**1. WHAT IS OOP (OBJECT-ORIENTED PROGRAMMING)?**

Object-Oriented Programming is a paradigm that organizes software design around **objects**, rather than functions and logic.  
Objects are instances of classes that contain **data (fields)** and **behaviors (methods)**.

C# is a fully OOP-based language and supports all four main principles:

|  |  |
| --- | --- |
| **Principle** | **Description** |
| **Encapsulation** | Hiding internal details of objects and exposing only necessary parts. |
| **Abstraction** | Showing only essential features and hiding background details. |
| **Inheritance** | Acquiring the properties and behavior of another class. |
| **Polymorphism** | One name, many forms — allows method overloading and overriding. |

**2. INHERITANCE**

**Definition:**

Inheritance is a mechanism that allows one class (child) to acquire the properties and behaviors of another class (parent).

**Syntax:**

class Parent

{

// base class code

}

class Child : Parent

{

// derived class code

}

**Types of Inheritance in C#:**

|  |  |  |
| --- | --- | --- |
| **Type** | **Example** | **Description** |
| **Single** | A → B | One parent, one child |
| **Multilevel** | A → B → C | Chain of inheritance |
| **Hierarchical** | A → B, A → C | One parent, multiple children |
| **Multiple** | Not supported directly (use interfaces) |  |

**Keywords:**

* base → Used to call parent class constructors or methods.
* protected → Members accessible in derived classes.

**Example:**

class Employee

{

public void Work() => Console.WriteLine("Working...");

}

class Developer : Employee

{

public void Code() => Console.WriteLine("Coding in C#...");

}

**3. POLYMORPHISM**

**Definition:**

Polymorphism means *"many forms"*. It allows methods to have the same name but behave differently depending on the context.

**Types:**

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Keyword** |
| **Compile-Time (Static)** | Method Overloading | Same method name, different parameters |
| **Runtime (Dynamic)** | Method Overriding | Redefined method in child class |

**Example 1 – Method Overloading:**

class Calculator

{

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

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

}

**Example 2 – Method Overriding:**

class Shape

{

public virtual void Draw() => Console.WriteLine("Drawing a shape...");

}

class Circle : Shape

{

public override void Draw() => Console.WriteLine("Drawing a circle...");

}

**💬 Keyword Summary:**

* virtual → Marks method in base class that can be overridden.
* override → Redefines base class method in child class.
* new → Hides base class method without overriding it.

**4. ABSTRACTION**

**Definition:**

Abstraction is the process of hiding internal implementation details and showing only the necessary features.

In C#, abstraction can be achieved using:

* **Abstract classes**
* **Interfaces**

**Abstract Class Example:**

abstract class Vehicle

{

public abstract void Start(); // abstract method

public void Stop() => Console.WriteLine("Vehicle stopped");

}

class Car : Vehicle

{

public override void Start() => Console.WriteLine("Car started ");

}

**Interface Example:**

interface IShape

{

double GetArea();

}

class Circle : IShape

{

public double radius = 5;

public double GetArea() => Math.PI \* radius \* radius;

}

**Key Points:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Abstract Class** | **Interface** |
| Implementation | Partial | None |
| Multiple Inheritance | Not | yes |
| Access Modifiers | Allowed | Not allowed in methods |

**5. ENCAPSULATION**

**Definition:**

Encapsulation is the process of **binding data and methods together** and protecting data from outside access.

**Example:**

class BankAccount

{

private double balance;

public void Deposit(double amount)

{

if (amount > 0) balance += amount;

}

public double GetBalance() => balance;

}

**Why Encapsulation?**

* Prevents unauthorized data access.
* Makes code maintainable and secure.
* Encourages modular structure.

**6. ACCESS MODIFIERS**

Access modifiers define the **scope** or **visibility** of members.

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Description** | **Scope** |
| **public** | Accessible anywhere | Global |
| **private** | Accessible only within class | Class |
| **protected** | Accessible in derived classes | Class + Child |
| **internal** | Accessible within same assembly | Project-level |
| **protected internal** | Protected + Internal | Class + Assembly |

**7. CONSTRUCTORS AND DESTRUCTORS**

**Constructor**

* Used to initialize objects.
* Automatically called when object is created.
* Has **same name** as class and **no return type**.

**Types:**

1. **Default Constructor** – No parameters.
2. **Parameterized Constructor** – With parameters.
3. **Static Constructor** – Used to initialize static data.
4. **Copy Constructor** – Copies data from one object to another.

**Example:**

class Student

{

public string Name;

public int Age;

public Student(string n, int a)

{

Name = n; Age = a;

}

public void Display() => Console.WriteLine($"{Name}, {Age}");

}

**Destructor**

* Used to clean up resources.
* Called automatically by Garbage Collector.

~Student()

{

Console.WriteLine("Destructor called");

}

**8. STATIC CLASSES AND MEMBERS**

**Static Class:**

A class declared with the keyword static cannot be instantiated and contains only static members.

**Example:**

static class MathHelper

{

public static int Square(int x) => x \* x;

}

**Notes:**

* Access directly using ClassName.MethodName.
* Commonly used for utility/helper functions.

**9. SEALED CLASS**

**Definition:**

A **sealed** class cannot be inherited.  
Used when you want to prevent further derivation for security or performance reasons.

**Example:**

sealed class Logger

{

public void Log(string msg) => Console.WriteLine("Log: " + msg);

}

**10. NAMESPACES**

**Definition:**

A **namespace** is a container that organizes classes and avoids name conflicts.

**Example:**

namespace DhruvTraining.Day3

{

class Demo

{

public void Show() => Console.WriteLine("Inside DhruvTraining namespace");

}

}

Use using DhruvTraining.Day3; to access it in another file.

**11. GARBAGE COLLECTOR (CLR CONCEPT RECAP)**

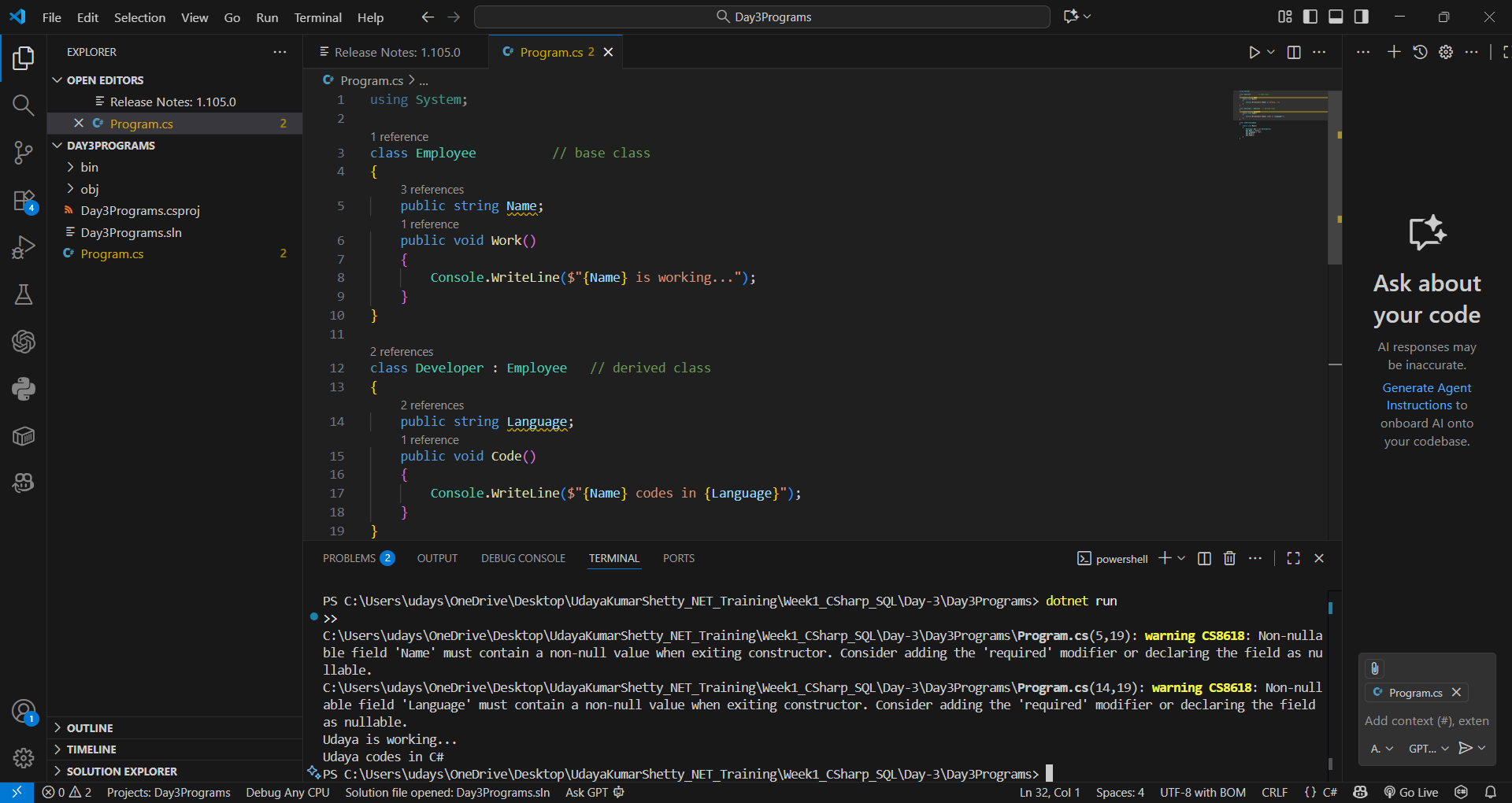
* CLR (Common Language Runtime) manages memory automatically.
* **Garbage Collector (GC)** removes unreferenced objects.
* You can force GC using:
* GC.Collect();
* GC.WaitForPendingFinalizers();
* But normally, it’s handled automatically by the runtime.

**SUMMARY TABLE**

|  |  |  |
| --- | --- | --- |
| **Concept** | **Keyword** | **Example** |
| Inheritance | : | class B : A { } |
| Polymorphism | virtual, override | Method overriding |
| Abstraction | abstract, interface | Abstract classes, interfaces |
| Encapsulation | private, public | Data hiding |
| Static Class | static | Utility methods |
| Sealed Class | sealed | Prevent inheritance |
| Constructor | same as class name | Initialization |
| Destructor | ~ClassName() | Cleanup |
| Namespace | namespace | Logical grouping |

**Snapshots:**

**Inheritance**

****

**Polymorphism (Overloading + Overriding)**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Abstraction**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Encapsulation**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Access Modifiers Example**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Constructors & Destructors**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Static Class & Members**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Sealed Class**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Namespace Example**

**A screenshot of a computer

AI-generated content may be incorrect.**