Topic 1: Abstract Class with Abstract and Concrete Methods (Any four)

Problem Statement:
Create an abstract class `Shape` with abstract methods `area()` and `perimeter()`. Provide a concrete method `displayInfo()`.
Create subclasses `Circle` and `Rectangle` that implement the abstract methods. Test the implementation by creating objects and displaying results.

Hints:

● Use `abstract` keyword for `Shape` class.

● Implement `area()` and `perimeter()` in subclasses.

● Call `displayInfo()` from subclass objects.

```
abstract class Shape {

    abstract double area();

    abstract double perimeter();

    void displayInfo() {

        System.out.println("Shape details:");

    }

}



class Circle extends Shape {

    double radius;

    Circle(double radius) {

        this.radius = radius;
    }

    double area() {

        return Math.PI * radius * radius;

    }

    double perimeter() {

        return 2 * Math.PI * radius;

    }

}


class Rectangle extends Shape {

    double length, width;

    Rectangle(double length, double width) {

        this.length = length;

        this.width = width;

    }

    double area() {

        return length * width;

    }

    double perimeter() {

        return 2 * (length + width);

    }

}
```

```
public class ShapeDemo {

    public static void main(String[] args) {

        Shape c = new Circle(5);

        Shape r = new Rectangle(4, 6);

        c.displayInfo();

        System.out.println("Circle Area: " + c.area());

        System.out.println("Circle Perimeter: " + c.perimeter());

        r.displayInfo();

        System.out.println("Rectangle Area: " + r.area());

        System.out.println("Rectangle Perimeter: " + r.perimeter());

    }

}
```

Topic 2: Interface Implementation in Multiple Classes

Problem Statement:
Create an interface `Playable` with methods `play()` and `pause()`.
Create two classes `MusicPlayer` and `VideoPlayer` that implement this interface.
Demonstrate polymorphism by storing objects in a `Playable` reference and invoking methods.

Hints:

● Use `interface` keyword.

● Implement both methods in each class.

- Use `Playable ref = new MusicPlayer();` to test polymorphism.

```
interface Playable {
    void play();
    void pause();
}

class MusicPlayer implements Playable {
    public void play() {
        System.out.println("Playing music");
    }
    public void pause() {
        System.out.println("Pausing music");
    }
}

class VideoPlayer implements Playable {
    public void play() {
        System.out.println("Playing video");
    }
    public void pause() {
        System.out.println("Pausing video");
    }
}

public class PlayableDemo {
    public static void main(String[] args) {
        Playable p1 = new MusicPlayer();
        Playable p2 = new VideoPlayer();
        p1.play();
        p1.pause();
        p2.play();
        p2.pause();
    }
}
```

Topic 3: Abstract Class + Interface Together

Problem Statement:
Create an abstract class `Vehicle` with abstract method `start()` and a concrete method `stop()`.
Create an interface `Fuel` with method `refuel()`.
Create class `Car` that extends `Vehicle` and implements `Fuel`. Test all

methods. **Hints:**

- Use `abstract class` for `Vehicle`.
- Implement `refuel()` from `Fuel` interface in `Car`.
- Show method calls of `start()`, `stop()`, and `refuel()`.

```
abstract class Vehicle {

    abstract void start();

    void stop() {

        System.out.println("Vehicle stopped");

    }

}



interface Fuel {

    void refuel();

}



class Car extends Vehicle implements Fuel {

    void start() {
        System.out.println("Car started");

    }

    public void refuel() {

        System.out.println("Car refueled");

    }

}


public class VehicleFuelDemo {

    public static void main(String[] args) {

        Car c = new Car();

        c.start();

        c.refuel();

        c.stop();

    }

}
```

Topic 4: Interface Inheritance (Extending Interface)

Problem Statement:
Create an interface `Animal` with method `eat()`.
Create another interface `Pet` that extends `Animal` and adds method `play()`.
Create a class `Dog` that implements `Pet`. Demonstrate interface inheritance in action.

Hints:

- Use `interface Pet extends Animal`.
- `Dog` must implement both `eat()` and `play()`.
- Create object of `Dog` and test.

```
interface Animal {

    void eat();

}



interface Pet extends Animal {

    void play();

}



class Dog implements Pet {

    public void eat() {

        System.out.println("Dog is eating");

    }

    public void play() {

        System.out.println("Dog is playing");

    }

}



public class AnimalPetDemo {
        public static void main(String[] args) {

            Dog d = new Dog();

            d.eat();

            d.play();

        }

    }
```

Topic 5: Abstraction in Real-world Example

Problem Statement:
Create an abstract class `BankAccount` with abstract method `calculateInterest()` and concrete method `deposit()`.
Create subclasses `SavingsAccount` and `CurrentAccount` that provide specific interest calculation logic.
Test the program by creating objects and calling methods.

Hints:

- Define `abstract void calculateInterest();` in `BankAccount`.

- Override `calculateInterest()` differently in `SavingsAccount` and `CurrentAccount`.
- Use constructor to set balance and test deposit/interest methods.

Topic 6: Multiple Interfaces with Same Method Name

Problem Statement:
Create two interfaces `Printer` and `Scanner`, each having a method `connect()`.
Create a class `AllInOneMachine` that implements both interfaces and provides its own implementation for `connect()`.
Demonstrate how a single class can resolve method name conflicts and handle multiple interfaces.

Hints:

- Use `interface Printer` and `interface Scanner`.
- Both will have a method `void connect()`.
- In `AllInOneMachine`, implement both `connect()` methods (since they have same signature, one method will serve both).
- Create objects and test with references:
    - `Printer p = new AllInOneMachine();`
    - `Scanner s = new AllInOneMachine();`