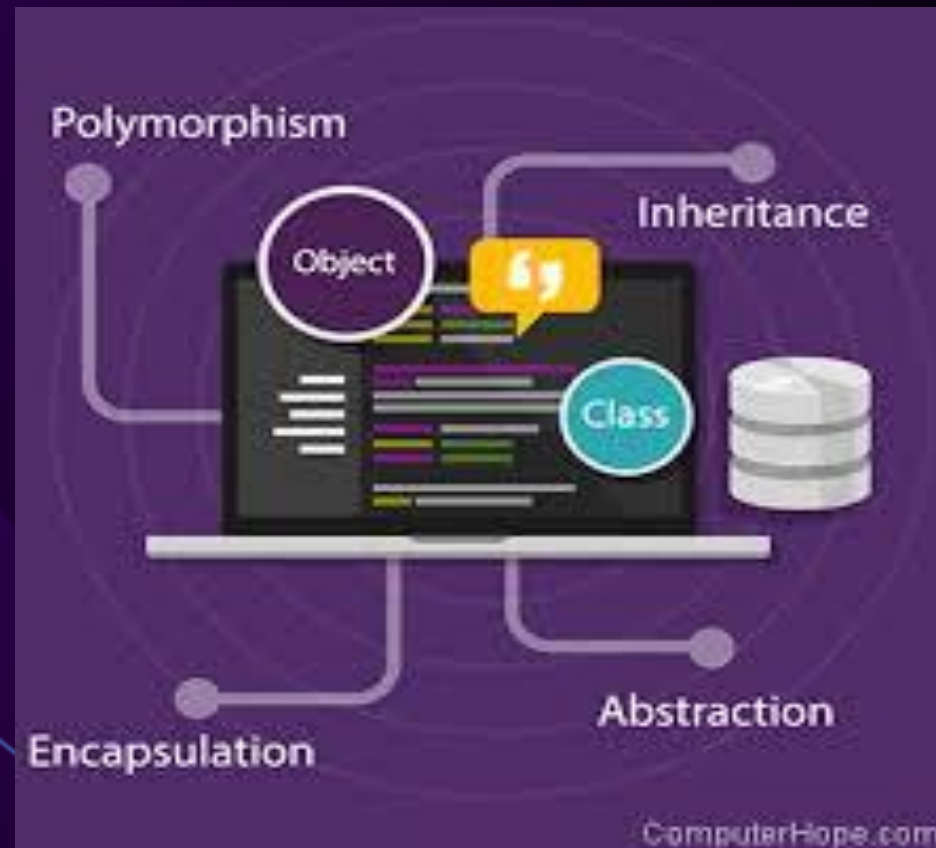



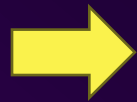
OBJECT ORIENTED PROGRAMMING



INTRODUCTION: PROCEDURAL AND OOPS

 programming paradigm based on procedures or routines (also called functions). Step by step execution

1. Code is organized into functions.



2. Global Data concept. No data Privacy



3. Not well organized

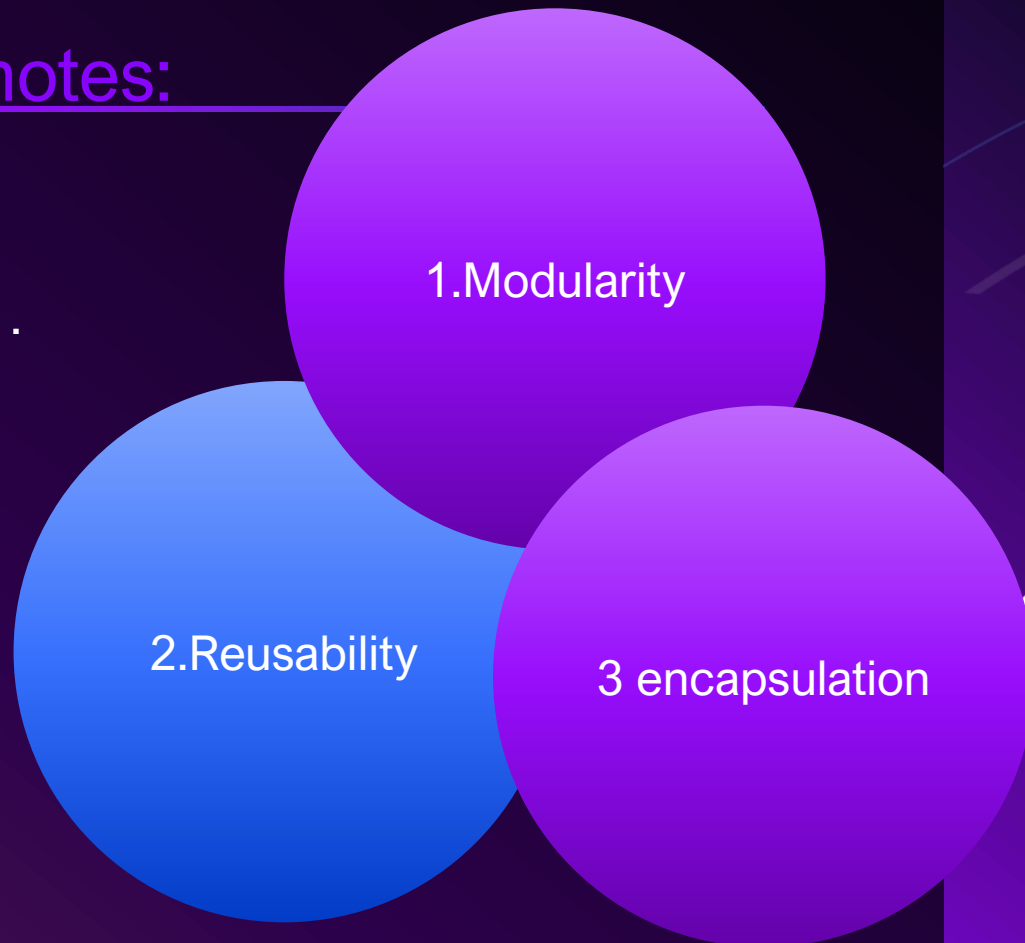


4. Can be unmanageable with project grows

OBJECT ORIENTED PROGRAMMING

OOP is a programming paradigm based on the concept of "objects", which contain both data and behavior.

It promotes:



Class

Data

Behaviour

PROCEDURAL VS OBJECT ORIENTED

```
internal class Program
{
    static void Main(string[] args)
    {
        DisplayResult();
    }

    static void DisplayResult()
    {
        // Output logic here
    }
}
```

```
class ResultDisplayer
{public void DisplayResult() {// Output logic here}
}
class Program
{
    static void Main(string[] args)
    {
        ResultDisplayer displayer = new
        ResultDisplayer();
        displayer.DisplayResult();
    }
}
```

PROCEDURAL VS OBJECT ORIENTED

Properties	procedural programming	Object oriented programming
Focus	Functions / Procedures	Objects and Classes
Data Handling	Global data access	Encapsulated in objects
Code Reusability	Limited	High (via inheritance, polymorphism)
Examples	C, Pascal	C#, Java, C++
Maintainability	Harder for large systems	Easier

The Four Pillars



1. Encapsulation – Combines data and related methods within a single class, restricting direct access using access modifiers
For example, private, public, and protected in C# to control what is accessible from outside the class.

```
class Person
{
    private string name; // can't be accessed directly from outside
    public string Name    // controlled access
    {
        get { return name; }
        set { name = value; }
    }
}
```

2. Abstraction: Hides complex internal logic and exposes only essential features — like using abstract or interface in C# to define what to do, not how.

```
abstract class Animal {  
    public abstract void Speak();  
}
```


3.Inheritance: Allows a class to inherit properties and methods from another class — like using : and base keywords in C# to promote code reuse.

```
class Animal {  
  
    public void Eat() { }  
  
}  
  
class Dog : Animal {  
  
    public void Bark() { }  
  
}
```

4. Polymorphism : One interface, multiple implementations . Achieved using

1.Method overriding(Run time Polymorphism)

2. Method overloading(Compile time Polymorphism)

Method Overriding

```
class Animal {  
  
    public virtual void Speak() {}  
  
}  
  
class Cat : Animal {  
  
    public override void Speak() {}  
  
}
```

Method Overloading

```
class Calculator  
{  
    public void Add(int a, int b) { }  
  
    public void Add(double a, double b) { }  
  
    public void Add(int a, int b, int c) { }  
}
```

CLASS AND OBJECT

Class: A class is a user-defined blueprint from which objects are created.

Object: Instance of a class. It represents a real-world entity.

```
class Car {  
    public string Color;  
    public void Drive() {  
        Console.WriteLine("Car is driving.");  
    }  
}
```

Class Definition

```
Car myCar = new Car();  
myCar.Color = "Red";  
myCar.Drive();
```

Object Creation

Constructor:

- A constructor is a special method that is called when an object is created.
- ~~same name as class name~~
- no return type.

Types of Constructor

1. Default constructor
2. Parameterized constructor
3. Static Constructor

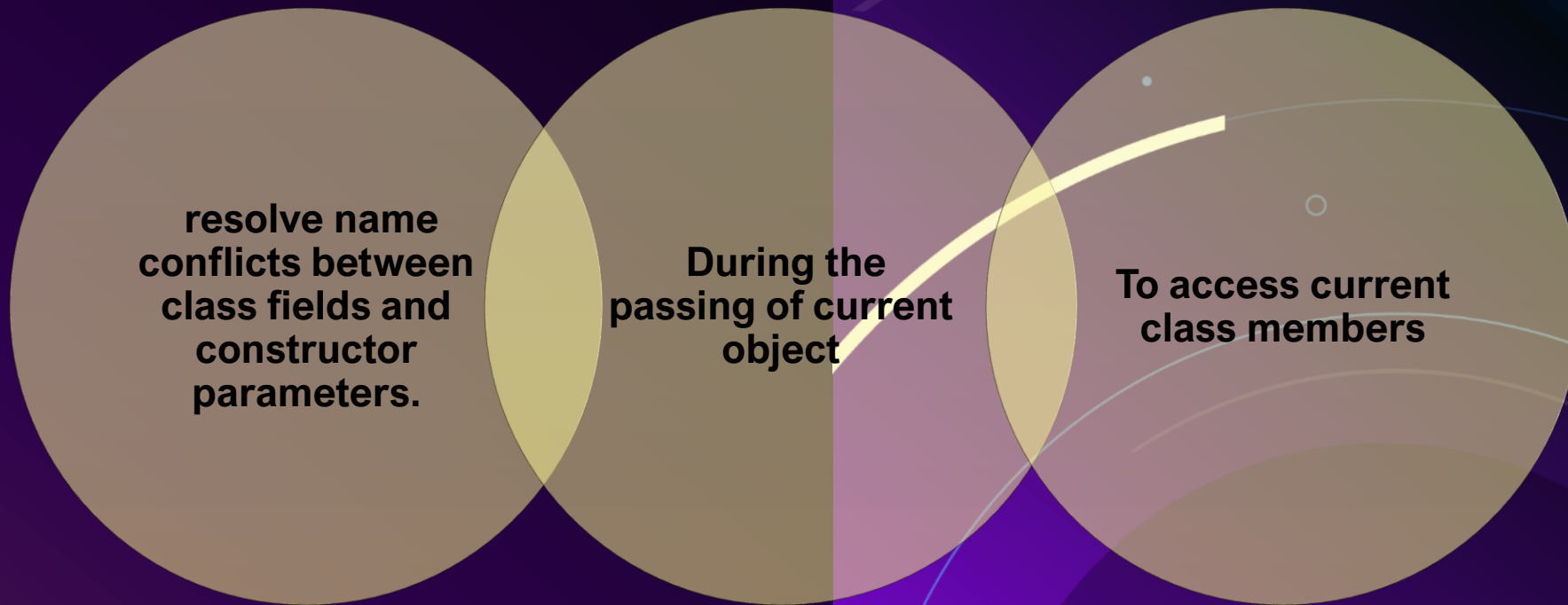
```
class Student
{
    public string Name;
    public Student(string name)
    {
        Name = name;
    }
}
```

this KEYWORD

Definition:

this refers to the current instance of the class in which it is used.

Use Cases of this:



DESTRUCTORS IN C#

Definition:

Destructors are used to clean up resources before the object is destroyed.

Key Points:

- Defined using a tilde (~) and the class name.
- Called by the garbage collector automatically.
- Rarely used.

PROPERTIES IN C#

Definition:

Properties provide a flexible way to read, write, or compute the values of **private fields** in a class while maintaining encapsulation.

Types of Properties:

Standard Properties: Includes a private field with explicit get and set blocks. Useful when custom logic is needed inside the accessors.

Auto-Implemented Properties: No need for a separate backing field; C# handles it internally. Ideal for simple get-set scenarios without additional logic.

STANDARD PROPERTIES

Example:

```
class Person {  
    private int age;  
    public int Age {  
        get { return age; }  
        set {  
            if (value >= 0)  
                age = value;  
        }  
    }  
}
```

AUTO-IMPLEMENTED PROPERTIES

Auto-implemented properties allows to declare properties quickly without writing a separate backing field.

~~The compiler generates a hidden, private field to store the value internally.~~

Use

- 1.Simplifies code by reducing boilerplate.
- 2.Ideal when no extra logic is needed in get or set
- 3.Improves readability and maintainability.

STANDARD PROPERTIES and AUTO-IMPLEMENTED PROPERTIES

○

```
private string _name;

public string Name {
    get { return _name; }
    set { _name = value; }
}
```

```
public string Name { get; set; }
```

For ReadOnly variable :

```
Public string Name{ get ;
Private set; }
```

○

●

•

GUID

Guid (Globally Unique Identifier) is a 128-bit unique identifier used to uniquely identify objects or records across systems.

Use cases:-

- 1.To generate unique IDs for database records.
- 2.Used in distributed systems where uniqueness is critical.
- 3.Often used as primary keys in systems where auto-increment IDs are not reliable.

- 1.Initializes a new instance of the Guid structure.

```
public static Guid NewGuid();
```

- 2.Print two unique global identifier

```
Guid g = Guid.NewGuid();  
Console.WriteLine(g);  
Console.WriteLine(Guid.NewGuid());
```

REGEX

In C#, Regex is a class in the System.Text.RegularExpressions namespace used to match, search, and manipulate strings using regular expressions.

<https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference>

```
using System;
using System.Text.RegularExpressions;

class Program
{
    static void Main()
    {
        string input = "Email: test@example.com";
        Regex emailRegex = new
        Regex(@"\b\w+@\w+\.\w+\b");

        Console.WriteLine(emailRegex.IsMatch(input) ?
        "Valid email found." : "No valid email found.");
    }
}
```

ACCESS MODIFIERS

Member of the Class:

Public -> complete Access

Private (Default) -> Accessible in same class only.

Protected -> Accessible in same class and child classes.

For Class:

Internal(Default) -> Accessible in same Assembly.

Public -> Accessible Outside the Assembly also .
Specially needed during developing Libraries.

Protected Internal -> Accessible in Same Assembly and by derived classes.

INHERITANCE

Allows a class to inherit members from another class.

Syntax

```
class Animal
{
    public void Eat() => Console.WriteLine("Animal eats");
}

class Dog : Animal
{
    public void Bark() => Console.WriteLine("Dog barks");
}
```

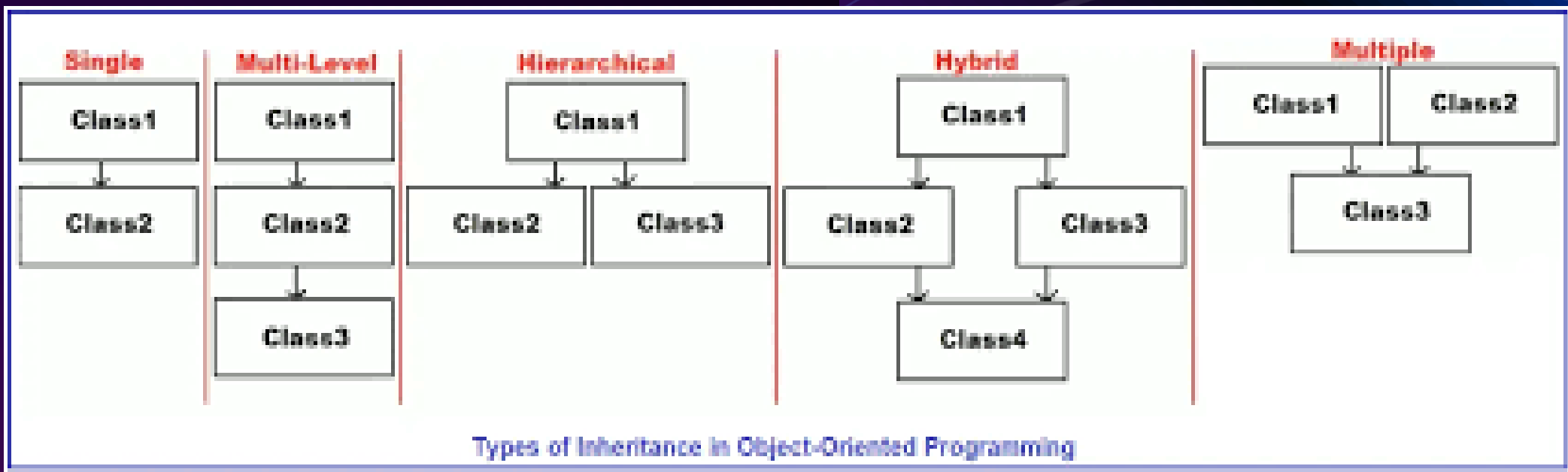
=> Is called Lambda operator Allows **short, single-line** definitions for methods, properties, constructors.

Also Used to create **anonymous functions** (inline methods), often passed to LINQ or delegates.

All class are inherited from **System.Object()** class implicitly

TYPES OF INHERITANCE

Multiple and hybrid inheritance is not supported in c# we can achieve similar functionalities using interfaces



base KEYWORD

The **base** keyword in C# is used to access members (methods, constructors, or fields) of a base (parent) class from within a derived (child) class.

If the parent class has only a parameterized constructor and no default constructor, then the child class constructor will throw an error unless we explicitly call the base constructor using the base keyword, since by default, C# tries to call the default constructor of the base class.

```
class Parent
{
    public Parent(string msg)
    {
        Console.WriteLine("Parent: " + msg);
    }
}

class Child : Parent
{
    public Child() : base("Hello from child") // Explicitly calling base constructor
    {
        Console.WriteLine("Child created");
    }
}
```

METHOD OVERRIDING

Method overriding allows a derived class to provide a specific implementation of a method that is already defined in its base class.

Key Points:

1. Use virtual keyword in base class method.
2. Use override keyword in derived class method.
3. Enables runtime polymorphism.

```
class Animal
{
    public virtual void Speak()
    {
        Console.WriteLine("Animal speaks");
    }
}

class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Dog barks");
    }
}
```

IS-A AND HAS-A RELATIONSHIP

The IS-A relationship represents inheritance, where a derived class inherits from a base class. It describes a generalization-specialization relationship. Improves code Reusability

```
class Animal{  
    public void Eat()=>Console.WriteLine("Eating");  
}  
  
class Dog : Animal // Dog IS-A Animal  
{  
    public void Bark() => Console.WriteLine("Barking...");  
}
```

The "Has-A" relationship, also known as composition or aggregation, describes a relationship where one class contains a reference to an instance of another class. This signifies that an object of the one class "has" an object of the another class .

```
class Engine {  
    void start() {  
        System.out.println("Engine started.");  
    }  
}  
  
class Car {  
    private Engine engine; // Car Has-A Engine  
  
    public Car() {  
        this.engine = new Engine(); // Initialize the Engine  
    }  
}
```

SO WHY HAS-A RELATIONSHIP??..

During creating an Employee class and want to store two different addresses:

1.ResidentialAddress and

2.OfficeAddress

So Instead of inheriting from 2 Address class (which wouldn't be Possible s, **Inheritance only allows one base class in C#** and also logically an employee is not an address). compose the Employee class using the Address class ie use Has A relationship.

SEALED CLASS

A sealed class is a class that cannot be inherited. Once a class is marked sealed, no other class can derive from it. Sealed method can not be overridden . Sealed class Only for Instance create.

Useful in case of security or design reason

```
sealed class FinalReport  
{  
    public void Generate() => Console.WriteLine("Report generated.");  
}  
  
class CustomReport : FinalReport { } // ❌ Will give error
```

STATIC CLASS AND STATIC MEMBERS

Static Class

1. cannot be instantiated (As only static members no need to instantiate)

2. can contain only static members (methods, properties, fields, etc.).

3. Useful for utility /helper class.

```
static class MathHelper {  
  
    public static int Square(int x) => x * x;  
  
}
```

Example of System Defined Static Class:

System.Math() , **System.Console()**

Static Members

1. Belong to the class, not instances.

2. Shared across all instances.

```
class Counter {  
    public static int Count = 0;  
    public Counter() {  
        Count++;  
    }  
}
```


STATIC CONSTRUCTOR

1. Initializes static members.
2. Called once, automatically before any access to other members .

```
class MyClass{  
  
    static int staticCounter;  
  
    int instanceld;  
  
    static MyClass(){  
  
        staticCounter = 100;  
  
        Console.WriteLine("Static constructor  
called.")}
```

```
        instanceld = ++staticCounter;  
        Console.WriteLine($"Instance constructor  
called. ID = {instanceld}");  
    }  
}  
class Manager  
{  
    public Static void main()  
    {  
        MyClass obj1 = new MyClass();  
        MyClass obj2 = new MyClass();  
    }  
}
```

Static constructor called.
Instance constructor called. ID = 101
Instance constructor called. ID = 102

PARTIAL CLASS

1. partial keyword allows a class to be split across multiple files or sections.
2. All parts must use the partial keyword and be within the same namespace and accessibility.
3. The compiler merges all partial class parts into a single class.
4. Used in tools like WinForms, Entity Framework, etc., to separate user code and generated code.

```
using System;
```

```
public partial class MyClass// File1  
{  
    public void MethodA() =>  
        Console.WriteLine("Method A called");  
}
```

```
public partial class MyClass// File2  
{  
    public void MethodB() =>  
        Console.WriteLine("Method B called");  
}
```

```
class Program  
{  
    static void Main()  
    {  
        MyClass obj = new MyClass();  
        obj.MethodA();  
        obj.MethodB();  
    }  
}
```

ABSTRACT CLASS

1. Used to define a base class with incomplete implementation.
2. Cannot be instantiated directly (as there is abstract method inside class so can't be instantiated).
3. Can contain both abstract (no body) and non-abstract (with body) members.

```
abstract class Animal
{
    public abstract void MakeSound(); // abstract method
    public void Eat() => Console.WriteLine("Eating...");
}
```

ABSTRACT METHOD

1. Declared in abstract class.
2. No body (implementation) in base class.
3. Must be overridden in derived class.
4. Used in interface

```
class Dog : Animal {  
    public override void MakeSound() {  
        Console.WriteLine("Bark");  
    }  
}
```

Object.ToString() method

1. `Object.ToString()` by default returns the fully qualified name of the class (like `ConsoleApp3.Program`)
2. When we want print an object directly (e.g., `Console.WriteLine(obj)`), it internally calls `ToString()`.

```
Program obj = new Program();  
Console.WriteLine(obj);           // Calls obj.ToString()  
Console.WriteLine(obj.ToString()); // Explicit call
```



ConsoleApp3.Program
ConsoleApp3.Program

2. Overriding ToString() in a Custom Class:

The Employee class overrides `ToString()`. This allows printing of actual employee details instead of just a class name.

```
public override string ToString()  
{  
    return $"{EmpId}, {Name}, {Salary}";  
}
```



10256, Scott, 45000

OBJECT INITIALIZER

An Object Initializer allows to assign values to properties of an object at the time of creation, without explicitly calling a constructor or writing multiple lines.

Reduce unnecessary code and provide more readable, clear syntax.

```
Person person = new Person {Name = "Alice",Age =30};
```

Using Object Initializer

```
Person person = new Person();  
person.Name = "Alice";  
person.Age = 30;
```

Using Constructor



THANK YOU