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.

```
import java.util.*;

abstract class Employee {
    private int employeeId;
    private String name;
    private double baseSalary;

    public Employee(int employeeId, String name, double baseSalary) {
        this.employeeId = employeeId;
```

```
        this.name = name;
        this.baseSalary = baseSalary;
    }

    public double getBaseSalary() {
        return baseSalary;
    }

    public abstract double calculateSalary();

    public void displayDetails() {
        System.out.println("Employee ID: " + employeeId);
        System.out.println("Name: " + name);
        System.out.println("Base Salary: " + baseSalary);
        System.out.println("Total Salary: " + calculateSalary());
    }
}

interface Department {
    void assignDepartment(String department);
    String getDepartmentDetails();
}

class FullTimeEmployee extends Employee implements Department {
    private String department;

    public FullTimeEmployee(int employeeId, String name, double baseSalary)
{
        super(employeeId, name, baseSalary);
    }

    @Override
    public double calculateSalary() {
        return getBaseSalary() * 1.2;     }

    @Override
    public void assignDepartment(String department) {
        this.department = department;
    }

    @Override
```

```
    public String getDepartmentDetails() {
        return "Department: " + department;
    }
}

class PartTimeEmployee extends Employee implements Department {
    private int workHours;
    private double hourlyRate;
    private String department;

    public PartTimeEmployee(int employeeId, String name, double hourlyRate,
int workHours) {
        super(employeeId, name, 0);
        this.hourlyRate = hourlyRate;
        this.workHours = workHours;
    }

    @Override
    public double calculateSalary() {
        return hourlyRate * workHours;
    }

    @Override
    public void assignDepartment(String department) {
        this.department = department;
    }

    @Override
    public String getDepartmentDetails() {
        return "Department: " + department;
    }
}

public class EmployeeManagementSystem {
    public static void main(String[] args) {
        List<Employee> employees = new ArrayList<>();
        FullTimeEmployee emp1 = new FullTimeEmployee(1, "Alice", 50000);
        emp1.assignDepartment("HR");

        PartTimeEmployee emp2 = new PartTimeEmployee(2, "Bob", 200, 20);
        emp2.assignDepartment("Support");
```

```
        employees.add(emp1);
        employees.add(emp2);

        for (int i = 0; i < employees.size(); i++) {
            employees.get(i).displayDetails();
            if (employees.get(i) instanceof Department) {
                Department dept = (Department) employees.get(i);
                System.out.println(dept.getDepartmentDetails());
            }
            System.out.println("--------------------------------");
        }
    }
}
```

---

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`.

```
import java.util.*;

abstract class Product {
    private int productId;
    private String name;
    private double price;

    public Product(int productId, String name, double price) {
        this.productId = productId;
```

```
        this.name = name;
        this.price = price;
    }

    public double getPrice() { return price; }

    public abstract double calculateDiscount();

    public void displayDetails() {
        System.out.println("ID: " + productId + ", Name: " + name + ",
Price: " + price);
    }
}

interface Taxable {
    double calculateTax();
    String getTaxDetails();
}

class Electronics extends Product implements Taxable {
    public Electronics(int id, String name, double price) {
        super(id, name, price);
    }

    public double calculateDiscount() {
        return getPrice() * 0.1;
    }

    public double calculateTax() {
        return getPrice() * 0.18;
    }

    public String getTaxDetails() {
        return "Electronics Tax: 18%";
    }
}

class Clothing extends Product implements Taxable {
    public Clothing(int id, String name, double price) {
        super(id, name, price);
    }
```

```
    public double calculateDiscount() {
        return getPrice() * 0.15;
    }

    public double calculateTax() {
        return getPrice() * 0.05;
    }

    public String getTaxDetails() {
        return "Clothing Tax: 5%";
    }
}

class Groceries extends Product {
    public Groceries(int id, String name, double price) {
        super(id, name, price);
    }

    public double calculateDiscount() {
        return getPrice() * 0.05;
    }
}

public class ECommercePlatform {
    public static void main(String[] args) {
        List<Product> products = new ArrayList<>();
        products.add(new Electronics(1, "Laptop", 50000));
        products.add(new Clothing(2, "T-shirt", 1000));
        products.add(new Groceries(3, "Rice", 2000));

        for (int i = 0; i < products.size(); i++) {
            Product p = products.get(i);
            p.displayDetails();

            double discount = p.calculateDiscount();
            double tax = (p instanceof Taxable) ?
((Taxable)p).calculateTax() : 0;

            System.out.println("Discount: " + discount);
            System.out.println("Tax: " + tax);
```

```
            System.out.println("Final Price: " + (p.getPrice() + tax -
discount));
            System.out.println("-----------------------------------");
        }
    }
}
```

---

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.

```
import java.util.*;

abstract class Vehicle {
    private String vehicleNumber;
    private String type;
    private double rentalRate;

    public Vehicle(String vehicleNumber, String type, double rentalRate) {
        this.vehicleNumber = vehicleNumber;
        this.type = type;
        this.rentalRate = rentalRate;
    }
```

```
    public double getRentalRate() { return rentalRate; }
    public String getVehicleNumber() { return vehicleNumber; }

    public abstract double calculateRentalCost(int days);
}

interface Insurable {
    double calculateInsurance();
    String getInsuranceDetails();
}

class Car extends Vehicle implements Insurable {
    public Car(String number, double rate) {
        super(number, "Car", rate);
    }

    public double calculateRentalCost(int days) {
        return getRentalRate() * days;
    }

    public double calculateInsurance() {
        return 1500;
    }

    public String getInsuranceDetails() {
        return "Car Insurance: ₹1500";
    }
}

class Bike extends Vehicle implements Insurable {
    public Bike(String number, double rate) {
        super(number, "Bike", rate);
    }

    public double calculateRentalCost(int days) {
        return getRentalRate() * days;
    }

    public double calculateInsurance() {
        return 500;
    }
```

```
    public String getInsuranceDetails() {
        return "Bike Insurance: ₹500";
    }
}

class Truck extends Vehicle implements Insurable {
    public Truck(String number, double rate) {
        super(number, "Truck", rate);
    }

    public double calculateRentalCost(int days) {
        return getRentalRate() * days + 1000; // Extra maintenance
    }

    public double calculateInsurance() {
        return 2500;
    }

    public String getInsuranceDetails() {
        return "Truck Insurance: ₹2500";
    }
}

public class VehicleRentalSystem {
    public static void main(String[] args) {
        List<Vehicle> vehicles = new ArrayList<>();
        vehicles.add(new Car("MH01AB1234", 1000));
        vehicles.add(new Bike("MH02XY9876", 300));
        vehicles.add(new Truck("MH03TR4567", 2000));

        for (int i = 0; i < vehicles.size(); i++) {
            Vehicle v = vehicles.get(i);
            System.out.println("Vehicle Number: " + v.getVehicleNumber());
            System.out.println("Rental Cost (5 days): ₹" +
v.calculateRentalCost(5));

            if (v instanceof Insurable) {
                System.out.println(((Insurable)v).getInsuranceDetails());
                System.out.println("Insurance Amount: ₹" +
((Insurable)v).calculateInsurance());
```

```
            }
            System.out.println("---------------------------------");
        }
    }
}
```

---

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.

```
abstract class BankAccount {
    private String accountNumber;
    private String holderName;
    private double balance;

    public BankAccount(String accountNumber, String holderName, double
balance) {
        this.accountNumber = accountNumber;
        this.holderName = holderName;
        this.balance = balance;
    }

    public void deposit(double amount) {
        balance += amount;
    }
```

```
    public void withdraw(double amount) {
        if (amount <= balance) balance -= amount;
        else System.out.println("Insufficient balance!");
    }

    public double getBalance() { return balance; }

    public abstract double calculateInterest();

    public void displayAccountInfo() {
        System.out.println("Account Number: " + accountNumber);
        System.out.println("Holder Name: " + holderName);
        System.out.println("Balance: ₹" + balance);
    }
}

interface Loanable {
    void applyForLoan(double amount);
    boolean calculateLoanEligibility();
}

class SavingsAccount extends BankAccount implements Loanable {
    public SavingsAccount(String accNo, String name, double bal) {
        super(accNo, name, bal);
    }

    public double calculateInterest() {
        return getBalance() * 0.04;
    }

    public void applyForLoan(double amount) {
        System.out.println("Savings Account Loan Applied: ₹" + amount);
    }

    public boolean calculateLoanEligibility() {
        return getBalance() >= 5000;
    }
}

class CurrentAccount extends BankAccount implements Loanable {
    public CurrentAccount(String accNo, String name, double bal) {
```

```
        super(accNo, name, bal);
    }

    public double calculateInterest() {
        return getBalance() * 0.02;
    }

    public void applyForLoan(double amount) {
        System.out.println("Current Account Loan Applied: ₹" + amount);
    }

    public boolean calculateLoanEligibility() {
        return getBalance() >= 10000;
    }
}

public class BankingSystem {
    public static void main(String[] args) {
        BankAccount[] accounts = {
            new SavingsAccount("S123", "Alice", 10000),
            new CurrentAccount("C456", "Bob", 15000)
        };

        for (int i = 0; i < accounts.length; i++) {
            accounts[i].displayAccountInfo();
            System.out.println("Interest: ₹" +
accounts[i].calculateInterest());

            if (accounts[i] instanceof Loanable) {
                Loanable loan = (Loanable) accounts[i];
                loan.applyForLoan(20000);
                System.out.println("Loan Eligibility: " +
loan.calculateLoanEligibility());
            }
            System.out.println("---------------------------------");
        }
    }
}
```

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.

```
abstract class LibraryItem {
    private int itemId;
    private String title;
    private String author;

    public LibraryItem(int itemId, String title, String author) {
        this.itemId = itemId;
        this.title = title;
        this.author = author;
    }

    public abstract int getLoanDuration();

    public void getItemDetails() {
        System.out.println("Item ID: " + itemId + ", Title: " + title + ",
Author: " + author);
    }
}

interface Reservable {
    void reserveItem();
    boolean checkAvailability();
}
```

```
class Book extends LibraryItem implements Reservable {
    private boolean available = true;

    public Book(int id, String title, String author) {
        super(id, title, author);
    }

    public int getLoanDuration() {
        return 14;
    }

    public void reserveItem() {
        available = false;
    }

    public boolean checkAvailability() {
        return available;
    }
}

class Magazine extends LibraryItem implements Reservable {
    private boolean available = true;

    public Magazine(int id, String title, String author) {
        super(id, title, author);
    }

    public int getLoanDuration() {
        return 7;
    }

    public void reserveItem() {
        available = false;
    }

    public boolean checkAvailability() {
        return available;
    }
}

class DVD extends LibraryItem implements Reservable {
```

```
    private boolean available = true;

    public DVD(int id, String title, String author) {
        super(id, title, author);
    }

    public int getLoanDuration() {
        return 3;
    }

    public void reserveItem() {
        available = false;
    }

    public boolean checkAvailability() {
        return available;
    }
}

public class LibraryManagementSystem {
    public static void main(String[] args) {
        LibraryItem[] items = {
            new Book(1, "Java Basics", "James Gosling"),
            new Magazine(2, "Tech Monthly", "John Smith"),
            new DVD(3, "Inception", "Christopher Nolan")
        };

        for (int i = 0; i < items.length; i++) {
            items[i].getItemDetails();
            System.out.println("Loan Duration: " +
items[i].getLoanDuration() + " days");

            if (items[i] instanceof Reservable) {
                Reservable res = (Reservable) items[i];
                res.reserveItem();
                System.out.println("Available after reservation? " +
res.checkAvailability());
            }
            System.out.println("---------------------------------");
        }
    }
```

```
}
```

---

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.

```
abstract class FoodItem {
    private String itemName;
    private double price;
    private int quantity;

    public FoodItem(String name, double price, int qty) {
        this.itemName = name;
        this.price = price;
        this.quantity = qty;
    }

    public double getPrice() { return price; }
    public int getQuantity() { return quantity; }

    public abstract double calculateTotalPrice();

    public void getItemDetails() {
```

```
        System.out.println("Item: " + itemName + ", Price: ₹" + price + ",
Qty: " + quantity);
    }
}

interface Discountable {
    double applyDiscount();
    String getDiscountDetails();
}

class VegItem extends FoodItem implements Discountable {
    public VegItem(String name, double price, int qty) {
        super(name, price, qty);
    }

    public double calculateTotalPrice() {
        return getPrice() * getQuantity();
    }

    public double applyDiscount() {
        return calculateTotalPrice() * 0.05;
    }

    public String getDiscountDetails() {
        return "5% Discount on Veg";
    }
}

class NonVegItem extends FoodItem implements Discountable {
    public NonVegItem(String name, double price, int qty) {
        super(name, price, qty);
    }

    public double calculateTotalPrice() {
        return (getPrice() + 20) * getQuantity(); // Additional charge for
non-veg
    }

    public double applyDiscount() {
        return calculateTotalPrice() * 0.1;
    }
```

```
    public String getDiscountDetails() {
        return "10% Discount on Non-Veg";
    }
}

public class FoodDeliverySystem {
    public static void main(String[] args) {
        FoodItem[] items = {
            new VegItem("Paneer Tikka", 200, 2),
            new NonVegItem("Chicken Biryani", 250, 1)
        };

        for (int i = 0; i < items.length; i++) {
            items[i].getItemDetails();
            double total = items[i].calculateTotalPrice();

            if (items[i] instanceof Discountable) {
                Discountable d = (Discountable) items[i];
                double discount = d.applyDiscount();
                System.out.println(d.getDiscountDetails());
                System.out.println("Total after Discount: ₹" + (total -
discount));
            }

            System.out.println("---------------------------------");
        }
    }
}
```

---

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.

```
abstract class Patient {
    private int patientId;
    private String name;
    private int age;

    public Patient(int id, String name, int age) {
        this.patientId = id;
        this.name = name;
        this.age = age;
    }

    public abstract double calculateBill();

    public void getPatientDetails() {
        System.out.println("Patient ID: " + patientId + ", Name: " + name +
", Age: " + age);
    }
}

interface MedicalRecord {
    void addRecord(String record);
    void viewRecords();
}

class InPatient extends Patient implements MedicalRecord {
    private double roomCharge = 3000;
    private int days = 5;
    private String record;

    public InPatient(int id, String name, int age) {
        super(id, name, age);
    }
```

```
    public double calculateBill() {
        return roomCharge * days;
    }

    public void addRecord(String record) {
        this.record = record;
    }

    public void viewRecords() {
        System.out.println("Record: " + record);
    }
}

class OutPatient extends Patient implements MedicalRecord {
    private double consultationFee = 500;
    private String record;

    public OutPatient(int id, String name, int age) {
        super(id, name, age);
    }

    public double calculateBill() {
        return consultationFee;
    }

    public void addRecord(String record) {
        this.record = record;
    }

    public void viewRecords() {
        System.out.println("Record: " + record);
    }
}

public class HospitalManagementSystem {
    public static void main(String[] args) {
        Patient[] patients = {
            new InPatient(1, "Amit", 45),
            new OutPatient(2, "Rita", 29)
        };
```

```
        for (int i = 0; i < patients.length; i++) {
            patients[i].getPatientDetails();
            System.out.println("Bill: ₹" + patients[i].calculateBill());

            if (patients[i] instanceof MedicalRecord) {
                MedicalRecord mr = (MedicalRecord) patients[i];
                mr.addRecord("General Check-up");
                mr.viewRecords();
            }

            System.out.println("---------------------------------");
        }
    }
}
```

---

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.

```
abstract class RideVehicle {
    private String vehicleId;
```

```
    private String driverName;
    private double ratePerKm;

    public RideVehicle(String id, String driver, double rate) {
        this.vehicleId = id;
        this.driverName = driver;
        this.ratePerKm = rate;
    }

    public double getRatePerKm() { return ratePerKm; }

    public void getVehicleDetails() {
        System.out.println("Vehicle ID: " + vehicleId + ", Driver: " +
driverName);
    }

    public abstract double calculateFare(double distance);
}

interface GPS {
    String getCurrentLocation();
    void updateLocation(String newLoc);
}

class Car extends RideVehicle implements GPS {
    private String location = "Stand A";

    public Car(String id, String driver, double rate) {
        super(id, driver, rate);
    }

    public double calculateFare(double distance) {
        return getRatePerKm() * distance;
    }

    public String getCurrentLocation() {
        return location;
    }

    public void updateLocation(String newLoc) {
        location = newLoc;
```

```
    }
}

class Bike extends RideVehicle implements GPS {
    private String location = "Stand B";

    public Bike(String id, String driver, double rate) {
        super(id, driver, rate);
    }

    public double calculateFare(double distance) {
        return getRatePerKm() * distance;
    }

    public String getCurrentLocation() {
        return location;
    }

    public void updateLocation(String newLoc) {
        location = newLoc;
    }
}

public class RideHailingApp {
    public static void main(String[] args) {
        RideVehicle[] rides = {
            new Car("CAR01", "John", 15),
            new Bike("BIKE01", "Alex", 8)
        };

        for (int i = 0; i < rides.length; i++) {
            rides[i].getVehicleDetails();
            System.out.println("Fare for 10km: ₹" +
rides[i].calculateFare(10));

            if (rides[i] instanceof GPS) {
                GPS gps = (GPS) rides[i];
                System.out.println("Current Location: " +
gps.getCurrentLocation());
                gps.updateLocation("New Stand");
                System.out.println("Updated Location: " +
```

```
gps.getCurrentLocation());
            }

            System.out.println("---------------------------------");
        }
    }
}
```