Core OOP Principles – Abstraction & Interface

Practice Problems (Any Three)

Problem 1: Vehicle with Abstract Class

Problem Statement:
Create an abstract class `Vehicle` with an abstract method `start()`. Subclasses `Car` and `Bike` will extend `Vehicle` and provide their own implementations for `start()`. Demonstrate abstraction by using `Vehicle` references to call the methods.

```
abstract class Vehicle {

    public abstract void start();

    public void fuelType() {

        System.out.println("Uses fuel");

    }

}

class Car extends Vehicle {

    public void start() {

        System.out.println("Car starts with key");

    }

}
```

```
class Bike extends Vehicle {

    public void start() {

        System.out.println("Bike starts with kick");

    }

}

public class VehicleTest {

    public static void main(String[] args) {

        Vehicle v1 = new Car();

        v1.start();

        v1.fuelType();

        Vehicle v2 = new Bike();

        v2.start();

        v2.fuelType();

    }

}
```

Problem 2: Bank Account with Abstract Methods

Problem Statement:

Design an abstract class `BankAccount` with abstract method `calculateInterest()`. Subclasses `SavingsAccount` and `CurrentAccount` should implement it differently. Demonstrate abstraction by handling different account types.

Understanding: Abstract class with both abstract and non-abstract methods.

```
abstract class BankAccount {

    protected double balance;

    public BankAccount(double balance) {

        this.balance = balance;

    }

    public abstract void calculateInterest();

    public void displayBalance() {

        System.out.println("Balance: " + balance);

    }

}

class SavingsAccount extends BankAccount {

    public SavingsAccount(double balance) {

        super(balance);

    }

    public void calculateInterest() {

        double interest = balance * 0.04;
```

```
        System.out.println("Interest: " + interest);
    }
}
class CurrentAccount extends BankAccount {
    public CurrentAccount(double balance) {
        super(balance);
    }
    public void calculateInterest() {
        double interest = balance * 0.02;
        System.out.println("Interest: " + interest);
    }
}
public class BankTest {
    public static void main(String[] args) {
        BankAccount s = new SavingsAccount(10000);
        s.displayBalance();
        s.calculateInterest();
        BankAccount c = new CurrentAccount(10000);
```

```
        c.displayBalance();

        c.calculateInterest();

    }

}
```

Problem 3: Interface for Payment Gateway

Problem Statement:
Create an interface `PaymentGateway` with methods `pay()` and `refund()`. Implement this interface in `CreditCardPayment` and `UPIPayment`. Demonstrate multiple payment methods using interfaces.

Understanding: Interface implementation and abstraction through contracts.

```
interface PaymentGateway {

    void pay(double amount);

    void refund(double amount);

}

class CreditCardPayment implements PaymentGateway {

    public void pay(double amount) {

        System.out.println("Paid via Credit Card: " + amount);

    }

    public void refund(double amount) {
```

```
        System.out.println("Refund to Credit Card: " + amount);
    }
}
class UPIPayment implements PaymentGateway {
    public void pay(double amount) {
        System.out.println("Paid via UPI: " + amount);
    }
    public void refund(double amount) {
        System.out.println("Refund to UPI: " + amount);
    }
}
public class PaymentTest {
    public static void main(String[] args) {
        PaymentGateway c = new CreditCardPayment();
        c.pay(2000);
        c.refund(500);
        PaymentGateway u = new UPIPayment();
        u.pay(1500);
```

```
        u.refund(300);

    }

}
```

Problem 4: Multiple Interfaces with Devices

Problem Statement:
Create two interfaces: `Camera` with method `takePhoto()` and `MusicPlayer` with method `playMusic()`. A class `SmartPhone` should implement both. Demonstrate multiple interface implementations.

Understanding: Multiple inheritance via interfaces.

Problem 5: Abstract Employee Class with Bonus Calculation

Problem Statement:
Create an abstract class `Employee` with data members `name` and `salary`. Add an abstract method `calculateBonus()`. Subclasses `Manager` and `Developer` should

implement the method differently. Demonstrate abstraction with real-world employee roles.

Understanding: Abstract class, common data members, constructor, and abstract method implementation.