**Lecture 2: Object-Oriented Programming in C#**

**Topics Covered:**

* Classes and Objects

**Class:**

* A **class** is like a **blueprint** or **template** used to create **objects**.
* It defines **properties (data)** and **methods (functions)** that an object can have.

Think of a **class** like a **Car design** – it tells what a car should have (like engine, wheels, color) and what it should do (like drive, stop).

csharp

CopyEdit

class Car

{

public string Brand;

public int Speed;

public void Drive()

{

Console.WriteLine("The car is driving...");

}

}

**🔷 Object:**

* An **object** is an **instance of a class**.
* Using the class as a blueprint, you can create many objects with different values.

csharp

CopyEdit

Car myCar = new Car(); // Creating an object

myCar.Brand = "Toyota";

myCar.Speed = 100;

myCar.Drive(); // Output: The car is driving...

* Constructors

**What is a Constructor?**

A **constructor** is a **special method** in a class that is **automatically called** when an object of the class is created.

**✅ Purpose:**

* To **initialize the object** (set initial values to variables).

**🧩 Basic Rules:**

* Constructor **name = class name**.
* It **has no return type**, not even void.
* It can be **overloaded** (multiple constructors in the same class with different parameters).

**🔄 Types of Constructors in C#**

**1. Default Constructor**

* Takes **no parameters**.
* Provided automatically if you don't write any constructor.
* You can also define it yourself.

csharp

CopyEdit

class Person

{

public string Name;

// Default Constructor

public Person()

{

Name = "Default Name";

Console.WriteLine("Default constructor called");

}

}

Person p1 = new Person(); // Output: Default constructor called

**2. Parameterized Constructor**

* Takes parameters to initialize fields.

csharp

CopyEdit

class Person

{

public string Name;

public int Age;

// Parameterized Constructor

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

Person p2 = new Person("Ajay", 25);

Console.WriteLine(p2.Name); // Output: Ajay

**3. Copy Constructor**

* Copies values from another object.
* Not provided by default in C#. You write it yourself.

csharp

CopyEdit

class Person

{

public string Name;

public Person(string name)

{

Name = name;

}

// Copy Constructor

public Person(Person obj)

{

Name = obj.Name;

}

}

Person p1 = new Person("Ajay");

Person p2 = new Person(p1);

Console.WriteLine(p2.Name); // Output: Ajay

* Access Modifiers

**What are Access Modifiers?**

Access modifiers **control the visibility** or **access level** of **classes, variables, methods, and properties** in C#.

**✅ Types of Access Modifiers:**

**1. public**

* Accessible **from anywhere** (inside or outside the class, in any project).

csharp

CopyEdit

public class Car

{

public string Brand;

}

**2. private**

* Accessible **only within the same class**.

csharp

CopyEdit

class Car

{

private int speed; // Only this class can use it

}

**3. protected**

* Accessible **within the same class and in derived (child) classes**.

csharp

CopyEdit

class Vehicle

{

protected string fuelType;

}

class Car : Vehicle

{

void Show()

{

fuelType = "Petrol"; // Allowed

}

}

**4. internal**

* Accessible **within the same project/assembly** but **not outside**.

csharp

CopyEdit

internal class Engine

{

internal int horsepower;

}

**5. protected internal**

* Accessible **within the same assembly** and also in **derived classes outside the assembly**.

csharp

CopyEdit

protected internal class Bike

{

protected internal void Start() { }

}

**6. private protected**

* Accessible **within the same class** and **derived classes**, but **only if they're in the same assembly**.

csharp

CopyEdit

class Parent

{

private protected void Display() { }

}

**📊 Summary Table:**

| **Modifier** | **Same Class** | **Derived Class (Same Assembly)** | **Other Class (Same Assembly)** | **Other Assembly** |
| --- | --- | --- | --- | --- |
| public | ✅ | ✅ | ✅ | ✅ |
| private | ✅ | ❌ | ❌ | ❌ |
| protected | ✅ | ✅ | ❌ | ❌ |
| internal | ✅ | ✅ | ✅ | ❌ |
| protected internal | ✅ | ✅ | ✅ | ✅ (inherited) |
| private protected | ✅ | ✅ | ❌ | ❌ |

* Inheritance

**What is Inheritance?**

**Inheritance** is a core concept of **Object-Oriented Programming (OOP)**, allowing one class to **inherit properties and methods** from another class. This promotes **code reusability** and **extensibility**.

**🔹 Base Class (Parent Class)**

* The class whose properties and methods are inherited.

**🔹 Derived Class (Child Class)**

* The class that inherits from the base class.

**🏗️ Syntax for Inheritance:**

csharp

CopyEdit

class Animal // Base Class

{

public void Eat()

{

Console.WriteLine("Eating...");

}

}

class Dog : Animal // Derived Class

{

public void Bark()

{

Console.WriteLine("Barking...");

}

}

**✅ Key Points:**

1. **Single Inheritance**
   * A class inherits from only one base class.
   * Example: class Dog : Animal
2. **Multilevel Inheritance**
   * A class inherits from a derived class, forming a chain of inheritance.
   * Example: class Dog : Mammal, class Mammal : Animal
3. **Hierarchical Inheritance**
   * Multiple classes inherit from a single base class.
   * Example: class Dog : Animal, class Cat : Animal
4. **Multiple Inheritance (Not directly supported in C#)**
   * A class inherits from more than one base class.
   * In C#, this is achieved through **interfaces** (since C# does not support multiple class inheritance).
5. **Hybrid Inheritance (via Interfaces)**
   * A class implements multiple interfaces, simulating multiple inheritance.
   * Example: class Dog : IAnimal, IDomestic

* Polymorphism (method overloading and overriding)

**Polymorphism** in C# is one of the core concepts of **Object-Oriented Programming (OOP)**, allowing objects of different types to be treated as objects of a common base type. The main idea is that **different classes can provide different implementations of the same method or property**.

**Types of Polymorphism in C#:**

**1. Compile-time Polymorphism (Static Polymorphism)**

* **Method Overloading**: The ability to define multiple methods with the same name but different parameters.

**Example**:

csharp

CopyEdit

class Calculator

{

// Method Overloading

public int Add(int a, int b) => a + b;

public double Add(double a, double b) => a + b;

}

class Program

{

static void Main()

{

Calculator calc = new Calculator();

Console.WriteLine(calc.Add(2, 3)); // int Add method

Console.WriteLine(calc.Add(2.5, 3.7)); // double Add method

}

}

* **Operator Overloading**: The ability to define custom behavior for operators like +, -, \*, etc.

**Example**:

csharp

CopyEdit

class Complex

{

public int Real { get; set; }

public int Imaginary { get; set; }

// Operator Overloading

public static Complex operator +(Complex c1, Complex c2)

{

return new Complex { Real = c1.Real + c2.Real, Imaginary = c1.Imaginary + c2.Imaginary };

}

}

**2. Run-time Polymorphism (Dynamic Polymorphism)**

* **Method Overriding**: This occurs when a derived class provides its own implementation of a method that is already defined in its base class. The base class method should be marked as virtual, and the derived class method should be marked as override.

**Example**:

csharp

CopyEdit

class Animal

{

public virtual void Speak() => Console.WriteLine("Animal speaks");

}

class Dog : Animal

{

// Method Overriding

public override void Speak() => Console.WriteLine("Dog barks");

}

class Program

{

static void Main()

{

Animal animal = new Dog();

animal.Speak(); // Outputs: Dog barks

}

}

**Key Differences:**

| **Type** | **Compile-time Polymorphism** | **Run-time Polymorphism** |
| --- | --- | --- |
| **Nature** | Resolved during compilation (early binding) | Resolved during runtime (late binding) |
| **Technique** | Method Overloading, Operator Overloading | Method Overriding (Inheritance + Virtual/Override) |
| **Performance** | More efficient (as it’s resolved early) | Slightly less efficient (resolved at runtime) |

**Advantages of Polymorphism:**

* **Code Reusability**: Methods and properties can be reused in different contexts.
* **Flexibility**: Code is more flexible and can handle new classes and objects without changes to the existing code.
* **Maintainability**: Easier to update or extend the code.
* Interfaces and Abstract Classes

**Abstraction** in C# is one of the four fundamental principles of **Object-Oriented Programming (OOP)**, and it refers to **hiding the complex implementation details** and showing only the essential features or functionality. In simple terms, abstraction allows you to focus on what an object **does** rather than how it does it.

**Types of Abstraction in C#:**

1. **Abstract Classes**
2. **Interfaces**

**1. Abstract Classes**

An **abstract class** cannot be instantiated (you cannot create an object directly from it), and it is intended to be inherited by other classes. Abstract classes can contain **abstract methods** (methods without implementation) and **concrete methods** (methods with implementation).

**Key Points:**

* Abstract methods do not have a body.
* Derived classes must implement abstract methods unless they are also abstract.
* Abstract classes can contain fields, properties, and constructors.
* You cannot instantiate an abstract class directly.

**Syntax:**

csharp

CopyEdit

abstract class Animal

{

// Abstract method (must be implemented in derived class)

public abstract void Sound();

// Concrete method (can be used as is)

public void Eat()

{

Console.WriteLine("Eating...");

}

}

class Dog : Animal

{

// Implementing the abstract method

public override void Sound()

{

Console.WriteLine("Barks");

}

}

class Program

{

static void Main()

{

// Animal animal = new Animal(); // Error: Cannot instantiate abstract class

Dog dog = new Dog();

dog.Sound(); // Output: Barks

dog.Eat(); // Output: Eating...

}

}

**2. Interfaces**

An **interface** defines a contract that a class must adhere to. Unlike abstract classes, interfaces can only contain method signatures (without any implementation), properties, events, or indexers. Classes and structs that implement an interface **must provide an implementation** for all its members.

**Key Points:**

* Interfaces cannot contain fields, constructors, destructors, or static members.
* A class can implement multiple interfaces.
* A class must implement all members of the interface.

**Syntax:**

csharp

CopyEdit

interface IAnimal

{

void Sound(); // Interface method (does not have a body)

}

class Dog : IAnimal

{

// Implementing the interface method

public void Sound()

{

Console.WriteLine("Barks");

}

}

class Program

{

static void Main()

{

IAnimal animal = new Dog();

animal.Sound(); // Output: Barks

}

}

**Key Differences Between Abstract Classes and Interfaces:**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Instantiation** | Cannot be instantiated | Cannot be instantiated |
| **Implementation** | Can have method implementations | Cannot have method implementations (only signatures) |
| **Multiple Inheritance** | Can inherit from one abstract class only | Can implement multiple interfaces |
| **Fields** | Can contain fields | Cannot contain fields |
| **Constructors** | Can have constructors | Cannot have constructors |