify an access specifier for a type then by default the access modifier will be internal.

Note: If we don’t specify an access specifier in C# then for Types the default is internal and for type members it is private.

**Encapsulation in C#**

**Encapsulation in C# with Examples**

In this article, I am going to discuss the **Encapsulation in C#**with Examples. Please read our previous article, before proceeding to this article where we discussed the [**Access Specifies in C#**](https://dotnettutorials.net/lesson/access-specifiers-csharp/) with examples. Encapsulation is one of the fundamental principles of Object-Oriented Programming. As part of this article, you will understand the following pointers in detail.

1. **What is Encapsulation?**
2. **How can we implement encapsulation in C#?**
3. **Encapsulation Examples using Accessors and Mutators**
4. **What are the advantages of providing variable access via setter and getter methods?**
5. **What is the problem if we don’t follow encapsulation in C# while designing a class?**
6. **Implementing Encapsulation Using Properties**

**What is  Encapsulation in C#?**

The process of binding the data and functions together into a single unit (i.e. class) is called encapsulation in C#. Or you can say that the process of defining a class by hiding its internal data members from outside the class and accessing those internal data members only through publicly exposed methods (setter and getter methods) or properties with proper validations is called encapsulation.

**Note:** Data encapsulation is also called data hiding because by using this principle we can hide the internal data from outside the class.

**How can we implement Encapsulation in C#?**

In C# Encapsulation is implemented

1. By declaring the variables as private (to restrict its direct access from outside the class)
2. By defining one pair of public setter and getter methods or properties to access private variables.

We declare variables as private to stop accessing them directly from outside the class. The public setter and getter methods or properties are used to access the private variables from outside the class with proper validations. If we provide direct access to variables then we cannot validate the data before storing it in the variable.

So the point that you need to remember is by implementing encapsulation in c#, we are protecting or securing the data.

**Encapsulation in C# using Accessors and Mutators:**

Let us see an example to understand this concept. In the following example, we declare the balance variable as private in the Bank class, and hence it can not be accessed directly outside of the Bank class. In order to access this balance variable, we have exposed two public methods i.e. getBalance and setBalance. The getBalance method (Accessors) is used to fetch the value store in the balance variable whereas the setBalance method (Mutator) is used to set the value in the balance variable.

**namespace** *EncapsulationDemo*

**{**

**public** **class** Bank

**{**

//hiding class data by declaring the variable as private

**private** **double** balance;

//creating public setter and getter methods

**public** **double** getBalance**()**

**{**

//add validation logic if needed

**return** balance;

**}**

**public** **void** setBalance**(double** balance**)**

**{**

// add validation logic to check whether data is correct or not

this.balance = balance;

**}**

**}**

**class** BankUser

**{**

**public** **static** **void** Main**()**

**{**

Bank SBI = new Bank**()**;

SBI.setBalance**(**500**)**;

Console.WriteLine**(**SBI.getBalance**())**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**What are the advantages of providing variable access via setter and getter methods in C#?**

We can validate the user-given data before it is storing in the variable. In the above program for balance variable –ve value is not allowed. So we can validate the given amount value before storing it in the balance variable. If we provide direct access to the balance variable it is not possible to validate the given amount value.

**What is the problem if we don’t follow encapsulation in C# while designing a class?**

If we don’t the encapsulation principle while designing the class, then we cannot validate the user-given data according to our business requirement as well as it is very difficult to handle future changes.

Let us understand this with an example. Assume in the initial project requirement, the client did not mention that the application should not allow the negative number to store in that variable. So, we give direct access to the variable and the user can store any value to it as shown in the below program.

**namespace** *EncapsulationDemo*

**{**

**public** **class** Example

**{**

**public** **int** x;

**}**

**class** Sample

**{**

**public** **static** **void** Main**()**

**{**

Example e = new Example**()**;

e.x = 50;

Console.WriteLine**(**e.x**)**;

e.x = -10;

Console.WriteLine**(**e.x**)**;

Console.WriteLine**(**"Press any key to exist"**)**;

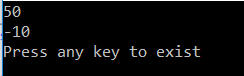
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



That’s it. It works as expected. Later, in the future, the customer wants that the application should not allow negative numbers. Then we should validate the user-given values before storing them into the variable.

**Hence the application architecture should be like below**

**namespace** *EncapsulationDemo*

**{**

**public** **class** Example

**{**

**private** **int** x;

**public** **int** getX**()**

**{**

**return** x;

**}**

**public** **void** setX**(int** x**)**

**{**

**if** **(**x **>** 0**)**

**{**

this.x = x;

**}**

**else**

**{**

Console.WriteLine**(**"Please Pass a positive value"**)**;

**}**

**}**

**}**

**class** Sample

**{**

**public** **static** **void** Main**()**

**{**

Example e = new Example**()**;

//We cannot use the variable directly here

// e.x = 50; //Compile time errr

// Console.WriteLine(e.x); //Compile time errr

e.setX**(**10**)**;

Console.WriteLine**(**e.getX**())**;

e.setX**(**-50**)**;

Console.WriteLine**(**e.getX**())**;

Console.WriteLine**(**"Press any key to exist"**)**;

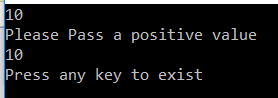
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Implementing Encapsulation in C# using Properties:**

The Properties are a new language feature introduced in C#. Properties in C# help in protecting a field of a class by reading and writing the values to it. The first approach i.e. setter and getter itself is good but Encapsulation in C# can be accomplished much smoother with properties.

Let us understand how to implement Encapsulation in C# using properties with an example. In the below example, inside the Bank class, we marked the balance variable as private to restrict direct access from outside the Bank class. In order to access the balance variable, we have exposed the Balance property by declaring it as public. Now from outside the Bank class, we can access the balance private variable through the public exposed Balance property.

**namespace** *EncapsulationDemo*

**{**

**public** **class** Bank

**{**

**private** **double** balance;

**public** **double** Balance

**{**

**get**

**{**

**return** balance;

**}**

**set**

**{**

// validate the value

**if** **(value** **<** 0**)**

**{**

Console.WriteLine**(**"value cannot be negative"**)**;

**}**

**else**

**{**

balance = **value**;

**}**

**}**

**}**

**}**

**class** BankUser

**{**

**public** **static** **void** Main**()**

**{**

Bank SBI = new Bank**()**;

SBI.Balance = 100;

Console.WriteLine**(**SBI.Balance**)**;

SBI.Balance = -50;

Console.WriteLine**(**SBI.Balance**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

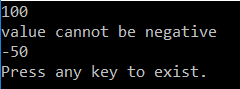
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Abstraction in C#**

**Abstraction in C# with Real-Time Examples**

In this article, I am going to discuss **Abstraction in C#** with Real-Time Examples. Please read our previous article before proceeding to this article where we discussed [**Encapsulation in C#**](https://dotnettutorials.net/lesson/encapsulation-csharp/) with examples. The **Abstraction in C#** is one of the fundamental OOPs principles which acts as a supporting principle. That means the Abstraction Principle in C# makes sure that all other three principles (Encapsulation, Polymorphism, and Inheritance) are working together to give the final shape of the project.

**What is Abstraction in C#?**

The process of representing the essential features without including the background details is called Abstraction. In simple words, we can say that it is a process of defining a class by providing necessary details to call the object operations (i.e. methods) by hiding or removing its implementation details is called abstraction in C#. It means we need to expose what is necessary and compulsory and we need to hide the unnecessary things from the outside world. In C# we can hide the member of a class by using private access modifiers.

**Let us understand Abstraction in C# with a Real-time Example.**

Let us understand this with a car example. As we know a car is made of many things, such as the name of the car, the color of the car, gear, breaks, steering, silencer, the battery of the car, engine of the car, etc. Now you want to ride a car. So to ride a car what are the things you should know. The things a car driver should know are as follows.

1. Name of the Car
2. The color of the Car
3. Gear
4. Break
5. Steering

So these are the things that should be exposed and know by the car driver before riding the car. The things which should be hidden to a Car rider as are follows

1. The engine of the car
2. Diesel Engine
3. Silencer

So these are the things which should be hidden from a car driver.

**Now let’s implement what we discussed with a program using C#**

**namespace** *AbstractionDemo*

**{**

**public** **class** Car

**{**

**private** string \_CarName = "Honda City";

**private** string \_CarColur = "Black";

**public** string CarName

**{**

**set**

**{**

\_CarName = **value**;

**}**

**get**

**{**

**return** \_CarName;

**}**

**}**

**public** string CarColur

**{**

**set**

**{**

\_CarColur = **value**;

**}**

**get**

**{**

**return** \_CarColur;

**}**

**}**

**public** **void** Steering**()**

**{**

Console.WriteLine**(**"Streering of the Car"**)**;

**}**

**public** **void** Brakes**()**

**{**

Console.WriteLine**(**"Brakes of the Car"**)**;

**}**

**public** **void** Gear**()**

**{**

Console.WriteLine**(**"Gear of the Car"**)**;

**}**

**private** **void** CarEngine**()**

**{**

Console.WriteLine**(**"Engine of the Car"**)**;

**}**

**private** **void** DiesalEngine**()**

**{**

Console.WriteLine**(**"DiesalEngine of the Car"**)**;

**}**

**private** **void** Silencer**()**

**{**

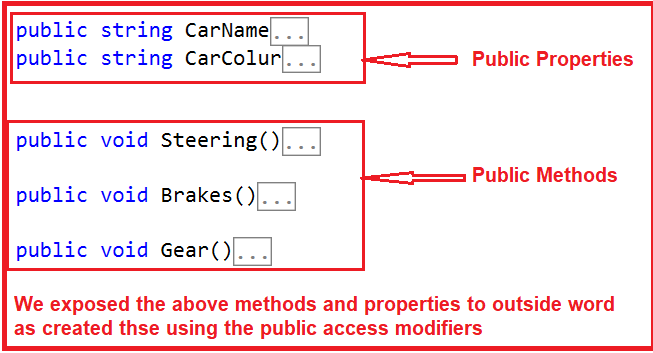
Console.WriteLine**(**"Silencer of the Car"**)**;

**}**

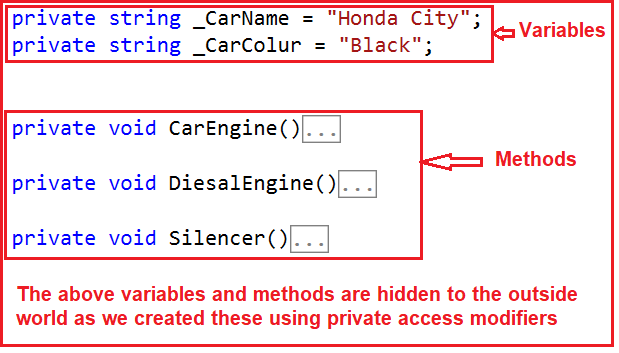
**}**

**}**

As shown in the above example, you can see that the necessary methods and properties are exposed by using the “public” access modifier whereas the unnecessary methods and properties hidden from outside the world by using the “private” access modifier as shown in the below image.



As you can see in the above image, the methods and properties which we want to expose to the outside world are created using the public access specifier. Now from outside the class, we can create the object of this Car class and can access the above methods and properties that we will see after a while. Have a look at the following image.



As shown in the above image, the methods and variables which don’t want to expose to the outside world are created using the private access modifier. Now from outside the class, we can create the instance of the Car class but we cannot access the above methods and variables.

**Consuming the Car Class:**

Let us create an instance of the Car class within the Main method of the Program class and then let’s try to invoke the public and private members of the Car class.

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

//Creating an instance of Car

Car CarObject = new Car**()**;

//Accessing the Public Properties and methods

string CarName = CarObject.CarName;

string CarColur = CarObject.CarColur;

CarObject.Brakes**()**;

CarObject.Gear**()**;

CarObject.Steering**()**;

//Try to access the private variables and methods

//Compiler Error, 'Car.\_CarName' is inaccessible due to its protection level

CarObject.\_CarName;

//Compiler Error, 'Car.CarEngine' is inaccessible due to its protection level

CarObject.CarEngine**()**;

**}**

**}**

As you can see, we can access the public members of the Car class using the Car instance. But while we are accessing the private members of the Car class using the same Car instance, we get Compiler Error. Hence, this proofs that, we have exposed the necessary methods and properties to outside the world or class by hiding the unnecessary members of the class by using the Abstraction in C#.

**What is the difference between Abstraction and Encapsulation in C#?**

Encapsulation is the process of hiding irrelevant data from the user or you can say Encapsulation is used to protect the data. For example, whenever we buy a mobile, we can never see how the internal circuit board works. We are also not interested to know how the digital signal converts into the analog signal and vice versa. So from a Mobile user’s point of view, these are some irrelevant pieces of information, This is the reason why they are encapsulated inside a cabinet.

In C# programming, we will do the same thing. We will create a cabinet and keep all the irrelevant information that should not be exposed to the user of the class.

Coming to abstraction in C#, It is just the opposite of Encapsulation. What it means, it is a mechanism that will show only the relevant information to the user. If we consider the same mobile example. Whenever we buy a mobile phone, we can see and use many different types of functionalities such as a camera, calling function, recording function, mp3 player, multimedia, etc. This is nothing but an example of abstraction in C#. The reason is we are only seeing the relevant information instead of their internal work.

**Inheritance in C#**

**Inheritance in C# with Examples**

In this article, I am going to discuss **Inheritance in C#** with Examples. Inheritance is one of the OOPs principles and this principle addresses the extensibility problem. Please read our [**Class and Object in C#**](https://dotnettutorials.net/lesson/class-and-objects-csharp/) article before proceeding to this article. As part of this article, we are going to discuss the following pointers.

1. **What is inheritance?**
2. **Types of inheritance in C#**
3. **Why do we need inheritance in C#?**
4. **How to make use of inheritance in application development?**

**What is inheritance in C#?**

The process of creating a new class from an existing class such that the new class acquires all the properties and behaviors of the existing class is called inheritance. The properties (or behaviors) are transferred from which class is called the superclass or parent class or base class whereas the class which derives the properties or behaviors from the superclass is known as a subclass or child class or derived class. In simple words, inheritance means to take something that is already made (or available).

Inheritance is the concept that is used for code reusability and changeability purpose. Here changeability means overriding the existed functionality or feature of the object or adding more functionality to the object.

**Classification of inheritance supported by C#.NET**

C#.NET classified the inheritance into two categories, such as

1. **Implementation inheritance.**
2. **Interface inheritance**

**Implementation inheritance:**This is the commonly used inheritance. Whenever a class is derived from another class then it is known as implementation inheritance.

**Interface inheritance:**This type of inheritance is taken from Java. Whenever a class is derived from an interface then it is known as interface inheritance.

**Types of inheritance in C#**

Inheritance is classified into 5 types. They are as follows.

1. **Single Inheritance:** When a class is derived from a single base class then the inheritance is called single inheritance.
2. **Multilevel Inheritance:** When a derived class is created from another derived class, then that type of inheritance is called multilevel inheritance.
3. **Hierarchical Inheritance:** When more than one derived class is created from a single base class then it is called Hierarchical inheritance.
4. **Hybrid Inheritance:** Hybrid Inheritance is the inheritance that is the combination of any single, hierarchical, and multilevel inheritances.
5. **Multiple Inheritance:** When a derived class is created from more than one base class then such type of inheritance is called multiple inheritances. But multiple inheritances are not supported by .net using classes and can be done using interfaces.

**Note:** Handling the complexity that causes due to multiple inheritances is very complex. Hence it was not supported in dot net with class and it can be done with interfaces.

**Rules to be considered while working with inheritance in C#**

**Rule1:** In inheritance, the constructor of the parent class must be accessible to its child class otherwise the inheritance will not possible because when we create the child class object first it goes and calls the parent class constructor so that the parent class variable will be initialized and we can consume them under the child class.

**NOTE:** The reason why a child class internally calls its parent class constructor is to initialize parent class and can consume them under child class.

**Rule2**: In inheritance, the child classes can consume the parent class members but the parent class does not consume child class members that are purely defined in the child class.

**Rule3:**Just like the object of a class can be assigned to a variable of the same class to make it as a reference, it can also be assigned a variable of its parent to make it as a reference so that the reference starts consuming memory of the object assigned to it, but now also using that we control access child class pure members.

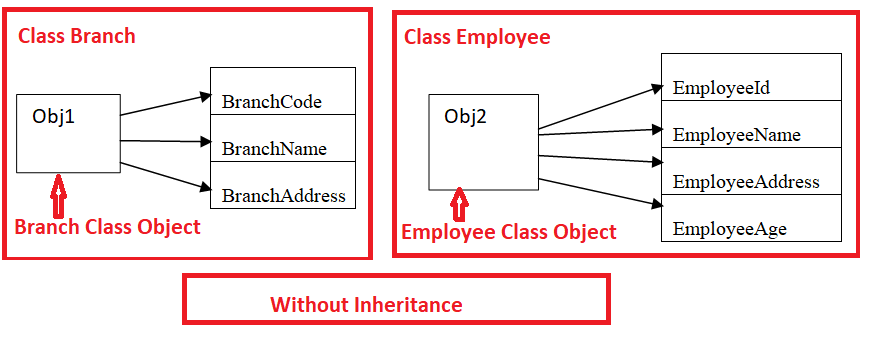
**NOTE:** A parent class object can never be assigned to a child class variable. A parent class reference i.e. created by using a child class object can be converted back into a child class reference if required by performing an explicit conversion.

**Why do we need inheritance in C#?**

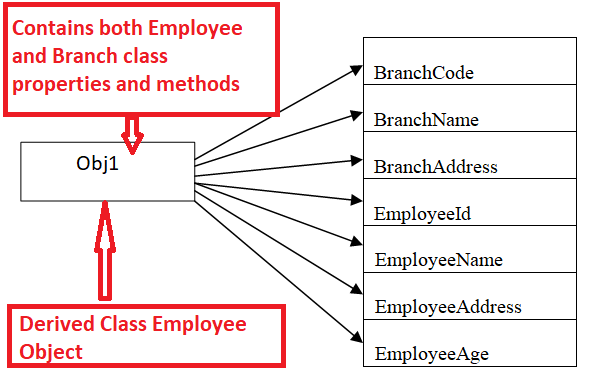
Let us understand why do we need inheritance in C# with an example. Assume that a company has “n” no of branches and asked to computerize branches details of the company, then we create a class like **Class Branch** having the data member (Data fields or variables) BranchCode, BranchName, and BranchAddress and also methods (functions) like GetBranchData() and DisplayBranchData().

Later after some period of time, the company also asked to computerize employee details of each branch. Then we create a class like **Class Employee** having the data member EmployeeId, EmployeeName, EmployeeAddress, EmployeeAge, and also methods like GetEmployeeData() and DisplayEmployeeData().

If we create two classes without inheritance we need to create the object for every class individually like below



Here Obj1 is the object of Class Branch class and Obj2 is the object of Class Employee. So it becomes very difficult to identify which employee belongs to which branch and integrate the Branch class object with the Employee class object. So if we derive the Employee class from the Branch class we create the object to the derived class Employee then it will represent both classes and will maintain the reference to the members of both base and derived classes.



Example:

Let us implement the example we discussed using inheritance. The complete code is given below.

**namespace** *InheritanceDemo*

**{**

**class** Branch

**{**

**int** BranchCode;

string BranchName, BranchAddress;

**public** **void** GetBranchData**()**

**{**

Console.WriteLine**(**"ENTER BRANCH DETAILS:"**)**;

Console.WriteLine**(**"ENTER BRANCH CODE"**)**;

BranchCode = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"ENTER BRANCH NAME"**)**;

BranchName = Console.ReadLine**()**;

Console.WriteLine**(**"ENTER BRANCH ADDRESS"**)**;

BranchAddress = Console.ReadLine**()**;

**}**

**public** **void** DisplayBranchData**()**

**{**

Console.WriteLine**(**"BRANCH CODE IS : " + BranchCode**)**;

Console.WriteLine**(**"BRANCH NAME IS : " + BranchName**)**;

Console.WriteLine**(**"BRANCH ADDRESS IS : " + BranchAddress**)**;

**}**

**}**

**class** Employee : Branch

**{**

**int** EmployeeId, EmployeeAge;

string EmployeeName, EmployeeAddress;

**public** **void** GetEmployeeData**()**

**{**

Console.WriteLine**(**"ENTER EMPLYEE DETAILS:"**)**;

Console.WriteLine**(**"ENTER EMPLOYEE ID"**)**;

EmployeeId = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"ENTER EMPLOYEE AGE"**)**;

EmployeeAge = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"ENTER EMPLOYEE NAME"**)**;

EmployeeName = Console.ReadLine**()**;

Console.WriteLine**(**"ENTER EMPLOYEE ADDRESS"**)**;

EmployeeAddress = Console.ReadLine**()**;

**}**

**public** **void** DisplayEmployeeData**()**

**{**

Console.WriteLine**(**"EMPLOYEE ID IS : " + EmployeeId**)**;

Console.WriteLine**(**"EMPLOYEE NAME IS : " + EmployeeName**)**;

Console.WriteLine**(**"EMPLOYEE ADDRESS IS : " + EmployeeAddress**)**;

Console.WriteLine**(**"EMPLOYEE AGE IS : " + EmployeeAge**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Employee obj1 = new Employee**()**;

obj1.GetBranchData**()**;

obj1.GetEmployeeData**()**;

obj1.DisplayBranchData**()**;

obj1.DisplayEmployeeData**()**;

Console.WriteLine**(**"Press any key to exist."**)**;

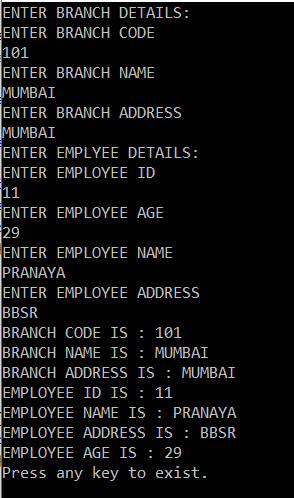
Console.ReadKey**()**;

**}**

**}**

**}**

**OUTPUT:**



In the above example, we made the functions GetEmployeeData() and DisplayEmployeeData() of class Branch as public because to access from outside the class i.e. from Class Employee. And the data field BranchCode, BranchName, and BranchAddress are private by default so they are accessible within the same class only.

But if you don’t want to give accessibility of the base class members to the non-derived class (in this case class Program) and would like to give derived class (Employee) then we need to use protected to the members.

**And our code would be like as…**

**namespace** *InheritanceDemo*

**{**

**class** Branch

**{**

**int** BranchCode;

string BranchName, BranchAddress;

**protected** **void** GetBranchData**()**

**{**

Console.WriteLine**(**"ENTER BRANCH DETAILS:"**)**;

Console.WriteLine**(**"ENTER BRANCH CODE"**)**;

BranchCode = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"ENTER BRANCH NAME"**)**;

BranchName = Console.ReadLine**()**;

Console.WriteLine**(**"ENTER BRANCH ADDRESS"**)**;

BranchAddress = Console.ReadLine**()**;

**}**

**protected** **void** DisplayBranchData**()**

**{**

Console.WriteLine**(**"BRANCH CODE IS : " + BranchCode**)**;

Console.WriteLine**(**"BRANCH NAME IS : " + BranchName**)**;

Console.WriteLine**(**"BRANCH ADDRESS IS : " + BranchAddress**)**;

**}**

**}**

**class** Employee : Branch

**{**

**int** EmployeeId, EmployeeAge;

string EmployeeName, EmployeeAddress;

**public** **void** GetEmployeeData**()**

**{**

//to call the base class method use base keyword

**base**.GetBranchData**()**;

Console.WriteLine**(**"ENTER EMPLYEE DETAILS:"**)**;

Console.WriteLine**(**"ENTER EMPLOYEE ID"**)**;

EmployeeId = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"ENTER EMPLOYEE AGE"**)**;

EmployeeAge = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"ENTER EMPLOYEE NAME"**)**;

EmployeeName = Console.ReadLine**()**;

Console.WriteLine**(**"ENTER EMPLOYEE ADDRESS"**)**;

EmployeeAddress = Console.ReadLine**()**;

**}**

**public** **void** DisplayEmployeeData**()**

**{**

**base**.DisplayBranchData**()**;

Console.WriteLine**(**"EMPLOYEE ID IS : " + EmployeeId**)**;

Console.WriteLine**(**"EMPLOYEE NAME IS : " + EmployeeName**)**;

Console.WriteLine**(**"EMPLOYEE ADDRESS IS : " + EmployeeAddress**)**;

Console.WriteLine**(**"EMPLOYEE AGE IS : " + EmployeeAge**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Employee obj1 = new Employee**()**;

//Here we cannot access the Branch class method as they are now protected

// obj1.GetBranchData(); //Will give Compile time error

obj1.GetEmployeeData**()**;

// obj1.DisplayBranchData(); // will give compile time error

obj1.DisplayEmployeeData**()**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

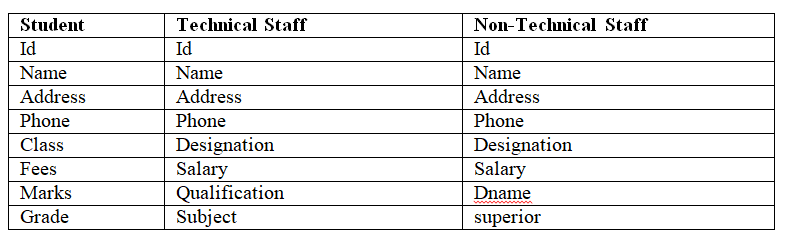
**How to make use of inheritance in application development?**

Generally, when we develop an application we will be following a process as follows.

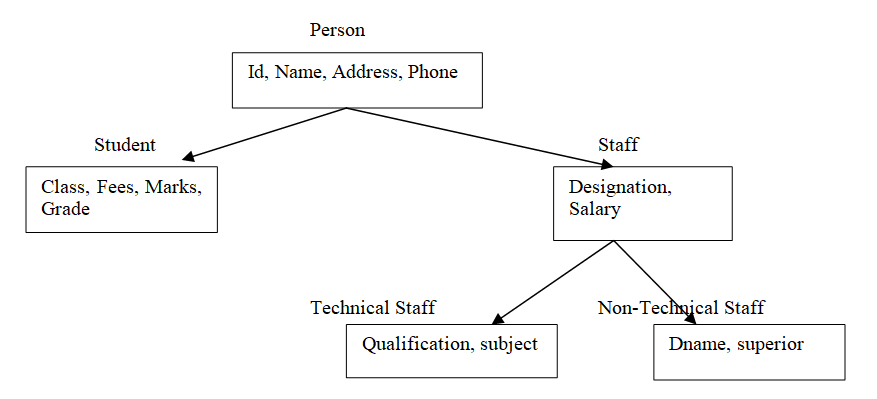
1. Identify the entity associated with the application
2. Identify the attribute that is associated with the application.
3. Now separate the attribute of each entity in a hierarchical order without having any duplicate.
4. Convert those entities into classes.

**Example:**

Let us understand inheritance in C# with one real-time example. Suppose we are developing an application for school the attributes of the entity will be as following



Now separate the attribute of that entity based on the hierarchy as follows.



**Now define the class representing the entity as following**

**namespace** *InheritanceDemo*

**{**

**public** **class** Person

**{**

**int** Id;

string Name;

string Address;

string Phone;

**}**

**public** **class** Student : Person

**{**

string Class;

string Fees;

string Marks;

string Grade;

**}**

**public** **class** Staff : Person

**{**

string Designation;

**double** Salary;

**}**

**public** **class** Technical : Staff

**{**

string Qualification;

string Subject;

**}**

**public** **class** NonTechnical : Staff

**{**

string Dname;

string Superior;

**}**

**}**

**Interface in C#**

**Interface in C# with Examples**

In this article, I am going to discuss one of the most important concepts i.e. **Interface in C#** with Examples. Please read our previous article where we discussed [**Inheritance in C#.**](https://dotnettutorials.net/lesson/inheritance-c-sharp/) At the end of this article, I am sure, you will understand what is Interface, why we need an interface, and how to use interface in C# with real-time examples.

**Why do we need an interface in C#?**

We know the concept of multiple inheritances where one class is derived from more than one superclasses. For example a definition likes

**class A: B, C**  
**{**  
**}**

But this concept is not supported by .NET with classes. Since a large no of real-time applications require the use of multiple inheritances, where we inherit properties and behaviors from several different classes. That’s why .NET provides an alternative approach known as the interface to support the concept of multiple inheritances.

**What is an interface in C#?**

The Interface in C# is a **fully un-implemented class** used for declaring a set of operations of an object. So, we can define an interface as a pure abstract class which allows us to define only abstract methods. The abstract method means a method without body or implementation.

**What is the need for an interface when we have the abstract class to define abstract methods?**

.NET doesn’t support multiple inheritances with classes. So we must use interface as the superclass to develop abstraction for supporting multiple inheritances. If we define an abstract class in place of an interface, a service provider cannot implement multiple specifications so that the service provider cannot have multiple businesses.

**What are the different types of Inheritance?**

A class can be inherited either from another class or from an interface also. So inheritance can be divided into two categories

1. **Implementation inheritance**
2. **Interface inheritance**

If a class is inheriting from another class we call it implementation inheritance and the main concept of implementation inheritance is child classes can consume the members of its parent class.

On the other hand, if a class is inheriting from an interface we call it as Interface Inheritance but interface inheritance does not provide any reusability because here we are not consuming the members of the parent under the child. The child is only implementing the parent’s members.

**How to declare an interface in C#?**

By using the keyword **interface** we can declare an interface.

// SYNTAX:

**public** **interface** InterfaceName

**{**

//only abstract methods

**}**

// For example

**public** **interface** Example

**{**

**void** show**()**;

**}**

Here the keyword interface tells that Example is an interface containing one abstract method i.e. show(). By default, the members of an interface are public and abstract. An interface can contain

1. **Abstract methods**
2. **Properties**
3. **Indexes**
4. **Events**

**An interface cannot contain**

1. **Non-abstract functions**
2. **Data fields**
3. **Constructors**
4. **Destructors**

**Should I use public access modifiers for interface methods?**

.NET interface methods are implicitly public by default, even if they belong to nested interfaces. Non-public modifiers are not valid or necessary for interface methods. So the compiler will fail and warn you in this case. Nested interfaces may be declared protected or private but not the interface methods.

**Can an interface implement an abstract class?**

No. In .NET an interface cannot implement an abstract class. An interface may only extend a super interface. However, an abstract class can implement an interface because an abstract class can contain both abstract methods and concrete methods.

**Can an interface be declared as sealed?**

No, it is not permitted to declare an interface as sealed; it will cause a compilation error. This is a .NET language design decision. Interface types are intended to be implemented and can be extended without restriction.

**Is more than one interface are allowed to implement a class?**

Yes, a class can implement multiple interfaces; this is an effective way to achieve multiple inheritances in .NET. But a class can extend only one superclass.

**Is it necessary to implement all interface methods?**

It is not necessary for a class that implements an interface to implement all its methods, but in this case, the class must be declared as abstract.

**How interface is different from a class in C#?**

An interface is different from a class in the following ways:

1. We cannot instantiate an interface.
2. An interface does not contain any constructor or data fields or destructor, etc.
3. All of the methods of an interface are abstract and public by default.
4. An interface is not extended by a class; it is implemented by a class.
5. An interface can extend multiple interfaces.

**What are the similarities between the interface and abstract class in C#?**

An interface is similar to an abstract class in the following ways

1. Both interface and the abstract class cannot be instantiated means we cannot create the object.
2. But we can create a reference variable for both interface and abstract class.
3. The subclass should implement all abstract methods.
4. Both cannot be declared as sealed.

**What is the main difference between interface and abstract class in C#?**

The main difference is to be answered in the interview is as follows. The interface is a fully un-implemented class used for declaring a set of operations of an object. The abstract class is a partially implemented class. It implements some of the operations of an object. These implemented operations are common for all next-level subclasses. The remaining operations are implemented by the next level subclasses according to their requirement.

The interface allows us to develop multiple inheritances. So we must start object design with interface whereas abstract class does not support multiple inheritances so it always comes next to interface in the object creation process.

**What are the differences between the interface and abstract class in C#?**

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| It is a partially implemented class. It allows us to define both concrete and abstract methods. | It is a fully un-implemented class. It allows us to define only abstract methods. |
| It provides both reusability and forcibility | It provides the only forcibility |
| It should be declared as abstract by using the abstract keyword, abstract methods should also contain the abstract keyword. | It should be created by using the keyword interface. Declaring its methods as abstract is optional because by default the methods of an interface are abstract. The compiler places abstract keywords at the time of program compilation. |
| A class that contains one or more abstract functions is called abstract class. | The class which contains all the abstract functions is known as an interface. |
| Its member’s default accessibility modifier is private and can be changed to any of the other accessibility modifiers. | Its member’s default accessibility modifier is public and cannot be changed. |
| It is possible to declare data fields in an abstract class. | But it is not possible to declare any data fields in an interface. |
| An abstract class can contain the non-abstract function. | An interface cannot contain non-abstract functions. |
| An abstract class can inherit from another abstract class or from an interface. | An interface can inherit from only other interfaces but cannot inherits from the abstract class. |
| It can have inner classes | It can also have inner classes. |
| An abstract class cannot be used to implement multiple inheritances. | An interface can be used to implement multiple inheritances. |
| Abstract class members can have access modifiers. | Interface members cannot have access modifiers. |

**Rules to follow while working with the interface**

While working with an interface, we must follow the below rules.

1. The interface cannot have concrete methods, violation leads to CE: interface methods cannot have a body.
2. We cannot declare interface members as private or protected members violation leads to CE:” modifier is not allowed here”.
3. An interface cannot be instantiated but its reference variable can be created for storing its subclass object reference.
4. We cannot declare the interface as sealed it leads to CE: “illegal combination of modifier interface and final”.
5. The class derived from the interface should implement all abstract methods of the interface otherwise it should be declared as abstract else it leads to a compile-time error.
6. The subclass should implement the interface method with public keyword because interface methods default accessibility modifier is public.
7. In an interface, we cannot create fields variable violation leads to a compile-time error.

We have discussed a lot of theories. Let’s move towards practical examples.

**Let’s see a simple example for better understanding.**

**namespace** *InterfaceDemo*

**{**

**public** **interface** A

**{**

**void** method1**()**;

**void** method2**()**;

**}**

**interface** B : A

**{**

**void** method3**()**;

**}**

**class** MyClass : B

**{**

**public** **void** method1**()**

**{**

Console.WriteLine**(**"implement method1"**)**;

**}**

**public** **void** method2**()**

**{**

Console.WriteLine**(**"implement method2"**)**;

**}**

**public** **void** method3**()**

**{**

Console.WriteLine**(**"implement method3"**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

MyClass obj = new MyClass**()**;

obj.method1**()**;

obj.method2**()**;

obj.method3**()**;

Console.WriteLine**(**"Press any key to exist."**)**;

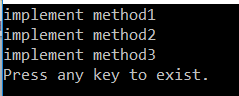
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Example2: WAP TO CALCULATE AREA OF RECTANGLE, CIRCLE USING INTERFACE.**

**namespace** *InterfaceDemo*

**{**

**public** **interface** Area

**{**

**void** area**(double** a, **double** b**)**;

**}**

**class** Rectangle : Area

**{**

**public** **void** area**(double** a, **double** b**)**

**{**

**double** areaRectangle;

areaRectangle = a \* b;

Console.WriteLine**(**"the area of rectangle is :" + areaRectangle**)**;

**}**

**}**

**class** Circle : Area

**{**

**static** **double** PI = 3.14;

**public** **void** area**(double** a, **double** b**)**

**{**

**double** areaCircle;

areaCircle = PI \* a \* a;

Console.WriteLine**(**"the area of Circle is :" + areaCircle**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Area a = new Rectangle**()**;

a.area**(**5, 6**)**;

a = new Circle**()**;

a.area**(**7, 0**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

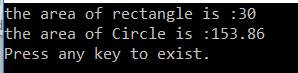
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



In the next article, I am going to discuss [**multiple inheritances using interfaces in C#**](https://dotnettutorials.net/lesson/multiple-inheritance-csharp/). Here, in this article, I try to explain **Interface in C#** with examples. I hope this article will help you with your need. I would like to have your feedback. Please post your feedback, question, or comments about this Interface in C# with Examples article.

**Multiple Inheritance in C#**

**Multiple Inheritance in C# with Examples**

In this article, I am going to discuss **Multiple Inheritance in C#** with Examples. Please read our previous article where we discussed [**Interface in C#**](https://dotnettutorials.net/lesson/interface-c-sharp/). As we discussed earlier multiple inheritances are not supported through classes but supported through interfaces. It is not supported through classes because we will get the ambiguity problem.

**What is the ambiguity problem in C#?**

Let us understand this with an example. Please have a look at the following sample code.

Class 1 **{**Test**(){}}**

Class 2 **{**Test**(){}}**

Class 3 : 1, 2**{}**

In the above case, class 3 is inheriting from class 1 and class 2 and both these two classes contain a method with the same name and same signature so while consuming the method ambiguity arises to understand which class method has to be executed. But, if a class is inheriting from interfaces we don’t have any ambiguity problem because the class is not consuming the members of its parent interfaces but only implementing the parent’s members.

**interface** 1**{**Test**()**;**}**

**interface** 2**{**Test**()**;**}**

**class** A : 1, 2**{}**

In the above case, the class is not consuming the interface members. It is only implementing the interface members so if we face any ambiguity with interface members that can be resolved in child class in two different ways.

If the method is implemented for a single time under the class both the interface will assume that the implemented method belongs to them and executes and there will not be any ambiguity.

The method can be implemented separately for each interface also under the class by providing the method name with the interface name. But in this case, while calling the method we should compulsorily use the interface reference that is created using the object of a class. We will discuss this with an example.

**Example: Let us see an example to understand multiple inheritances in C#.**

**namespace** *InterfaceDemo*

**{**

**public** **interface** Interface1

**{**

**void** Test**()**;

**void** Show**()**;

**}**

**public** **interface** Interface2

**{**

**void** Test**()**;

**void** Show**()**;

**}**

**class** ImplementInterafce : Interface1, Interface2

**{**

**public** **void** Test**()**

**{**

Console.WriteLine**(**"Test method is implemented"**)**;

**}**

**public** **void** Show**()**

**{**

Console.WriteLine**(**"Show mwthod is implemented"**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

ImplementInterafce obj = new ImplementInterafce**()**;

obj.Test**()**;

obj.Show**()**;

Interface1 obj1 = new ImplementInterafce**()**;

obj1.Test**()**;

obj1.Show**()**;

Interface2 obj2 = new ImplementInterafce**()**;

obj2.Test**()**;

obj2.Show**()**;

Console.WriteLine**(**"Press any key to exist."**)**;

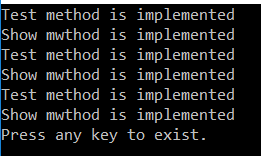
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**What is Explicit Interface Implementation in C#?**

When each interface method is implemented separately under the child class by providing the method name along with the interface name then it is called Explicit Interface Implementation. But in this case, while calling the method we should compulsorily use the interface reference that is created using the object of a class. Let see an example to understand explicit interface implementation in C#. While implementing the Interface methods explicitly, it is not allowed us to use the public access specifiers.

**namespace** *InterfaceDemo*

**{**

**public** **interface** Interface1

**{**

**void** Test**()**;

**void** Show**()**;

**}**

**public** **interface** Interface2

**{**

**void** Test**()**;

**void** Show**()**;

**}**

**class** ImplementInterafce : Interface1, Interface2

**{**

//public modifier is not allowed

**void** Interface1.Test**()**

**{**

Console.WriteLine**(**"Test mthod of interafce1 is implemented"**)**;

**}**

**void** Interface1.Show**()**

**{**

Console.WriteLine**(**"Show mwthod of interafce1 is implemented"**)**;

**}**

**void** Interface2.Test**()**

**{**

Console.WriteLine**(**"Test mthod of interface2 is implemented"**)**;

**}**

**void** Interface2.Show**()**

**{**

Console.WriteLine**(**"Show mwthod of interafce2 is implemented"**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

ImplementInterafce obj = new ImplementInterafce**()**;

//obj.Test(); //not possible

//obj.Show(); //not possible

**((**Interface1**)**obj**)**.Test**()**;

**((**Interface1**)**obj**)**.Show**()**;

Interface1 obj1 = new ImplementInterafce**()**;

obj1.Test**()**;

obj1.Show**()**;

Interface2 obj2 = new ImplementInterafce**()**;

obj2.Test**()**;

obj2.Show**()**;

Console.WriteLine**(**"Press any key to exist."**)**;

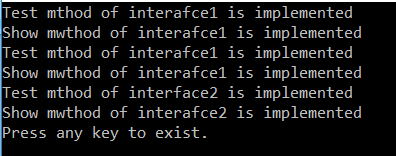
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Some Important points to remember:**

**Point1:**

We create an interface using the interface keyword. Just like classes, the interface also contains properties, methods, delegates, or events, but only declarations and no implementation. For example,  
interface ICustomer  
{  
      void Print();  
}

**Point2:**

It is a compile-time error to provide the implementation for any interface members. For example.  
interface ICustomer  
{  
    void Print(){}   //Compile time error  
}

**Point3:**

Interface members are public by default and they don’t allow explicit access modifiers. For example:  
interface ICustomer  
{  
     public void Print(); //Compile time error  
}

**Point4:**

The interface cannot contain fields. For example:  
interface ICustomer  
{  
     int ID; //Compile time error  
     void Print();  
}

**Point5:**

When a class or a struct inherits from an interface then it’s the responsibility of the class or struct to provide the implementation for all interface members otherwise we get a compile-time error. For example:  
interface ICustomer  
{  
       void Print();  
}  
class Customer : ICustomer //Compile time error  
{  
}

**When do you choose interface over an abstract class or vice versa?**

If we want some implementation that will be the same for all the derived classes, then it is better to go for an abstract class instead of an interface. With the interface, we can move our implementation to any class that implements the interface. With the abstract class, we can share the implementation for all the derived classes in one central place, and thus avoid code duplication in the derived classes.

**Can an interface inherit from another interface?**

Yes, an interface can inherit from another interface. It is possible for a class to inherit an interface multiple times, through base classes or interfaces it inherits. In this case, the class can only implement the interface one time, if it is declared as part of the new class. If the inherited interface is not declared as part of the new class, its implementation is provided by the base class that declared it. It is possible for a base class to implement interface members using virtual members; in that case, the class inheriting the interface can change the interface behavior by overriding the virtual members.

**Can you create an instance of an interface?**

No, you cannot create an instance of an interface.

**If a class inherits an interface, what are the 2 options available for that class?**

**Option 1:** Provide Implementation for all the members inherited from the interface.

**namespace** *Interfaces*

**{**

**interface** Interface1

**{**

**void** Interface1Method**()**;

**}**

**class** BaseClass1 : Interface1

**{**

**public** **void** Interface1Method**()**

**{**

Console.WriteLine**(**"Interface1 Method"**)**;

**}**

**public** **void** BaseClass1Method**()**

**{**

Console.WriteLine**(**"BaseClass1 Method"**)**;

**}**

**}**

**}**

**Option 2:** If the class does not wish to provide Implementation for all the members inherited from the interface, then the class has to be marked as abstract.

**namespace** *Interfaces*

**{**

**interface** Interface1

**{**

**void** Interface1Method**()**;

**}**

**abstract** **class** BaseClass1 : Interface1

**{**

**abstract** **public** **void** Interface1Method**()**;

**public** **void** BaseClass1Method**()**

**{**

Console.WriteLine**(**"BaseClass1 Method"**)**;

**}**

**}**

**}**

**A class inherits from 2 interfaces and both the interfaces have the same method name as shown below. How should the class implement the drive method for both Car and Bus interfaces?**

**namespace** *Interfaces*

**{**

**interface** Car

**{**

**void** Drive**()**;

**}**

**interface** Bus

**{**

**void** Drive**()**;

**}**

**class** Demo : Car, Bus

**{**

//How to implement the Drive() Method inherited from Bus and Car

**}**

**}**

By using explicitly Interface Implementation. To implement the Drive() method use the fully qualified name as shown in the example below. To call the respective interface drive method typecast the demo object to the respective interface and then call the drive method.

**namespace** *Interfaces*

**{**

**interface** Car

**{**

**void** Drive**()**;

**}**

**interface** Bus

**{**

**void** Drive**()**;

**}**

**class** Demo : Car, Bus

**{**

**void** Car.Drive**()**

**{**

Console.WriteLine**(**"Drive Car"**)**;

**}**

**void** Bus.Drive**()**

**{**

Console.WriteLine**(**"Drive Bus"**)**;

**}**

**static** **void** Main**()**

**{**

Demo DemoObject = new Demo**()**;

**((**Car**)**DemoObject**)**.Drive**()**;

**((**Bus**)**DemoObject**)**.Drive**()**;

**}**

**}**

**}**

**Abstract Class and Abstract Methods in C#**

**Abstract Class and Abstract Methods in C#**

In this article, I am going to discuss **Abstract class and Abstract methods in C#** with Examples. Please read our previous article, where we discussed [**Inheritance in C#**](https://dotnettutorials.net/lesson/multiple-inheritance-csharp/). At the end of this article, you will understand what are abstract class and abstract methods, why do we need abstract class and abstract methods, and how to implement this in C#.

**What is an Abstract Class in C#?**

A class that is declared by using the keyword **abstract** is called an abstract class. An abstract class is a partially implemented class used for implementing some of the operations of an object which are common for all next-level subclasses and the remaining abstract methods to be implemented by the next-level subclasses. So it contains both abstract methods and concrete methods including variables, properties, and indexers.

It is always created as a superclass next to the interface in the object inheritance hierarchy for implementing common operations from an interface. An abstract class may or may not have abstract methods. But if a class contains an abstract method then it must be declared as abstract.

An abstract class cannot be instantiated directly. It’s compulsory to create/derive a subclass from the abstract class in order to provide the functionality to its abstract functions.

**What is the abstract method in C#?**

A method that does not have a body is called an abstract method. It is declared with the modifier abstract. It contains only a **declaration/signature** and does not contain the implementation/body/definition of the method. An abstract function should be terminated with a semicolon. Overriding an abstract function is compulsory.

**Why should the method have an abstract keyword if it does not have a body in C#?**

In a class, we are allowed only to define a class with the body. Since we are changing its default behavior (which means removing its body) it must have the abstract keyword in its prototype.

**Points to Remember while working with an abstract class in C#**

1. An abstract class can contain both abstract methods and non-abstract (concrete) methods.
2. It can contain both static and instance variables.
3. The abstract class cannot be instantiated but its reference can be created.
4. If any class contains abstract methods then it must be declared by using the keyword abstract.
5. An abstract class can contain sealed methods but an abstract method or class cannot be declared as sealed.
6. A subclass of an abstract class can only be instantiated if it implements all of the abstract methods of its superclass. Such classes are called concrete classes to differentiate them from abstract classes.

**When should a class be declared as abstract?**

A class should be declared as abstract

1. If the class has any abstract methods
2. If it does not provide implementation to any of the abstract methods it inherited
3. If it does not provide implementation to any of the methods of an interface

**When to use the abstract method in C#?**

Abstract methods are usually declared where two or more subclasses are expected to fulfill a similar role in a different manner. Often the subclasses are required to fulfill an interface, so the abstract superclass might provide several of the interface methods, but leave the subclasses to implement their own variations of the abstract methods.

**Child class of abstract class:**

1. Implements all the abstract methods of the parent
2. Then only we can consume the non-abstract methods of the parent.

**Rules Of Abstract Method and Abstract Class in C#:**

**Rule1:** If a method does not have the body, then it should be declared as abstract using the abstract modifier else it leads to a compile-time error: “**must declare a body because it is not marked abstract, extern, or partial**”

**public** **class** Example

**{**

// compile time error: must declare a body because it is not marked abstract, extern, or partial

**void** m1**()**;

**}**

**Rule2:**If a class has an abstract method it should be declared as abstract by using the keyword abstract else it leads to a compile-time error: ‘**Example.m1()’ is abstract but it is contained in non-abstract class ‘Example**‘.

**public** **class** Example

**{**

//'Example.m1()' is abstract but it is contained in non-abstract class 'Example'

**public** **abstract** **void** m1**()**;

**}**

**The correct syntax is**

**public** **abstract** **class** Example

**{**

**public** **abstract** **void** m1**()**;

**}**

**Rule3:** If a class is declared as abstract it cannot be instantiated violation leads to compile-time Error.

**public** **abstract** **class** Example

**{**

**public** **abstract** **void** m1**()**;

**public** **static** **void** Main**(**String**[]** args**)**

**{**

//Cannot create an instance of the abstract class or interface 'Example'

Example e = new Example**()**;

**}**

**}**

**Error: Cannot create an instance of the abstract class or interface ‘Example’**

**Rule4:** The sub-classes of an abstract class should override all the abstract methods or it should be declared as abstract else it leads to the compile-time error:

**namespace** *AbstractDemo*

**{**

**public** **abstract** **class** Example

**{**

**public** **abstract** **void** m1**()**;

**public** **abstract** **void** m2**()**;

**}**

//'Sample' does not implement inherited abstract member 'Example.m2()'

**public** **class** Sample : Example

**{**

**public** **override** **void** m1**()**

**{**

Console.WriteLine**(**"m1 method"**)**;

**}**

**}**

**}**

**Solutions:**

**Declare the class as abstract as shown below**

**public** **abstract** **class** Sample : Example

**{**

**public** **override** **void** m1**()**

**{**

Console.WriteLine**(**"m1 method"**)**;

**}**

**}**

**Override both abstract methods as shown below**

**public** **class** Sample : Example

**{**

**public** **override** **void** m1**()**

**{**

Console.WriteLine**(**"m1 method"**)**;

**}**

**public** **override** **void** m2**()**

**{**

Console.WriteLine**(**"m2 method"**)**;

**}**

**}**

**Why abstract class can not be instantiated?**

Because it is not a fully implemented class as its abstract methods cannot be executed. If the compiler allows us to create the object for an abstract class we can invoke the abstract method using that object which cannot be executed by CLR at runtime. Hence to restrict calling abstract methods, the compiler does not allow us to instantiate an abstract class.

**Who will provide the implementation (body) for abstract methods?**

Sub-class developers provide the body for abstract methods according to their business requirements. Basically, in projects, abstract methods (method prototype) are defined by the superclass developer and they are implemented by sub-class developers.

**What type of member can we define in an abstract class?**

We can define all static and non-static members including properties, fields, indexes, and also abstract methods.

**Will abstract class members be created when a subclass object is created?**

Yes**,** its non-static members get memory when its concrete sub-class object is created.

**How can we execute static and non-static concrete members of the abstract class?**

Static members can be executed directly from its main method and its non-static members are executed by using its concrete sub-class object.

**Can we declare an abstract method as static?**

No, we are not allowed to declare an abstract method as static. It leads to **Compile Time Error: illegal combination of modifier abstract and static**.

If the compiler allows us to declare it as static, it can be invoked directly which cannot be executed by CLR at runtime. Hence to restrict in calling abstract methods compiler does not allow us to declare an abstract method as static. For example

**namespace** *AbstractDemo*

**{**

**public** **abstract** **class** Example

**{**

//A static member 'Example.m1()' cannot be marked as override, virtual, or abstract

**public** **static** **abstract** **void** m1**()**;

**public** **abstract** **void** m2**()**;

**}**

**}**

**Can we declare an abstract method as sealed?**

No, because it should be allowed to override in subclasses. It leads to Compile Time Error:**cannot be sealed because it is not an override**. For example:

**namespace** *AbstractDemo*

**{**

**public** **abstract** **class** Example

**{**

// 'Example.m1()' cannot be sealed because it is not an override

**public** **sealed** **abstract** **void** m1**()**;

**public** **abstract** **void** m2**()**;

**}**

**}**

**Can we declare an abstract method as private?**

No, because it should be inherited to subclasses. It leads to Compile Time Error: **virtual or abstract members cannot be private**. For example:

**namespace** *AbstractDemo*

**{**

**public** **abstract** **class** Example

**{**

//'Example.m1()': virtual or abstract members cannot be private

**private** **abstract** **void** m1**()**;

**public** **abstract** **void** m2**()**;

**}**

**}**

**Can we declare a concrete class as abstract?**

Yes, it is allowed. Defining a class as abstract is a way of preventing someone from instantiating a class that is supposed to be extended first. To ensure our class non-static members are only accessible via sub-class objects we should declare the concrete class as abstract. Let see an example for a better understanding

**namespace** *AbstractDemo*

**{**

Public **abstract** **class** Example

**{**

**public** **static** **void** m1**()**

**{**

Console.WriteLine**(**"Example m1 method"**)**;

**}**

**public** **void** m2**()**

**{**

Console.WriteLine**(**"Example m2 method"**)**;

**}**

**}**

**}**

**Calling concrete abstract members from normal class:**

**namespace** *AbstractDemo*

**{**

**public** **class** Sample

**{**

**static** **void** Main**(**String**[]** args**)**

**{**

Example.m1**()**;

//CE: cannot be instantiated class Example

Example e = new Example**()**;

//not possible

e.m2**()**;

**}**

**}**

**}**

**Calling concrete abstract class members from sub-classes**

**namespace** *AbstractDemo*

**{**

**public** **class** Sample : Example

**{**

**static** **void** Main**(**String**[]** args**)**

**{**

Example.m1**()**;

Sample e = new Sample**()**;

e.m2**()**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Differences between overriding methods and abstract methods in C#?**

1. The concept of the abstract method is near similar to the concept of method overriding because in method overriding if a Parent class contains any **virtual** methods in it, then those methods can be re-implemented under the child class by using the **override** modifier.
2. In a similar way, if a parent class contains any **abstract** methods in it, those abstract methods must be implemented under the child class by using the same **override** modifier.
3. The main difference between method overriding and abstract method is in the case of method overriding the child class re-implementing the method is optional but in the case of the abstract method, the child class implementing the method is mandatory.

**What is the need for abstract classes in application development?**

The concepts of abstract methods and abstract classes are an extension to the inheritance wherein inheritance we have been discussing that with the help of a parent class we can provide property to the child class that can be consumed by the child classes which gives us re-usability.

Along with the parent providing property to the children, the parent can also impose the restriction on the children with the help of abstract methods so that all the child classes have to full fill the restriction without failing.

**Example:**

In the below example, the abstract class MyClass has one abstract method which has got various implementations in sub-classes.

**namespace** *AbstractDemo*

**{**

**public** **abstract** **class** MyClass

**{**

**public** **abstract** **void** calculate**(double** x**)**;

**}**

**class** Sub1 : MyClass

**{**

**public** **override** **void** calculate**(double** x**)**

**{**

Console.WriteLine**(**"SQUARE ROOT IS " + Math.Sqrt**(**x**))**;

**}**

**}**

**public** **class** Sub2 : MyClass

**{**

**public** **override** **void** calculate**(double** x**)**

**{**

Console.WriteLine**(**"SQUARE is :" + **(**x \* x**))**;

**}**

**}**

**public** **class** Sub3 : MyClass

**{**

**public** **override** **void** calculate**(double** x**)**

**{**

Console.WriteLine**(**"CUBE is :" + **(**x \* x \* x**))**;

**}**

**}**

**class** Test

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Sub1 obj1 = new Sub1**()**;

Sub2 obj2 = new Sub2**()**;

Sub3 obj3 = new Sub3**()**;

obj1.calculate**(**9**)**;

obj2.calculate**(**9**)**;

obj3.calculate**(**9**)**;

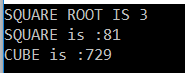
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Example:**

In the below example, we calculate the electricity bills for commercial and domestic plans using abstract class and abstract methods.

**namespace** *AbstractDemo*

**{**

**public** **abstract** **class** Plan

**{**

**protected** **double** rate;

**public** **abstract** **void** getRate**()**;

**public** **void** calculation**(int** units**)**

**{**

Console.Write**(**"BILL AMOUNT FOR " + units + " UNITS is: Rs."**)**;

Console.WriteLine**(**rate \* units**)**;

**}**

**}**

**class** CommercialPlan : Plan

**{**

**public** **override** **void** getRate**()**

**{**

rate = 5.00;

**}**

**}**

**class** DomesticlPlan : Plan

**{**

**public** **override** **void** getRate**()**

**{**

rate = 2.50;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Plan p;

Console.WriteLine**(**"COMMERCIAL CONNECTION"**)**;

p = new CommercialPlan**()**;

p.getRate**()**;

p.calculation**(**250**)**;

Console.WriteLine**(**"DOMESTIC CONNECTION"**)**;

p = new DomesticlPlan**()**;

p.getRate**()**;

p.calculation**(**150**)**;

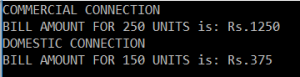
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Example to show the use of abstract methods in C#.**

**namespace** *AbstractDemo*

**{**

**abstract** **class** ClsEmployee

**{**

**protected** **int** EmpId, EAge;

**protected** string EName, EAddress;

**public** **abstract** **void** GetEmpData**()**;

**public** **virtual** **void** DisplayEmpData**()**

**{**

Console.WriteLine**(**"Employee Id Is: " + this.EmpId**)**;

Console.WriteLine**(**"Employee Name Is: " + this.EName**)**;

Console.WriteLine**(**"Employee Address Is: " + EAddress**)**;

Console.WriteLine**(**"Employee Age is: " + this.EAge**)**;

**}**

**public** ClsEmployee**()**

**{**

Console.WriteLine**(**"ABSTRACT CLASS CONSTRUCTOR"**)**;

**}**

**}**

**class** ClsManager : ClsEmployee

**{**

**double** Bonus, CA;

**public** **override** **void** GetEmpData**()**

**{**

Console.WriteLine**(**"ENTER MANAGER DETAILS"**)**;

Console.WriteLine**(**"Enter the Manager Id: "**)**;

EmpId = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"Enter the Manager name: "**)**;

EName = Console.ReadLine**()**;

Console.WriteLine**(**"Enter the manager Bonus: "**)**;

Bonus = **double**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"Enter the Manager CA: "**)**;

CA = **double**.Parse**(**Console.ReadLine**())**;

**}**

**public** **override** **void** DisplayEmpData**()**

**{**

Console.WriteLine**(**"manager id is: " + EmpId**)**;

Console.WriteLine**(**"manager name is: " + EName**)**;

Console.WriteLine**(**"manager bonus is: " + Bonus**)**;

Console.WriteLine**(**"Manager CA is: " + CA**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

ClsManager cm = new ClsManager**()**;

cm.GetEmpData**()**;

cm.DisplayEmpData**()**;

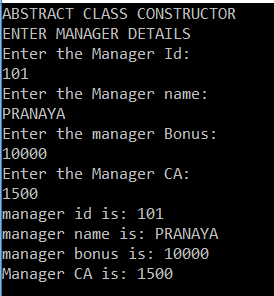
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Summary of Abstract Class and Abstract Methods in C#:**

1. A method that does not have a body is called an abstract method and the class that is declared by using the keyword abstract is called an abstract class. If a class contains an abstract method, then it must be declared as abstract.
2. An abstract class can contain both abstract and non-abstract methods. If a child class of an abstract class wants to consume any non-abstract methods of its parent, should implement all abstract methods of its parent.
3. An abstract class is never usable to itself because we cannot create the object of an abstract class. The members of an abstract class can be consumed only by the child class of the abstract class.

**Polymorphism in C#**

**Polymorphism in C# with Real-Time Examples**

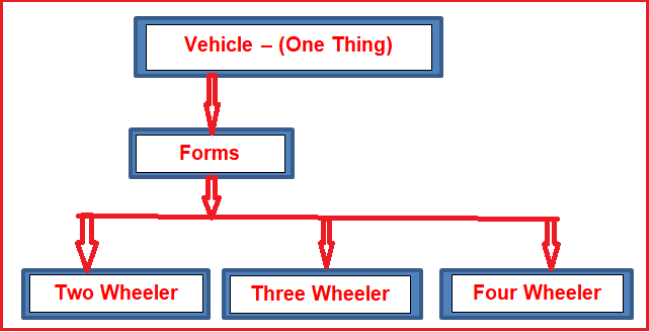
In this article, I am going to discuss **Polymorphism in C#**with Real-Time Examples. Please read our previous where we discussed [**Abstract Class and Abstract Methods in C#**](https://dotnettutorials.net/lesson/abstract-class-abstract-methods-csharp/) with Examples. At the end of this article, you will understand the following polymorphism pointers in detail.

1. **What is Polymorphism?**
2. **Why do we need Polymorphism?**
3. **Types of Polymorphism in C#?**
4. **What is Compile-Time Polymorphism?**
5. **What is Runtime Polymorphism?**

**Note:** Polymorphism is one of the primary pillars of object-oriented programming.

**What is Polymorphism in C#?**

The word polymorphism is derived from the Greek word, where Poly means many and morph means faces/ behaviors. So, the word polymorphism means the ability to take more than one form. Please have a look at the following diagram. As shown in the below diagram, the vehicle is something that has various forms; two-wheeler, three-wheeler, and four-wheeler, and so on. So this is one example of polymorphism.



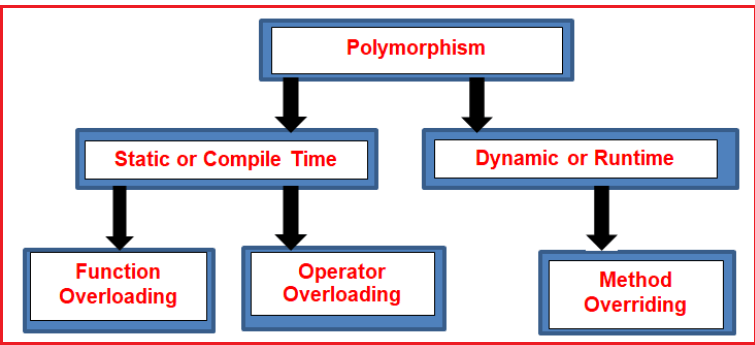
Technically we can say that when a function shows different behaviors when we passed different types and numbers of input values, then it is called Polymorphism in C#. So behaving in different ways depending on the input received is known as polymorphism i.e. whenever the input changes, automatically the output or the behavior also changes.

**Types of Polymorphism in C#**

There are two types of polymorphism in C#

1. **Static Polymorphism / Compile-Time Polymorphism / Early Binding**
2. **Dynamic Polymorphism / Run-Time Polymorphism / Late Binding**

The following diagram shows different types of polymorphisms in C# with their examples.



The polymorphism in C# can be implemented using the following three ways.

1. [**Function Overloading**](https://dotnettutorials.net/lesson/function-overloading-csharp/)
2. [**Function Overriding**](https://dotnettutorials.net/lesson/function-overriding-csharp/)
3. [**Function Hiding**](https://dotnettutorials.net/lesson/function-hiding-csharp/)

**Note:** While working with Polymorphism in C# we need to understand two things i.e. what happens at the time of compilation and what happens at the time of execution for a method call. Is the method going to be executed from the same class at run-time which is bounded to the class at the compile-time or is the method going to be executed from a different class at run-time rather than the class bounded at compile time? Let us proceed and see the answer.

**What is Compile-Time Polymorphism in C#?**

The function call bounded to the class at the time of compilation, if the function is going to be executed from the same bounded class at run-time, then it is called Compile-Time Polymorphism in C#. This happens in the case of [**Method Overloading**](https://dotnettutorials.net/lesson/function-overloading-csharp/) because, in case of overloading each method will have a different signature, and based on the method call, we can easily recognize the method which matches the method signature.

It is also called [**Static Polymorphism or Early Binding**](https://dotnettutorials.net/lesson/function-overloading-csharp/) as at the Compilation time we will be able to know from which class the method going to be executed.

**What is Runtime Polymorphism in C#?**

The function call bounded to the class at the time of compilation, if the function is going to be executed from a different class (Parent Class) at run-time rather than the class bounded at compilation-time, then it is called Run-Time Polymorphism. This happens in the case of [**Method Overriding**](https://dotnettutorials.net/lesson/function-overriding-csharp/)because, in the case of Overriding, we have multiple methods with the same signature i.e. Parent Class and the Child class having the same method implementation. So, in this case, we will be able to know at runtime from which class the method is going to be executed.

It is also called [**Dynamic Polymorphism or Late Binding**](https://dotnettutorials.net/lesson/function-overriding-csharp/) as at Run-time we will be able to know from which class the method is going to be executed.

**Method Overloading in C#**

**Method Overloading in C# with Examples**

In this article, I am going to discuss **Method Overloading in C#** with Examples. Please read our previous article before proceeding to this article where we discussed the basics of [**Polymorphism in C#**](https://dotnettutorials.net/lesson/polymorphism-csharp/). At the end of this article, you will have a very good understanding of the following pointers related to Method Overloading.

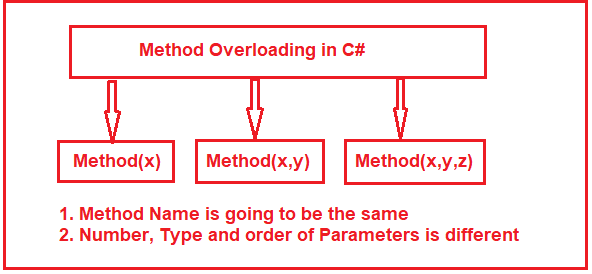
1. **What is Method Overloading in C#?**
2. **When should we overload methods?**
3. **What are the advantages of using Method Overloading in C#?**
4. **When is a method considered as an overloaded method?**
5. **What is the execution control flow of overloaded methods in C#?**
6. **What is Inheritance based overloading?**
7. **Real-time scenarios where you need to use Method Overloading?**

**Note:** The point that you need to keep in mind is function overloading and method overloading terms are interchangeably used. Method overloading is one of the common ways to implement **Compile-Time Polymorphism in C#**.

**What is Method Overloading or Function Overloading in C#?**

If we are defining multiple methods with the same name but with a different signature in a class or in the Parent and Child class, then it is called Method Overloading in C#. That means C#.NET allows us to create a method in the derived class with the same name as the method name defined in the base class.

In simple words, we can say that the **Method Overloading in C#** allows a class to have multiple methods with the same name but with a different signature. The functions or methods can be overloaded based on the number, type (int, float, etc), order, and kind (Value, Ref or Out) of parameters. For better understanding, please have a look at the below image.



The signature of a method consists of the name of the method and the data type, number, order, and kind (Value, Ref or Out) of parameters.

**Note:**The point that you need to keep in mind is that the signature of a method does not include the return type and the params modifiers. So it is not possible to overload a method just based on the return type and params modifier.

We can compare the function overloading with a person overloading. For example, if a person has already some work to do and if we are assigning some additional work to that person then the person’s work will be overloaded. In the same way, a function will have already some work to do and if we are assigning some additional work to that function, then we can say that the function is overloaded.

**When should we overload methods in C#?**

If you want to execute the same logic but with different types and numbers of arguments, then you need to overload the methods. For example, if you want to add two integers, two floats, and two strings, then you need to define three methods with the same name as shown in the below example.

**namespace** *PolymorphismDemo*

**{**

**class** Program

**{**

**public** **void** **add(int** a, **int** b**)**

**{**

Console.WriteLine**(**a + b**)**;

**}**

**public** **void** **add(float** x, **float** y**)**

**{**

Console.WriteLine**(**x + y**)**;

**}**

**public** **void** **add(**string s1, string s2**)**

**{**

Console.WriteLine**(**s1 + s2**)**;

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

Program obj = new Program**()**;

obj.add**(**10, 20**)**;

obj.add**(**10.5f, 20.5f**)**;

obj.add**(**"pranaya", "kumar"**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

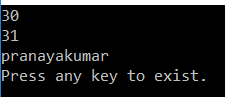
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**What are the advantages of using Method Overloading in C#? Or what are the disadvantages if we define methods with a different name?**

If we overload the methods, then the user of our application gets comfort feeling in using the method with an impression that he/she calling one method by passing different types of values. The best example for us is the system-defined “**WriteLine()**” method. It is an overloaded method, not a single method taking different types of values.

**When is a method considered as an overloaded method in C#?**

If two methods have the same method name but with different signatures, then those methods are considered overloaded methods. Then the rule we should check is both methods must have different parameter **types/numbers/orders**. But there is no rule on return type, non-accessibility modifier and accessibility modifier means overloading methods can have their own return type, non-accessibility modifier, and accessibility modifier because overloading methods are different methods

**Can we overload methods in the same class?**

Yes, it is possible. No Compile Time Error, and no Runtime Error. Methods can be overloaded in the same or in super and sub classes because overloaded methods are different methods. But we can’t override a method in the same class it leads to Compile Time Error: “**method is already defined**” because overriding methods are the same methods with a different implementation.

**What is the execution control flow of overloaded methods?**

The compiler always checks for the called method definition in the reference variable type class with the given argument type parameter. So in searching and executing a method definition, we must consider both reference variable type and argument type. The Referenced variable type for deciding from which class method should be to bind. Argument type for deciding which overloaded method should be to bind.

**For example:**  
**B b = new B();**  
**b.m1(50) => b.m1(int);**

In the above method call, the compiler searches **m1()** method definition in the “**B”** class with integer parameter at the time of program compilation, and if it found that method then it binds that method definition. The compiler will searches in the B class because the type of the reference variable b is B type.

**A a = new B();**  
**a.m1(50); => a.m1(int);**

In the above method call, at the time of compilation, the compiler will search **m1()** method definition in the “**A”** class with an integer parameter not in the B class even though the object is B. This is because, at compilation time, the compiler will check only the reference variable type, not the object type it holds. And here, the reference variable a type is A and it holds the object whose type is B.

**What is Inheritance-Based Overloading in C#?**

A method that is defined in the parent class can also be overloaded under its child class. It is called Inheritance-Based Overloading in C#. See the following example for a better understanding. As you can see in the below code, we have defined the add method twice in the ADD1 class and also defined the add method in the child ADD2 class. Here, notice every add method taking different types of parameters.

**namespace** *PolymorphismDemo*

**{**

**class** ADD1

**{**

**public** **void** **add(int** a, **int** b**)**

**{**

Console.WriteLine**(**a + b**)**;

**}**

**public** **void** **add(float** x, **float** y**)**

**{**

Console.WriteLine**(**x + y**)**;

**}**

**}**

**class** ADD2 : ADD1

**{**

**public** **void** **add(**string s1, string s2**)**

**{**

Console.WriteLine**(**s1 + s2**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

ADD2 obj = new ADD2**()**;

obj.add**(**10, 20**)**;

obj.add**(**10.5f, 20.5f**)**;

obj.add**(**"pranaya", "kumar"**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

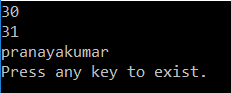
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Note:** To overload a parent class method under its child class the child class does not require any permission from its parent class.

**Real-life Scenario of Method Overloading in C#**

Suppose you are working on a maintenance project. And you are going to work on a class where already some parameterized constructors have been defined and you need to pass some additional parameters. So what you will do, either add the required parameter with one of the already defined constructors or add a new constructor as per your requirement. In such cases, you should not add the required parameter with the already defined constructor because this may disturb your other class dependency structure. So what you will do is create a new constructor with the required parameter. That new constructor that you are creating is nothing but the constructor overloading.

**Example: Constructor Overloading in C#**

Please have a look at the following example. Here, we are creating three different versions of the Constructor, and each constructor taking a different number of parameters and this is called Constructor Overloading in C#.

**using** *System;*

**namespace** *ConstructorOverloading*

**{**

**class** ConstructorOverloading

**{**

**int** x, y, z;

**public** ConstructorOverloading**(int** x**)**

**{**

this.x = 10;

**}**

**public** ConstructorOverloading**(int** x, **int** y**)**

**{**

this.x = x;

**}**

**public** ConstructorOverloading**(int** x, **int** y, **int** z**)**

**{**

this.x = x;

**}**

**}**

**class** Test

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

ConstructorOverloading obj1 = new ConstructorOverloading**(**10**)**;

ConstructorOverloading obj2 = new ConstructorOverloading**(**10, 20**)**;

ConstructorOverloading obj3 = new ConstructorOverloading**(**10, 20, 30**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Method Overriding in C#**

**Method Overriding in C# with Examples**

In this article, I am going to discuss **Method Overriding in C#** with examples. Please read our previous article where we discussed [**Method Overloading**](https://dotnettutorials.net/lesson/function-overloading-csharp/)in detail. Here in this article, we are going to discuss the following pointers with examples.

1. **What is Method Overriding in C#?**
2. **When do we need to override a method in C#?**
3. **When a subclass method is treated as an overridden method in C#?**
4. **How a method is Overridden in C#?**
5. **How to execute the superclass method if it is overridden in the subclass?**
6. **What are the differences between Method Overloading and Method Overriding in C#?**

**Note:** The terms Function Overriding and Method Overriding are interchangeably used. Function Overriding is another approach to implement Polymorphism (i.e. Run-Time Polymorphism) in C#.

**What is Method Overriding in C#?**

The process of re-implementing the superclass non-static and non-private method in the subclass with the same signature is called Function Overriding or Method Overriding in C#. The same signature means the name and the parameters should be the same. The implementation of the subclass overrides (i.e. replaces) the implementation of the superclass method.

The point that you need to keep in mind is that the overriding method is always going to be executed from the current class object. The superclass method is called the overridden method and the sub-class method is called the overriding method.

**When do we need to override a method in C#?**

If the superclass method logic is not fulfilling the sub-class business requirements, then the subclass needs to override that method with the required business logic. Usually, in most real-time applications, the superclass methods are implemented with generic logic which is common for all the next-level sub-classes.

**When is a sub-class method treated as an overriding method in C#?**

If a method in sub-class contains the same signature as the superclass non-private non-static method, then the subclass method is treated as the overriding method and the superclass method is treated as the overridden method.

**How can we override a parent class method under child class in C#?**

If you want to override the parent class method in its child class, first the method in the parent class must be declared as **virtual** by using the keyword **virtual,** then only the child classes get the permission for overriding that method. Declaring the method as **virtual** is marking the method as overridable. If the child class wants to override the parent class virtual method then the child class can do it with the help of the **override** modifier. But overriding the method under child class is not mandatory for the child classes.

Syntax:  
**Class1:  
Public virtual void show(){} //virtual function (overridable)  
Class2: Class1  
Public override void show(){} //overriding**

**Example: Method Overriding in C#**

Let us see an example for understanding method overriding in C#. Please have a look at the below code. Here the class “class1” is the parent class and in this class, we defined one method i.e. show() method by using the virtual keyword which marks this method to be overridable in the child class. The class “Class2” is derived from the class “Class1” and hence it becomes a child class of class “Class1” and as soon as it becomes a child class, it got permission to override the overridable method show(). As you can see in the child class we override the show method using the override modifier.

**namespace** *PolymorphismDemo*

**{**

**class** Class1

**{**

**public** **virtual** **void** show**()**

**{**

Console.WriteLine**(**"Super class show method"**)**;

**}**

**}**

**class** Class2 : Class1

**{**

**public** **override** **void** show**()**

**{**

Console.WriteLine**(**"Sub class override show method"**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Class2 obj = new Class2**()**;

obj.show**()**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Execution Flow:**

Now let us understand when we create the object and call the show method as shown below, what happens internally?  
**Class2 obj = new Class2();  
obj.show();**In the above method call, the compiler will search show() method definition in the class “Class2” as the reference type of obj variable is Class2. As it found the method definition in the class “Class2”, so the compilation is successful. At run time, the CLR will check the method implementation in class “Class2” as the reference variable obj holds the Class2 object reference. And it will execute the method from that class. And once you execute the above code, you will get the following output.

Function Overriding in C#

**Example: Parent class Reference Variable holding subclass object reference.**

The following is the same example as the previous one. But here, the only change is the parent class reference variable holds the sub-class object reference.

**using** *System;*

**namespace** *PolymorphismDemo*

**{**

**class** Class1

**{**

**public** **virtual** **void** show**()**

**{**

Console.WriteLine**(**"Super class show method"**)**;

**}**

**}**

**class** Class2 : Class1

**{**

**public** **override** **void** show**()**

**{**

Console.WriteLine**(**"Sub class override show method"**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Class1 obj = new Class2**()**;

obj.show**()**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Execution Flow:**

Now let us understand the execution flow of the above example. When we create the object and call the show method as shown below, what happens internally?  
**Class1 obj = new Class2();  
obj.show();**In the above method call, the compiler will search show() method definition in the class “Class1” as the reference type of obj variable is Class1. As it found the method definition in the class “Class1”, so the compilation is successful. At run time, the CLR will check the method implementation in class “Class2” as the reference variable obj holds the Class2 object reference. And it will execute the method from that class. And once you execute the above code, you will get the following output.

Function Overriding in C#

**How can we execute the superclass method if it is overridden in the sub-class in C#?**

Once we re-implement the parent class methods under the child class, then the object of the child class calls its own method but not its parent class method. But if you want to still consume or call the parent class’s methods from the child class, then it can be done in two different ways.

By creating the parent class object under the child class, we can call the parent class methods from the child class, or by using the **base** keyword, we can call parent class methods from the child class, but **this** and **base** keyword cannot be used under the static block.

**Using the base keyword to call the parent class methods in C#:**

Let us see an example for a better understanding. As you can see in the below code, from the child class show method we call the parent class show method using **base.show()** method call.

**namespace** *PolymorphismDemo*

**{**

**class** Class1

**{**

**public** **virtual** **void** show**()**

**{**

Console.WriteLine**(**"Super class show method"**)**;

**}**

**}**

**class** Class2 : Class1

**{**

**public** **override** **void** show**()**

**{**

**base**.show**()**;

Console.WriteLine**(**"Sub class override show method"**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Class2 obj = new Class2**()**;

obj.show**()**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Function Overriding in C#

**What is the difference between Method Overloading and Method Overriding in C#?**

|  |  |
| --- | --- |
| **Method Overloading in C#** | **Method Overriding in C#** |
| It is an approach of defining multiple methods with the same name but with a different signature. | It is an approach of defining multiple methods with the same name and with the same signature. |
| Overloading a method can be performed within a class or within the child classes also. | Overriding of methods is not possible within the same class it must be performed under the child classes. |
| To overload a parent class method under the child class, the child class does not require permission from the parent. | To override a parent class method under the child class, first, the child class requires explicit permission from its parent. |
| This is all about defining multiple behaviors to a method. | This is all about changing the behavior of a method. |
| Used to implement static polymorphism. | Used to implement dynamic polymorphism. |
| This is a code refinement technique. | This is a code replacement technique. |
| No separate keywords are used to implement function overloading. | Use the **virtual** keyword for the base class function and **override**keyword in the derived class function to implement function overriding. |

**Method Hiding in C#**

**Method Hiding in C# with Examples**

In this article, I am going to discuss **Method Hiding in C#** with Examples. Please read our previous article, where we discussed [**Method Overriding in C#**](https://dotnettutorials.net/lesson/function-overriding-csharp/) with examples. At the end of this article, you will understand what exactly Method Hiding is and when and how to use Method Hiding in C#?

**What is Method Hiding in C#?**

When we use the **new** keyword to hide a base class member, then it is called **Method Hiding in C#**. We will get a compiler warning if we miss the new keyword. This is also used for re-implementing a parent class method under child class. Re-implementing parent class methods under child classes can be done using two different approaches, such as

1. **Method Overriding**
2. **Method Hiding**

In the first case, we re-implement the parent class methods under child classes with the permission of parent class because here in parent class the method is declared as **virtual** giving permission to the child classes for overriding the methods using the **override** modifier.

In the 2nd approach, we re-implement the method of parent class even if those methods are not declared as virtual that is without parent permission we are re-implementing the methods.

**The syntax for Method Hiding in C#:**

Please have a look at the following image to understand the syntax of Function Hiding in C#.



**Note:** Using the **new** keyword for re-implementing the methods in the child class is optional and if used will give information to hiding.

**Example to understand Method Hiding in C#:**

Please have a look at the following example to understand the concept of Method Hiding in C#. As you can see, in the parent class we have not used the virtual keyword in the method signature of the display method. In the child class, we have re-implement the method using the new keyword.

**namespace** *PolymorphismDemo*

**{**

**class** Class1

**{**

**public** **void** display**()**

**{**

Console.WriteLine**(**"Super class display method"**)**;

**}**

**}**

**class** Class2 : Class1

**{**

**public** new **void** display**()**

**{**

Console.WriteLine**(**"Sub class display method"**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Class2 obj = new Class2**()**;

obj.display**()**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Method Hiding in C#

**Example: Method Hiding in C#**

In the below example, we are creating the object of Class2 and storing it in the Class1 reference variable. In this case method overriding, the method is going to be executed from the reference type.

**namespace** *PolymorphismDemo*

**{**

**class** Class1

**{**

**public** **void** display**()**

**{**

Console.WriteLine**(**"Super class display method"**)**;

**}**

**}**

**class** Class2 : Class1

**{**

**public** new **void** display**()**

**{**

Console.WriteLine**(**"Sub class display method"**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Class1 obj = new Class2**()**;

obj.display**()**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: Super class display method**

**What is the difference between Method Overriding and Method Hiding in C#?**

A parent class method can be redefined under its child class using two different approaches.

1. **Method Overriding.**
2. **Method Hiding.**

In Method Overriding parent class gives permission for its child class to override the method by declaring it as **virtual**. Now the child class can override the method using the **Override** keyword as it got permission from the parent.

In Method hiding also, the parent class methods redefined under the child classes even if they were not declared as Virtual in the parent class, and in this case, we need to re-implement the method in the child class by using the ‘**new**‘ keyword.

In method overriding a base class reference variable pointing to a child class object will invoke the overridden method in the child class. In method hiding a base class reference variable pointing to a child class object will invoke the hidden method in the base class.

New is reference-type specific, overriding is object-type specific.

**Partial Classes and Partial Methods in C#**

**Partial Classes and Partial Methods in C# with Examples**

In this article, I am going to discuss **Partial Classes and Partial Methods in C#** with Examples. Please read our previous article, where we discussed [**Method Hiding in C#**](https://dotnettutorials.net/lesson/function-hiding-csharp/). At the end of this article, you will understand what are Partial classes and Partial Methods and when and how to use Partial Classes and Partial Methods in C# with Examples.

**What are Partial Classes in C#?**

Partial Classes are 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.

Partial classes allow us to split a class definition into 2 or more files. It is also possible to split the definition of a struct or an interface over two or more source files. Each source file will contain a section of the class definition, and all parts are combined into a single class when the application is compiled.

**Understanding Partial Classes in C# with an Example:**

Create a console application. Add a class file with the name **Employee.cs** to the project. Copy and paste the following code in the **Employee.cs** class file.

**namespace** *PartialDemo*

**{**

**public** **class** Employee

**{**

**private** string \_firstName;

**private** string \_lastName;

**private** **double** \_salary;

**private** string \_gender;

**public** string FirstName

**{**

**get** **{** **return** \_firstName; **}**

**set** **{** \_firstName = **value**; **}**

**}**

**public** string LastName

**{**

**get** **{** **return** \_lastName; **}**

**set** **{** \_lastName = **value**; **}**

**}**

**public** **double** Salary

**{**

**get** **{** **return** \_salary; **}**

**set** **{** \_salary = **value**; **}**

**}**

**public** string Gender

**{**

**get** **{** **return** \_gender; **}**

**set** **{** \_gender = **value**; **}**

**}**

**public** **void** DisplayFullName**()**

**{**

Console.WriteLine**(**@"Full Name is : {0} {1}", \_firstName, \_lastName**)**;

**}**

**public** **void** DisplayEmployeeDetails**()**

**{**

Console.WriteLine**(**"Employee Details : "**)**;

Console.WriteLine**(**@"First Name : {0}", \_firstName**)**;

Console.WriteLine**(**@"Last Name : {0}", \_lastName**)**;

Console.WriteLine**(**@"Gender : {0}", \_gender**)**;

Console.WriteLine**(**@"Salary : {0}", \_salary**)**;

**}**

**}**

**}**

This is a very simple Employee class having 4 private fields, 4 public properties, and 2 public methods. Let’s use the above class in our Main method. Please modify the Program class which contains the Main method as shown below.

**namespace** *PartialDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Employee emp = new Employee**()**;

emp.FirstName = "Pranaya";

emp.LastName = "Rout";

emp.Salary = 100000;

emp.Gender = "Male";

emp.DisplayFullName**()**;

emp.DisplayEmployeeDetails**()**;

Console.WriteLine**(**"Press any key to exist."**)**;

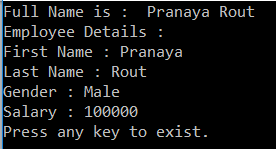
Console.ReadKey**()**;

**}**

**}**

**}**

**Run the application and see it will give you the below output.**



**Splitting the above class definition into 2 files.**

Now we will split the Employee class definition into two files. One class file is going to contain, all the private fields and public properties and the other class file is going to contain the two public methods.

First, delete the **Employee.cs** class file from the project. Then we need to add two class files with the name **PartialEmployeeOne** and **PartialEmployeeTwo**. To do so, right-click on the project and add a class file with the name **PartialEmployeeOne.cs** and copy and paste the following code. Notice that in the above code, the **PartialEmployee** class is marked with the **partial**keyword and it contains only the 4 private fields and the 4 public properties. Next, we need to add **PartialEmployeeTwo.cs**.

**namespace** *PartialDemo*

**{**

**public** **partial** **class** PartialEmployee

**{**

**private** string \_firstName;

**private** string \_lastName;

**private** **double** \_salary;

**private** string \_gender;

**public** string FirstName

**{**

**get** **{** **return** \_firstName; **}**

**set** **{** \_firstName = **value**; **}**

**}**

**public** string LastName

**{**

**get** **{** **return** \_lastName; **}**

**set** **{** \_lastName = **value**; **}**

**}**

**public** **double** Salary

**{**

**get** **{** **return** \_salary; **}**

**set** **{** \_salary = **value**; **}**

**}**

**public** string Gender

**{**

**get** **{** **return** \_gender; **}**

**set** **{** \_gender = **value**; **}**

**}**

**}**

**}**

**Note:** Here the class file name is **PartialEmployeeOne.cs** but the class name is **PartialEmployee**

**Adding** **PartialEmployeeTwo.cs**

To do so, right-click on the project and add a class file with the name **PartialEmployeeTwo.cs** and copy and paste the following code. Notice that in the above code, the **PartialEmployee** class is also marked with the **partial**keyword and it contains only the two public methods. Here, we are able to access the private fields, **\_firstName**, **\_lastName, \_salary, and \_gender**, that are defined in **PartialEmployeeOne.cs** file.

**namespace** *PartialDemo*

**{**

**public** **partial** **class** PartialEmployee

**{**

**public** **void** DisplayFullName**()**

**{**

Console.WriteLine**(**@"Full Name is : {0} {1}", \_firstName, \_lastName**)**;

**}**

**public** **void** DisplayEmployeeDetails**()**

**{**

Console.WriteLine**(**"Employee Details : "**)**;

Console.WriteLine**(**@"First Name : {0}", \_firstName**)**;

Console.WriteLine**(**@"Last Name : {0}", \_lastName**)**;

Console.WriteLine**(**@"Gender : {0}", \_gender**)**;

Console.WriteLine**(**@"Salary : {0}", \_salary**)**;

**}**

**}**

**}**

Here the class file name is **PartialEmployeeTwo.cs** but the class name is **PartialEmployee.**Now Modify the Main method of the **Program** class as shown below to use the **PartialEmployee** class.

**namespace** *PartialDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

PartialEmployee emp = new PartialEmployee**()**;

emp.FirstName = "Pranaya";

emp.LastName = "Rout";

emp.Salary = 100000;

emp.Gender = "Male";

emp.DisplayFullName**()**;

emp.DisplayEmployeeDetails**()**;

Console.WriteLine**(**"Press any key to exist."**)**;

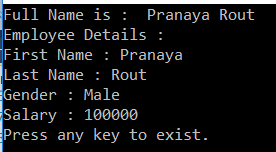
Console.ReadKey**()**;

**}**

**}**

**}**

**Now run the application and see the out.**



**When do we need to use Partial Classes in C#?**

There are several situations when splitting a class definition is desirable

1. When working on large projects, splitting a class over separate files allows multiple programmers to work on it simultaneously.
2. When working with automatically generated source code, the code can be added to the class without having to recreate the source file. Visual Studio uses this approach when creating windows form, Web service wrapper code, and so on.

In My Project, I am using Entity Framework Database First Approach. In that case, the Entity Framework will create the models i.e. the classes based on the database and it creates the classes as partial classes. Next, I want to do some modifications with the auto-generated partial classes like adding some additional property or adding some attribute. But, if I do the modification with the auto-generated partial classes, then my changes will be lost when I update the EDMX file. So, what I generally do it, create a partial class and in that partial class, I do all the customization.

**Rules to follow when working with Partial Classes in C#:**

All the parts spread across different class files, must use the **partial**keyword. Otherwise, a compiler error is raised. **Missing partial modifier. Another partial declaration of this type exists.**

All the parts spread across different files, must have the **same access specifiers**. Otherwise, a compiler error is raised. **Partial declarations have conflicting accessibility modifiers.**

If any of the parts are declared as abstract, then the **entire type is considered as abstract** or if any of the parts are declared as sealed, **then the entire type is considered as sealed** or if any of the parts inherit a class, **then the entire type inherits that class.**

**C# does not support multiple class inheritance.** Different parts of the partial class must not specify different base classes. The following code will raise a compiler error stating – **Partial declarations must not specify different base classes.**

**public** **class** Employee

**{**

**}**

**public** **class** Customer

**{**

**}**

**public** **partial** **class** PartialClass : Employee

**{**

**}**

**public** **partial** **class** PartialClass : Customer

**{**

**}**

Different parts of the partial class can specify different base interfaces and the final type **implements all of the interfaces listed by all of the partial declarations.**In the example below **PartialClass** needs to provide the implementation for both **IEmployee** and **ICustomer** interface methods.

**public** **interface** IEmployee

**{**

**void** EmployeeMethod**()**;

**}**

**public** **interface** ICustomer

**{**

**void** CustomerMethod**()**;

**}**

**public** **partial** **class** PartialClass : IEmployee

**{**

**public** **void** EmployeeMethod**()**

**{**

//Method Implementation

**}**

**}**

**public** **partial** **class** PartialClass : ICustomer

**{**

**public** **void** CustomerMethod**()**

**{**

//Method Implementation

**}**

**}**

**Note:** Any members that are declared in a partial definition are available to all of the other parts of the partial class. Once we understand Partial Classes in C#, let’s understand Partial Methods in C#.

**What are Partial Methods in C#?**

A partial class or a struct can contain partial methods. A partial method is created using the same **partial** keyword. Let us understand partial methods with an example. Create a console application. Add a class file with the name **PartialClassOne.cs** to the project. Copy and paste the following code.

**namespace** *PartialDemo*

**{**

**partial** **class** PartialClass

**{**

// Declaration of the partial method.

**partial** **void** PartialMethod**()**;

// A public method calling the partial method

**public** **void** PublicMethod**()**

**{**

Console.WriteLine**(**"Public Method Invoked"**)**;

PartialMethod**()**;

**}**

**}**

**}**

Notice that the **PartialMethod()** definition has the partial keyword and does not have a body(implementation) only the signature. The implementation of a partial method is optional. If we don’t provide the implementation, the compiler removes the signature and all calls to the method.

The implementation can be provided in the same physical file or in another physical file that contains the partial class. In this example, the partial **PartialMethod()** is invoked within the **PublicMethod()**.

**Copy and paste the following code in the Main() method of the** **Program class**.

**namespace** *PartialDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

PartialClass SC = new PartialClass**()**;

SC.PublicMethod**()**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Partial Class and Partial Methods in C#

When we run the application, now notice that we don’t get a compiler error, in spite of not having an implementation for the partial **PartialMethod()**. Since the implementation of the partial method is missing, the compiler will remove the signature and all calls to the method.

Now add a class file with the name **PartialClassTwo**.**cs**. Copy and paste the following code. The implementation of the partial method is provided here.

**namespace** *PartialDemo*

**{**

**partial** **class** PartialClass

**{**

// Partial method implemented

**partial** **void** PartialMethod**()**

**{**

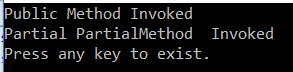
Console.WriteLine**(**"Partial PartialMethod Invoked"**)**;

**}**

**}**

**}**

Now, run the console application and notice the output. The partial method and the public method messages are printed on the console as shown below.



**So, a Partial Method declaration consists of two parts in C#.**

1. The definition (only the method signature ending with a semi-colon, without method body)
2. The implementation.

Partial methods in C# are private by default and it is a compile-time error to include any access modifiers, including private. The following code will raise an error stating – **A partial method cannot have access modifiers or the virtual, abstract, override, new, sealed, or extern modifiers.**

**partial** **class** PartialClass

**{**

**private** **partial** **void** PartialMethod**()**;

**}**

It is a compile-time error to include declaration and implementation at the same time for a partial method. The code below produces a compile-time error – **No defining declaration found for implementing declaration of partial method ‘PartialDemo.PartialClass.partialMethod()’**

**partial** **class** PartialClass

**{**

**partial** **void** PartialMethod**()**

**{**

Console.WriteLine**(**"PartialMethod Implemented"**)**;

**}**

**}**

A partial method return type must be void. Including any other return type is a compile-time error – **Partial methods must have a void return type**

**partial** **class** PartialClass

**{**

**partial** **int** PartialMethod**()**;

**}**

A partial method must be declared within a partial class or partial struct. A non-partial class or struct cannot include partial methods. The signature of the partial method declaration must match with the signature of the implementation.

A partial method can be implemented only once. Trying to implement a partial method more than once raises a compile-time error – **A partial method may not have multiple implementing declarations.**

**Sealed Class in C#**

**Sealed Class and Sealed Methods in C# with Examples**

In this article, I am going to discuss **Sealed Class and Sealed Methods in C#**with Examples. Please read our previous article where we discussed [**Partial Classes and Partial Methods in C#**](https://dotnettutorials.net/lesson/partial-classes-partial-methods-csharp/). At the end of this article, you will understand what exactly Sealed Class in C# and Sealed Methods in C# are and when and how to use them with examples.

**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 also define the class that is declared using the sealed modifier is known as the sealed class and a sealed class cannot be inherited by any other class. For example:

**sealed class Class1 {}**  
**class class2 : Class1{} //invalid**

To make any class a sealed class we need to use the keyword sealed.

**Points to Remember while working with Sealed Class**

1. A sealed class is completely opposite to an abstract class.
2. This sealed class cannot contain abstract methods.
3. It should be the bottom-most class within the inheritance hierarchy.
4. A sealed class can never be used as a base class.
5. The sealed class is specially used to avoid further inheritance.
6. The keyword sealed can be used with classes, instance methods, and properties.

**Note:** Even if a sealed class cannot be inherited we can still consume the class members from any other class by creating the object of the class.

**Sealed Methods in C#**

The method that is defined in a parent class, if that method cannot be overridden under a child class, we call it a sealed method. By default, every method is a sealed method because overriding is not possible unless the method is not declared as virtual in the parent class. If a method is declared as virtual in a class, any child class of it can have the right to override that method. For Example:

**namespace** *SealedDemo*

**{**

**class** class1

**{**

**public** **virtual** **void** show**()** **{** **}**

**}**

**class** class2 : class1

**{**

**public** **override** **void** show**()** **{** **}**

**}**

**class** class3 : class2

**{**

**public** **override** **void** show**()** **{** **}**

**}**

**}**

In the above case even if the first child is not overriding the method the second child can still override the method. When a child class is overriding its parent class virtual methods the child class uses the sealed modifier in the method so that further overriding of the method will not be possible i.e. child classes cannot override the methods.

**For example:**

**namespace** *SealedDemo*

**{**

**class** class1

**{**

**public** **virtual** **void** show**()** **{** **}**

**}**

**class** class2 : class1

**{**

**public** **sealed** **override** **void** show**()** **{** **}**

**}**

**class** class3 : class2

**{**

//'class3.show()': cannot override inherited member 'class2.show()' because it is sealed

**public** **override** **void** show**()** **{** **}** //Invalid

**}**

**}**

**Let’s understand one example for better understanding.**

**namespace** *SealedDemo*

**{**

**public** **class** Employee

**{**

**protected** **int** Eid, Eage;

**protected** string Ename, Eaddress;

**public** **virtual** **void** GetEmployeeData**()**

**{**

Console.WriteLine**(**"ENTER EMPLOYEE DETAILS:"**)**;

Console.WriteLine**(**"ENTER THE ID"**)**;

Eid = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"ENTER THE NAME"**)**;

Ename = Console.ReadLine**()**;

Console.WriteLine**(**"ENTER THE ADDRESS"**)**;

Eaddress = Console.ReadLine**()**;

Console.WriteLine**(**"ENTER THE AGE"**)**;

Eage = **int**.Parse**(**Console.ReadLine**())**;

**}**

**public** **virtual** **void** DisplayEmployeeData**()**

**{**

Console.WriteLine**(**"\nEMPLOEE DETAILS ARE:"**)**;

Console.WriteLine**(**"EMPLOYEE ID IS: " + Eid**)**;

Console.WriteLine**(**"EMPLOYEE NAME IS: " + Ename**)**;

Console.WriteLine**(**"EMPLOYEE ADDRESS IS:" + Eaddress**)**;

Console.WriteLine**(**"EMPLOYEE AGE IS : " + Eage**)**;

**}**

**}**

**public** **sealed** **class** Manager : Employee

**{**

**double** Bonus, CA;

**public** **override** **void** GetEmployeeData**()**

**{**

Console.WriteLine**(**"ENTER MANAGER DETAILS:"**)**;

Console.WriteLine**(**"ENTER THE ID"**)**;

Eid = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"ENTER THE NAME"**)**;

Ename = Console.ReadLine**()**;

Console.WriteLine**(**"ENTER THE BONUS"**)**;

Bonus = **double**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"ENTER THE CA"**)**;

CA = Convert.ToDouble**(**Console.ReadLine**())**;

**}**

**public** **override** **void** DisplayEmployeeData**()**

**{**

Console.WriteLine**(**"MANAGER DETAILS ARE:"**)**;

Console.WriteLine**(**"EMPLOYEE ID IS: " + Eid**)**;

Console.WriteLine**(**"EMPLOYEE NAME IS: " + Ename**)**;

Console.WriteLine**(**"MANAGER BONUS IS: " + Bonus**)**;

Console.WriteLine**(**"MANAGER CA IS : " + CA**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Manager m1 = new Manager**()**;

m1.GetEmployeeData**()**;

m1.DisplayEmployeeData**()**;

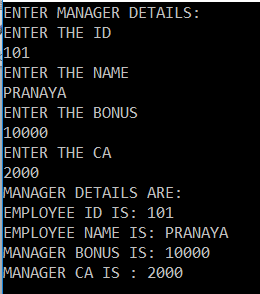
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**When should a method be declared as sealed in C#?**

If we don’t want to allow subclasses to override the superclass method and to ensure that all sub-classes use the same superclass method logic then that method should be declared as sealed. The sealed method cannot be overridden in sub-classes violation leads to a compile-time error.

**What is the difference between the private and sealed methods in C#?**

The private method is not inherited whereas the sealed method is inherited but cannot be overridden in C#. So, a private method cannot be called from sub-classes whereas a sealed method can be called from sub-classes. The same private method can be defined in sub-class and it does not lead to error.

**When should a class be declared as sealed in C#?**

In the below situations we must define the class as sealed

1. If we don’t want to override all the methods of our class in sub-classes.
2. And if we don’t want to extend our class functionality.

**What are the differences between an abstract class and a sealed class in C#?**

|  |  |  |
| --- | --- | --- |
| **SL NO** | **ABSTRACT CLASS** | **SEALED CLASS** |
| 1. | A class that contains one or more abstract methods is known as an abstract class. | A class from which it is not possible to derive a new class is known as a sealed class. |
| 2. | The abstract class can contain abstract and non-abstract methods. | The sealed class can contain non-abstract methods; it cannot contain abstract and virtual methods. |
| 3. | Creating a new class from an abstract class is compulsory to consume. | It is not possible to create a new class from a sealed class. |
| 4. | An abstract class cannot be instantiated directly; we need to create the object for its child classes to consume an abstract class. | We should create an object for a sealed class to consume its members. |
| 5. | We need to use the keyword abstract to make any class abstract. | We need to use the keyword sealed to make any class as sealed. |
| 6. | An abstract class cannot be the bottom-most class within the inheritance hierarchy. | The sealed class should be the bottom-most class within the inheritance hierarchy. |

**Extension Methods in C#**

**Extension Methods in C# with Examples**

In this article, I am going to discuss the **Extension Methods in C#** with examples. Please read our previous article where we discussed [**Sealed Class and Sealed Methods in C#**](https://dotnettutorials.net/lesson/sealed-class-methods-csharp/). At the end of this article, you will understand what exactly C# Extension Methods are and when and how to use these extension methods in C#?

**What are Extension Methods in C#?**

It is a new feature that has been added in C# 3.0 which allows us to add new methods into a class without editing the source code of the class i.e. if a class consists of a set of members in it and in the future if you want to add new methods into the class, you can add those methods without making any changes to the source code of the class.

Extension methods can be used as an approach to extending the functionality of a class in the future if the source code of the class is not available or we don’t have any permission in making changes to the class.

Before extension methods, inheritance is an approach that used for extending the functionality of a class i.e. if we want to add any new members into an existing class without making a modification to the class, we will define a child class to that existing class and then we add new members in the child class.

In the case of an extension method, we will extend the functionality of a class by defining the methods, we want to add into the class in a new class and then bind them to an existing class.

Both these approaches can be used for extending the functionalities of an existing class whereas, in inheritance, we call the method defined in the old and new classes by using object of the new class whereas, in the case of extension methods, we call the old and new methods by using object of the old class.

**Extension Methods Example in C#:**

Let us understand C# Extension Methods with an example. Create a console application and then add a class file with the name **OldClass.cs** and then copy and paste the following code in it.

**public** **class** OldClass

**{**

**public** **int** x = 100;

**public** **void** Test1**()**

**{**

Console.WriteLine**(**"Method one: " + this.x**)**;

**}**

**public** **void** Test2**()**

**{**

Console.WriteLine**(**"Method two: " + this.x**)**;

**}**

**}**

Now our requirement is to add three new methods to the class **OldClass**. But we don’t want to change the source code of **OldClass**. Then we can achieve this with the help of extension methods. Let’s create a new class with the name **NewClass.cs** and then copy and paste the following code in it.

**public** **static** **class** NewClass

**{**

**public** **static** **void** Text3**(**this OldClass O**)**

**{**

Console.WriteLine**(**"Method Three"**)**;

**}**

**public** **static** **void** Text4**(**this OldClass O, **int** x**)**

**{**

Console.WriteLine**(**"Method Four: " + x**)**;

**}**

**public** **static** **void** Text5**(**this OldClass O**)**

**{**

Console.WriteLine**(**"Method Five:" + O.x**)**;

**}**

**}**

Let us first test the application, then we will understand the extension methods. Now to test whether the methods are accessed using the old class objects or not, add a class Program.CS and write the following code

**public** **class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

OldClass obj = new OldClass**()**;

obj.Test1**()**;

obj.Test2**()**;

//Calling exrension methods

obj.Text3**()**;

obj.Text4**(**10**)**;

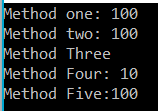
obj.Text5**()**;

Console.ReadLine**()**;

**}**

**}**

Now, run the application and see everything is working as expected and it will display the following output.



**Points to Remember while working with C# Extension methods:**

1. Extension methods must be defined only under the **static class.**
2. As an extension method is defined under a static class, compulsory that the method should be defined as static whereas once the method is bound with another class, the method changes into non-static.
3. The first parameter of an extension method is known as the binding parameter which should be the name of the class to which the method has to be bound and the binding parameter should be prefixed with **this** keyword.
4. An extension method can have only one binding parameter and that should be defined in the first place of the parameter list.
5. If required, an extension method can be defined with a normal parameter also starting from the second place of the parameter list.

**Extension Method Real-time Example:**

Let us see one real-time scenario where you can use the extension method. As we know string is a built-in class provided by .NET Framework. That means the source code of this class is not available to us and hence we can change the source code of the string class. Now our requirement is to add a method to the String class i.e. GetWordCount() and that method will return the number of words present in a string and we should call this method as shown in the below image.

C# Extension Methods Real-time Example

You can achieve the above using Extension Methods. First, create a class with the name **StringExtension** and then copy and paste the following code into it. As you can see, here we created the class as static and hence the GetWordCount as static and provide the first parameter as the string class name so that we can call this method on the String class object.

**namespace** *ExtensionMethodsDemo*

**{**

**public** **static** **class** StringExtension

**{**

**public** **static** **int** GetWordCount**(**this string inputstring**)**

**{**

**if** **(**!string.IsNullOrEmpty**(**inputstring**))**

**{**

string**[]** strArray = inputstring.Split**(**' '**)**;

**return** strArray.Count**()**;

**}**

**else**

**{**

**return** 0;

**}**

**}**

**}**

**}**

Once you have created the extension method, now you can use that method on the String class object. So, modify the Main method of the Program class as shown below.

**namespace** *ExtensionMethodsDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string myWord = "Welcome to Dotnet Tutorials Extension Methods Article";

**int** wordCount = myWord.GetWordCount**()**;

Console.WriteLine**(**"string : " + myWord**)**;

Console.WriteLine**(**"Count : " + wordCount**)**;

Console.Read**()**;

**}**

**}**

**}**

That’s it. Now run the application and you should get the output as expected as shown in the below image.

Real-time Example of Extension Methods in C#