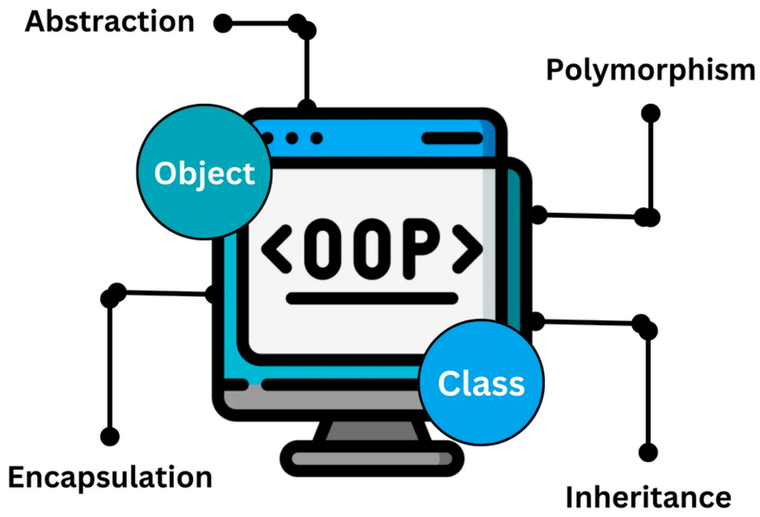
Session 3 : OOP Recap

Md Abdullah All Naim

Software Engineer Level II
Associate Program Manager @ CodeCamp by Astha.IT

Table of Contents



Programming Paradigm

A programming paradigm is a **methodology or style of programming** that defines how programs are structured and how problems are **solved**.

Programming paradigms have evolved to manage complexity, improve performance, and enhance developer productivity.

Each paradigm **Procedural, Object-Oriented, Functional, and Declarative** provides distinct ways to model problems, each with its own advantages and trade-offs.

Selecting the appropriate paradigm depends on the **problem domain, system requirements, and long-term maintainability** of the project.

Procedural vs OOP

Procedural Programming

Procedural Programming is a paradigm where:

- Program is written as a **sequence of steps**
- Logic is divided into functions (procedures)
- Data is separate from functions
- **Functions operate on shared or global data**
- Focus is on how the program works

👉 Control flow is more important than data structure.

❌ Issues

- balance can be **modified** from anywhere
- Business rules are not strongly **protected**
- **Debugging** becomes difficult in large codebases

```
// Shared data
double balance = 0;

// Procedure
void Deposit(double amount)
{
    if (amount > 0)
        balance += amount;
}

// Procedure
void Withdraw(double amount)
{
    if (amount <= balance)
        balance -= amount;
}

// Procedure
double GetBalance()
{
    return balance;
}
```

Procedural vs OOP

Object-Oriented Programming (OOP)

Object-Oriented Programming is a paradigm where:

- Program is organized around objects
- Objects contain data + behavior
- Data is encapsulated and protected
- **Focus is on who is responsible for what**
- Models real-world entities naturally

```
class BankAccount
{
    // Encapsulated data
    private double _balance;
    public void Deposit(double amount)
    {
        if (amount <= 0)
            throw new ArgumentException("Invalid deposit amount");

        _balance += amount;
    }

    public void Withdraw(double amount)
    {
        if (amount > _balance)
            throw new InvalidOperationException("Insufficient balance");

        _balance -= amount;
    }

    public double GetBalance()
    {
        return _balance;
    }
}
```

✅ Benefits

- Data is **protected**
- Rules are enforced inside the object
- Easy to extend and maintain
- Safer for team development

Why OOP?

1. Modularity and Organization

- Organizes code into **logical units**, making it easier to understand.

2. Reusability

- Allows code **reuse** through inheritance and shared components.

3. Maintainability

- Simplifies **debugging and testing** by isolating changes to specific classes.

4. Scalability

- Supports building **large, complex applications** that can grow over time.

5. Security and Encapsulation

- Protects data by controlling **access and preventing misuse.**

6. Collaboration

- Enables teams to work **independently** on different parts of the code.

CLASS & OBJECT

Class and Object are the **core building blocks** of Object-Oriented Programming (OOP).

A class is a blueprint or template that defines:

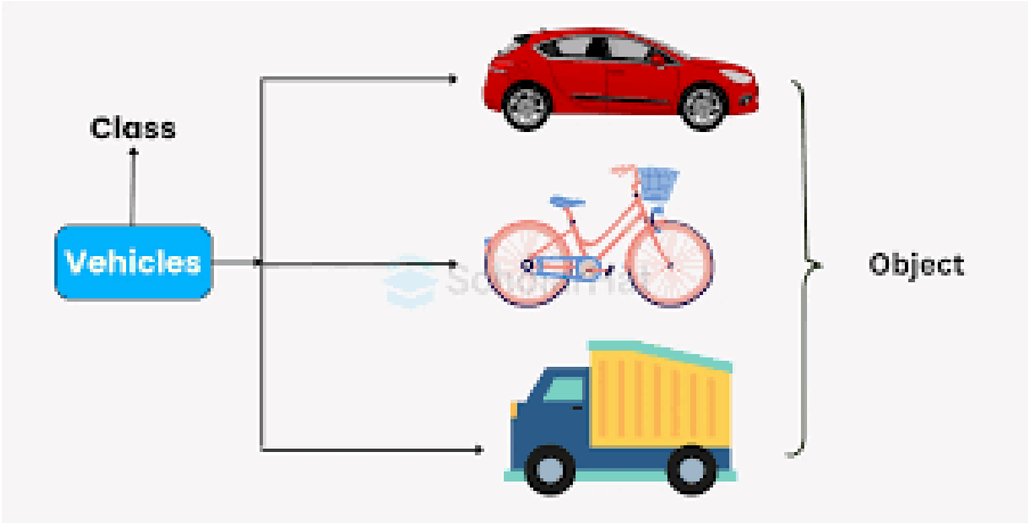
- **Properties (variables / attributes)**
- **Behaviors (methods / functions)**

👉 **A class does not occupy memory until an object is created.**

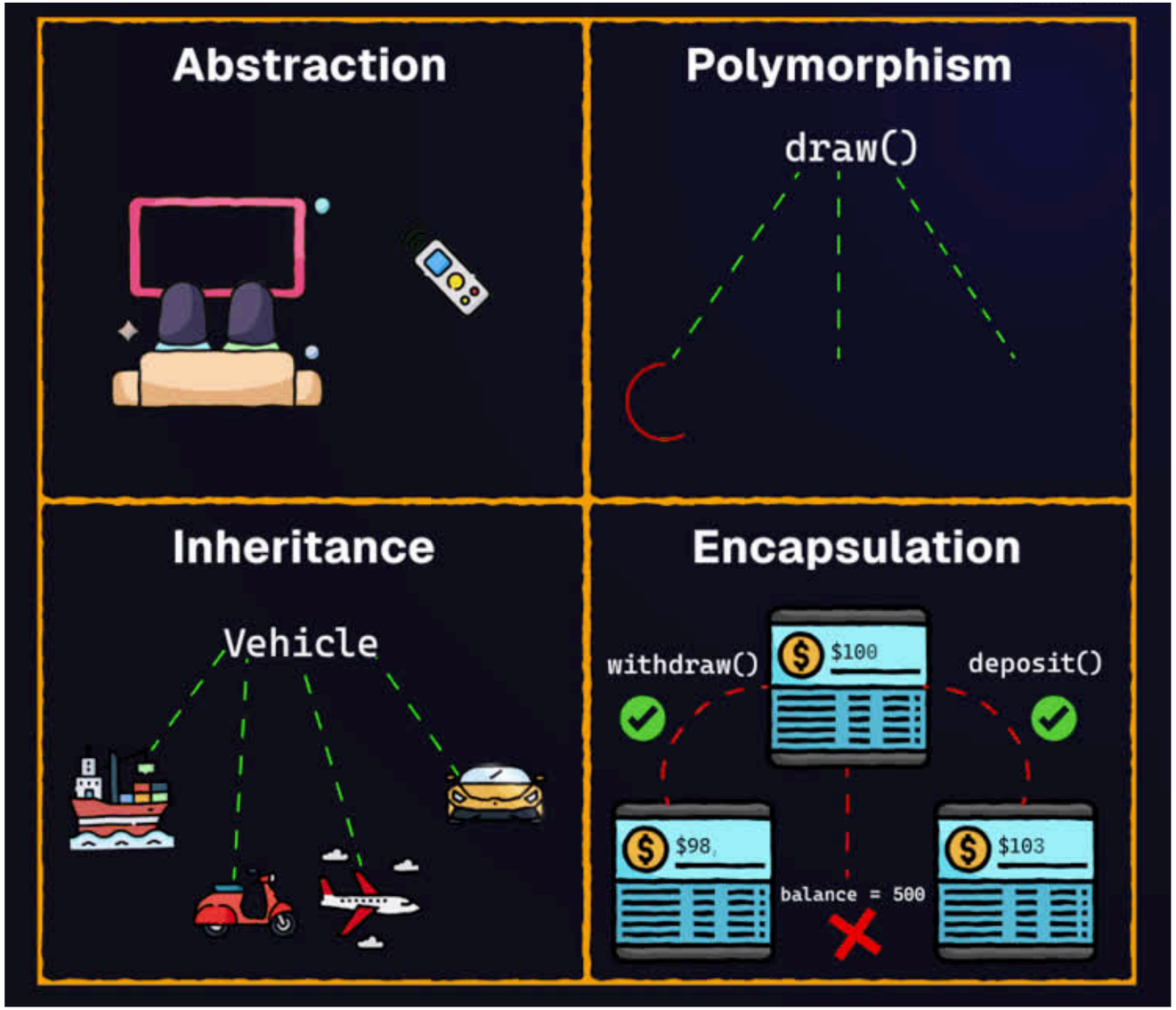
🚗 Object

An object is a real instance of a class. It:

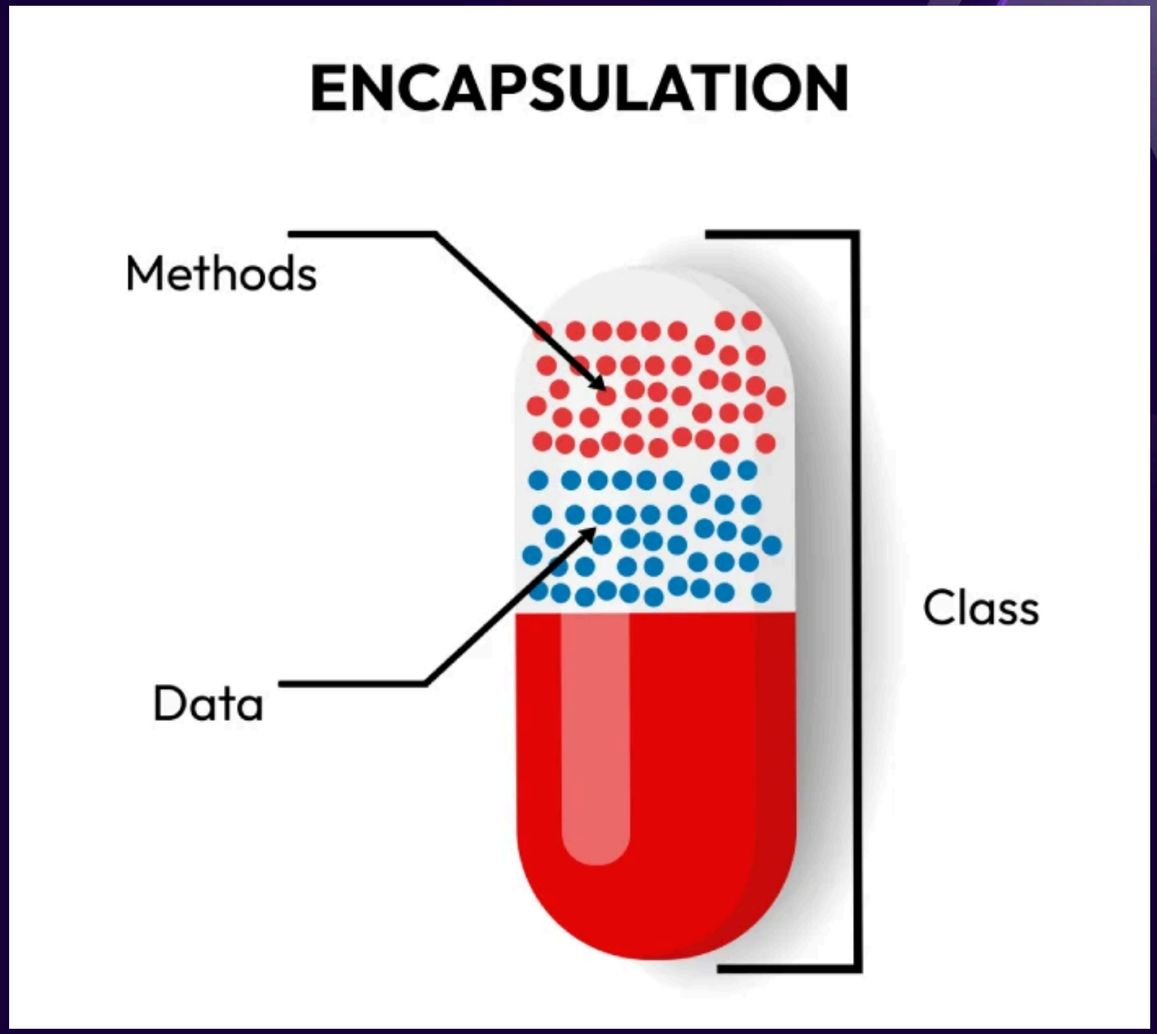
- **Occupies memory**
- Has actual values
- Can call class methods



OOP PILLARS


Abstraction
Polymorphism
draw()
Inheritance
Vehicle
Encapsulation
withdraw()
$100
deposit()
$98
$103
balance = 500

Encapsulation



ENCAPSULATION

Encapsulation means **wrapping data and methods together** and **restricting direct access** to internal details.

Key Principles:

- **Hide** internal state using private fields
- Expose controlled access through public properties (**minimizes coupling**)
- **Validate** data before allowing changes
- Improves **security, modularity, and maintainability**.

ENCAPSULATION

Access modifiers control visibility of fields and methods:

Modifier	Access Level
public	Accessible everywhere
private	Accessible only within the class
protected	Accessible within the class and derived classes
internal	Accessible within the same assembly
protected internal	Accessible in derived classes or same assembly

ENCAPSULATION

Property Accessors (get, set, init) :

Properties provide controlled access to private fields.

- get → Read access
- set → Write access with optional validation
- init → **Can be set only during object initialization (C# 9+)**

```
class Person
{
    private int _age; // Private field
    public int Age
    {
        get { return _age; } // Getter: allows read access
        set
        {
            if (value < 0 || value > 120)
                throw new ArgumentException("Invalid age"); // Validation
            _age = value; // Only valid data is assigned
        }
    }
}
```

```
class Student
{
    public string Name { get; init; }
    // Can be set only during object initialization
}

var s = new Student { Name = "Naim" };
// s.Name = "Rahman"; // ❌ Error: cannot modify after init
```

ENCAPSULATION

Auto-implemented vs Full Properties

Auto-Implemented Properties are a short syntax where:

- The compiler automatically creates a hidden backing field
- You don't write validation or logic
- Best for simple data holding

```
class Student
{
    public string Name { get; set; }
    public int Age { get; set; }
}


# ✔ Behind the scenes, the compiler creates something like:
private string _name;
private int _age;
```

✅ Why they exist

- **Reduce** boilerplate code
- **Improve** readability
- **Faster** development

Use Auto-implemented Properties When:

- **No validation** or business rules are needed
- The property only stores and returns a value
- Creating:
  - **DTOs**
  - **Models**

ENCAPSULATION

Auto-implemented vs Full Properties

Full Property

- Gives full control over data access
- Uses an explicit backing field
- Allows custom logic in get and set

Use Full Properties When:

- **Validation** is required
- Data is **sensitive** (Price, Age, Balance)
- Business **rules** must be enforced
- Value depends on **logic or calculation**

```
public class Product
{
    private decimal _price;
    public string Name { get; set; }

    public decimal Price
    {
        get { return _price; }
        set
        {
            if (value < 0)
                throw new ArgumentException("Price cannot be negative.");

            _price = value;
        }
    }
}
```

Encapsulation (Example)

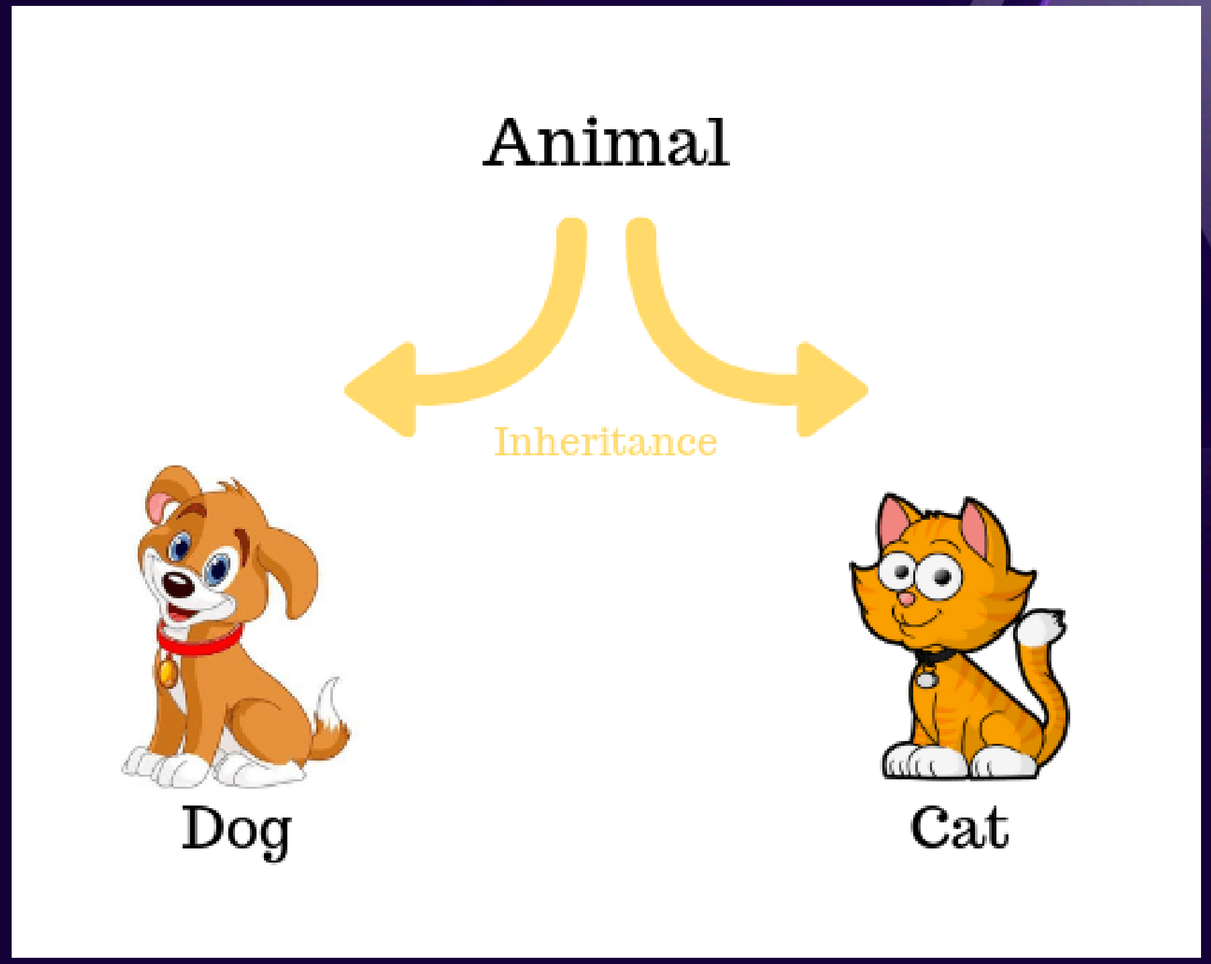
```
class BankAccount
{
    private decimal _balance; // Private field: cannot be accessed directly from outside

    public BankAccount(decimal initialBalance) // Constructor to initialize account
    {
        if (initialBalance < 0)
            throw new ArgumentException("Balance cannot be negative"); // Protect invariant
        _balance = initialBalance;
    }

    public void Withdraw(decimal amount) // Public method controls changes
    {
        if (amount <= 0)
            throw new ArgumentException("Invalid withdrawal amount"); // Validate input
        if (_balance < amount)
            throw new InvalidOperationException("Insufficient funds"); // Protect state

        _balance -= amount; // Update internal state safely
    }
    public decimal Balance => _balance; // Public getter provides controlled read access
}
```

Inheritance



INHERITANCE

Inheritance is an OOP principle that allows a class (derived) to reuse members(fields and methods) of another class (base).

Benefits of Inheritance

Code Reusability

- Inheritance allows a derived class to reuse methods and properties of a base class.
- Reduces code duplication and saves development time.

Polymorphism Support

- Inheritance enables polymorphism, letting the program decide at runtime which implementation to use.

Logical Hierarchy / “Is-A” Relationship

- Inheritance models real-world relationships.
- Makes code easier to read, maintain, and understand.

```
class Animal { }
class Dog : Animal { } // Dog “is-an” Animal
```

INHERITANCE

Easier Maintenance

- Shared behavior resides in one place (base class).
- Changing behavior in the base class automatically propagates to derived classes.

Extensibility

- Derived classes can add new features without changing the base class.
- Supports open/closed principle (software entities open for extension but closed for modification).

Reduces Code Redundancy

- Common code in base class eliminates the need to write the same code in multiple classes.

INHERITANCE

virtual, override, and new keyword differences

1. virtual

- Used in base class to indicate that a method can be overridden in a derived class.

2. override

- Used in derived class to replace the base class's virtual method.

3. new

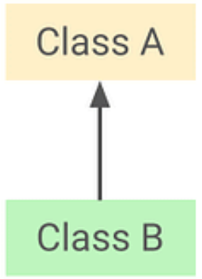
- Used in derived class to hide the base class method instead of overriding it.

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

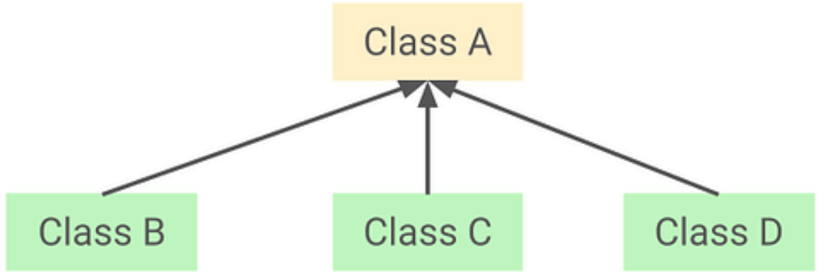
```
class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Dog barks");
    }
}

// Usage
Animal a = new Dog();
a.Speak();  // Output: Dog barks
```

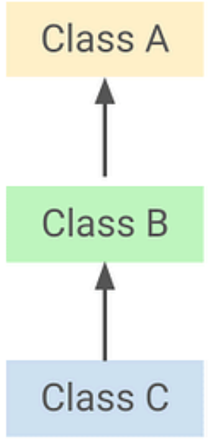
INHERITANCE



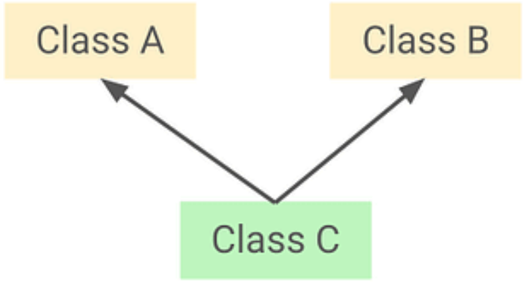
Single Inheritance



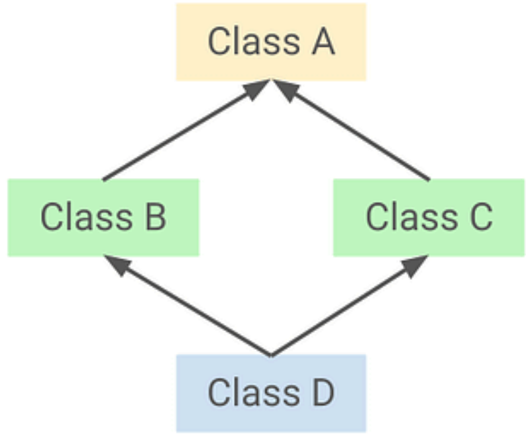
Hierarchical inheritance



Multilevel Inheritance



Multiple Inheritance



Hybrid Inheritance

TYPES OF INHERITANCE

Single Inheritance

- A class inherits from one base class

```
class Vehicle { }
class Car : Vehicle { } // Single inheritance
```

Multilevel Inheritance

- Chain of inheritance

```
class Vehicle { }
class Car : Vehicle { }
class SportsCar : Car { } // Multilevel inheritance
```

Hierarchical Inheritance

- Multiple classes inherit from the same base class

```
class Vehicle { }
class Car : Vehicle { }
class Bike : Vehicle { }
```

⚠️ **Note:**

- C# does not allow multiple inheritance with classes (diamond problem)
- **Use interfaces to achieve multiple inheritance**

TYPES OF INHERITANCE

Hybrid Inheritance is a combination of two or more types of inheritance:

- Single Inheritance: Derived class inherits from one base class
- Multilevel Inheritance: Chain of inheritance (A → B → C)
- Hierarchical Inheritance: Multiple classes inherit from the same base

Hybrid inheritance occurs when a system combines these types in a single class hierarchy.

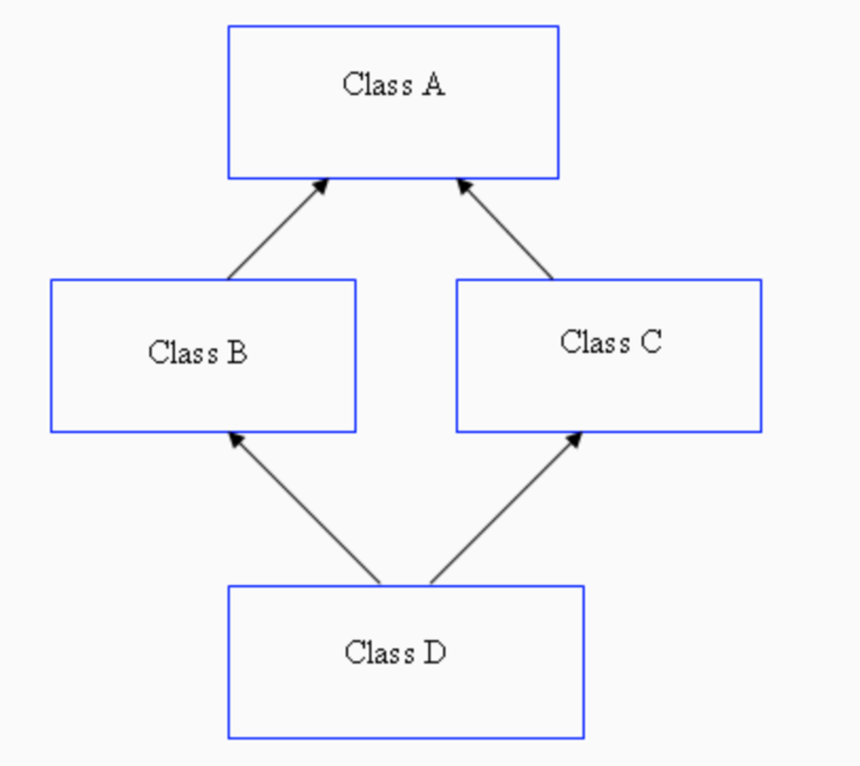
```
public interface IEngine
{
    void StartEngine();
}

public class Vehicle
{
    public void Drive()
    {
        Console.WriteLine("The vehicle is driving.");
    }
}

public class Car : Vehicle, IEngine
{
    public void StartEngine()
    {
        Console.WriteLine("The car engine starts.");
    }
}

public class ElectricCar : Car
{
    public void ChargeBattery()
    {
        Console.WriteLine("The electric car is charging.");
    }
}
```

DO YOU KNOW
WHAT IS DIAMOND PROBLEM?



- If B and C both override a method from A, and D inherits from both

Inheritance (Example)

```
// Base class
public class BaseEntity
{
    public int Id { get; set; }
    public DateTime CreatedAt { get; set; }
    public DateTime UpdatedAt { get; set; }

    public BaseEntity()
    {
        CreatedAt = DateTime.Now;
        UpdatedAt = DateTime.Now;
    }

    public void UpdateTimestamp()
    {
        UpdatedAt = DateTime.Now;
        Console.WriteLine($"Entity {Id} updated at {UpdatedAt}");
    }
}

// Subclass: Product
public class Product : BaseEntity
{
    public string Name { get; set; }
    public decimal Price { get; set; }

    public Product(int id, string name, decimal price)
    {
        Id = id;
        Name = name;
        Price = price;
    }

    public void DisplayProduct()
    {
        Console.WriteLine($"Product: {Name}, Price: {Price:C}");
    }
}
```

```
// Subclass: Customer
public class Customer : BaseEntity
{
    public string FullName { get; set; }
    public string Email { get; set; }

    public Customer(int id, string fullName, string email)
    {
        Id = id;
        FullName = fullName;
        Email = email;
    }

    public void DisplayCustomer()
    {
        Console.WriteLine($"Customer: {FullName}, Email: {Email}");
    }
}
```

DRAWBACKS OF INHERITANCE

- **Tight Coupling**
- **Code Reusability:** If Bird has a fly() method, a subclass Penguin inherits it even though penguins can’t fly.
- **Inflexibility :** Once a class hierarchy is set, it’s difficult to change without affecting all subclasses
- **Multiple Inheritance:** Creates ambiguity. When a class inherits from more than one parent, conflicts may occur
- **Fragile base class problem :** When a class inherits from more than one parent, conflicts may occur

DRAWBACKS OF INHERITANCE

- **Liskov substitution principle violations :** You should be able to use a child class wherever the parent class is expected—without breaking things.

```
class Bird {
    void fly() { print("Flying"); }
}

class Penguin : Bird {
    void fly() { print("Can't fly"); }  // violates expectation
}
```

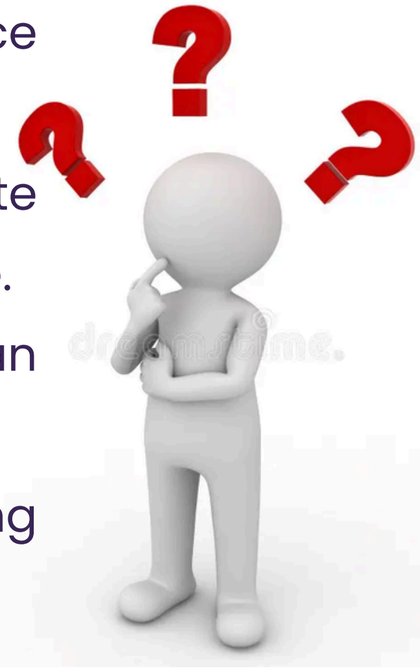
- **Parent: Bird 🐦 -> can fly ✈️**
- **Child: Penguin 🐧 -> can’t fly ❌**

WHEN TO USE INHERITANCE

WHEN TO USE INHERITANCE

A rule of thumb is, choose inheritance when an object needs all of the behavior of the parent class.

- **Clear "Is-A" Relationship** : Use inheritance when there is a natural "is-a" relationship between the base and derived classes. For example, a Dog is an Animal.
- **Shared Behavior** : When multiple classes share common behavior, inheritance allows you to define the shared functionality in a base class.
- **Polymorphism** : Use inheritance to enable polymorphism, allowing you to write flexible code that works with objects of different types through a common interface.
- **Code Reuse** : When you want to reuse code across multiple classes, inheritance can help reduce duplication.
- **Extending Functionality** : Use inheritance to extend the functionality of an existing class without modifying its code.



But who can solve the drawbacks of inheritance?

THANK YOU!