Day 2: Object Oriented Programming (OOPs) Basics

1. The Evolution of Programming (Why do we need OOPs?)

Before understanding OOPs, we must understand the problems with previous programming paradigms.

- **Machine Language (Binary):**
  - Code was written in 0s and 1s.
  - **Problem:** Highly error-prone and tedious. If you miss one zero, the whole code breaks.
- **Assembly Language:**
  - Used mnemonics (English keywords) like MOV, ADD.
  - **Problem:** Still tightly coupled with hardware. Not scalable for large applications.
- **Procedural Programming (e.g., C language):**
  - Introduced functions, loops, and logic blocks. It viewed code as a "Recipe" (Step-by-step instructions).
  - **Problem:** Good for small tasks but failed in **Real World Modeling** and **Data Security** for complex enterprise applications (like Uber, Zomato).

---

2. Introduction to OOPs

Definition: OOPs is a programming paradigm based on the concept of "Objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).

Core Idea:

Just like the real world is made of objects (Laptop, Car, Human) that interact with each other, our code should also be a collection of interacting objects.

Class vs. Object

- **Object:** A real-world entity. It has two main things:
  1. **Characteristics (Attributes/Data):** What it looks like (Brand, Color, Engine type).
  2. **Behavior (Methods/Functions):** What it does (Start, Stop, Accelerate).
- **Class:** The **Blueprint** or Template to create an object.

Real-World Example: The Car

Feature	Real World (Car)	Programming Term
Blueprint	The design on paper	Class
Entity	The actual physical car	Object
Details	Brand: BMW, Color: Black	Attributes (Variables)
Actions	Drive, Brake, Honk	Methods (Functions)

---

3. The 4 Pillars of OOPs

(This lecture covers the first two pillars: Abstraction and Encapsulation)

Pillar 1: Abstraction
"Hiding the complexity, showing only the essential."

Definition: Abstraction hides unnecessary implementation details from the user and only shows the necessary features of the object.

Real-World Analogy (The Car Dashboard):

- To drive a car, you only need to know how to use the **Steering Wheel, Accelerator, and Brake**.
- You **DO NOT** need to know how the internal combustion engine works, how the fuel is injected, or how the pistons move.
- The Car provides an **Interface** (Dashboard) to interact with, hiding the complex **Implementation** (Engine) under the hood.

Code Concept:

We use **Abstract Classes** and **Interfaces**.

- We define methods like startEngine() but we don't need to show the complex logic of *how* the engine starts to the user who just wants to drive.

Key Takeaway: Abstraction is about **Data Hiding** (Hiding design complexity).

Pillar 2: Encapsulation
"wrapping data and methods into a single unit (Capsule)."

Definition: Encapsulation is the bundling of data (variables) and methods (functions) that operate on the data into a single unit (Class). It also restricts direct access to some of an object's components.

Real-World Analogy (The Medical Capsule):

- Just like a capsule holds medicine inside a coating, a **Class** holds variables and methods together.

Why Encapsulation? (Data Security)

- **Problem:** If you have a car, you shouldn't be able to manually change the **Odometer** (Kilometer reading) from 50,000km to 0km. That would be fraud.
- **Solution:** We make the critical data **Private** so it cannot be accessed directly from outside.
- We provide **Public methods (Getters and Setters)** to access or modify it with rules (Validation).

Access Modifiers:

1. **Public:** Accessible by everyone.
2. **Private:** Accessible ONLY within the class. (Secure).
3. **Protected:** Accessible within the class and child classes.

Code Example (C++ Style Logic):

C++

```
class SportsCar {
   // 1. Data is Private (Security)
   private:
      int currentSpeed;

   // 2. Methods are Public (Interface)
   public:
      // Setter: Allows controlling how data is changed
      void setSpeed(int speed) {
         if (speed < 0) {
            return; // Validation: Speed cannot be negative
         }
         currentSpeed = speed;
      }

      // Getter: Allows reading the data
      int getSpeed() {
         return currentSpeed;
      }
```

```
};
```

> **Key Takeaway:** Encapsulation is about **Data Security** (Protecting data from unauthorized access).

Summary: Abstraction vs. Encapsulation

Feature	Abstraction	Encapsulation
Focus	Focuses on **Hiding Complexity** (Design level).	Focuses on **Data Security** (Implementation level).
Goal	"I don't need to know how it works."	"I shouldn't be able to mess with the internal data directly."
Example	Using a TV remote without knowing electronics.	Wrapping wires in plastic so you don't get shocked.