C#(oops)

O.O.P.S (Object Oriented Programming Structure)

To create a Real-world Application using programming we use concept of oops. It is a concept that is used to create a complex Software.

1. Class :- A class is a blueprint of an object that defines its basic structure and behavior just like a template.
2. Object :- An Object is an instance of a class that has its own identity , state and behavior.
3. Classes and Objects allow developers to efficiently create , modify and maintain a large amount of data.

# Data Abstraction

It is a process of hiding complex implementation details and showing only necessary information to the outside world . It helps in reducing complexity and increase the efficiency of code. Hides Implementation details and only shows essential features to the user (using abstract classes and interface)

```
using System;

public abstract class Shape
{
   public abstract void Draw(); // Abstract method, no implementation here
}

public class Circle : Shape
{
   public override void Draw()
```

```
    {
        Console.WriteLine("Drawing a Circle");
    }
}

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();

        Shape rectangle = new Rectangle();
        rectangle.Draw();
    }
}
```

#Data Encapsulation

Data Encapsulation is a mechanism of wrapping data and code that operate on data into a single unit , such as a class . It provides security and protects our code and data from outside . Hides data by restricting direct access and allows modification only through methods(using private variables and getter/setter)

```
using System;

public class Person
{
    // Private fields
    private string name;
    private int age;

    // Public properties (encapsulation)
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public int Age
    {
        get { return age; }
        set { age = value; }
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Create a Person object
        Person person = new Person();

        // Set properties using the setters
        person.Name = "John";
        person.Age = 30;

        // Get properties using the getters
        Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");
    }
}
```

#Inheritance

Inheritance allows one class (the child class) to inherit the properties and behaviors of another class (the parent class).

Inheritance can save time when writing code by allowing classes to share common functionality.

→ There are four types of inheritance in C# Single, Multiple , Hierarchical (child classes inherit from a single parent base class) and Multi-level.

```
//Multi-level Inheritance
using System;

// Base class
class Animal
{
    public void Speak()
    {
        Console.WriteLine("Animal makes a sound.");
    }
}

// Derived class 1
class Mammal : Animal
{
    public void Walk()
    {
        Console.WriteLine("Mammal walks.");
    }
}

// Derived class 2
class Dog : Mammal
{
    public void Bark()
    {
        Console.WriteLine("Dog barks.");
    }
}

class Program
{
    static void Main()
    {
        Dog dog = new Dog();
        dog.Speak(); // From Animal
        dog.Walk();  // From Mammal
        dog.Bark();  // From Dog
    }
}
```

```
//Multiple Inheritance
using System;

// Interface 1
interface IAnimal
{
    void Speak();
}

// Interface 2
interface IMammal
{
    void Walk();
}

// Class implementing both interfaces
class Dog : IAnimal, IMammal
{
    public void Speak()
    {
        Console.WriteLine("Dog makes a sound.");
    }

    public void Walk()
    {
        Console.WriteLine("Dog walks.");
    }
}
```

```
class Program
{
    static void Main()
    {
        Dog dog = new Dog();
        dog.Speak();  // From IAnimal
        dog.Walk();   // From IMammal
    }
}
```

#Access Modifier

1. `public`

- **Access**: Accessible from **anywhere** (both within the same assembly and from other assemblies).
- **Use case**: When you want to expose the member or type to all parts of the program.

```
public int number;
public void Method() { }
```

2. `private`

- **Access**: Accessible **only within the same class** or struct.
- **Use case**: When you want to restrict access to the member to within the class or struct.

```
private int number;
private void Method() { }
```

3. `protected`

- **Access**: Accessible within the **same class** and **derived classes** (subclasses).
- **Use case**: When you want to allow access within the class and its subclasses, but not from outside.

```
protected int number;
protected void Method() { }
```

4. `internal`

- **Access**: Accessible **only within the same assembly** (project).
- **Use case**: When you want to expose members to other types in the same project, but hide them from other assemblies.

```
internal int number;
internal void Method() { }
```

5. `protected internal`

- **Access**: Accessible from **derived classes** (like `protected`) **and from any class in the same assembly** (like `internal`).
- **Use case**: When you want to allow access within the same assembly and by derived classes, even in other assemblies.

```
protected internal int number;
protected internal void Method() { }
```

6. `private protected`

- **Access**: Accessible **only within the same class or derived classes in the same assembly**.
- **Use case**: To limit access to the class and its derived classes within the same assembly.

```
private protected int number;
private protected void Method() { }
```

#Polymorphism

Polymorphism and inheritance are key concepts in C# that allow for greater flexibility and functionality in programming. In this presentation, we'll explore their definitions , examples , and applications in depth.

Polymorphism allows code reusability, expanding the functionality of your application with fewer lines of code. It also enables dynamic binding and easier maintenance.

1. Function Overloading : - It is when multiple functions in a class share the same name , but have different parameters. It can simplify code and make it more readable , as well as give functionality to different data type.

Example : - In a calculator application , different over-loadings of the Add function could exists , allowing for the addition of two integers or two doubles.

```
using System;

class Program
{
    // Function to add two integers
    public static int Add(int a, int b)
    {
        return a + b;
    }

    // Overloaded function to add three integers
    public static int Add(int a, int b, int c)
    {
        return a + b + c;
    }
    // Overloaded function to add two doubles
    public static double Add(double a, double b)
    {
        return a + b;
    }

    static void Main()
    {
        // Calling the overloaded functions
        Console.WriteLine(Add(5, 10));           // Calls Add(int, int)
        Console.WriteLine(Add(5, 10, 15));       // Calls Add(int, int, int)
        Console.WriteLine(Add(5.5, 10.5));       // Calls Add(double, double)
    }
}
```

2. Function Overriding :- It is when a subclass provides its own implementation of a method already defined by its parent class. Function overriding allows for more flexibility and customization of code , especially in larger applications where code reuse can save time and resources.

Example:- If a parent class defines a GetInfo method , a subclass of that parent class can override the GetInfo method and provide its own implementation.

```
using System;

class Animal
{
    // Base class method
    public virtual void Speak()
    {
        Console.WriteLine("The animal makes a sound.");
    }
}

class Dog : Animal
```

```
{
    // Overriding the base class method
    public override void Speak()
    {
        Console.WriteLine("The dog barks.");
    }
}

class Program
{
    static void Main()
    {
        Animal animal = new Animal();
        animal.Speak();  // Calls the base class method

        Dog dog = new Dog();
        dog.Speak();     // Calls the overridden method in Dog

        Animal animalDog = new Dog();
        animalDog.Speak();  // Calls the overridden method in Dog (polymorphisr
    }
}
```

#Abstract Classes

In C#, an abstract class is a class that cannot be instantiated. It provides a base for subclasses but requires them to provide their own implementations for certain methods.

```
using System;

abstract class Animal
{
    // Abstract method (must be implemented in derived classes)
    public abstract void Speak();

    // Regular method (can be used as-is or overridden)
    public void Eat()
    {
        Console.WriteLine("The animal is eating.");
    }
}

class Dog : Animal
{
    // Implementing the abstract method
    public override void Speak()
    {
        Console.WriteLine("The dog barks.");
    }
}

class Program
{
    static void Main()
    {
        // Creating an object of the Dog class, which is derived from Animal
        Dog dog = new Dog();

        // Calling the implemented method
        dog.Speak();  // Calls the overridden Speak method in Dog

        // Calling the regular method from the abstract class
        dog.Eat();    // Calls the Eat method from Animal
    }
}
```

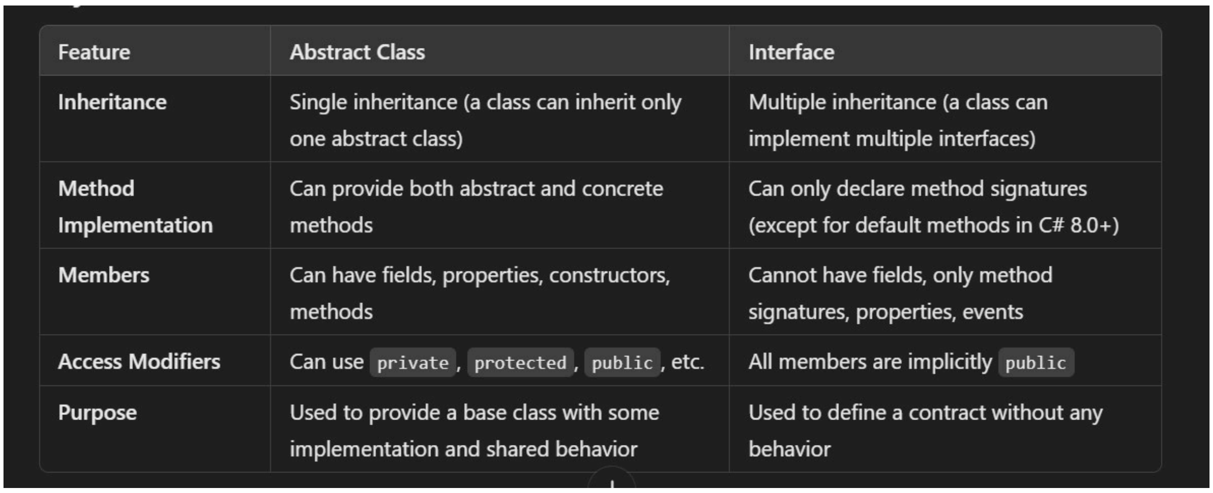
#Interfaces

Interfaces define a set of methods and properties that a class must implement. They provide a contract for future implementations , enabling code to be written that works with objects that share common behaviors but may have differing implementations.

```
using System;

interface IAnimal
{
    // Interface method (does not have a body)
    void Speak();
}

class Dog : IAnimal
{
    // Implementing the interface method
    public void Speak()
    {
        Console.WriteLine("The dog barks.");
    }
}

class Cat : IAnimal
{
    // Implementing the interface method
    public void Speak()
    {
        Console.WriteLine("The cat meows.");
    }
}

class Program
{
    static void Main()
    {
        IAnimal dog = new Dog();
        dog.Speak();  // Calls Speak method for Dog

        IAnimal cat = new Cat();
        cat.Speak();  // Calls Speak method for Cat
    }
}
```

#Difference between Abstract classes and Interfaces.

Abstract classes can provide implementation details , while interfaces cannot. A class can inherit from only one class , but it can implement multiple interfaces. Abstract classes can have constructors
,while interfaces cannot.



Feature	Abstract Class	Interface
Inheritance	Single inheritance (a class can inherit only one abstract class)	Multiple inheritance (a class can implement multiple interfaces)
Method Implementation	Can provide both abstract and concrete methods	Can only declare method signatures (except for default methods in C# 8.0+)
Members	Can have fields, properties, constructors, methods	Cannot have fields, only method signatures, properties, events
Access Modifiers	Can use `private`, `protected`, `public`, etc.	All members are implicitly `public`
Purpose	Used to provide a base class with some implementation and shared behavior	Used to define a contract without any behavior