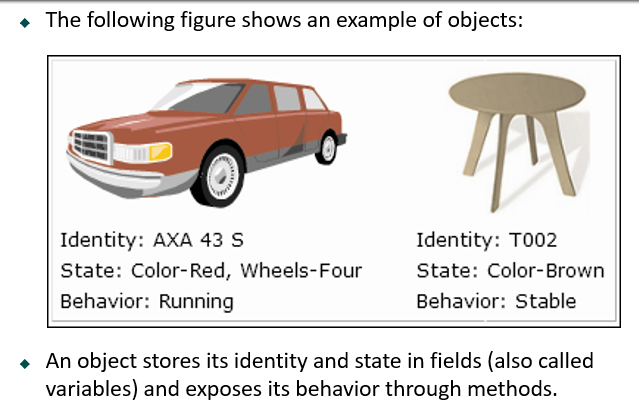
**CLASSES AND OBJECTS IN C#**

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



* The concept of classes in the real world can be extended to the programming world, similar to the concept of objects.
* In object-oriented programming languages like C#, a class is a template or blueprint which defines the state and behavior of all objects belonging to that class.
* A class comprises fields, properties, methods, and so on, collectively called data members of the class. In C#, the class declaration starts with the class keyword followed by the name of the class.

**CONSTRUCTORS IN C#**

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

**Default Constructor**

A constructor which has not defined any parameters or we can say without any parameters is called default constructor. It initializes the same value of every instance of class.

**Parameterized Constructor**

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

**STATIC CONSTRUCTOR IN C#**

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

**KEY POINTS OF STATIC CONSTRUCTOR**

* Only one static constructor can be created in the class.
* It is called automatically before the first instance of the class created.
* We cannot call static constructor directly. CLR (COMMON LANGUAGE RUNTIME)

**PRIVATE CONSTRUCTOR IN C#**

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

* One use of a private constructor is when we have only static members.
* Once we provide a constructor that is either private or public or any, the compiler will not add the parameter-less public constructor to the class.
* In the presence of parameterless private constructor you cannot create a default constructor.
* We cannot inherit the class in which we have a private constructor.
* We can have parameters in private constructor. YES
* **COPY CONSTRUCTOR IN C#**
* The constructor which creates an object by copying variables from another object is called a copy constructor. The purpose of a copy constructor is to initialize a new instance to the values of an existing instance.
* In c#, Copy Constructor is a parameterized constructor which contains a parameter of same class type. The copy constructor in C# is useful whenever we want to initialize a new instance to the values of an existing instance.
* In simple words, we can say copy constructor is a constructor which copies a data of one object into another object.

**CONSTRUCTOR IN INHERITANCE**

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

**STATIC CLASS IN C#**

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

**DESTRUCTORS IN C# PROGRAMMING**

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

Following are the features of destructors:

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

**INHERITANCE IN C# PROGRAMMING**

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

**PERMANENT EMPLOYEES**

**Int EmpId;**

**String EmpName;**

**Int EmpAge;**

**Int EmpContactNo;**

**Int permanentSalary;**

**Int permanentHours;**

**VISITING EMPLOYEES**

**Int EmpId;**

**String EmpName;**

**Int EmpAge;**

**Int EmpContactNo;**

**Int visitingSalary;**

**Int visitingHours;**

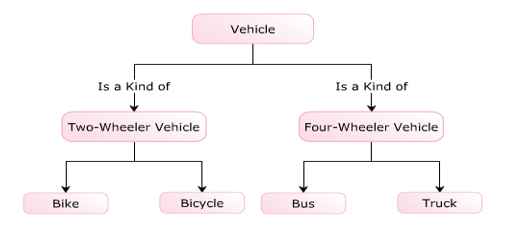
**EMPLOYEE**

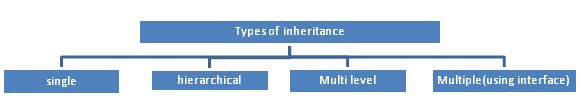
**Int EmpId;**

**String EmpName;**

**Int EmpAge;**

**Int EmpContactNo;**



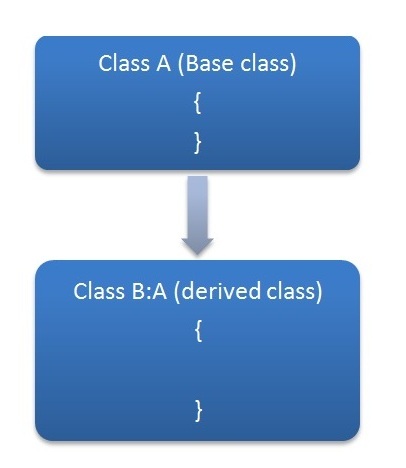
* The following are the types of inheritance in C#.

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

* **BASE CLASS** - is the class from which features are to be inherited into another class.
* **DERIVED CLASS** - it is the class in which the base class features are inherited.

**SINGLE INHERITANCE**

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



**WHAT ARE PROPERTIES IN C#**

* PROPERTIES ALLOW YOU TO CONTROL THE ACCESSIBILITY OF A CLASS VARIABLES, AND ARE THE RECOMMENDED WAY TO ACCESS VARIABLES FROM THE OUTSIDE IN C#.
* A PROPERTY IS MUCH LIKE A COMBINATION OF A VARIABLE AND A METHOD - IT CAN'T TAKE ANY PARAMETERS, BUT YOU ARE ABLE TO PROCESS THE VALUE BEFORE IT'S ASSIGNED TO OUR RETURNED.
* PROPERTIES ARE LIKE DATA FIELDS (VARIABLES), BUT HAVE LOGIC BEHIND THEM.
* FROM THE OUTSIDE, THEY LOOK LIKE ANY OTHER MEMBER VARIABLE.
  + BUT THEY ACT LIKE A MEMBER FUNCTION.
* DEFINED LIKE A FIELD, WITH “GET” AND “SET” ACCESSORS CODE ADDED.
* PROPERTIES ARE ALSO USED FOR ENCAPSULATION.

**TYPES OF PROPERTIES IN C#**

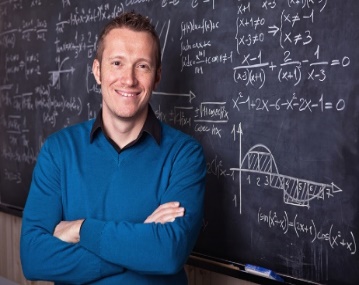
* READ / WRITE PROPERTIES
* READ ONLY PROPERTIES
* WRITE ONLY PROPERTIES
* AUTO IMPLEMENTED PROPERTIES

**STATIC PROPERTY IN C#**

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

**POLYMORPHISM IN C#**

* Polymorphism is one of the four pillars of Object Oriented Programming.
* Polymorphism in C# is a concept by which we can perform a single action by different ways.
* Polymorphism is derived from 2 Greek words: POLY and MORPHS.
* The word "poly" means many and "morphs" means forms.
* So polymorphism means many forms.

**REAL WORLD EXAMPLE**

FATHER

SON

HUSBAND

TEACHER

**THERE ARE TWO TYPES OF POLYMORPHISM**

1. STATIC POLYMORPHISM (COMPILE TIME POLYMORPHISM)
2. DYNAMIC POLYMORPHISM (RUN TIME POLYMORPHISM)

**STATIC POLYMORPHISM (COMPILE TIME POLYMORPHISM) IN C#**

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

C# provides two techniques to implement static polymorphism. They are −

* METHOD OR FUNCTION OVERLOADING
* OPERATOR OVERLOADING

**METHOD OR FUNCTION OVERLOADING**

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

**DYNAMIC OR RUNTIME POLYMORPHISM IN C#**

* RUN TIME POLYMORPISM IS ACHIEVED BY METHOD OVERRIDING.
* METHOD OVERRIDING ALLOWS US TO HAVE VIRTUAL AND ABSTRACT METHODS IN THE BASE USING DERIVED CLASSES WITH THE SAME NAME AND THE SAME PARAMETER.

**C# METHOD OVERRIDING**

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

**METHOD HIDING IN C# PROGRAMMING**

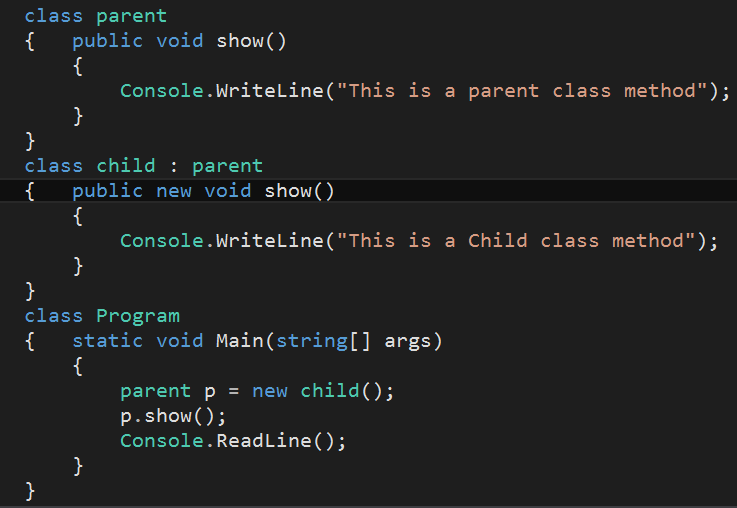
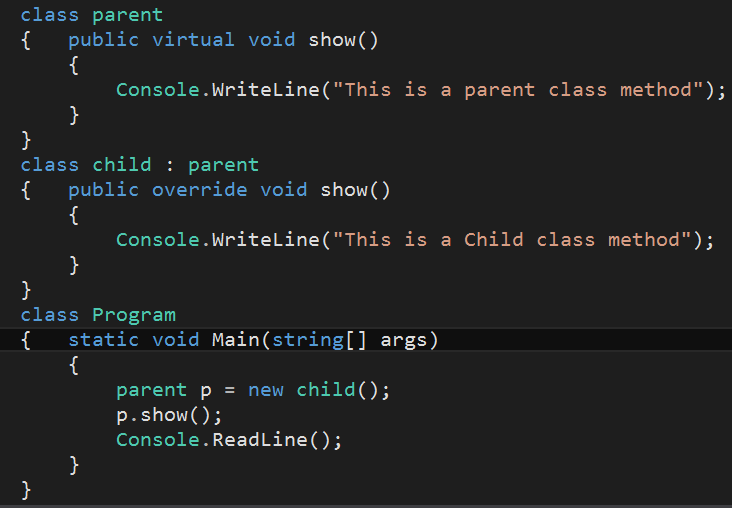
* Method hiding occurs in inheritance relationship when base class and derived class both have a method with same name and same signature.
* When we create the object of derived class it will hide the base class method and will call its own method and this is called method hiding or **name hiding** in C# inheritance.
* We use “new” keyword in derived function name to show that implementation of the function in derived class is intentional and derived class no longer want to use base class method.

**NOTE:** If we do not use “new” keyword then compiler will raise only warning, but, program will work fine.

**DIFFERENT WAYS TO CALL A HIDDEN BASE CLASS MEMBER FROM DERIVED CLASS**

1. USE **BASE** KEYWORD
2. CAST CHILD TYPE TO PARENT TYPE AND INVOKE THE HIDDEN MEMBER.
3. ParentClass PC = new ChildClass();

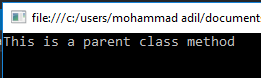
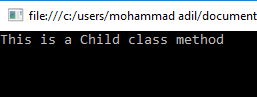
PC.hiddenMethod();

* PARENT CLASS CAN HAVE THE REFERENCE OF ITS CHILD CLASS.
* WHEN WE CREATE THE OBJECT OF CHILD CLASS, PARENT CLASS OBJECT IS ALSO CREATED.
* **DIFFERENCE BETWEEN METHOD HIDING AND METHOD OVERRIDING IN C#**
* 

IN **METHOD OVERRIDING**, A BASE CLASS REFERENCE VARIABLE POINTING TO A CHILD CLASS OBJECT, WILL INVOKE THE OVERRIDDEN METHOD Of THE CHILD CLASS.

**METHOD HIDING OUTPUT**

**METHOD HIDING**



**METHOD OVERRIDING OUTPUT**

IN **METHOD HIDING**, A BASE CLASS REFERENCE VARIABLE POINTING TO A CHILD CLASS OBJECT, WILL INVOKE THE HIDDEN METHOD OF THE BASE CLASS.

**METHOD OVERRIDING**

**SEALED CLASS IN C# PROGRAMMING**

A sealed class is a class that prevents inheritance.

The features of a sealed class are as follows:

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

**Purpose of Sealed Classes**

* Consider a class named **SystemInformation** that consists of critical methods that affect the working of the operating system.
* You might not want any third party to inherit the class **SystemInformation** and override its methods, thus, causing security and copyright issues.
* Here, you can declare the **SystemInformation** class as sealed to prevent any change in its variables and methods.

**SEALED METHODS IN C#**

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

**STEPS TO REMEMBER FOR SEALED METHODS:**

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

**DELEGATES IN C# PROGRAMMING**

* **DELEGATE MEANING FROM GOOGLE:** A PERSON SENT OR AUTHORIZED TO REPRESENT OTHERS, IN PARTICULAR AN ELECTED REPRESENTATIVE SENT TO A CONFERENCE.
* DELEGATE IS A TYPE WHICH HOLDS A METHOD’S REFERENCE IN AN OBJECT.
* IT IS ALSO CALLED FUNCTION POINTER.
* DELEGATE IS OF REFERENCE TYPE.
* DELEGATE SIGNATURE SHOULD BE AS SAME AS THE METHOD SIGNATURE REFERENCING BY A DELEGATE.
* DELEGATE CAN POINT TO A PARAMETERIZED METHOD OR NON-PARAMTERIZED METHOD.
* DELAGATE HAS NO IMPLEMENTATION MEANS NO BODY WITH { }.
* WE CAN USE **INVOKE()** METHOD WITH DELEGATES.
* DELEGATES ARE USED TO ENCAPSULATE METHODS.
* IN THE .NET FRAMEWORK, A DELEGATE POINTS TO ONE OR MORE METHODS. ONCE YOU INSTANTIATE THE DELEGATE, THE CORRESPONDING METHODS INVOKE.
* DELEGATES ARE OBJECTS THAT CONTAIN REFERENCES TO METHODS THAT NEED TO BE INVOKED INSTEAD OF CONTAINING THE ACTUAL METHOD NAMES
* USING DELEGATES, YOU CAN CALL ANY METHOD, WHICH IS IDENTIFIED ONLY AT RUN-TIME.
* A DELEGATE IS LIKE HAVING A GENERAL METHOD NAME THAT POINTS TO VARIOUS METHODS AT DIFFERENT TIMES AND INVOKES THE REQUIRED METHOD AT RUN-TIME.
* IN C#, INVOKING A DELEGATE WILL EXECUTE THE REFERENCED METHOD AT RUN-TIME.
* TO ASSOCIATE A DELEGATE WITH A PARTICULAR METHOD, THE METHOD MUST HAVE THE SAME RETURN TYPE AND PARAMETER TYPE AS THAT OF THE DELEGATE.