🔒 Encapsulation in C# (OOP Pillar 1)

Aspect	Description	Code Example / Notes
Definition	Encapsulation is the bundling of data (fields) and behaviors (methods) that operate on the data into a single unit (class), hiding the internal state.	Makes it harder for external code to depend on internal implementation.
Purpose	- Protect internal state- Control access to fields- Improve code maintainability and flexibility	
Access Modifiers	- `private`: accessible only within the same class- `public`: accessible from outside- `protected`: accessible in derived classes- `internal`: accessible within same assembly	Used to restrict or allow access to members.
Properties	Use C# properties (`get` / `set`) instead of exposing fields directly. This adds a layer of control over how a field is read/written.	`csharp private int _age; public int Age { get => _age; set => _age = value; }`
Auto-Properties	C# allows auto-implemented properties to reduce boilerplate when no extra logic is needed.	`public string Name { get; set; }`
Read-only properties	Encapsulation can be enforced using read-only (`get` only) or `private set`.	`public string Id { get; private set; }`
Backing Fields	Traditional way of implementing a property using a private field	`private string _name; public string Name { get => _name; set => _name = value; }`
Constructor Use	Set required fields via constructor to ensure objects are always in a valid state	`public Car(string brand) { _brand = brand; }`
Validation in Setters	Encapsulation allows adding validation logic inside `set` accessor	`set { if (value < 0) throw new Exception(); }`

Immutability	Achieved by making fields `readonly` and using constructor injection only	Useful for domain-driven design and thread safety.
.NET 6 Enhancements	Init-only setters: `public string Name { get; init; }` allow setting values only during object creation	Promotes immutability with concise syntax

🧬 Inheritance in C# (OOP Pillar 2)

Aspect	Description	Code Example / Notes
Definition	Inheritance allows a class (derived/child class) to acquire members (fields, properties, methods) from another class (base/parent class).	Enables **code reuse**, **polymorphism**, and **hierarchical relationships**.
Base Class	A class that provides common functionality for one or more derived classes.	`public class Animal { public void Eat() { } }`
Derived Class	A class that inherits from another class and optionally adds new members or overrides base behavior.	`public class Dog : Animal { public void Bark() { } }`
Syntax	Use `:` to indicate inheritance in class declaration.	`class Child : Parent`
Access to Base Members	Derived classes can access all `public` and `protected` members of the base class.	`base.Method()` to explicitly call base implementation
sealed Keyword	Prevents further inheritance from a class.	`public sealed class FinalClass { }`
virtual Keyword	Marks a method or property in the base class as *overridable*.	`public virtual void Speak() { }`
override Keyword	Allows the derived class to replace the base implementation of a `virtual` method or property.	`public override void Speak() { Console.WriteLine("Bark"); }`
new Keyword	Hides a base class member with a new implementation in the derived class, without overriding.	`public new void Speak() { }`
Object Class	All types in C# ultimately derive from the `System.Object` class.	Includes methods like `ToString()`, `Equals()`, `GetHashCode()`, etc.

Constructors	Base class constructors can be called using `base()` in the derived class.	`public Dog(string name) : base(name) { }`
Abstract Classes	Cannot be instantiated. Used to define a base class that enforces a contract for subclasses.	`public abstract class Shape { public abstract void Draw(); }`
Inheritance Limit	C# supports **single inheritance** for classes (a class can inherit from only one base class).	However, **multiple interfaces** can be implemented.
IS-A Relationship	Inheritance expresses a strong **IS-A** relationship (e.g., `Dog` IS-A `Animal`).	Check with `is` or `as` keyword

---

👇 Example

```
public class Animal
{
    public virtual void Speak() => Console.WriteLine("Some sound");
}

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

Animal animal = new Dog();
animal.Speak(); // Output: Bark
```

🎭 Polymorphism in C# (OOP Pillar 3)

Aspect	Description	Code Example / Notes
Definition	Polymorphism allows objects of different types to be treated as instances of a common base type, while behaving differently.	“Many forms” – same interface, different behavior.
Types	1. **Compile-time (static)** polymorphism – achieved via method overloading.2. **Runtime (dynamic)** polymorphism – achieved via method overriding.	
Method Overloading	Same method name, different parameter list in the **same class**.	`void Print(int x)void Print(string s)`
Method Overriding	Derived class provides a specific implementation of a base class’s `virtual` method using `override`.	See `Speak()` example below.
virtual Keyword	Marks a method in the base class as eligible to be overridden.	`public virtual void Draw() { ... }`
override Keyword	Used in the derived class to redefine a virtual method.	`public override void Draw() { ... }`
new Keyword	Hides the base class method without participating in polymorphism.	`public new void Draw() { ... }`
abstract Classes	Often used to define base classes with abstract methods (no implementation) that **must** be implemented in derived classes.	`abstract void Draw();`
Interfaces	Interfaces define a contract that any implementing class must fulfill, allowing polymorphic use of objects across unrelated class hierarchies.	`public interface IDrawable { void Draw(); }`
Base Class References	Polymorphism enables storing derived objects in base class variables.	`Shape s = new Circle(); s.Draw();`

Casting	Polymorphic objects can be cast from base to derived (via `as`, `is`, or direct casting) to access specific members.	`(Circle)s` or `s as Circle`
The `object` Class	Since everything in C# derives from `object`, polymorphism even applies at the top level.	You can treat any object as `object`, and then downcast.

👇 Example – Runtime Polymorphism with `virtual` / `override`

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

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

public class Cat : Animal
{
    public override void Speak() => Console.WriteLine("Meow");
}

public void MakeItSpeak(Animal animal)
{
    animal.Speak(); // Calls the overridden method depending on runtime type
}

MakeItSpeak(new Dog()); // → Bark
MakeItSpeak(new Cat()); // → Meow
```

⚖️ Overloading vs Overriding

Feature	Overloading	Overriding
Where it happens	Within the **same class**	Across **base/derived class** hierarchy
Purpose	Multiple forms of the same method name	Change behavior in derived class
Parameters	Must be different	Must have **same** signature
Polymorphic?	❌ No (static dispatch at compile-time)	✅ Yes (dynamic dispatch at runtime)
Uses `virtual`/`override`?	❌ No	✅ Yes

🧩 Abstraction in C# (OOP Pillar 4)

Aspect	Description	Code Example / Notes
Definition	Abstraction is the process of hiding **complex implementation details** and exposing only **essential features** of an object.	Think: *what* an object does, not *how* it does it.
Purpose	- Reduce complexity- Focus on relevant behavior- Enforce contracts between objects	Helps create cleaner and more understandable interfaces.
How It's Achieved	In C#:1. **Abstract classes**2. **Interfaces**	Both can define a "contract" that derived classes must follow.
Abstract Class	Cannot be instantiated. Can contain **abstract** (no implementation) and **concrete** methods.	`public abstract class Shape { public abstract void Draw(); }`
Abstract Method	Must be overridden in derived class.	`public abstract void Draw();`
Interface	Pure abstraction — defines **only** members, with **no implementation** (until C# 8 default members).	`public interface IDrawable { void Draw(); }`
Keyword	`abstract` (class or method), `interface`	Interface = full abstraction, abstract class = partial abstraction
Usage	Expose only the operations you want the caller to see. Hide internal logic.	`void Save(IStorage storage)` – caller doesn't need to know if it's File or DB storage.
Polymorphism Link	Abstraction often works **with polymorphism** – allows different implementations of the same interface or base class.	Enables code like `List<IAnimal>.ForEach(a => a.Speak());`
Real World Analogy	A TV remote — you interact with buttons (abstracted behavior), but you don't know (or care) how signals are transmitted internally.	

👇 Abstract Class Example

```
public abstract class Shape
{
    public string Color { get; set; }

    public abstract void Draw(); // Abstract method
}

public class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine($"Drawing a {Color} circle.");
    }
}
```

👇 Interface Example

```
public interface IShape
{
    void Draw();
}

public class Rectangle : IShape
{
    public void Draw()
    {
        Console.WriteLine("Drawing a rectangle.");
    }
}
```

🆚 Abstract Class vs Interface (Summary Table)

Feature	Abstract Class	Interface
Instantiation	❌ No	❌ No
Constructors	✅ Yes	❌ No
Fields	✅ Can have fields	❌ No fields
Implementation	✅ Can have implementation	❌ Only declarations (C# < 8)
Multiple Inheritance	❌ Only one abstract class allowed	✅ Multiple interfaces
Accessibility Modifiers	✅ Members can have access modifiers	❌ All members are public by default
When to Use	When base implementation is shared	When only a contract is needed