OOPS CONCEPTS

Classes and Objects:

Classes act as blueprints defining properties (data) and behaviors (methods) of objects. Objects are individual instances of these classes, holding their own unique data and behaving according to class methods.

Encapsulation:

Combining data and related methods within a class restricts unauthorized access, protecting data integrity and promoting modularity.

Inheritance:

Classes can inherit properties and methods from parent classes, enabling code reuse and specialization.

Polymorphism:

Objects can take on different forms and provide different responses based on their type, allowing for flexible and dynamic code.

Abstraction:

Programs focus on essential characteristics and functionalities, hiding complex internal details for cleaner and easier code management.

Association:

Represents relationships between objects. Imagine a "Student" object associated with a "Course" object through an enrollment system. Associations can be one-to-one, one-to-many, or many-to-many.

Aggregation

Aggregation, also known as "has-a" relationship, represents a whole-part relationship between two classes in object-oriented programming. In Java, one object (the whole) owns another object (the part) and manages its lifecycle.

Composition:

Stronger than aggregation, where the "part" objects become entirely dependent on the "whole" object and cannot exist independently. Imagine a "Document" object composing "Paragraph" objects that wouldn't have meaning outside the document.

SOLID

1**). Single Responsibility Principle (SRP):**

A class should have one and only one reason to change. This means each class should focus on a single specific task or responsibility, avoiding being cluttered with unrelated functionalities.

2. Open-Closed Principle (OCP):

Software entities (classes, modules, functions) should be open for extension, but closed for modification. This allows you to add new functionalities without changing existing code, improving flexibility and avoiding potential regressions.

3. Liskov Substitution Principle (LSP):

Objects of a supertype should be replaceable with objects of its subtypes without altering the correctness of the program. This ensures subclasses follow the same behaviors as their parent classes, promoting consistency and reliability.

4. Interface Segregation Principle (ISP):

Clients shouldn't be forced to depend on methods they don't use. Large interfaces should be split into smaller, more specific ones, allowing clients to depend only on the functionalities they actually need.

5. Dependency Inversion Principle (DIP):

High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions. This decouple high-level code from low-level implementation details, improving flexibility and making it easier to change implementation without impacting the higher levels.