

Object Oriented Programming System (OOPs)

Table of Contents

Introduction.....	2
OOPs	3
Class and Object.....	5
Abstraction.....	7
Encapsulation	17
Inheritance.....	21
Polymorphism	30
Reference link	36

Create By:

Afjal Hossain

[afjal.swe@gmail.com]

Software Engineer at Faztrack Technology, LLC

Object Oriented Programming System (OOPs)

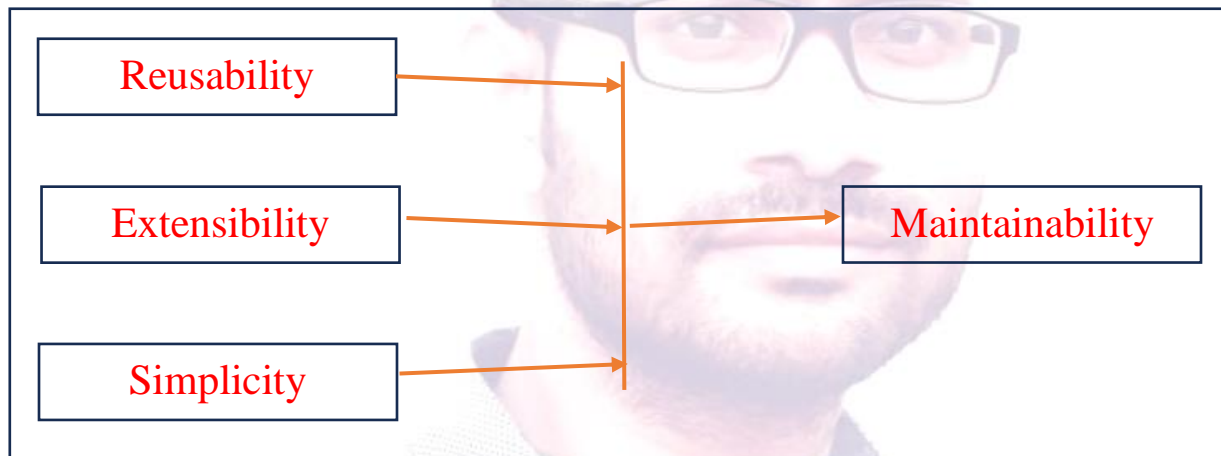
Introduction

In C#. Object-Oriented Programming, commonly known as OOPs, is a technique, not a technology. It means it doesn't provide any syntaxes or APIs; instead, it provides suggestions to design and develop objects in programming languages.

Object Oriented Programming system(OOPs) concept comes as a solution to modular programming problems.

Now, we look at the problems of Modular programming:

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



Reusability: In Modular Programming, we must write the same code or logic at multiple places, increasing code duplication. Later, if we want to change the logic, we must change it everywhere.

Extensibility: It is not possible in modular 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 an entirely new function and then change the function as per your requirement.

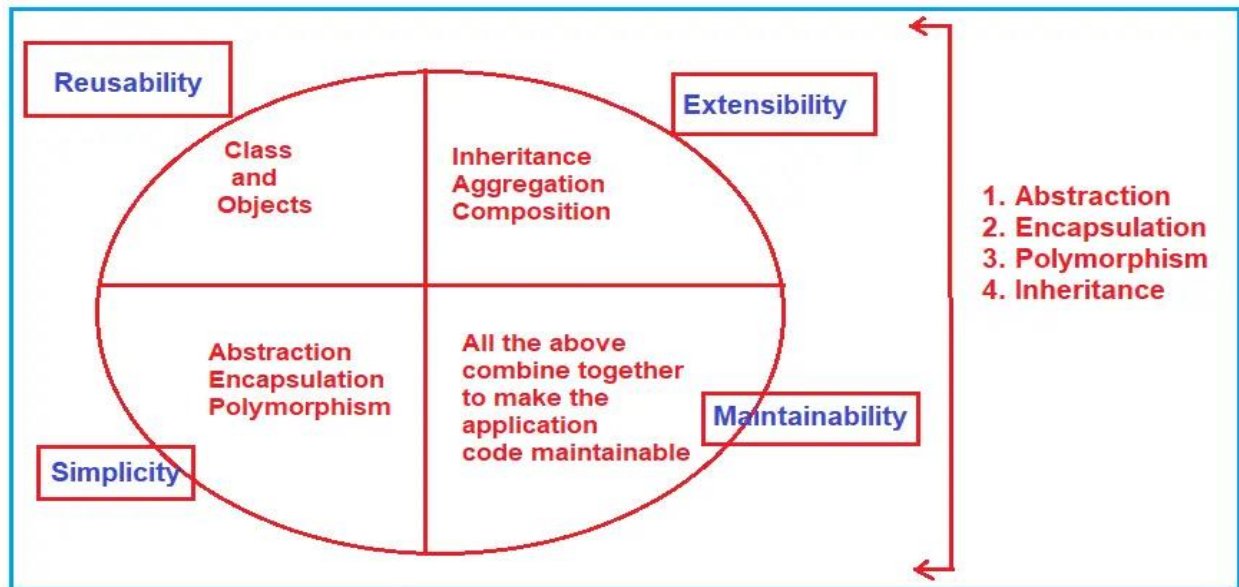
Simplicity: As extensibility and reusability are impossible in Modular Programming, we usually end up with many functions and scattered code.

Maintainability: As we don't have Reusability, Extensibility, and Simplicity in modular Programming, it is very difficult to manage and maintain the application code.

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

Object Oriented Programming System (OOPs)

Let's understand with the following image...



Reusability:

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

Extensibility:

Suppose you have a function and want to extend it with some new features that were impossible with functional programming. You have to create an entirely new function and then change the whole function to whatever you want. OOPs, this problem is addressed using 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 modular programming, we end up with lots of functions and scattered code, and from anywhere we can access the functions, security is less. In OOPs, this problem is addressed using Abstraction, Encapsulation, and Polymorphism concepts.

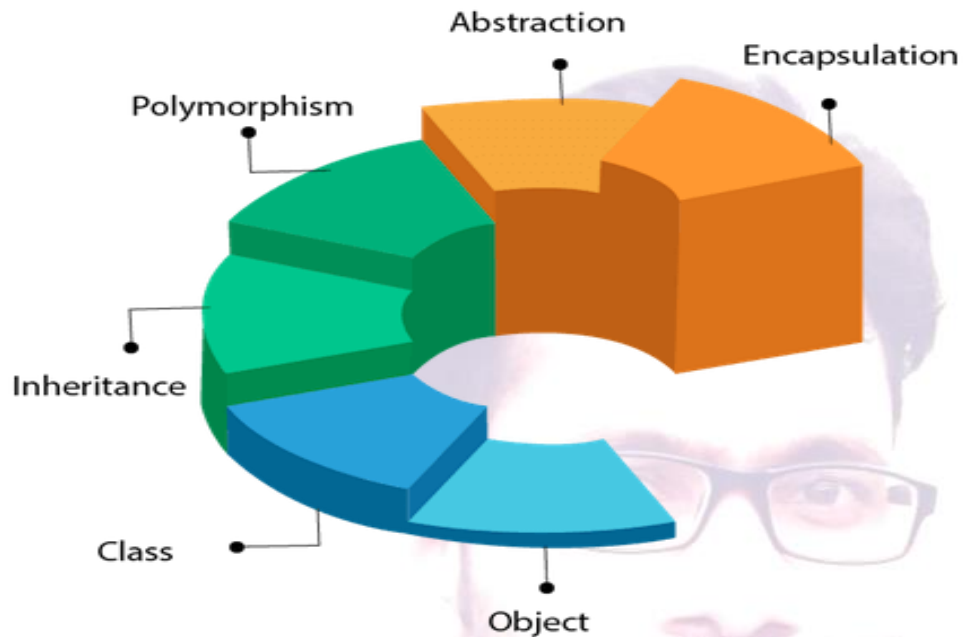
Maintainability:

As OOPs address Reusability, Extensibility, and Simplicity, we have good, maintainable, and clean code, increasing the application's maintainability.

OOPs

Object Oriented Programming System (OOPs)

OOPs (Object-Oriented Programming System)



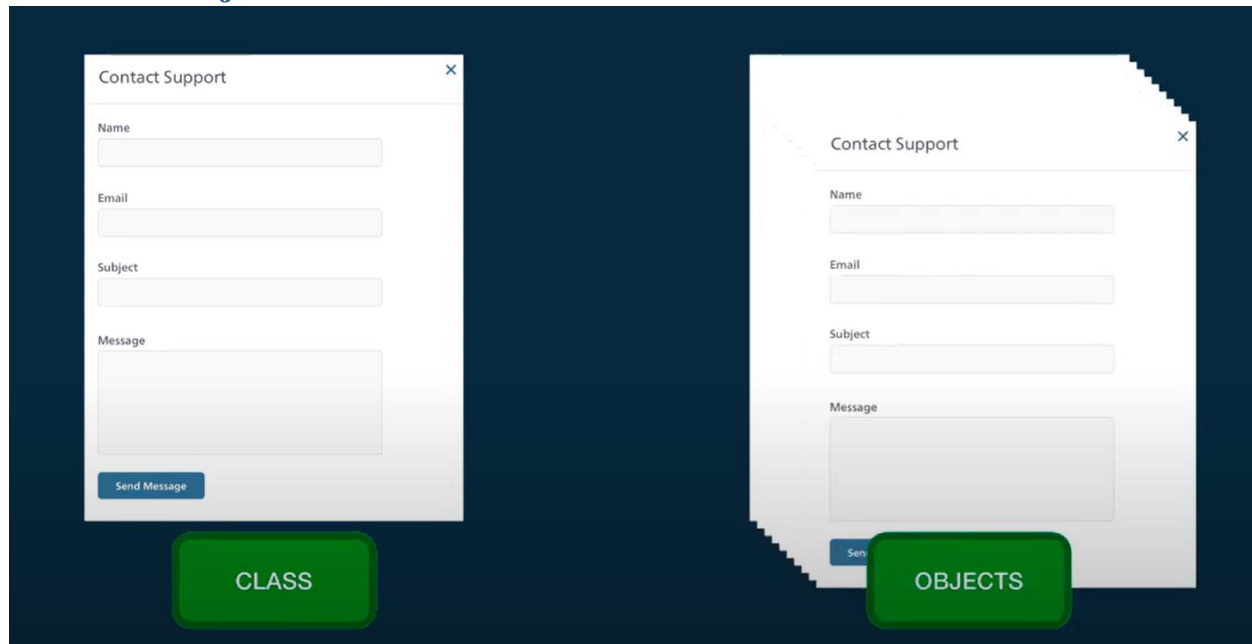
“OOPs stands for Object Oriented Programming System. It is a methodology or a kind of design philosophy that helps to write the program where we specify the code in the form of classes and objects.”

OOPs concept involves:

- Class
- Object
- Abstraction
- Polymorphism
- Inheritance
- Encapsulation

Object Oriented Programming System (OOPs)

Class and Object



“Class is just a Factory to Create Multiple Object without Code Duplication and Object is an instance of Class.”

Class: A class is a collection of objects and represents a description of objects that share same attributes and actions or behaviors.

Here is the syntax and declaration example of Class:

```
public class Student {  
    //your code goes here...  
}
```

There are 4 types of classes that we can use in C#:

1. **Partial class** – Allows its members to be divided or shared with multiple .cs files. It is denoted by the keyword Partial.
2. **Sealed class** – It is a class that cannot be inherited. To access the members of a sealed class, we need to create the object of the class. It is denoted by the keyword Sealed.
3. **Abstract class** – It is a class whose object cannot be instantiated. The class can only be inherited. It should contain at least one method. It is denoted by the keyword abstract.
4. **Static class** – It is a class that does not allow inheritance. The members of the class are also static. It is denoted by the keyword static. This keyword tells the compiler to check for any accidental instances of the static class.

Object Oriented Programming System (OOPs)

Object: Object is an instance of a class. It is a logical as well as a physical entity. Object has three features: State, Behavior & Identity.

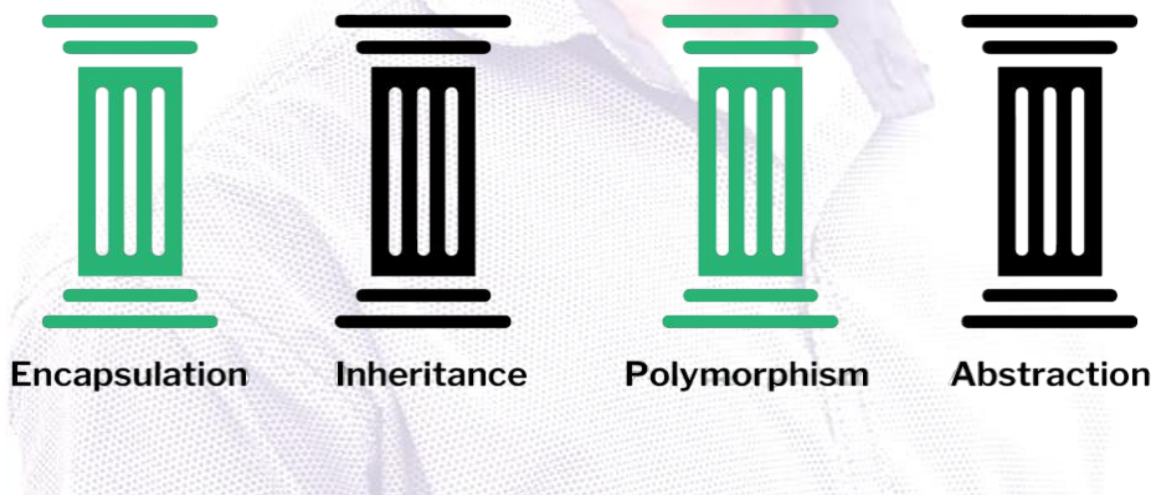
Consider a Student class here...

Student
- _name: string - _age: int
+ Student (): void + DoLearn (): bool <<Property>> + Name (): string + Age (): int

Student **objStudent** = new Student ();

According to the above sample we can say that the object called **objStudent** object has been created from the student class.

Pillars of OOPs

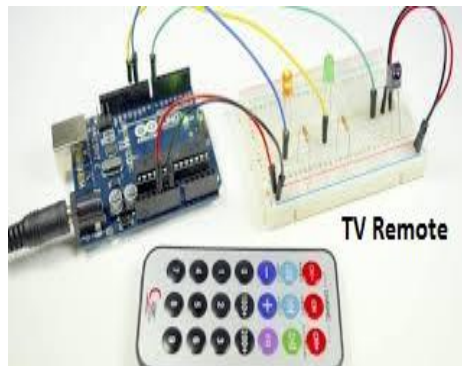


In this part, we will be focusing on four principal of OOPs concepts:

Object Oriented Programming System (OOPs)

Abstraction

Abstraction is a process of hiding the implementation details and showing only functionality to the user.



“TV or Car remote is the example for Abstraction. TV or Car remote is assembled from the collection of circuits, but they don't show to the user all circuits behind the remote, they only provide remote to the user to use it. When the user presses the key on remote the channel gets changed. They provide only necessary information to the user. This concept is called Data Abstraction.”

Abstraction can be achieved by two ways:

1. Abstract class
2. Interface

Abstract class and interface both can have abstract methods which are necessary for abstraction.

The following are some of the key points –

- You cannot create an instance of an abstract class.
- You cannot declare an abstract method outside an abstract class.
- When a class is declared sealed, it cannot be inherited, abstract classes cannot be declared sealed.

Example:

```
using system;
//Abstract class
public abstract class Shape
{
    public abstract void Draw(); //Abstract method
    public void DisplayArea(float area){
        Console.WriteLine("Area: "+ area);
    }
}
//Concrete class
public class Circle : Shape{
    public override void Draw(){
        Console.WriteLine("Drawing a circle");
    }
}
```

Object Oriented Programming System (OOPs)

```
    }  
}  
//Concrete class  
public class Rectangle : Shape{  
    public override void Draw(){  
        Console.WriteLine("Drawing a rectangle");  
    }  
}  
class Program{  
    static void Main(string[] args){  
        Shape circle = new Circle();  
        circle.Draw(); // Output: Drawing a circle  
        circle.DisplayArea(50.2f); // Output: Area: 50.2  
  
        Shape rectangle = new Rectangle();  
        rectangle.Draw(); //Output: Drawing a rectangle  
        rectangle.DisplayArea(25.5f) // Output: Area: 25.5  
    }  
}
```

Code Explanation:

In this example, we have an abstract class called Shape, which defines an abstract method, **Draw()**, and a non-abstract method **DisplayArea()**, that displays the area. The Draw() method is left abstract because each Shape will have its own drawing implementation.

The concrete classes Circle and Rectangle inherited from the Shape class and provide their own implementations of the Draw() method. They also inherit the DisplayArea() method.

In the **Main()** method, we create instances of Circle and Rectangle and call their Draw() and DisplayArea() methods. Since the abstract class Shape provides a common interface, we can treat the objects of both classes as Shape objects, allowing for abstraction.

By using abstract classes and interfaces, we can achieve abstraction in C# by focusing on the essential features and behavior of objects while hiding the implementation details.

Interface:

In C#, Interface is a **blueprint of a class**. The interface looks like a class but has no implementation. It contains declarations of events, indexers, methods and/or properties.

The reason interfaces only provide declarations is because they are inherited by structs and classes, that must provide an implementation for each interface member declared.

The implementation of the interface's members will be given by class who implements the interface implicitly or explicitly.

- Interfaces specify what a class must do and not how.

Object Oriented Programming System (OOPs)

- Interfaces can't have private members.
- By default, all the members of Interface are public and abstract.
- The interface will always be defined with the help of keyword '*interface*'.
- Interface cannot contain fields because they represent a particular implementation of data.
- *Multiple inheritance* is possible with the help of Interfaces but not with classes.

Syntax for Interface Declaration:

The following code demonstrates the syntax for interface declaration in C#:

```
interface IExampleInterface{  
    void Method1();  
    int Method2(string str);  
    string Property1 {get; set;}  
    event EventHandler MyEvent;  
}
```

In the above example, we have declared an interface named **IExampleInterface**, which contains four members – two methods, one property, and one event. Note that interfaces cannot have access modifiers because all members of an interface are public by default.

Advantages of Interface in C#

Some of the key advantages of an Interface are:

- **Code Reusability:** By using interfaces, we can create reusable code that multiple classes can implement. That reduces code duplication and makes our code more maintainable.
- **Loose Coupling:** Interfaces allow us to achieve loose coupling between different components of our application. It means that changes made to one component will not affect the other components that use it.
- **Polymorphism:** Interfaces allow us to achieve polymorphism in our code. It means we can write code that works with objects of different classes that implement the same interface.
- **Separation of Concerns:** Interfaces allow developers to separate the definition of a contract from the implementation details. It allows for a clean separation of concerns and promotes better code maintainability and modularity.
- **Testability:** Interfaces make it easier to test code by allowing developers to create mock objects that implement the same interface as the actual objects being tested. That helps to isolate and test individual components of a system.

Example 1:

```
// C# program to demonstrate working of  
// interface  
using System;
```

Object Oriented Programming System (OOPs)

```
// A simple interface
interface inter1{
    // method having only declaration
    // not definition
    void display ();
}
// A class that implements interface.
class testClass: inter1{
    // providing the body part of function
    public void display(){
        Console.WriteLine("View from display method");
    }
    // Main Method
    public static void Main (String []args){
        // Creating object
        testClass t = new testClass();
        // calling method
        t.display();
    }
}
```

Output:

View from display method

Example 2:

```
// C# program to illustrate the interface
using System;
// interface declaration
interface Vehicle {
    // all are the abstract methods.
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}
// class implements interface
class Bicycle : Vehicle{
    int speed;
    int gear;
    // to change gear
    public void changeGear(int newGear){
        gear = newGear;
    }
    // to increase speed
    public void speedUp(int increment){
        speed = speed + increment;
    }
    // to decrease speed
    public void applyBrakes(int decrement){
        speed = speed - decrement;
    }
}
```

Object Oriented Programming System (OOPs)

```
        public void printStates(){
            Console.WriteLine("speed: " + speed + " gear: " + gear);
        }
    }
    // class implements interface
    class Bike : Vehicle {
        int speed;
        int gear;
        // to change gear
        public void changeGear(int newGear){
            gear = newGear;
        }
        // to increase speed
        public void speedUp(int increment){
            speed = speed + increment;
        }
        // to decrease speed
        public void applyBrakes(int decrement){
            speed = speed - decrement;
        }
        public void printStates(){
            Console.WriteLine("speed: " + speed + " gear: " + gear);
        }
    }
}
class Program {
    // Main Method
    public static void Main (String [] args){
        // creating an instance of Bicycle
        // doing some operations
        Bicycle bicycle = new Bicycle();
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);
        Console.WriteLine("Bicycle present state :");
        bicycle.printStates();
        // creating instance of bike.
        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);
        Console.WriteLine("Bike present state :");
        bike.printStates();
    }
}
```

Output:

Bicycle present state :

speed: 2 gear: 2

Bike present state :

speed: 1 gear: 1

Object Oriented Programming System (OOPs)

Multiple Interfaces Implementation

A class in C# can implement multiple interfaces. To implement multiple interfaces, a class must list all the interfaces separated by commas after the “:” colon.

```
using System;

public interface IShape{
    double GetArea();
}

public interface IColor{
    string GetColor();
}

// Implementing multiple Interfaces
public class Rectangle : IShape, IColor{
    private double length;
    private double width;
    private string color;

    public Rectangle(double length, double width, string color){
        this.length = length;
        this.width = width;
        this.color = color;
    }

    public double GetArea(){
        return length * width;
    }

    public string GetColor(){
        return color;
    }
}

class Program{
    static void Main(string[] args){
        Rectangle rectangle = new Rectangle(5, 10, "Blue");
        Console.WriteLine("Area: " + rectangle.GetArea());
        Console.WriteLine("Color: " + rectangle.GetColor());

        Console.ReadKey();
    }
}
```

In the above example, we have two interfaces: **IShape** and **IColor**. The IShape interface defines a single method GetArea(), while the IColor interface defines a single

Object Oriented Programming System (OOPs)

method GetColor(). The Rectangle class implements both interfaces and provides implementations for both methods.

In the Main() method, we create a new instance of the Rectangle class and call the GetArea() and GetColor() methods on it, demonstrating that the class correctly implements both interfaces.

Output:

Area: 50

Color: Blue

Explicit Interface Implementation

In C#, we use explicit interface implementation to explicitly implement an interface member to avoid ambiguity when a class implements multiple interfaces having a member with the same name and signature.

By using explicit implementation, we can resolve the ambiguity between the members with the same name. The class must explicitly specify which interface member it is implementing.

Example:

Here is a simple full code example of explicit interface implementation in C#:

```
using System;
```

```
public interface IShape {  
    void Draw();  
}
```

```
public interface ICircle {  
    void Draw();  
}
```

```
public class Shape : IShape, ICircle {  
    void IShape.Draw() {  
        Console.WriteLine("Drawing a shape");  
    }  
  
    void ICircle.Draw() {  
        Console.WriteLine("Drawing a circle");  
    }  
}
```

```
class Program {
```

Object Oriented Programming System (OOPs)

```
static void Main(string[] args) {  
    IShape shape = new Shape();  
    ICircle circle = new Shape();  
  
    shape.Draw(); // Outputs "Drawing a shape"  
    circle.Draw(); // Outputs "Drawing a circle"  
}  
}
```

Default Interface Methods (C# 8.0)

Default interface methods are introduced in C# 8.0, which allows you to define default implementations of interface methods.

This feature is useful when you want to add a new method to an existing interface without breaking the code that implements it.

A default method in an interface is similar to a regular method but has a default implementation. The default implementation provides a fallback behavior for the interface method if a class that implements the interface does not provide its own implementation.

Here is an example of how to declare a default method in an interface:

Example of Default Interface method:

```
using System;  
  
public interface IExample{  
    void Print();  
    // Default method implementation  
    void SayHello() {  
        Console.WriteLine("Hello, World!");  
    }  
}  
  
public class Example : IExample{  
    public void Print(){  
        Console.WriteLine("This is an example.");  
    }  
}  
  
public class Program{  
    static void Main(string[] args){  
        Example ex = new Example();  
        ex.Print();  
        ex.SayHello(); // Calling the default method  
    }  
}
```

Object Oriented Programming System (OOPs)

```
}
```

Modifiers in Interfaces (C# 8.0)

Starting with C# 8.0, interfaces can have access modifiers just like classes. The available access modifiers for interfaces are public, internal, protected internal, and private. These access modifiers control the visibility of the interface members to the outside world.

A public interface can be accessed from anywhere, whereas an internal interface can only be accessed from within the same assembly.

A protected internal interface can be accessed from within the same assembly and from derived types located in other assemblies. Finally, we can only access a private interface within the same class or struct.

Example of access modifiers supported in C# Interface 8.0:

Let's consider an example where we have an interface named **IMyInterface** with four methods, each with a different access modifier.

```
using System;
namespace InterfaceAccessModifiersExample{
    public interface IExampleInterface{
        public void PublicMethod();

        void InternalMethod();

        protected void ProtectedMethod();

        private void PrivateMethod();

        static void StaticMethod(){
            Console.WriteLine("This is a static method in the interface");
        }
    }
}

public class ExampleClass : IExampleInterface{
    public void PublicMethod(){
        Console.WriteLine("This is a public method implementation in the class");
    }

    void IExampleInterface.InternalMethod(){
        Console.WriteLine("This is an internal method implementation in the class");
    }

    protected void IExampleInterface.ProtectedMethod(){
```

Object Oriented Programming System (OOPs)

```
        Console.WriteLine("This is a protected method implementation in the class");
    }

    private void IExampleInterface.PrivateMethod(){
        Console.WriteLine("This is a private method implementation in the class");
    }
}

class Program{
    static void Main(string[] args){
        ExampleClass obj = new ExampleClass();
        obj.PublicMethod();

        IExampleInterface iobj = obj;
        iobj.InternalMethod();

        ((IExampleInterface)obj).ProtectedMethod();

        // Compiler error, private method not accessible
        //((IExampleInterface)obj).PrivateMethod();

        // Calling a static method in the interface
        IExampleInterface.StaticMethod();
    }
}
```

Advantage of Interface:

- ✓ It is used to achieve loose coupling.
- ✓ It is used to achieve total abstraction.
- ✓ To achieve component-based programming
- ✓ To achieve multiple inheritance and abstraction.
- ✓ Interfaces add a plug and play like architecture into applications.

Abstract class vs interface

Abstract Class	Interface
Abstract class contains both declaration and definition part.	Interface contains only a declaration part.
Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
Abstract class contain constructor.	Interface does not contain constructor.
Abstract class can have access specifier for functions.	Interface cannot have access specifier for functions. It is public by default.
Abstract class is fast.	Interface is comparatively slow.

Object Oriented Programming System (OOPs)

Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
An abstract class cannot be instantiated.	The interface is absolutely abstract and cannot be instantiated.

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieve fully abstraction (100%).

When to use Abstract Interfaces or Classes?

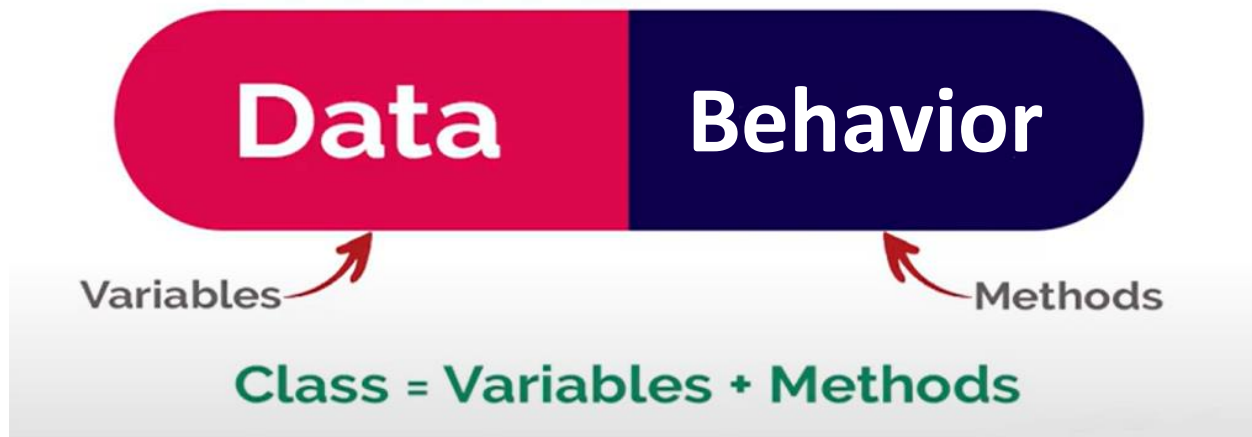
- According to the mentioned explanations, when we need multiple inheritance, we must use the Interface; Because this is not possible in Abstract classes.
- When we want to implement all the methods introduced in the base class completely in the derived class, we must use the Interface.
- When we face many changes in large projects, the use of the Abstract class is recommended; Because by changing it, changes are automatically applied to the derived classes.
- Because it is not possible to define other elements in the Interfaces other than the declaration of methods and properties, if we are required to use these elements, it is necessary to use Abstract classes.
- If we do not want to implement all the methods in the derived classes and code some of them in the parent class, we must use the Abstract class.
- In general, an interface defines the framework and capabilities of a class and is a contract; But the Abstract class determines the type of class. This difference helps programmers to determine when to use the two.

Encapsulation

It is a process of wrapping data (fields, properties) and behavior (methods) together in a single unit (class) that refers to hiding the implementation details of an object from other parts of the program.

Object Oriented Programming System (OOPs)

Encapsulation = Data + Behavior



In other words, encapsulation provides a way to protect the data from unauthorized access or modification and ensures that the object's state remains consistent.

Encapsulation in C# is achieved through access modifiers (Example: public, private, protected, and internal) to restrict the visibility of the class members (fields, properties, methods, and events) from the outside world.

A class's fields are typically marked as private, and its methods and properties are marked as public, protected, or internal based on the desired usage.

Example

Let's take a simple example to understand how encapsulation works in C#. Suppose you have a class named **BankAccount** that contains one private field **balance** and two public methods "Deposit" and "Withdraw" to manipulate the balance.

```
using System;
public class BankAccount{
    private decimal balance;
    public void Deposit(decimal amount){
        balance += amount;
    }
    public void Withdraw(decimal amount){
        if(balance < amount){
            Console.WriteLine("Insufficient funds.");
            return;
        }
    }
}
```

Object Oriented Programming System (OOPs)

```
        balance -= amount;
    }
    public decimal GetBalance(){
        return balance;
    }
}
class Program{
    static void Main(string[] args){
        BankAccount account = new BankAccount();
        account.Deposit(1000);
        account.Withdraw(500);
        decimal balance = account.GetBalance();
        Console.WriteLine("Account balance is: " + balance);
    }
}
```

In the above example, we can see that we can access the “Deposit” and “Withdraw” methods to modify the balance, but we cannot directly access the “balance” field.

We can use the **GetBalance()** method to retrieve the balance information.

Output:

Account balance is: 500

Encapsulation provides several benefits to C# developers, some of which are listed below:

- ✓ **Security:** Encapsulation in C# provides a way to protect the data from unauthorized access or modification, making the code more secure and robust.
- ✓ **Modularity:** Encapsulation promotes code modularity by grouping related data and behavior into a single unit, making the code more organized and easier to maintain.
- ✓ **Abstraction:** Encapsulation provides a way to abstract the implementation details of a class from the outside world, making it easier to understand and use the class without worrying about its internal implementation.
- ✓ **Flexibility:** Encapsulation allows for changes to be made to the implementation of a class without affecting the code that uses the class. It makes the code more flexible and adaptable to changing requirements.
- ✓ **Code Reusability:** Encapsulation promotes code reusability by allowing the same class to be used in multiple contexts without modifying its internal implementation.

The following ways we can achieve Encapsulation in C#:

Object Oriented Programming System (OOPs)

- **Access Modifiers:** [Access modifiers](#) (public, private, protected, internal, Protected Internal and Private Protected) restrict the visibility of the class members from the outside world.

.Net has six access Specifiers.

- **Public** -- Available anywhere in the program.
 - **Private** -- Only available within the class where it is declared.
 - **Protected** -- Available only to the current class or child classes that inherit from that class.
 - **Internal** -- Available only to the classes in the current project where it is defined.
[The internal keyword specifies that the object is accessible only inside its own assembly but not in other assemblies.]
 - **Protected Internal** -- The protected internal access modifier is a combination of protected and internal. As a result, we can access the protected internal member only in the same assembly or in a derived class in other assembly(project).
 - **Private Protected**-- Only available to the classes in the current project where it is defined and also to the children of all classes that inherit from the parent class within the same project.
- **Properties:** [Properties](#) provide a way to encapsulate the data by providing a controlled way of accessing and modifying the class members. Properties can have getter and setter methods that enforce validation and data consistency.
 - **Interfaces:** [Interfaces](#) provide a way to define a contract for a class without specifying its implementation details. It allows for loose coupling between classes and promotes code modularity and flexibility.

Abstraction Vs Encapsulation

Abstraction	Encapsulation
Abstraction solves the problem in the design level.	Encapsulation solves the problem in the implementation level.
Abstraction is outer layout in terms of design.	Encapsulation is inner layout in terms of implementation.
Abstraction is achieved through abstract classes and interfaces in C#.	Encapsulation is achieved through access modifiers, properties, and interfaces in C#.
For Ex: Outer look of a iPhone like it has a display screen.	For Ex: Inner implementation details of a iPhone, how display screen are connect with each other using circuits.

Object Oriented Programming System (OOPs)

Inheritance

It's a mechanism in which one object acquires all the states and behaviors of a parent object. It is a relationship or association between two classes that allows one class to inherit code from another class.



“Inheritance means a parent-child relationship where we can create a new class by using the existing class.”

Inheritance is the **IS-A** type of relationship.
(Example: Apple is a fruit; Car is a Vehicle)

Class	Description
Parent Class	In Inheritance, the class whose features are inherited is known as the parent class(or a generalized class, superclass, or base class)
Child Class	The class that inherits the existing class is known as a child class(or a specialized class, derived class, extended class, or subclass).
<i>Inheritance allows a derived class to extend the functionality of the base class as it can add its own fields, properties, and methods in addition to the base class data members.</i>	

Syntax

Inheritance uses “**:**” colon to make a relationship between parent class and child class. Here you can see in the below syntax.

```
<AccessModifier> class <BaseClassName>{  
    // Base class code  
}
```

```
<AccessModifier> class <DerivedClassName> : <BaseClassName>{
```

Object Oriented Programming System (OOPs)

```
// Derived class code  
}
```

Example:

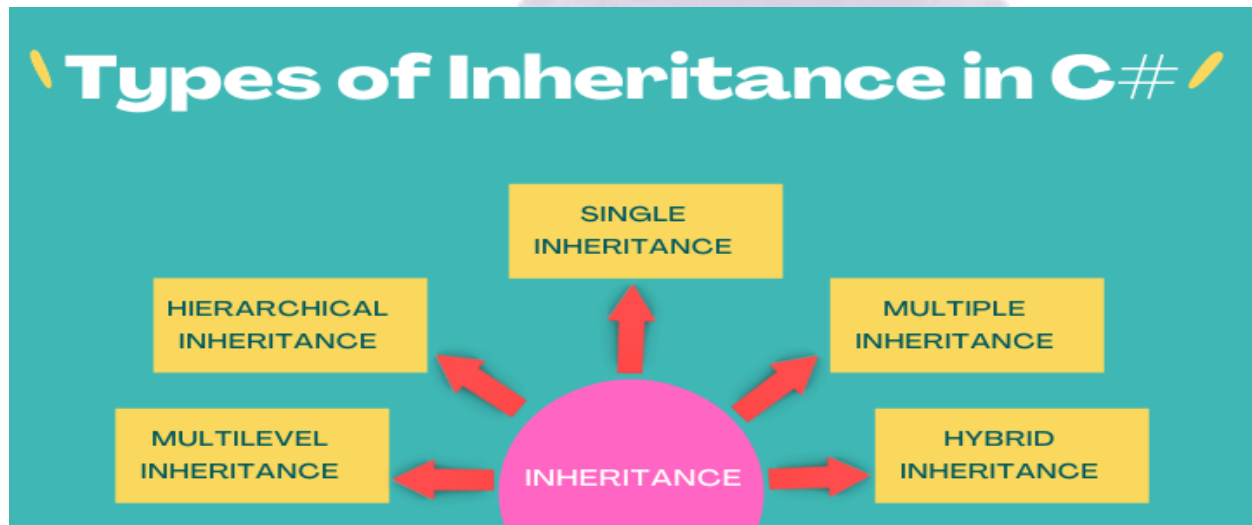
```
//Base class  
public class Phone{  
    public string Brand { get; set; }  
    public string Model { get; set; }  
    public void Call(string phoneNumber){  
        Console.WriteLine($"Calling {phoneNumber}...");  
    }  
}  
//derived class & inherited to base class  
public class SmartPhone : Phone{  
    public void BrowseInternet(){  
        Console.WriteLine("Browsing the internet...");  
    }  
    public void SendMessage(string phoneNumber, string message){  
        Console.WriteLine($"Sending message to {phoneNumber}: {message}");  
    }  
}  
//derived class & inherited to base class  
public class LandLinePhone : Phone {  
    // LandLine phones don't have internet browsing functionality or message functionality  
}  
//main class  
public class Program{  
    public static void Main(){  
        SmartPhone cellPhone = new SmartPhone{  
            Brand = "Apple",  
            Model = "Iphone 14"  
        };  
  
        LandLinePhone housePhone = new LandLinePhone{  
            Brand = "Panasonic",  
            Model = "KX-TG3411SX"  
        };  
        cellPhone Call("4259638547"); // Output: "Calling 4259638547..."  
        cellPhone SendMessage("2589634712", "Hi! I'm a new contributor of codecademy"); // Output:  
"Sending message to 2589634712: Hi! I'm a new contributor of codecademy."  
        cellPhone BrowseInternet(); // Output: "Browsing the internet..."  
        housePhone Call("4384938752"); // Output: "Calling 4384938752..."  
    }  
}
```

Advantages of Inheritance

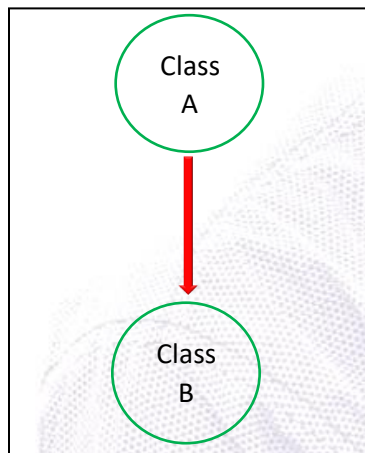
- **Code reusability:** It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

Object Oriented Programming System (OOPs)

- **Avoiding Duplication of Code:** Inheritance support reusability, which reduces duplication of the same code.
- **Polymorphic behavior:** Inheritance allows a derived class to override the methods of its base class. This allows developers of derived classes to customize or completely replace the behavior of base class methods.
- **Extensibility:** Extensibility is the ability to extend and derive new classes from the existing classes.



Single Inheritance: It's when a child class inherits the characteristics of only one parent class.



Single Inheritance

The following above image, class A serves as a base class for the derived class B.

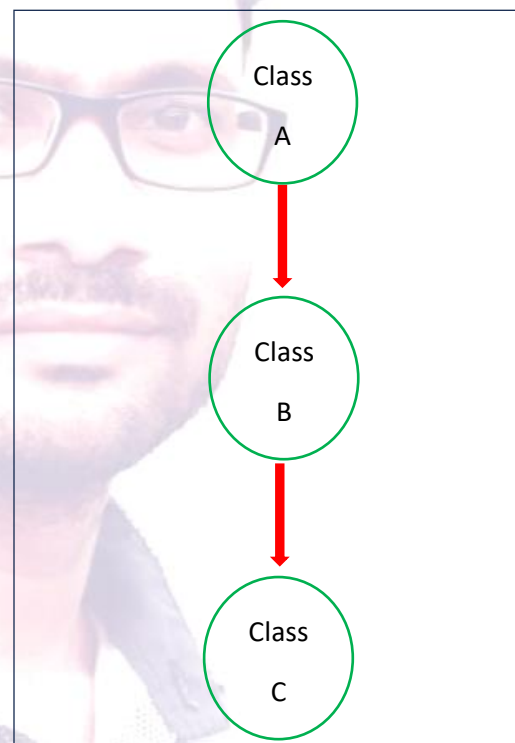
Object Oriented Programming System (OOPs)

```
public class A{  
    // Code Implementation  
}  
public class B: A{  
    // Code Implementation  
}
```

Multilevel Inheritance: Multilevel inheritance is when a child class inherits from a parent class which in turn inherits from another parent class. This forms a chain of inheritance. A common example in real life is the grand-parent, parent and grand-children relationship.

The following image is an example of multi-level inheritance in C#.

```
public class A  
{  
    // Code Implementation  
}  
public class B : A  
{  
    // Code Implementation  
}  
public class C : B  
{  
    // Code Implementation  
}
```



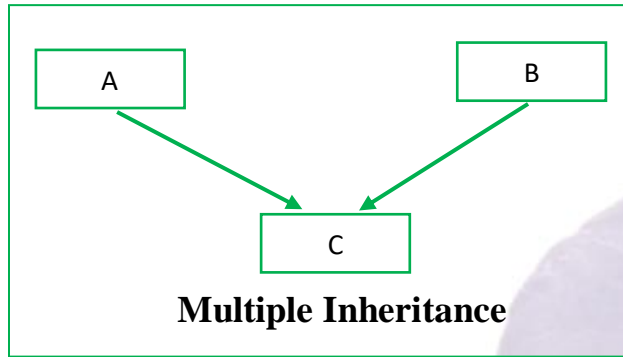
Multi-Label Inheritance

For example, if class “C” is derived from class “B”, and further class “B” is derived from class “A”, then class “C” will be able to inherit the members declared in both class “B” and class “A”.

Multiple Inheritance: Multiple inheritance allows a child class to have multiple base classes and inherit features from all of its parent classes.

However, **C# does not explicitly support multiple inheritance with classes, and attempting to do so will result in a compilation error.** Interfaces are the only way to implement multiple inheritance in C#.

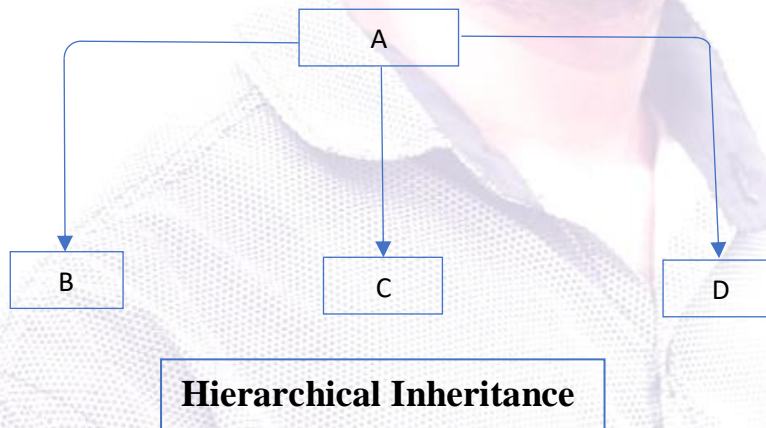
Object Oriented Programming System (OOPs)



The following is the below example of implementing multiple interfaces in C#.

```
interface A{  
    // Code implementation  
}  
interface B{  
    // Code implementation  
}  
// Derived class  
class C : A,B{  
    // Code Implementation  
}
```

Hierarchical Inheritance: It's when a single parent class has multiple children.



Example: 1

```
//Base class  
class A{  
    // Code implementation  
}  
// Derived class  
class B :A{
```

Object Oriented Programming System (OOPs)

```
// Code implementation
}  
  
// Derived class  
class C:A{  
    // Code Implementation  
}  
// Derived class  
class D:A{  
    // Code Implementation  
}
```

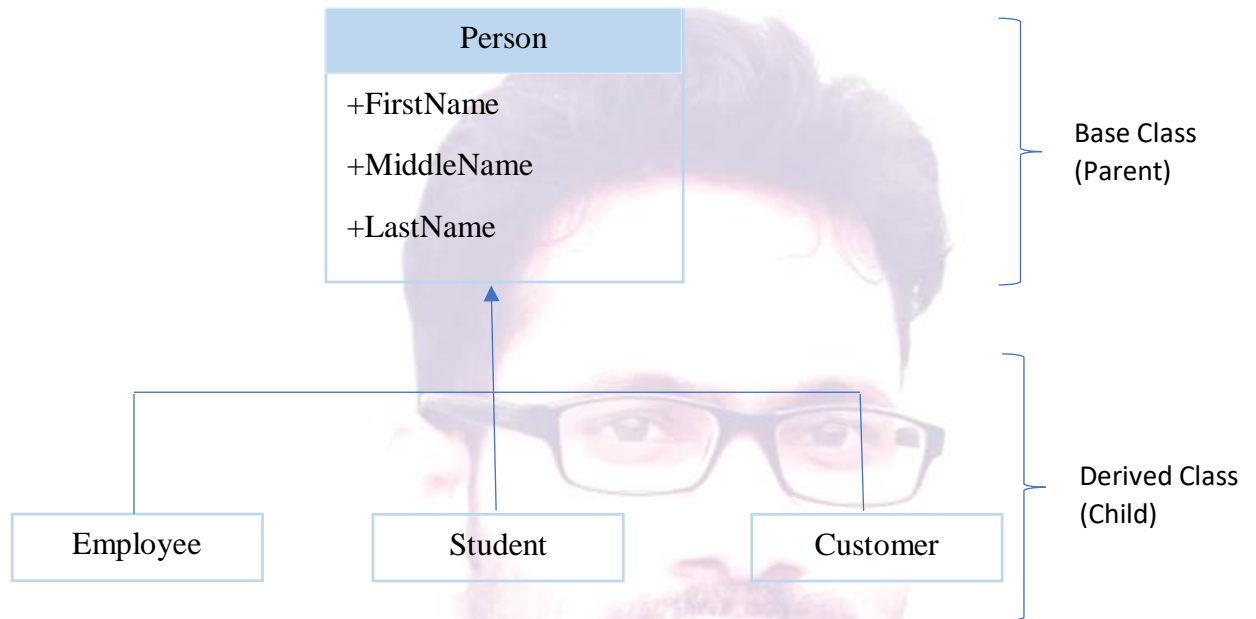
The following above example, class A refers to a base class for the derived class B, class C, and class D.

Example: 2

```
public class Animal {  
    // Parent class properties and methods  
}  
public class Dog : Animal{  
    // Animal class properties and methods accessible here due to  
    inheritance  
    // Dog class properties and methods  
}  
public class Cat : Animal{  
    // Animal class properties and methods accessible here due to  
    inheritance  
    // Cat class properties and methods  
}  
public class Lion : Animal{  
    // Animal class properties and methods accessible here due to  
    inheritance  
    // Lion class properties and methods  
}  
public class Mouse : Animal{  
    // Animal class properties and methods accessible here due to  
    inheritance  
    // Mouse class properties and methods  
}
```

Object Oriented Programming System (OOPs)

Example: 3



Code Explanation of above image:

```
class Person{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string GetFullName(){
        return FirstName + " " + LastName;
    }
}

class Employee : Person{
    public int EmployeeId { get; set; }
    public string CompanyName { get; set; }
}

class Student : Person{
    public int StudentId { get; set; }
    public int Age { get; set; }
}
```

Object Oriented Programming System (OOPs)

```
class Customer : Person{
    public int CustomerId { get; set; }
    public int Age { get; set; }
    public string InvoiceId {get; set;}
}

public class Program{
    public static void Main(){
        Employee emp = new Employee();
        emp.FirstName = "Steve";
        emp.LastName = "Jobs";
        emp.EmployeeId = 1;
        emp.CompanyName = "Apple";

        Console.WriteLine(emp.GetFullName()); //base class method
        Console.WriteLine(emp.EmployeeId);
        Console.WriteLine(emp.CompanyName);

        Student stdnt = new Student();
        stdnt.FirstName = "Jhon"
        stdnt.LastName = "Joe"
        stdnt.Age =25
        stdnt.StudentId =101
        Console.WriteLine(stdnt.GetFullName()); //base class method
        Console.WriteLine(stdnt.StudentId);
        Console.WriteLine(stdnt.Age);

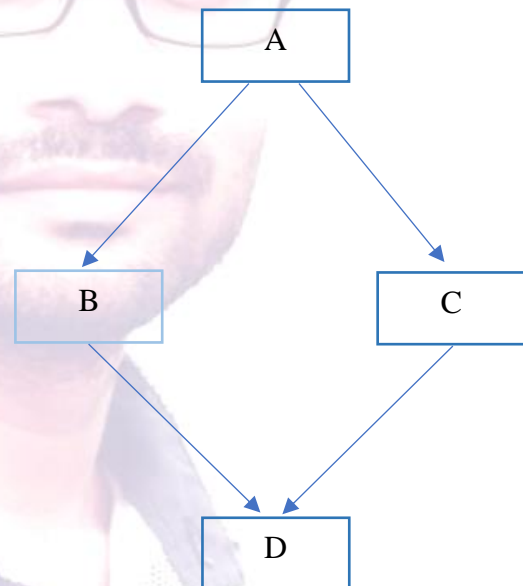
        Customer cust = new Customer ();
        cust.FirstName = "Jhon"
        cust.LastName = "Joe"
        cust.Age =25
        cust.StudentId =101
```


Object Oriented Programming System (OOPs)

```
    cust.InvoiceId ="In102"  
  
    Console.WriteLine(cust.GetFullName()); //base class method  
  
    Console.WriteLine(cust. CustomerId);  
  
    Console.WriteLine(cust.Age);  
  
    Console.WriteLine(cust.InvoiceId);  
  
    }  
}
```

Hybrid Label Inheritance: In C#, the hybrid inheritance is not supported with classes, but hybrid inheritance can be achieved through interfaces only. Actually, hybrid inheritance is a mix of multiple types of inheritance.

```
interface A  
{  
    // Code here  
}  
  
interface B :A  
{  
    // Code here  
}  
  
interface C : A  
{  
    // Code here  
}  
  
// Derived class  
class D :B, C  
{  
    // Code here  
}
```



Hybrid Inheritance

[Note: If you don't want other classes to inherit from a class, use the sealed keyword]

Object Oriented Programming System (OOPs)

If you try to access a **sealed** class, C# will generate an error:

```
sealed class Vehicle
{
    ...
}

class Car : Vehicle
{
    ...
}
```

The error message will be something like this:

```
'Car': cannot derive from sealed type 'Vehicle'
```

Key points about Inheritance in C#

- In C #, the **structure** does not support inheritance, but it can be inherited through interfaces.
- The **Object** class is the base class of all subclasses. All classes in C# are implicitly derived from the Object class.
- C# only supports single inheritance, but we can implement multiple inheritance through interfaces.
- A child class can't inherit the private members of its parent class until or unless the base class doesn't have public properties to access the private fields.
- A child class can inherit all the members of its base class, except the constructor, because the constructor is not a data member of the class. (Note: *A constructor of the base class can be called from the child class.*)

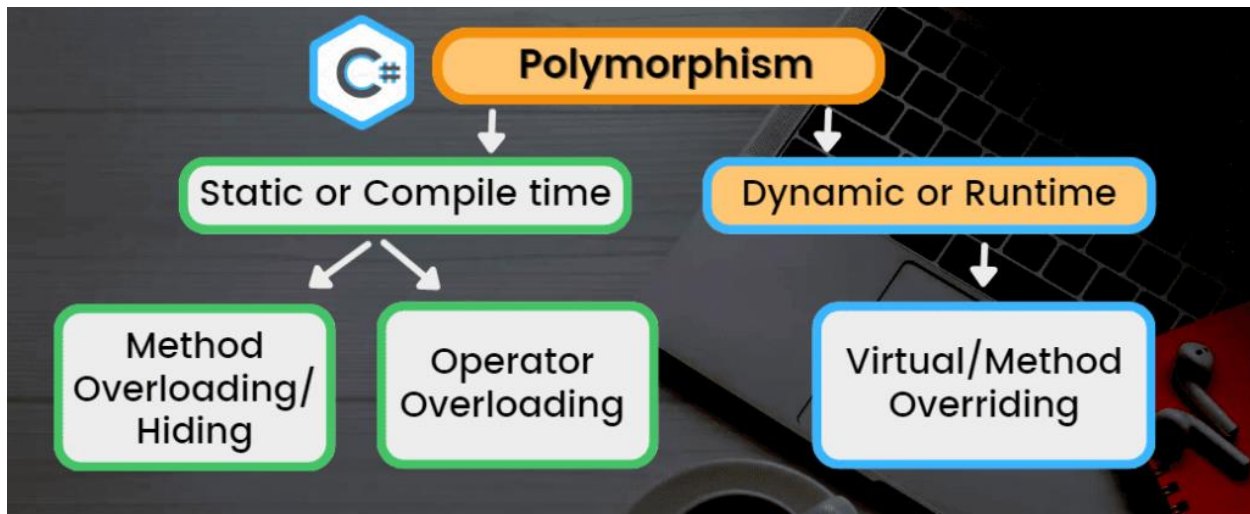
Polymorphism

Polymorphism is the ability of an object to take on many forms.

Polymorphism is one of the main key concepts of object-oriented programming after **encapsulation** and **inheritance**.

Now, we will discuss the different types of **polymorphism in C#**, how they work, how to implement them, and how to use polymorphism in our program code.

Object Oriented Programming System (OOPs)



There are two types of polymorphism like that.

1. Compile Time Polymorphism / Static Polymorphism (method overloading)
2. Run-Time Polymorphism / Dynamic Polymorphism (method overriding)

Compile Time Polymorphism:

Compile Time Polymorphism in C# refers to defining multiple methods with the same name but different parameters. It allows us to perform different tasks with the same method name by passing different parameters.

- Compile time polymorphism is also known as **method overloading**, early binding, or static binding.
- In compile-time polymorphism, the compiler identifies which method is needed to be called at compile-time rather than runtime.

Rules for Method Overloading:

Method overloading in C# is a feature of object-oriented programming that allows a class to have multiple methods with the same name, but with different signatures (parameters). There are several rules that must be followed when overloading a method:

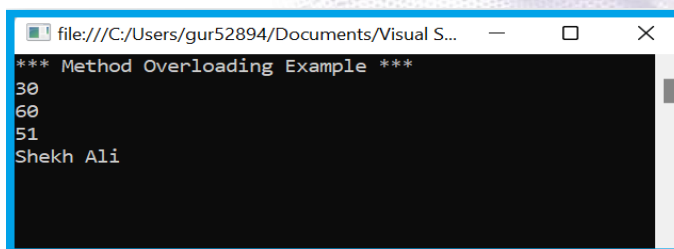
- **Method Signature:** The first rule to overload a method in C# is to change the method signature. Either the number of arguments, type of arguments, or order of arguments must be different types.
- **Method Return Type:** The return type of the method is not part of the method overloading, so simply changing the return type will not overload a method in C#.
- **Access modifier:** The access modifier of the overloaded method can be the same or different.

Object Oriented Programming System (OOPs)

The following example shows method overloading by defining multiple Add methods with different types and numbers of parameters.

```
using System;
namespace MethodOverloadingExample{
    class Program{
        public void Add(int num1, int num2){
            Console.WriteLine(num1 + num2);
        }
        public void Add(int num1, int num2, int num3){
            Console.WriteLine(num1 + num2 + num3);
        }
        public void Add(float x, float y){
            Console.WriteLine(x + y);
        }
        public void Add(string str1, string str2){
            Console.WriteLine(str1 + " " + str2);
        }
        static void Main(string[] args){
            Program program = new Program();
            Console.WriteLine($"*** Method Overloading Example ***");
            program.Add(10, 20);
            program.Add(10, 20, 30);
            program.Add(20.5f, 30.5f);
            program.Add("Shekh", "Ali");
            Console.ReadKey();
        }
    }
}
```

Output:



```
file:///C:/Users/gur52894/Documents/Visual S...
*** Method Overloading Example ***
30
60
51
Shekh Ali
```

Example: Invalid Method overloading using different return type

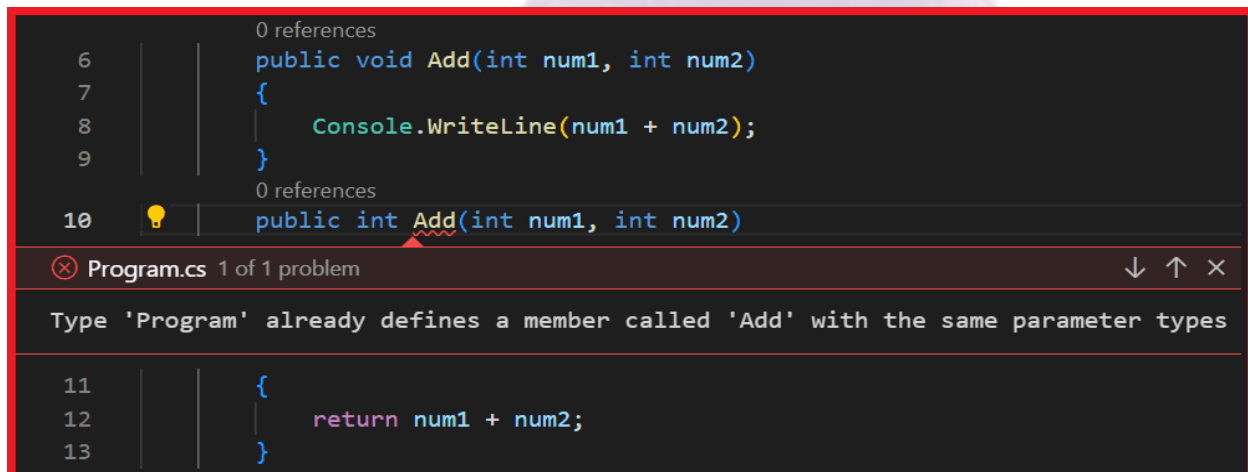
As we already know the return type of methods is not considered as method overloading. The following example will give a compile-time error.

```
public void Add(int num1, int num2){
```


Object Oriented Programming System (OOPs)

```
        Console.WriteLine(num1 + num2);  
    }  
    public int Add(int num1, int num2){  
        return num1 + num2;  
    }  
}
```

Once we run the application, we will get the compilation error:



```
6 | 0 references  
  | public void Add(int num1, int num2)  
7 | {  
8 |     Console.WriteLine(num1 + num2);  
9 | }  
  | 0 references  
10 | public int Add(int num1, int num2)  
    |  
    | x Program.cs 1 of 1 problem  
    | Type 'Program' already defines a member called 'Add' with the same parameter types  
    |  
11 | {  
12 |     return num1 + num2;  
13 | }
```

Dynamic / Runtime Polymorphism: Runtime time polymorphism is done using inheritance and virtual methods. Method overriding is called runtime polymorphism.

When overriding a method, you change the behavior of the method for the derived class. Overloading a method simply involves having another method with the same prototype.

Key points about method overriding and runtime polymorphism in C#:

- **Inheritance Requirement:** Method overriding is possible only when there is an inheritance relationship between the base class (parent class) and the derived class (child class).
- **Same Signature:** The method in the derived class must have the same signature (name, return type, and parameters) as the method in the base class that it intends to override.
- **Override Keyword:** The **override** keyword is used in the derived class to indicate that a method is intended to override a method in the base class.

“Runtime Polymorphism is also known as Dynamic binding or Late binding where the method that needs to be called is determined at the run-time rather than at compile-time.

We can achieve Runtime polymorphism using override & virtual keywords, and inheritance principles.”

Object Oriented Programming System (OOPs)

Runtime polymorphism Example1:

```
class Animal{
    public virtual void MakeSound(){
        Console.WriteLine("Animal makes a generic sound");
    }
}
class Cat : Animal{
    public override void MakeSound(){
        Console.WriteLine("Cat purrs");
    }
}
class Dog : Animal{
    public override void MakeSound(){
        Console.WriteLine("Dog barks");
    }
}
class Program{
    static void Main(){
        Animal pet1 = new Cat();
        Animal pet2 = new Dog();

        Console.WriteLine("Pet 1 sound:");
        pet1.MakeSound();           //Calls the overridden method in Cat class

        Console.WriteLine("\nPet 2 sound:");
        pet2.MakeSound();           //Calls the overridden method in Dog class
    }
}
```

Output:

Pet 1 sound:
Cat purrs
Pet 2 sound:
Dog barks

Code Explanation:

In this example, we have an **Animal** base class with a virtual method **MakeSound**. The **Cat** and **Dog** classes are derived from **Animal** and override the **MakeSound** method with their specific implementations. The **Main** method shows how you can use polymorphism to call the overridden methods based on the actual runtime type of the objects.

Method Overriding Example2:

Object Oriented Programming System (OOPs)

In the following code example, we have defined an Area() method with a virtual keyword in the base class named **Shape**, which allows the derived classes named **Circle** and **Square** to override this method using the override keyword.

```
using System;
namespace MethodOverridingExample{
    abstract class Shape{
        //Virtual method
        public virtual double Area(){
            return 0;
        }
    }
    class Circle : Shape{
        private double radius;
        public Circle(double radius){
            this.radius = radius;
        }
        //Override method
        public override double Area(){
            return (3.14*radius*radius);
        }
    }
    class Square : Shape{
        private double side;
        public Square(double square){
            side = square;
        }
        //Override method
        public override double Area(){
            return (side*side);
        }
    }
    class Program{
        static void Main(string[] args){
            Circle circle = new Circle(5);
            Square square = new Square(10.5);
            Console.WriteLine($"Area of Circle = {circle.Area()}");
            Console.WriteLine($"Area of Square = {square.Area()}");
            Console.ReadKey();
        }
    }
}
```

Output:

Area of Circle=78.5

Area of Square =110.25

Rules for method overriding:

Object Oriented Programming System (OOPs)

It's important to follow the following rules when overriding a method because they ensure that the child class maintains the expected behavior of the method and does not violate the contract established by the base or superclass.

- A child class method must have the same name, return type, and number and type of parameters as the method defined in the base class. This is required because the child class is overriding and providing a new implementation of an existing method and needs to maintain the same interface as the original method.
- The method in the child class or subclass must have the same or a more restrictive [access modifier](#) than the method in the base or superclass. For example, if the method in the base class is public, the method in the child class can also be public, but it cannot be less accessible, such as protected or private.
- The method in the child class should not be sealed. If the method in the base class is [sealed](#), it cannot be overridden in the child class. A sealed method is a method that cannot be overridden or modified by a child/subclass.

Difference between the Virtual method and the Abstract method

A Virtual method must always have a default implementation. However, it can be overridden in the derived class, though not mandatory. It can be overridden using override keyword.

An Abstract method does not have an implementation. It resides in the abstract class. It is mandatory that the derived class implements the abstract method. An override keyword is not necessary here though it can be used.

=====End=====

Reference link

- 1) https://www.youtube.com/playlist?list=PL_XxuZqN0xVCqNHQtxzS9LbeNRMG4AJmG
- 2) <https://www.youtube.com/watch?v=ZByRelird54>
- 3) <https://www.programmingwithshri.com/2018/04/abstraction-and-encapsulation-in-c.html>
- 4) <https://www.codecademy.com/resources/docs/c-sharp/inheritance>
- 5) <https://www.shekhali.com/>
- 6) <https://www.tutorialsteacher.com/csharp/oop>
- 7) <https://www.w3schools.com/cs/index.php>
- 8) <https://dotnettutorials.net/course/csharp-dot-net-tutorials/>