9. O.O.P Approach in C#.Net

The object oriented programming (OOP) is a programming model where Programs are organized around object and data rather than action and logic .OOP allow decomposition of a problem into a number of entities called Object and then builds data and objects around these objects.

Class:

A class is the core of any modern Object Oriented Programming language such as C#. In OOP languages it is must to create a class for representing data.

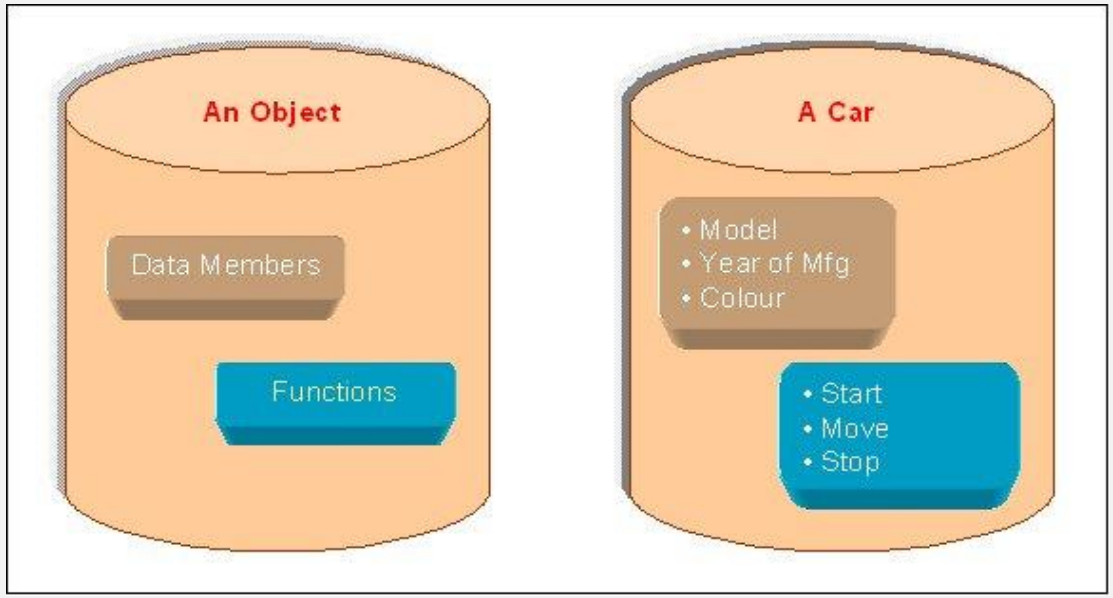
Class is a blueprint of an object that contains variables for storing data and functions to performing operations on these data.

Class will not occupy any memory space and hence it is only logical representation of data.

```
class <Identifier>
{
        //class body-class members
}
```

Object:

Objects are the basic run-time entities in an object oriented system. They may represent a person, place or any item that the program has to handle. "Object is a Software bundle of related variable and methods. “Object is an instance of a class”.



About Class and Object:

- Class will not occupy any memory space.
- Hence to work with the data represented by the class you must create a variable for the class, which is called as an object.
- When an object is created by using the keyword new, then memory will be allocated for the class in heap memory area. which is called as an instance.
- When an object is created without the keyword new, then memory will not be allocated in heap.

```
class Employee
{
}
```

Syntax to create an object of class Employee:-

```
Employee objEmp = new Employee();
```

All the programming languages supporting object oriented Programming will be supporting these four main concepts:

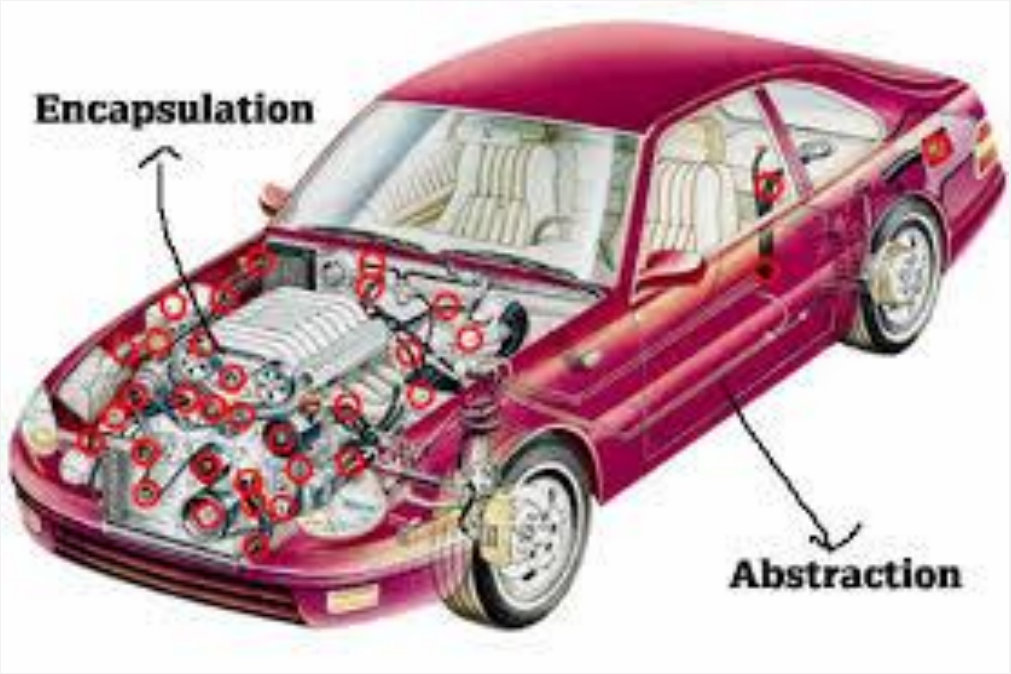
- ❑ Abstraction
- ❑ Encapsulation
- ❑ Inheritance
- ❑ Polymorphism

9.1 Abstraction:

Abstraction is just opposite of Encapsulation. Abstraction is mechanism to show only relevant data to the user. Consider the same mobile example again. Whenever you buy a mobile phone, you see their different types of functionalities as camera, mp3 player, calling function, recording function, multimedia etc. It is abstraction, because you are seeing only relevant information instead of their internal engineering.

9.2 Encapsulation:

Encapsulation is the process of hiding irrelevant data from the user. To understand encapsulation, consider an example of mobile phone. Whenever you buy a mobile, you don't see how circuit board works. You are also not interested to know how digital signal converts into analog signal and vice versa. These are the irrelevant information for the mobile user, that's why it is encapsulated inside a cabinet.



Abstraction	Encapsulation
Abstraction solves the problem in the design level.	Encapsulation solves the problem in the implementation level.
Abstraction is used for giving only relevant data.	Encapsulation is hiding the code and data into a single unit to protect the data from outer world.
Abstraction is set focus on the object instead of how it does it.	Encapsulation means hiding the internal details or mechanics of how an object does something.
Abstraction is outer layout in terms of design. For Example: - Outer Look of a iPhone, like it has a display screen.	Encapsulation is inner layout in terms of implementation. For Example: - Inner Implementation detail of a iPhone, how DisplayScreen are connect with each other using circuits

9.3 Inheritance:

When a class acquire the property of another class is known as inheritance. To create new Class using existing class and add its own functionality in new class.

For example, A Child acquire property of Parents.

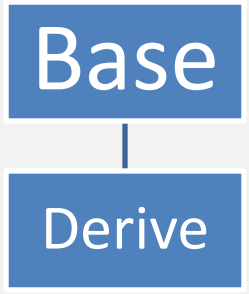
```
public class MobilePhone
{
    public void Calling()
    {
        //logic
    }

    public void SendSMS()
    {
        //logic
    }
}
```

```
public class Nokia1400 : MobilePhone
    {
            //body of NOKIA1400
    }
```

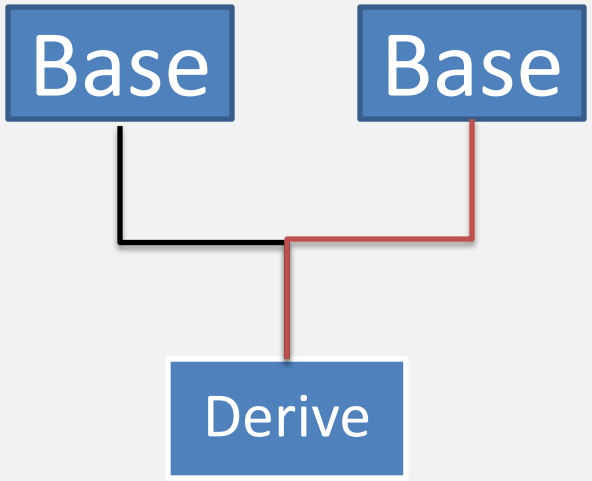
Demo: How to implement Inheritance.

Types of Inheritance:

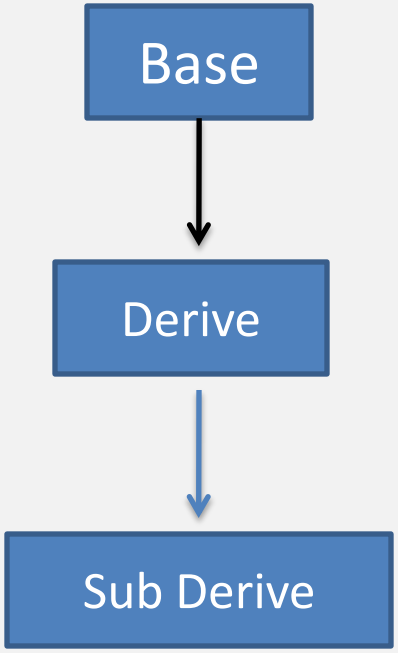
❑Single Inheritance:



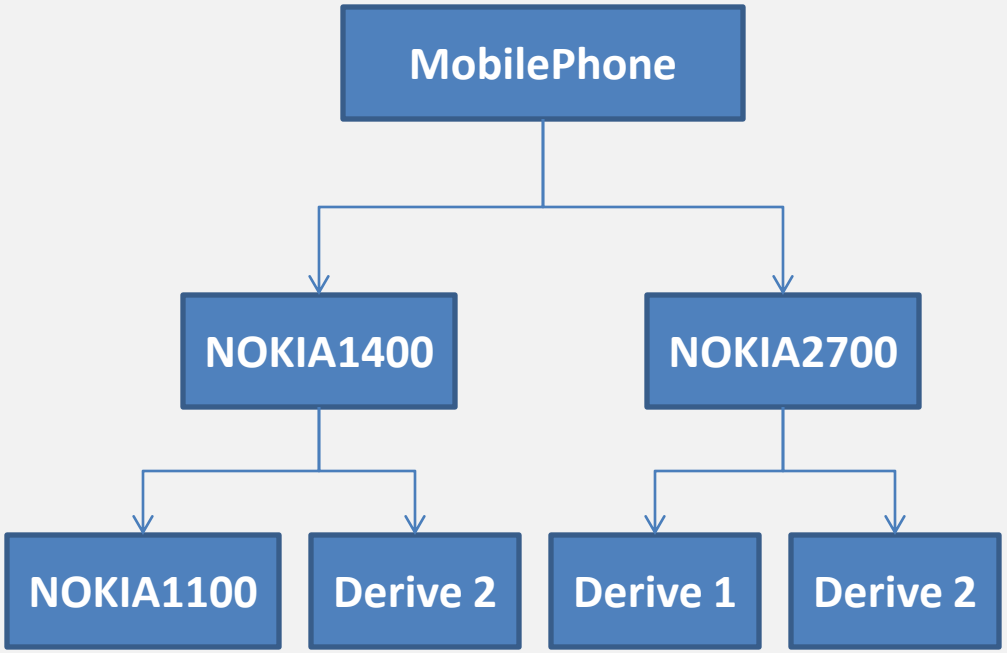
❑Multiple Inheritance:



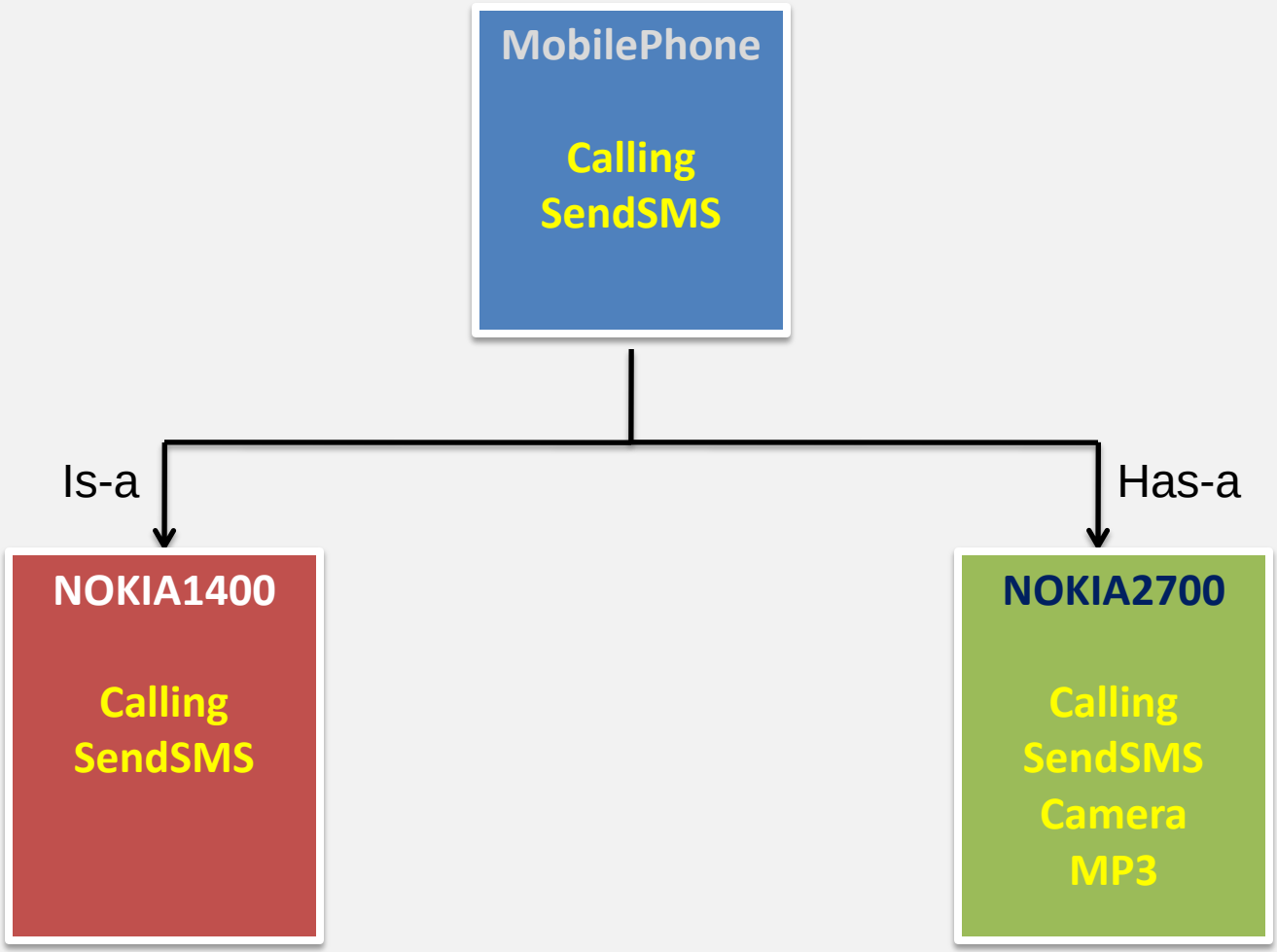
❑Multi Level:



❑Hybrid/Hierarchical:



❑ “Is-A and Has-A relationship between classes”.



9.4 Polymorphism:

Polymorphism means **one name many forms**.

One function behaves different forms.

In other words, "Many forms of a single object is called Polymorphism."

Real World Example of Polymorphism:

Person behaves SON in house at the same time that person behaves EMPLOYEE in office.

C# Example of Polymorphism:

- ❑ Function Overloading
- ❑ Virtual Function/Function Overriding
- ❑ Shadowing

9.4.1 Function Overloading:

- ❑ Multiple function with the same name but with different signatures.
- ❑ Return type is not part of signature hence it not affects overloading.

Demo: To Add two numbers and to add two string using function overloading.

9.4.2 Virtual Function/Function Overriding :

- ❑ If a function or a property in the base class is declared as virtual it can be overridden in any derived classes.
- ❑ Virtual function is use to change functionality of base class function in its derived class.

Keywords: **virtual [Base Version]**
override [Derived Version]

Demo: To Add two numbers and to add two string using virtual function.

9.4.3 Shadowing:

- This concept is used for suppress version of base class using **new** keyword.
- new keyword is not mandatory it is only to denote new version of a function.
- If new keyword used it will show a warning.

Demo: To Add two numbers and to add two string using function Shadowing.

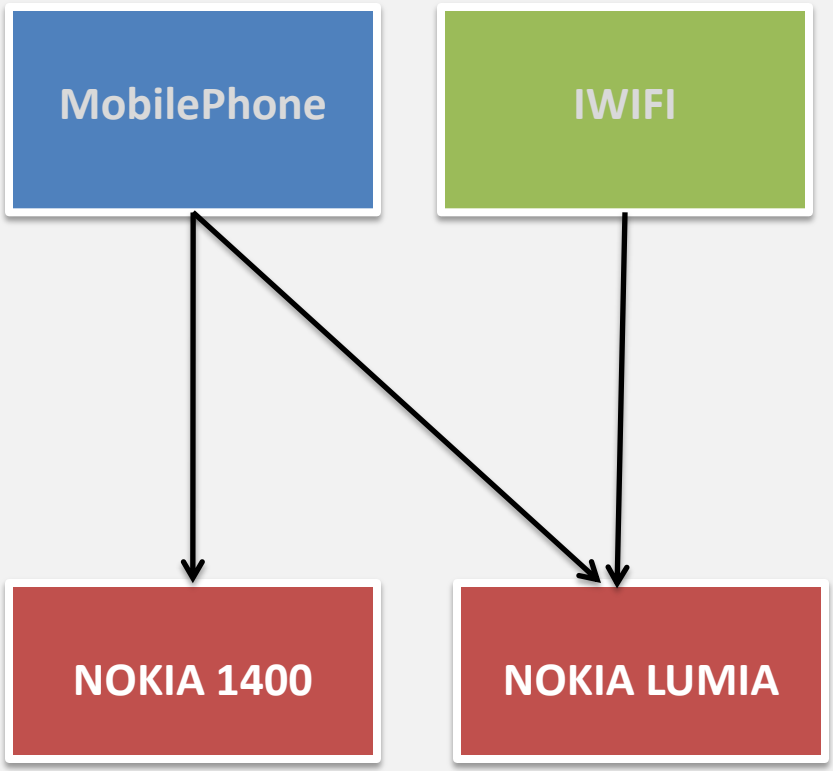
Comparison between Function Overloading, Overriding and Shadowing.

Term	Overloading	Overriding	Shadowing
Scope	both versions must Be in the same class.	Version must be separated in base class and derived class.	Version must be separated in base class and derived class.
Proto	Same name but with different signatures.	Name and signature must be same of both versions.	Name should be same but signatures can be different or signatures can also be same.
Keyword	No Keyword Required	Keywords :-Virtual and override	No Keyword Required
Required	Not mandatory to overload all function.	Not mandatory.	No Shadowing mandatory for all functions
Return Type	Return Type not matters	Return Type matters	Return Type not matters

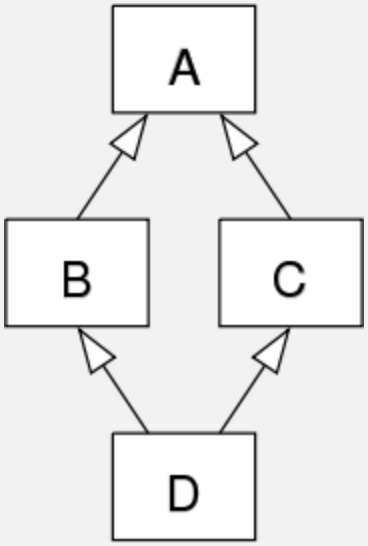
9.5 Interfaces

- An Interface is simply a collection of function prototypes. An Interface defines a contract. A class that implements an interface must adhere to that contract. An Interface Can inherit from multiple base interfaces and a class may implement multiple Interface . Interfaces are reference types.
- An interface is like an abstract class, except it does not contain any logic.
- it doesn’t have any constructor and its has only declarations of functions.
- by default all members are public inside interface hence no need to declare function with public keyword explicitly.
- interfaces can only contains declaration of functions and prop.
- it doesn't contain any variable or constant.
- All functionalities needs to be implemented in derived class which is declared Inside interface.
- There are two kind of implementation-implicit and explicit.

Demo: How to implement interface.


MobilePhone
IWIFI
NOKIA 1400
NOKIA LUMIA

Diamond problem with multiple inheritance



Comparison between abstract class and Interface.

Term	Abstract class	Interface
Content	It can contain declaration and implementation too.	It can not contain implementation. It only Contain declaration.
Default modifier	All member of abstract class are private by default.	All member of Interface are by default public.
restriction	It can contain const,field members, prop. and functions.	Only contain prop. and methods .

9.10 Constructors:

When you use the *new* operator it may be necessary to initialize field values to custom values. It is not necessary to define a C# constructor a default constructor is called if there are no programmer defined constructors. The default constructor does not do anything except creates your object.

- ❑ Constructor can be Inherited.
- ❑ Constructor can be Overload.
- ❑ Constructor does not have any return type.

Demo: Use of Constructor in student class for library Management.

9.11 Properties:

A program needs a way to access and manipulate **private** variables in a class. To do this we would rely on **get** and **set** accessors in C# **properties**. C# **properties** are blocks that give the client the ability to *set* and *get* private variables in a class.

- ❑ Properties can be ReadOnly or ReadWrite.
- ❑Properties can be Parameterized or Parameterless.

```
public class <Class Name>
  {
     private <type> <variable>;

     public  <return type> <property name>[ param]
          {
             get
                {
                 return <private variable>;
                 }
              set
                 {
                <private variable> = value;
                 }
          }
  }
```

- get/set does not have any access modifier they are public by default.
- Properties should be public.
- Property can be of any type

Demo: Create property allow to change contact no of student out side student class.

9.12 Class Access Modifiers:

Scope Modifiers

Sr.No	Modifiers	Scope
1.	Public	Entire assembly/project and out side project as well.
2.	internal (by default)	Within assembly/project only.

Other Modifiers

Sr.No	Modifiers	Description
1.	abstract	Must inherit. can not instantiate.
2.	sealed	Can not inherit must be instantiate.
3.	static	Not specified with any instance. Can not inherit.
4.	partial	Splits class into multiple files for security.

9.13 Scope Modifiers for Function/properties:

Sr.No	Modifiers	Scope/function
1.	public	Entire assembly/project and out side project as well.
2.	internal	Within assembly/project only.
3.	protected	Up to derived class only.
4.	Private (by default)	Within class only.

Minutes of Chapter

- Introduction to O.O.P in C#.Net.
- Features of O.O.P in C#.Net.
- Interfaces.
- Constructors.
- Properties.
- Class Access modifiers.
- Access modifiers of class members.
- Assemblies in C#.Net.