Lecture 10: Interfaces and Inner Classes in Java

Introduction to Interfaces

In Java, an interface is a reference type similar to a class, but it can contain only:

- Constants
- Method signatures
- Default methods
- Static methods
- Nested types

Interfaces enable abstraction by defining a set of methods that implementing classes must provide. They are fundamental to Java's object-oriented design.

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, achieving multiple inheritance in Java.
3. **Contract-Based Programming**: Interfaces act as a contract, ensuring that implementing classes adhere to predefined behaviors.

Why Use Interfaces?

- **Code Reusability**: Promotes polymorphism, allowing different objects to interact via a common interface.
- **Decoupling**: Helps separate components, enhancing modularity.
- **Standardization**: Ensures consistent method signatures across implementations.
- **Testability**: Facilitates easier mocking and testing in unit tests.

Example: Defining and Implementing an Interface

ITelephone.java (Interface Definition)

```
public interface ITelephone {
    void powerOn();
    void dial(int phoneNumber);
    void answer();
    boolean callPhone(int phoneNumber);
    boolean isRinging();
}
```

- `void powerOn()`: Simulates powering on the telephone.
- `void dial(int phoneNumber)`: Simulates dialing a number.
- `void answer()`: Represents answering the phone.
- `boolean callPhone(int phoneNumber)`: Attempts to call a number.

- `boolean isRinging()`: Checks if the phone is ringing.

DeskPhone.java (Implementation of ITelephone)

```
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 desk phone");
    }

    @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;
    }
}
```

- **Constructor**: Initializes the phone number.
- **Implemented Methods**: Provide behavior specific to a desk phone.

Main.java (Usage of Interface)

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

```
        ITelephone timsPhone;
        timsPhone = new DeskPhone(123456);
        timsPhone.powerOn();
        timsPhone.callPhone(123456);
        timsPhone.answer();
    }
}
```

- Declares a reference variable of type `ITelephone`.
- Instantiates a `DeskPhone` and calls its methods.

Inner Classes in Java

Inner classes are defined within the scope of another class. They are primarily used to:

- Logically group classes that will be used only in one place.
- Improve encapsulation and readability.

Types of Inner Classes

1. **Nested Static Classes**: Can be instantiated without an object of the enclosing class.
2. **Non-Static Inner Classes**: Associated with an instance of the enclosing class.
3. **Local Inner Classes**: Defined within a method or block.
4. **Anonymous Inner Classes**: Created for one-time use, typically to override methods.

Example: Inner Class Implementation

Gearbox.java

```
import java.util.ArrayList;

public class Gearbox {
    private ArrayList<Gear> gears;
    private int maxGears;
    private int currentGear = 0;
    private boolean clutchIsIn;

    public Gearbox(int maxGears) {
        this.maxGears = maxGears;
        this.gears = new ArrayList<>();
        Gear neutral = new Gear(0, 0.0);
        this.gears.add(neutral);
    }

    public void operateClutch(boolean in) {
        this.clutchIsIn = in;
    }

    public void addGear(int number, double ratio) {
        if ((number > 0) && (number <= maxGears)) {
            this.gears.add(new Gear(number, ratio));
        }
```

```
    }

    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;
        }
    }

    public double wheelSpeed(int revs) {
        if (clutchIsIn) {
            System.out.println("Scream!!!");
            return 0.0;
        }
        return revs * gears.get(currentGear).getRatio();
    }

    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;
        }
    }
}
```

- **Fields**: Tracks gears, maximum gears, current gear, and clutch state.
- **Methods**: Enable adding gears, changing gears, and calculating speed.
- **Inner Gear Class**: Represents individual gears with methods to compute speeds.

Main.java

```
public class Main {
    public static void main(String[] args) {
        Gearbox mcLaren = new Gearbox(6);
        mcLaren.addGear(1, 5.3);
        mcLaren.addGear(2, 10.6);
        mcLaren.addGear(3, 15.9);

        mcLaren.operateClutch(true);
        mcLaren.changeGear(1);
```

```
        mcLaren.operateClutch(false);
        System.out.println(mcLaren.wheelSpeed(1000));
    }
}
```

- Demonstrates adding gears and simulating clutch operation.

Key Takeaways

- **Interfaces**: Define contracts for classes, promoting polymorphism and modularity.
- **Inner Classes**: Enable grouping related classes for better encapsulation.
- Usage of interfaces and inner classes together can enhance the structure and maintainability of complex programs.