Object-Oriented Programming (OOP) - Detailed Explanation

1) What is OOP?

Object-Oriented Programming is a programming paradigm where a program is modeled as a collection of objects. Each object has state (attributes/data) and behavior (methods/functions). This approach makes it easier to map real-world entities into software.

2) Key Terms

- Class: A blueprint or template. - Object/Instance: A real-world example created from a class. - Attributes (State): Properties of the object. - Methods (Behavior): Actions the object can perform. - Constructor/Destructor: Special functions for setting up/cleaning an object. - Visibility: Controls access (public/private/protected).

3) Four Pillars of OOP

1. Encapsulation: Bundling data and methods together; hides internal details. 2. Abstraction: Shows only necessary features, hides complexity. 3. Inheritance: One class can reuse properties and behavior of another. 4. Polymorphism: Same interface behaves differently depending on the object.

4) Relationships Between Classes

- Association: General relationship (Teacher — Student). - Aggregation: Weak 'has-a' relationship (Library has Books). - Composition: Strong 'has-a' relationship (House has Rooms). - Dependency: 'Uses-a' relationship (ATM uses Account temporarily).

5) Design Principles (SOLID)

S - Single Responsibility: One job per class. O - Open/Closed: Open to extension, closed to modification. L - Liskov Substitution: Subclasses should replace base classes seamlessly. I - Interface Segregation: Clients only depend on what they use. D - Dependency Inversion: Depend on abstractions, not implementations.

6) Common Design Patterns

- Singleton: Only one instance exists. - Factory: Centralized object creation. - Observer: Publisher–subscriber model. - Strategy: Switch algorithms at runtime.

7) Benefits of OOP

- Modularity, Reusability, Real-world modeling, Maintainability, Testability.

8) Drawbacks of OOP

- Over-engineering, Performance overhead, Complex hierarchies, Not always best for functional/data-heavy tasks.

9) Best Practices

- Prefer composition over deep inheritance. - Keep classes small and cohesive. - Expose only what’s necessary. - Use interfaces/abstractions for flexibility.

10) Real-World Example (Bank System)

- Class: Account → has balance, owner. - Methods: deposit, withdraw. - Inheritance: SavingsAccount extends Account. - Polymorphism: Different accounts handle withdraw differently. - Composition: Bank has Branches, Branch has Accounts.