Name: MD. Atif Rahman Rudro

ID: IT-22002

When to use interface and when to use abstract class? Develop a story and write codes.

Story Introduction: Atif, a computer science undergraduate is passionate about vehicles and programming. ~~He~~ decides to simulate a real-world car using Object-Oriented Programming (OOP) concepts in Java. His goal is to create a realistic vehicle system that can be extended later for other classes.

Design Approach and OOP Concept Used

1) Vehicle Interface - Defines Behaviour : Atif starts by creating a vehicle interface. He knows that

i) Interfaces define contracts, not implementation

ii) They are useful when multiple unrelated classes need to follow the same behaviour.

Code

```
interface Vehicle {

    void startEngine();
    void stopEngine();

    void increment();
    void decrement();

}
```

2) Transport Abstract class - Shared Blueprint:

Atif defines an abstract class Transport, that acts as a base for all transport types: It:

i) Contains common fields like engineCapacity and wheels.

ii) Has a constructor to initialize the values.

iii) Implements a method showDetails.

iv) Has an abstract method honk() to be defined by subclasses:

Code

```
abstract class Transport {
        protected int engineCapacity;
        protected int wheels;

    public Transport(int engineCapacity, int wheels) {
        this.engineCapacity=engineCapacity;
        this.wheels = wheels;
    }

    public void showDetails () {
        System.out.println("Engine Capacity:" +
                engineCapacity + "cc");

        System.out.println("Number of wheels:"+
                    wheels);
    }
    public abstract void horn();
}
```

3) Engine class - Has-A Relationship (Composition):

Atif rebuilds a simple Engine class to ~~so~~ simulate the engine behaviour. He then include this in the Car class.

Code:

```
  class Engine {
      public void start(){
          System.out.println("Engine is starting...");
      }
      public void stop(){
          System.out.println("Engine is stopping...");
      }
}
```

4) Car Class – Is-A Transport and Is-A Vehicle:

Atif designs the Car class by:-

i) Extending Transport

ii) Implementing Vehicle

iii) Having an Engine object.

Code:

```
class Car extends Transport implements Vehicle {
    private Engine engine;
    private int speed;
  public Car(int engineCapacity, int wheels) {
      super(engineCapacity, wheels);
     this.engine= new Engine();
     this.speed=0;
  }
  public void startEngine (){
      engine.start();
      System.out.println("Car engine
                              started.");
  }
```

```
public void stopEngine(){

    engine.stop();
                                        stopped
    System.out.println("Car engine ~~stopped~~.");

}

public void increment(){

    speed=speed+10;

    System.out.println("Speed increased to: "

                +speed+"km/h");

}

public void decrement(){

    speed=speed-10;

    System.out.println("Speed decreased to: "+

                speed+"km/h");

}

public void honk(){

    System.out.println("Car says: Beep

                                Beep!");

}
```

```
public void showSpeed(){
    System.out.println("Current speed : " +
                        speed + "km/h");
    }
}
```

5) Main class – Testing and Garbage Collection :

In Main class, Atif :

i) Creates a Car object.

ii) Tests engine start/stop, speed control, honking.

iii) Dereferences object by setting it to null.

iv) Requests garbage collection using System.~~gcc~~ gc ();

Code:

```
public class Main{
    public static void main(String[] args){
        Car car1 = new Car(2000, 4);
        car1.startEngine();
        car1.showDetails();
car1.increment();
car1.increment();
car1.showSpeed();
car1.honk();
car1.decrement();
car1.showSpeed();
car1.stopEngine();

car1=null;
System.out.println("Requesting garbage
                          collection...");
System.gc();
try{
    Thread.sleep(1000);
}
catch(InterruptedExeption e){
    e.printStackTrace();
}
System.out.println("End of Main Method");
}
}
```