CodInClub

Powered By BridgeLabz

Best Programming Practices

Encapsulation

- Use `private` access modifiers for class fields to restrict direct access.
- Provide `public` getter and setter methods to access and modify private fields.
- Implement validation logic in setters to ensure data integrity.
- Use `final` fields and avoid setters for immutable classes.
- Follow naming conventions for methods (e.g., `getX`, `setX`).

Polymorphism

- Program to an interface, not an implementation.
- Ensure overridden methods adhere to the base class method's contract.
- Avoid explicit casting; rely on polymorphic behavior.
- Leverage covariant return types for overriding methods.
- Keep inheritance hierarchies shallow to maintain simplicity.

Interfaces

- Use interfaces to define a contract or behavior.
- Prefer default methods only when backward compatibility or shared implementation is necessary.
- Combine interfaces to create modular, reusable behaviors.
- Favor composition over inheritance when combining multiple behaviors.

Abstract Classes

- Use abstract classes for shared state and functionality among related classes.
- Avoid overusing abstract classes; use them only when clear shared behavior exists.
- Combine abstract classes with interfaces to separate behavior and implementation.
- Avoid deep inheritance hierarchies; keep designs flexible and maintainable.

General Practices

- Follow Java naming conventions for classes, methods, and variables.
- Document code with comments and Javadoc to improve readability.
- Ensure consistency and readability by adhering to team or industry coding standards.
- Apply SOLID principles, particularly Single Responsibility and Interface Segregation. (We will learn it in coming days)

Tips for Implementation

- **Encapsulation**: Ensure all sensitive fields are private and accessed through well-defined getter and setter methods. Include validation logic where applicable.
- **Polymorphism**: Use abstract class references or interface references to handle objects of multiple types dynamically.
- **Abstract Classes**: Use them to define a common structure and behavior while deferring specific details to subclasses.
- **Interfaces**: Use them to define additional capabilities or contracts that are not tied to the class hierarchy.

Problem Statements

1. Employee Management System

- **Description**: Build an employee management system with the following requirements:
  - Use an abstract class `Employee` with fields like `employeeId`, `name`, and `baseSalary`.
  - Provide an abstract method `calculateSalary()` and a concrete method `displayDetails()`.
  - Create two subclasses: `FullTimeEmployee` and `PartTimeEmployee`, implementing `calculateSalary()` based on work hours or fixed salary.
  - Use encapsulation to restrict direct access to fields and provide getter and setter methods.
  - Create an interface `Department` with methods like `assignDepartment()` and `getDepartmentDetails()`.
  - Ensure polymorphism by processing a list of employees and displaying their details using the `Employee` reference.

---

2. E-Commerce Platform

- **Description**: Develop a simplified e-commerce platform:
  - Create an abstract class `Product` with fields like `productId`, `name`, and `price`, and an abstract method `calculateDiscount()`.
  - Extend it into concrete classes: `Electronics`, `Clothing`, and `Groceries`.
  - Implement an interface `Taxable` with methods `calculateTax()` and `getTaxDetails()` for applicable product categories.
  - Use encapsulation to protect product details, allowing updates only through setter methods.

- Showcase polymorphism by creating a method that calculates and prints the final price (price + tax - discount) for a list of `Product`.

---

3. Vehicle Rental System

- **Description**: Design a system to manage vehicle rentals:
  - Define an abstract class `Vehicle` with fields like `vehicleNumber`, `type`, and `rentalRate`.
  - Add an abstract method `calculateRentalCost(int days)`.
  - Create subclasses `Car`, `Bike`, and `Truck` with specific implementations of `calculateRentalCost()`.
  - Use an interface `Insurable` with methods `calculateInsurance()` and `getInsuranceDetails()`.
  - Apply encapsulation to restrict access to sensitive details like insurance policy numbers.
  - Demonstrate polymorphism by iterating over a list of vehicles and calculating rental and insurance costs for each.

---

4. Banking System

- **Description**: Create a banking system with different account types:
  - Define an abstract class `BankAccount` with fields like `accountNumber`, `holderName`, and `balance`.
  - Add methods like `deposit(double amount)` and `withdraw(double amount)` (concrete) and `calculateInterest()` (abstract).
  - Implement subclasses `SavingsAccount` and `CurrentAccount` with unique interest calculations.
  - Create an interface `Loanable` with methods `applyForLoan()` and `calculateLoanEligibility()`.
  - Use encapsulation to secure account details and restrict unauthorized access.
  - Demonstrate polymorphism by processing different account types and calculating interest dynamically.

---

5. Library Management System

- **Description**: Develop a library management system:
  - Use an abstract class `LibraryItem` with fields like `itemId`, `title`, and `author`.
  - Add an abstract method `getLoanDuration()` and a concrete method `getItemDetails()`.
  - Create subclasses `Book`, `Magazine`, and `DVD`, overriding `getLoanDuration()` with specific logic.
  - Implement an interface `Reservable` with methods `reserveItem()` and `checkAvailability()`.
  - Apply encapsulation to secure details like the borrower's personal data.
  - Use polymorphism to allow a general `LibraryItem` reference to manage all items, regardless of type.

---

6. Online Food Delivery System

- **Description**: Create an online food delivery system:
  - Define an abstract class `FoodItem` with fields like `itemName`, `price`, and `quantity`.
  - Add abstract methods `calculateTotalPrice()` and concrete methods like `getItemDetails()`.
  - Extend it into classes `VegItem` and `NonVegItem`, overriding `calculateTotalPrice()` to include additional charges (e.g., for non-veg items).
  - Use an interface `Discountable` with methods `applyDiscount()` and `getDiscountDetails()`.
  - Demonstrate encapsulation to restrict modifications to order details and use polymorphism to handle different types of food items in a single order-processing method.

---

7. Hospital Patient Management

- **Description**: Design a system to manage patients in a hospital:
  - Create an abstract class `Patient` with fields like `patientId`, `name`, and `age`.
  - Add an abstract method `calculateBill()` and a concrete method `getPatientDetails()`.
  - Extend it into subclasses `InPatient` and `OutPatient`, implementing `calculateBill()` with different billing logic.

- Implement an interface `MedicalRecord` with methods `addRecord()` and `viewRecords()`.
- Use encapsulation to protect sensitive patient data like diagnosis and medical history.
- Use polymorphism to handle different patient types and display their billing details dynamically.

---

8. Ride-Hailing Application

- **Description**: Develop a ride-hailing application:
  - Define an abstract class `Vehicle` with fields like `vehicleId`, `driverName`, and `ratePerKm`.
  - Add abstract methods `calculateFare(double distance)` and a concrete method `getVehicleDetails()`.
  - Create subclasses `Car`, `Bike`, and `Auto`, overriding `calculateFare()` based on type-specific rates.
  - Use an interface `GPS` with methods `getCurrentLocation()` and `updateLocation()`.
  - Secure driver and vehicle details using encapsulation.
  - Demonstrate polymorphism by creating a method to calculate fares for different vehicle types dynamically.