Module 7: Abstraction & Interfaces (Day 19–20)

---

1. Abstract Classes

Definition:

An abstract class in Java is a class that **cannot be instantiated** (you cannot create objects directly from it). It is designed to be a **blueprint** for other classes and may contain abstract methods (methods without a body) that must be implemented by subclasses.

Key Features:

- Declared using the abstract keyword.
- Can contain both **abstract** and **concrete** (non-abstract) methods.
- Can have **constructors, instance variables, and static methods**.
- Subclasses **must override** all abstract methods unless they are also declared abstract.

When to Use Abstract Classes?

- When you want to **share common code** among closely related classes.
- When you need to define a **template** for future subclasses.

Real-World Analogy:

Imagine a blueprint of a vehicle. You can't use the blueprint to drive, but you can use it to build different types of vehicles like a car or bike, each with their own way of starting.

Example Code:

```
abstract class Vehicle {
    String name;

    Vehicle(String name) {
        this.name = name;
    }

    abstract void start();  // Abstract method

    void displayInfo() {    // Concrete method
        System.out.println("This is a " + name);
    }
}
```

```
class Car extends Vehicle {
    Car(String name) {
        super(name);
    }

    @Override
    void start() {
        System.out.println("Car starts with a key.");
    }
}

class Bike extends Vehicle {
    Bike(String name) {
        super(name);
    }

    @Override
    void start() {
        System.out.println("Bike starts with a kick.");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle car = new Car("Toyota");
        Vehicle bike = new Bike("Harley");

        car.displayInfo();
        car.start();
```

```
        bike.displayInfo();
        bike.start();
    }
}
```

Expected Output:

This is a Toyota

Car starts with a key.

This is a Harley

Bike starts with a kick.

---

2. Interfaces

Definition:

An interface in Java is a **fully abstract class** (before Java 8) that defines a **contract** for what a class can do, without saying how to do it.

Key Features (Java 7 & earlier):

- All methods are implicitly **public** and **abstract**.
- All variables are implicitly **public**, **static**, and **final**.
- A class implements an interface using the implements keyword.
- Supports **multiple inheritance**.

Java 8+ Enhancements:

- **Default Methods:** Use default keyword to provide method body.
- **Static Methods:** Use static keyword for utility methods.

When to Use Interfaces?

- When you need to support **multiple inheritance**.
- When you want to enforce a **contract** across multiple classes.

Real-World Analogy:

Think of an interface like a remote control interface that defines the buttons (methods). Different devices (TV, AC) implement them differently.

Example Code:

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

```
    void stop();

    default void displayType() {
        System.out.println("This is a media player.");
    }
}

class AudioPlayer implements Playable {
    public void play() {
        System.out.println("Audio is playing.");
    }

    public void pause() {
        System.out.println("Audio paused.");
    }

    public void stop() {
        System.out.println("Audio stopped.");
    }
}

class VideoPlayer implements Playable {
    public void play() {
        System.out.println("Video is playing.");
    }

    public void pause() {
        System.out.println("Video paused.");
    }

    public void stop() {
```

```
        System.out.println("Video stopped.");
    }
}

public class Main {
    public static void main(String[] args) {
        Playable audio = new AudioPlayer();
        Playable video = new VideoPlayer();

        audio.play();
        audio.pause();
        audio.displayType();

        video.play();
        video.stop();
    }
}
```

Expected Output:

Audio is playing.

Audio paused.

This is a media player.

Video is playing.

Video stopped.

---

3. Functional Interfaces (Java 8+)

Definition:

A functional interface is an interface that contains only **one abstract method**. It may contain multiple **default or static methods**.

Key Features:

- Used for **lambda expressions** and **method references**.
- Annotated with @FunctionalInterface (optional but recommended).

Examples:

- Runnable, Comparator, Callable, Predicate, Function

When to Use?

- When working with **lambda expressions** for cleaner, concise code.
- When using **Stream API** for functional programming.

Real-World Analogy:

Imagine a single-button machine. When you press it (invoke), it performs one action - very specific and powerful.

Example Code:

```
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        // Lambda expression for addition
        Calculator add = (a, b) -> a + b;

        // Lambda expression for multiplication
        Calculator multiply = (a, b) -> a * b;

        System.out.println("Addition: " + add.calculate(5, 3));
        System.out.println("Multiplication: " + multiply.calculate(5, 3));
    }
}
```

Expected Output:

```
Addition: 8
Multiplication: 15
```

Summary Table:

Feature	Abstract Class	Interface (Java 7)	Functional Interface
Instantiation	No	No	No
Methods	Abstract + Concrete	Only Abstract (pre-Java 8)	One Abstract Method
Variables	Any type	public static final	public static final
Inheritance	Single	Multiple	Single Abstract Method
Java 8+	No Change	Default/Static Methods	Lambda Support
