Object and Class

**Object** is a runtime entity, it is created at runtime.



In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class.

The **new** keyword allocates memory at runtime.

**Constructors**

A constructor is a special method of the class which gets automatically invoked whenever an instance of the class is created.

Like methods, a constructor also contains the collection of instructions that are executed at the time of Object creation.

It is used to assign initial values to the **data members** of the same class.

**Types of Constructor**

1. **Default Constructor**

A constructor with no parameters is called a default constructor.

The default constructor initializes all numeric fields to zero and all string and object fields to null inside a class.

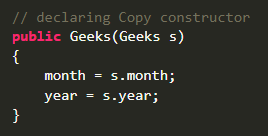
1. **Parameterized Constructor**

A constructor having at least one parameter is called as parameterized constructor. It can initialize each instance of the class to different values.

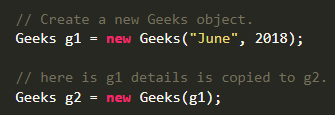
1. **Copy Constructor**

This constructor creates an object by copying variables from another object.

Its main use is to initialize a new instance to the values of an existing instance.



Above is the copy constructor of Geek class which takes Geek type object as parameter so that it can create object with same values when objet is created using new keyword.



1. **Private Constructor**

If a constructor is created with private specifier is known as Private Constructor.

It is **not possible** for other classes to **derive** from this class and also it’s not possible to **create an instance** of this class.

It is the implementation of **a singleton class pattern**.

use private constructor when we have only static members.

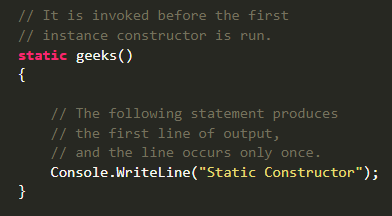
1. **Static Constructor**

A static constructor is used to **initialize** any **static data**, or to perform a particular action that needs to be **performed only once**.

It is called automatically **before** the first instance is created or any static members are referenced.

Static constructors have the following properties:

* A static constructor **doesn't** take access modifiers or have parameters.
* A class or struct can **only have one** static constructor.
* Static constructors **cannot be inherited or overloaded**.
* A static constructor **cannot be called directly** and is **only meant to be called by the common language runtime (CLR).** It is invoked automatically.
* The **user has no control** on when the static constructor is executed in the program.
* If a static constructor throws an exception, the runtime doesn't invoke it a second time, and the type will remain uninitialized for the lifetime of the application domain.
* If you don't provide a static constructor to initialize static fields, all static fields are initialized to their default value as listed in Default values of C# types.



**Destructor**

Destructors in C# are methods inside the class used to **destroy instances of that class** when they are no longer needed.

The Destructor is called implicitly by the .NET Framework’s Garbage collector and therefore **programmer has no control as when to invoke the destructor**.

An instance variable or an object is eligible for destruction when it is no longer reachable.

There **cannot be more than one** destructor in a class.

A Destructor has **no return type** and has exactly the same name as the class name (Including the same case).

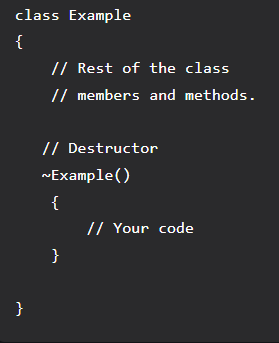
A Destructor **does not accept any parameters and modifiers**.

It **cannot be defined in Structures**. It is only used with classes.

It **cannot be overloaded or inherited**.

It is called when the program exits.

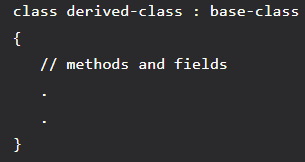
Internally, Destructor called the Finalize method on the base class of object.



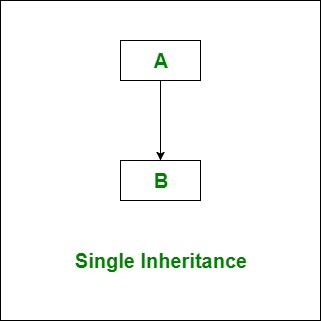
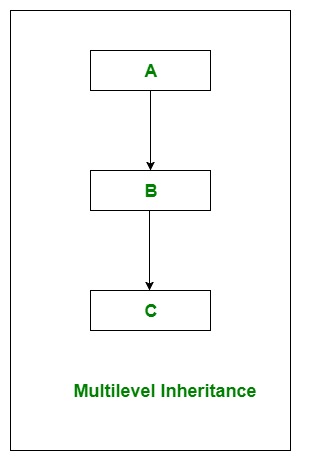
**Inheritance**

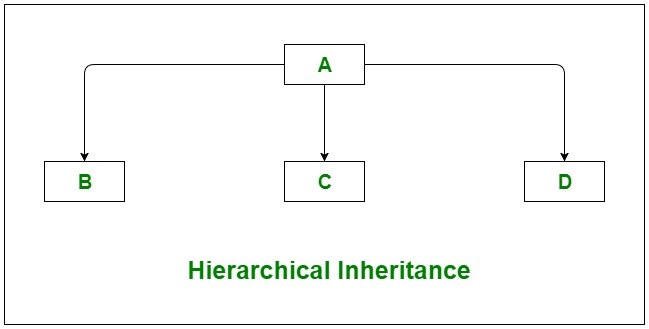
Inheritance is an important pillar of OOP (Object Oriented Programming).

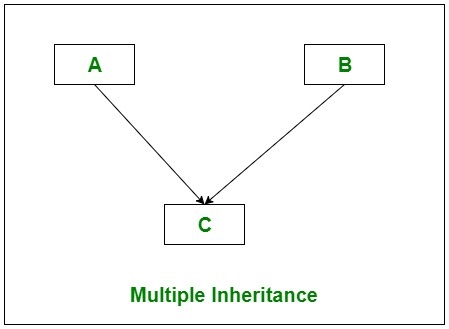
It is the mechanism in C# by which one class is allowed to **inherit the features (fields and methods) of another class**.



Types of Inheritance in C#

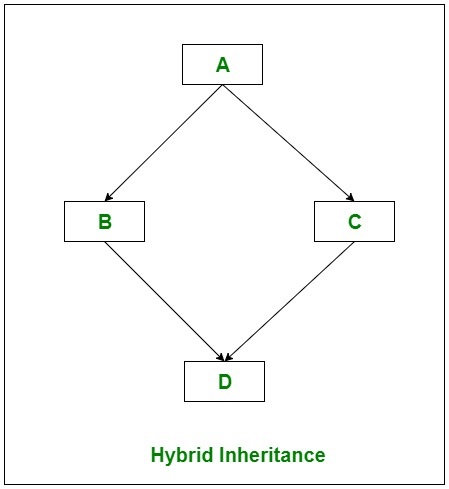




C# **does not support multiple inheritance** with classes. In C#, we can **achieve multiple inheritance only through Interfaces**.

In the image below, Class C is derived from interface A and B.

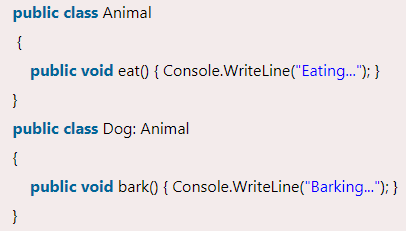
Same is the case with Hybrid Inheritance, we can achieve it only through interfaces.



Important facts about inheritance in C#

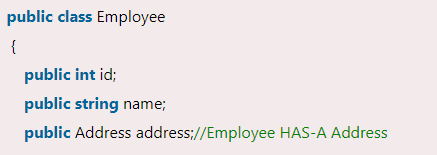
* Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.
* A subclass inherits all the members (fields, methods) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
* A subclass does not inherit the private members of its parent class. However, if the superclass has properties (get and set methods) for accessing its private fields, then a subclass can inherit.

**Inheritance:** IS-A Relationship.



Dog **IS-A** Animal.

**Aggregation:** HAS-A Relationship



Employee **HAS-A** Address

In case of inheritance, when you create object of Derived class, along with default constructor of derived class, default constructor of base class also gets called.

when you create object of Derived class using parameter, along with parameterized constructor of derived class, default constructor of base class also gets called.

In above both scenarios constructor of base class executes first.

**Encapsulation**

Encapsulation is defined as the wrapping up of data under a single unit.

It is the mechanism that **binds together code and the data** it manipulates.

In a different way, encapsulation is a protective shield that prevents the data from being accessed by the code outside this shield.

**Encapsulation can be achieved by**: Declaring all the variables in the class as private and using C# Properties in the class to set and get the values of variables.

**Abstraction**

Abstraction in C# is the process **to hide the internal details** and **showing functionality only**.

**Example:** Consider a real-life scenario of withdrawing money from ATM.

The user only knows that in ATM machine first enter ATM card, then enter the pin code of ATM card, and then enter the amount which he/she wants to withdraw and at last, he/she gets their money.

The user does not know about the inner mechanism of the ATM or the implementation of withdrawing money etc.

The user just simply knows how to operate the ATM machine; this is called abstraction.

Abstraction can be achieved by two ways:

1. Abstract class
2. Interface

First let us known about **Abstract Methods**

* A method which is declared **abstract** and has **no body** is called abstract method.
* It can be **declared inside the abstract class only**.
* Its implementation must be provided by derived classes.

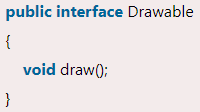


* An abstract method in C# **is internally a virtual method** so it can be overridden by the derived class.
* You **can't** use **static** and **virtual** modifiers in abstract method declaration.

**Abstract Class:**

* In C#, abstract class is a class which is declared abstract.
* It **can have abstract** and **non-abstract** methods.
* It cannot be instantiated.
* Its implementation must be provided by derived classes. Here, derived class is forced to provide the implementation of all the abstract methods.
* An abstract class **cannot be inherited by structures**.
* It **can contain** constructors or destructors.
* It can implement functions with non-Abstract methods.
* It cannot support multiple inheritances.
* It can’t be static.

**Interface:**

* Interface in C# is a blueprint of a class.
* It is like abstract class because all the methods which are declared inside the interface are abstract methods.
* It **cannot have method body** **and cannot be instantiated**.
* It is used to **achieve multiple inheritance** which can't be achieved by class.
* It is used to achieve **fully** **abstraction** because it cannot have method body.
* Its implementation must be provided by class or struct.
* The class or struct which implements the interface, must provide the implementation of all the methods declared inside the interface.
* 
* **Note**: Interface methods are public and abstract by default. You cannot explicitly use public and abstract keywords for an interface method.
* 