**Java Interfaces**

Interfaces are a fundamental concept in Java that define a contract for classes to implement. They promote loose coupling, enable polymorphism, and are crucial for creating flexible and maintainable code. This chapter will explore various aspects of interfaces in Java.

**1. Basic Interface Declaration and Implementation**

Let's start with a basic example of an interface and its implementation:

// Drawable.java

public interface Drawable {

void draw();

}

// Circle.java

public class Circle implements Drawable {

@Override

public void draw() {

System.out.println("Drawing a circle");

}

}

// Square.java

public class Square implements Drawable {

@Override

public void draw() {

System.out.println("Drawing a square");

}

}

// Main.java

public class Main {

public static void main(String[] args) {

Drawable circle = new Circle();

Drawable square = new Square();

circle.draw();

square.draw();

}

}

Output:

Drawing a circle

Drawing a square

In this example, Drawable is an interface implemented by both Circle and Square. This allows us to treat different shapes uniformly through the Drawable interface.

**2. Default Methods in Interfaces**

Java 8 introduced default methods in interfaces, allowing us to provide a default implementation for a method:

// Vehicle.java

public interface Vehicle {

void start();

default void stop() {

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

}

}

// Car.java

public class Car implements Vehicle {

@Override

public void start() {

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

}

}

// Main.java

public class Main {

public static void main(String[] args) {

Vehicle car = new Car();

car.start();

car.stop();

}

}

Output:

Car starting...

Vehicle stopping...

Here, stop() is a default method in the Vehicle interface, which Car inherits without needing to implement it explicitly.

**3. Static Methods in Interfaces**

Interfaces can also contain static methods:

// MathOperations.java

public interface MathOperations {

static int add(int a, int b) {

return a + b;

}

static int subtract(int a, int b) {

return a - b;

}

}

// Main.java

public class Main {

public static void main(String[] args) {

System.out.println("5 + 3 = " + MathOperations.add(5, 3));

System.out.println("5 - 3 = " + MathOperations.subtract(5, 3));

}

}

Output:

5 + 3 = 8

5 - 3 = 2

Static methods in interfaces are called directly on the interface, without needing an implementing class.

**4. Private Methods in Interfaces**

Java 9 introduced private methods in interfaces, allowing for better code organization within the interface:

// Logger.java

public interface Logger {

default void logInfo(String message) {

log("INFO", message);

}

default void logWarning(String message) {

log("WARNING", message);

}

private void log(String level, String message) {

System.out.println("[" + level + "] " + message);

}

}

// ConsoleLogger.java

public class ConsoleLogger implements Logger {}

// Main.java

public class Main {

public static void main(String[] args) {

Logger logger = new ConsoleLogger();

logger.logInfo("This is an information message");

logger.logWarning("This is a warning message");

}

}

Output:

[INFO] This is an information message

[WARNING] This is a warning message

The private log method in the Logger interface helps avoid code duplication in the default methods.

**5. Multiple Interface Implementation**

A class can implement multiple interfaces:

// Swimmable.java

public interface Swimmable {

void swim();

}

// Flyable.java

public interface Flyable {

void fly();

}

// Duck.java

public class Duck implements Swimmable, Flyable {

@Override

public void swim() {

System.out.println("Duck is swimming");

}

@Override

public void fly() {

System.out.println("Duck is flying");

}

}

// Main.java

public class Main {

public static void main(String[] args) {

Duck duck = new Duck();

duck.swim();

duck.fly();

}

}

Output:

Duck is swimming

Duck is flying

Here, Duck implements both Swimmable and Flyable interfaces, demonstrating multiple interface implementation.

**6. Functional Interfaces**

Functional interfaces, introduced in Java 8, have exactly one abstract method and can be used with lambda expressions:

// Greeting.java

@FunctionalInterface

public interface Greeting {

void greet(String name);

}

// Main.java

public class Main {

public static void main(String[] args) {

Greeting casualGreeting = name -> System.out.println("Hey, " + name + "!");

Greeting formalGreeting = name -> System.out.println("Good day, " + name + ".");

casualGreeting.greet("Arjun");

formalGreeting.greet("Mrs. Sonali");

}

}

Output:

Hey, Arjun!

Good day, Mr. Sonali.

The @FunctionalInterface annotation is optional but helps catch errors if more than one abstract method is accidentally added.

**Conclusion**

Interfaces in Java 17 are powerful tools for designing flexible and modular systems. They allow you to define contracts for classes, provide default and static methods, and even support functional programming paradigms. By mastering interfaces, you can create more maintainable and extensible Java applications.