OOPs

Object-Oriented Programming (OOPs) is a programming **paradigm** or methodology, not a specific language. It's a way of designing software by modeling real-world entities as **objects**.

Instead of focusing on procedures and logic (like in procedural programming), OOPs focuses on the objects that developers want to manipulate. An object contains both **data** (attributes or properties) and **code** (methods or functions) that operate on that data.

- **Real-world Analogy:** Think of a car.
    - **Data/Attributes:** Its color, model, current speed, and fuel level.
    - **Code/Methods:** Functions like `startEngine()`, `accelerate()`, `brake()`.

The primary goal of OOPs is to bind the data and the functions that work on it together so that no other part of the code can access this data, except for that function. This helps in creating more manageable, reusable, and secure code.

Four Pillars of OOPs

These four principles are the foundation of any object-oriented programming language.

1. Encapsulation

Encapsulation is the practice of bundling an object's data (attributes) and the methods that operate on that data into a single unit, known as a **class**. It also involves restricting direct access to an object's data, a principle called **data hiding**.

- To protect the internal state of an object from outside interference. You expose a public interface (methods) for others to interact with your object, but you keep the internal implementation details private.
- Think of a medical capsule. The plastic casing (the class) encloses the medicine (the data) and protects it from the outside world. You don't interact with the powdered medicine directly; you interact with it via the capsule.
- A `BankAccount` class has a `private` attribute called `balance`.
    - we cannot directly change the balance from outside the class (e.g., `myAccount.balance = 1000000`).
    - Instead, we must use `public` methods like `deposit(amount)` and `withdraw(amount)`. These methods can contain important logic, like checking for sufficient funds before allowing a withdrawal, thus protecting the integrity of the `balance` data.

2. Abstraction

Abstraction means hiding the complex implementation details and showing only the essential features to the user. It focuses on **what** an object does rather than **how** it does it.

- To simplify complex systems by modeling classes appropriate to the problem and providing a clear, simple interface for them.
- Driving a car. You use the steering wheel, accelerator, and brake pedals to drive. You don't need to know the complex details of the engine, the fuel injection system, or the transmission to operate the car. The pedals and steering wheel are an **abstraction** that hides the complexity.
- When you use a library or framework, you call a method like `database.connect()`. You know it connects to the database, but you don't need to know the low-level details of how it creates network sockets, handles authentication packets, or manages connection pools. The `connect()` method is an abstraction.

3. Inheritance

Inheritance is a mechanism where a new class (the **child** or **subclass**) acquires the properties and behaviors (attributes and methods) of an existing class (the **parent** or **superclass**). This creates an "is-a" relationship.

- To promote code reusability. You can write common code once in a parent class and have multiple child classes reuse it, while also being able to add their own unique features.
- Think of vehicles. A `Car` *is-a* `Vehicle`. A `Bicycle` *is-a* `Vehicle`.
    - The `Vehicle` parent class can have attributes like `speed` and `color`, and methods like `move()`.
    - The `Car` child class inherits these and adds its own specific attributes like `numberOfDoors` and methods like `playRadio()`.
    - The `Bicycle` child class also inherits from `Vehicle` but adds a method like `ringBell()`.
- In GUI frameworks, you might have a base `Button` class. You could then create specialized buttons like `ImageButton` or `SubmitButton` that inherit all the basic properties of a `Button` but add their own specific logic.

4. Polymorphism

The word polymorphism means "many forms". It is the ability of a method or an object to take on many different forms. In practice, it means a single action can be performed in different ways.

- To allow a single interface to be used for a general class of actions. This makes the code more flexible and decoupled.
- Think of the action "speak". A person will "speak" differently than a dog, and a dog will "speak" differently than a cat. The action is the same, but the implementation is different based on the object performing it.
- Types of polymorphism
    1. **Compile-time Polymorphism (Method Overloading):** This happens when you have multiple methods in the *same class* with the same name but **different parameters** (different type or number of arguments). The compiler decides which method to call at compile time.
        - `void add(int a, int b)`
        - `void add(double a, double b)`
        - `void add(int a, int b, int c)`
    2. **Run-time Polymorphism (Method Overriding):** This happens when a **subclass** provides a specific implementation for a method that is already defined in its **superclass**. The method name and parameters must be the same. The decision on which method to execute is made at runtime.
        - Parent class `Animal` has a `makeSound()` method.
        - Child class `Dog` **overrides** `makeSound()` to print "Woof!".
        - Child class `Cat` **overrides** `makeSound()` to print "Meow!".