16.4.2 Abstraction using interface

🧭 Introduction

Abstraction is a key concept in Object-Oriented Programming (OOP), allowing you to define the **"what"** without worrying about the **"how"**. In Java, abstraction is primarily achieved using **abstract classes** and **interfaces**. This blog focuses on **interfaces**, particularly their evolution and advanced features introduced in **Java 8** and **Java 9**.

We'll begin with basic concepts and gradually move towards more intermediate and advanced features like `default`, `static`, and `private` methods in interfaces.

🎯 Learning Abstraction with Interfaces — From Basic to Intermediate

🔰 Level 1: What is Abstraction?

Abstraction is the process of hiding internal implementation and showing only essential information.

Real-life analogy: When you use a TV remote, you press the power button to turn it on — you don't know (and don't care) how the internal circuits work.

In Java, this is achieved through:

- **Abstract classes** (partial abstraction)
- **Interfaces** (complete abstraction before Java 8)

🔰 Level 2: Basic Interface Syntax

```java
public interface Animal {
    void makeSound();
}
```

- This is a **pure abstract interface**.
- It declares a method `makeSound()` without implementation.

👇 Example:

```java
public class Dog implements Animal {
    public void makeSound() {
        System.out.println("Bark");
    }
}
```

✅ Output:

```
Bark
```

📝 **Key Takeaways**:

- Interfaces cannot have constructors.
- All methods are implicitly `public abstract` before Java 8.
- Classes use the `implements` keyword to inherit interfaces.

🔰 Level 3: Multiple Interface Inheritance

Java doesn't support multiple inheritance with classes, but **interfaces** solve this.

```java
public interface Flyable {
    void fly();
}

public interface Swimmable {
    void swim();
}

public class Duck implements Flyable, Swimmable {
    public void fly() {
        System.out.println("Duck flying...");
    }

    public void swim() {
        System.out.println("Duck swimming...");
    }
}
```

📝 **Key Takeaway**: A class can implement **multiple interfaces**.

🔰 Level 4: Interface Variables

All variables in interfaces are:

- `public`
- `static`
- `final`

```java
public interface Config {
    int TIMEOUT = 5000; // Equivalent to public static final int TIMEOUT = 5000;
}
```

🚀 Interface Evolution: Java 8 and 9

🔄 Why Change Interfaces?

Before Java 8, interfaces could only have abstract methods. If you needed to add a new method, all implementing classes would break.

To support **backward compatibility** and **method sharing**, Java 8 introduced:

- ✅ `default` methods
- ✅ `static` methods

And Java 9 added:

- ✅ `private` methods

🧩 Default Methods (Java 8)

✅ Syntax:

```java
public interface Vehicle {
    void start();

    default void stop() {
        System.out.println("Vehicle stopped.");
    }
}
```

✅ Use Case:

Add new methods to an interface **without breaking existing implementations**.

⚠️ Note:

- Can be overridden in implementing classes.

⚙️ Static Methods (Java 8)

✅ Syntax:

```java
public interface Logger {
    static void log(String message) {
        System.out.println("LOG: " + message);
    }
}
```

🔍 Key Points:

- Belongs to the interface, **not** to implementing classes.
- Called like: `Logger.log("Hello");`

🔐 Private Methods (Java 9)

✅ **Syntax:**

```java
public interface Helper {
    private static void audit(String msg) {
        System.out.println("AUDIT: " + msg);
    }

    static void process() {
        audit("Processing started");
    }
}
```

✅ **Use Case:**

- Used **internally** within `default` or `static` methods to avoid code duplication.

🛠️ Complete Example with All Types

```java
public interface PaymentGateway {

    void initiatePayment(double amount);

    default void validate() {
        System.out.println("Validating payment details...");
        log("Validation successful");
    }

    static void help() {
        System.out.println("For help, contact support@payments.com");
    }

    private void log(String message) {
        System.out.println("[LOG] " + message);
    }
}

public class PayPal implements PaymentGateway {

    @Override
    public void initiatePayment(double amount) {
        System.out.println("Payment of $" + amount + " initiated via PayPal.");
    }
}
```

🔎 Output:

```java
PaymentGateway gateway = new PayPal();
gateway.initiatePayment(100.0);
gateway.validate();
PaymentGateway.help();
```

🧠 Best Practices

- ✔ Use interfaces for defining contracts.
- ✔ Use default methods for backward compatibility, **not for business logic**.
- ✔ Prefer helper classes for complex logic over stuffing static methods into interfaces.
- ✔ Use private methods to clean up reusable internal logic in large interfaces.

✅ Summary Table

Feature	Since	Use Case	Overridable?	Visibility
Abstract	Java 1	Contract-only	✅ Yes	Public
Default	Java 8	Shared default behavior	✅ Yes	Public
Static	Java 8	Utility/helper methods	❌ No	Public
Private	Java 9	Internal method reuse	❌ No	Private

Document by ***Suyash*** 😇