**Introduction to Interfaces**

In Java, an interface is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces are a crucial part of Java's object-oriented programming model, as they allow the abstraction of functionality that multiple classes can implement.

**Key Features of Interfaces:**

1. **Abstract Methods**: Interfaces can declare methods that must be implemented by classes.
2. **Multiple Inheritance**: A class can implement multiple interfaces, enabling multiple inheritance in Java.
3. **Contract-Based Programming**: Interfaces define a contract that implementing classes must adhere to.

**Why Use Interfaces?**

1. **Code Reusability**: Interfaces provide a way to achieve polymorphism, allowing code to interact with objects through a common interface.
2. **Decoupling**: They help decouple the code, promoting better architecture and design patterns.
3. **Standardization**: Interfaces enforce consistent method signatures, making code predictable and easier to understand.
4. **Testability**: Using interfaces enables easier mocking and testing in unit tests.

**Example: Step-by-Step Explanation of Provided Code**

**1. ITelephone.java**

This file contains the ITelephone interface, defining the contract for any class implementing it.

public interface ITelephone {

void powerOn();

void dial(int phoneNumber);

void answer();

boolean callPhone(int phoneNumber);

boolean isRinging();

}

* **void powerOn()**: A method to simulate powering on the telephone.
* **void dial(int phoneNumber)**: Initiates dialing a phone number.
* **void answer()**: Represents answering the phone.
* **boolean callPhone(int phoneNumber)**: Attempts to call a number and returns whether the phone is ringing.
* **boolean isRinging()**: Checks if the phone is ringing.

**2. DeskPhone.java**

This class implements the ITelephone interface, providing concrete behavior for a desk phone.

public class DeskPhone implements ITelephone {

private int myNumber;

private boolean isRinging;

public DeskPhone(int myNumber) {

this.myNumber = myNumber;

}

@Override

public void powerOn() {

System.out.println("No action taken, desk phone does not have a power button");

}

@Override

public void dial(int phoneNumber) {

System.out.println("Now ringing " + phoneNumber + " on deskphone");

}

@Override

public void answer() {

if(isRinging) {

System.out.println("Answering the desk phone");

isRinging = false;

}

}

@Override

public boolean callPhone(int phoneNumber) {

if(phoneNumber == myNumber) {

isRinging = true;

System.out.println("Ring ring");

} else {

isRinging = false;

}

return isRinging;

}

@Override

public boolean isRinging() {

return isRinging;

}

}

* **DeskPhone(int myNumber)**: Constructor initializes the phone number.
* **powerOn()**: Outputs a message indicating the absence of a power button.
* **dial(int phoneNumber)**: Simulates dialing a number.
* **answer()**: Answers the call if the phone is ringing.
* **callPhone(int phoneNumber)**: Rings if the number matches the phone’s assigned number.
* **isRinging()**: Returns the ringing status.

**3. Main.java**

This file demonstrates the usage of the ITelephone interface and its implementation by DeskPhone.

public class Main {

public static void main(String[] args) {

ITelephone timsPhone;

timsPhone = new DeskPhone(123456);

timsPhone.powerOn();

timsPhone.callPhone(123456);

timsPhone.answer();

}

}

* **ITelephone timsPhone;**: Declares a reference variable for the interface.
* **timsPhone = new DeskPhone(123456);**: Creates an instance of DeskPhone.
* **timsPhone.powerOn();**: Calls the powerOn() method.
* **timsPhone.callPhone(123456);**: Simulates calling the phone.
* **timsPhone.answer();**: Answers the call.

**Lab Task**

**Objective:**

Create a new class MobilePhone that implements the ITelephone interface. The MobilePhone should have distinct behavior from DeskPhone, such as requiring the phone to be powered on before calling.

**Task Description:**

1. Implement a MobilePhone class that:
   * Requires powering on before any operation.
   * Provides implementations for all methods in ITelephone.
2. Test your implementation in a Main class.

**Steps to Achieve:**

1. Create the MobilePhone class:
   * Add fields for myNumber, isRinging, and isPoweredOn.
   * Implement all methods of ITelephone.
2. Update the Main class:
   * Instantiate and test a MobilePhone object.

**Expected Output:**

Phone is powered off. Cannot call.

Powering on the mobile phone.

Now ringing 987654 on mobile phone.

Answering the mobile phone.

**Submission Instructions:**

* Submit your MobilePhone.java and updated Main.java files.
* Ensure the code is well-commented and adheres to Java coding standards.

**Understanding Inner Classes in Java**

**Introduction**

Inner classes in Java are classes defined within the scope of another class. They are primarily used to logically group classes that will only be used in one place, enabling better encapsulation and readability. This lab explores the concept of inner classes through the implementation of a gearbox system.

The provided code consists of:

1. **Gearbox class**: Represents a gearbox with gears.
2. **Inner Gear class**: Models individual gears of the gearbox.
3. A **Main class** to test the functionality.

**Code Explanation**

**Gearbox.java**

java

Copy code

private ArrayList<Gear> gears; // Holds all gear objects

private int maxGears; // Maximum gears the gearbox can support

private int currentGear = 0; // Tracks the currently active gear

private boolean clutchIsIn; // Indicates if the clutch is engaged

* These variables are the key components of the Gearbox class.

**Constructor**

java

Copy code

public Gearbox(int maxGears) {

this.maxGears = maxGears;

this.gears = new ArrayList<>();

Gear neutral = new Gear(0, 0.0);

this.gears.add(neutral);

}

* Initializes the gearbox with a maximum number of gears.
* A default "neutral" gear is added with a ratio of 0.0.

**operateClutch Method**

java

Copy code

public void operateClutch(boolean in) {

this.clutchIsIn = in;

}

* Simulates clutch operation.

**addGear Method**

java

Copy code

public void addGear(int number, double ratio) {

if((number > 0) && (number <= maxGears)) {

this.gears.add(new Gear(number, ratio));

}

}

* Adds a new gear to the gearbox, ensuring it doesn’t exceed the allowed range.

**changeGear Method**

java

Copy code

public void changeGear(int newGear) {

if((newGear >= 0) && (newGear < this.gears.size()) && this.clutchIsIn) {

this.currentGear = newGear;

System.out.println("Gear " + newGear + " selected.");

} else {

System.out.println("Grind!");

this.currentGear = 0;

}

}

* Changes the current gear if the clutch is engaged and the gear is valid.

**wheelSpeed Method**

java

Copy code

public double wheelSpeed(int revs) {

if(clutchIsIn) {

System.out.println("Scream!!!");

return 0.0;

}

return revs \* gears.get(currentGear).getRatio();

}

* Computes the wheel speed based on engine revs and the gear ratio.

**Inner Gear Class**

java

Copy code

private class Gear {

private int gearNumber;

private double ratio;

public Gear(int gearNumber, double ratio) {

this.gearNumber = gearNumber;

this.ratio = ratio;

}

public double getRatio() {

return ratio;

}

public double driveSpeed(int revs) {

return revs \* this.ratio;

}

}

* Represents a single gear.
* Has methods to fetch the gear ratio and compute speed.

**Main.java**

**Usage of Inner Classes**

java

Copy code

Gearbox mcLaren = new Gearbox(6);

Gearbox.Gear first = mcLaren.new Gear(1, 12.3);

* Demonstrates how to create an instance of an inner class.
* first.driveSpeed(1000) calculates the speed for a given RPM using the first gear.

**Lab Task**

**Objective**

Enhance the Gearbox class to include reverse gear functionality and implement a method to simulate a gear shift sequence.

**Task Description**

1. **Add Reverse Gear**: Modify the addGear method to support a reverse gear with a gear number of -1 and a custom ratio.
2. **Simulate Gear Shifting**: Create a new method, shiftSequence, that shifts through all available gears (including reverse) and prints the respective speeds for a given RPM.

**Steps to Achieve**

1. Modify the addGear method to accept a gear number of -1.
2. Implement the shiftSequence method:
   * Iterate through all gears using a loop.
   * Print the speed for each gear using wheelSpeed.
3. Test the implementation in the Main class.

**Expected Outcome**

* Successfully add a reverse gear.
* Simulate gear shifting from reverse through all forward gears, with outputs like:

yaml

Copy code

Gear -1: Reverse speed = -12.3 RPM

Gear 0: Neutral, no movement.

Gear 1: Speed = 1230.0 RPM

...