16.4.1 Abstraction using Abstract class

🧭 Introduction

Abstraction is the object-oriented principle of exposing only **relevant details** and hiding the underlying complexity. Java offers two main tools to implement abstraction:

1. **Interfaces** – For complete abstraction (especially after Java 8+)
2. **Abstract Classes** – For partial abstraction with flexibility

This post focuses on **abstraction using abstract classes**, covering how and when to use them, along with real-world examples and comparisons.

🎯 Learning Abstraction Using Abstract Classes — From Basic to Intermediate

---

🔰 Level 1: What is an Abstract Class?

An **abstract class** in Java is a class that cannot be instantiated and may contain **abstract methods** (without a body) as well as **concrete methods** (with implementation).

```
abstract class Animal {
    abstract void makeSound(); // abstract method

    void eat() {
        System.out.println("This animal eats food."); // concrete method
    }
}
```

- **Cannot be instantiated** directly
- Can have constructors
- Can have fields, methods, static methods, etc.
- Supports **partial abstraction**

🔰 Level 2: Abstract Class Example

```
abstract class Vehicle {
    abstract void start(); // abstract method

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

```

```java
class Car extends Vehicle {
    void start() {
        System.out.println("Car started.");
    }
}
```

```java
Vehicle v = new Car();
v.start(); // Car started.
v.stop();  // Vehicle stopped.
```

🔰 Level 3: Abstract Classes with Constructors

Yes! **Abstract classes can have constructors**, which can be used by subclasses.

```java
abstract class Shape {
    String color;

    Shape(String color) {
        this.color = color;
        System.out.println("Shape constructor called");
    }

    abstract double area();
}

```

```java
class Circle extends Shape {
    double radius;

    Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }

    double area() {
        return Math.PI * radius * radius;
    }
}

```

🔰 Level 4: Abstract vs Concrete Methods

```java
abstract class Employee {
    abstract void work();      // Abstract

    void logHours() {         // Concrete
        System.out.println("Logging work hours...");
    }
}
```

📝 **Key Point**: You can mix both in an abstract class, offering flexibility.

🔰 Level 5: Access Modifiers in Abstract Classes

- You can use **public**, **protected**, or **private** access modifiers.
- Abstract methods **cannot be private** (they must be accessible to subclasses).

```java
abstract class Demo {
    protected abstract void display(); // Valid
    // private abstract void secret(); // Invalid
}
```

🔰 Level 6: Final and Static Methods

✅ Final Methods:

- Cannot be overridden in subclasses.

```java
abstract class Machine {
    final void powerOn() {
        System.out.println("Machine powered on");
    }
}
```

✅ Static Methods:

- Belong to the class, not the object.

```java
abstract class Logger {
    static void log(String msg) {
        System.out.println("LOG: " + msg);
    }
}
```

🔰 Level 7: Abstract Class Hierarchy and Inheritance

- An abstract class **can extend another abstract class**.
- A concrete subclass must implement **all** inherited abstract methods.

```java
abstract class Animal {
    abstract void sound();
}

abstract class Mammal extends Animal {
    abstract void walk();
}

class Human extends Mammal {
    void sound() {
        System.out.println("Talks");
    }

    void walk() {
        System.out.println("Walks on two legs");
    }
}
```

🛠️ Real-World Example

```java
abstract class Payment {
    abstract void processPayment(double amount);

    void printReceipt() {
        System.out.println("Receipt printed.");
    }
}

class CreditCardPayment extends Payment {
    void processPayment(double amount) {
        System.out.println("Processing credit card payment: $" + amount);
    }
}
```

```java
Payment payment = new CreditCardPayment();
payment.processPayment(250.0);
payment.printReceipt();
```

⚖️ Abstract Class vs Interface

Feature	Abstract Class	Interface
Methods	Can have abstract and concrete	Java 7: only abstract Java 8+: default, static
Constructors	✅ Yes	❌ No
Multiple Inheritance	❌ No (single class)	✅ Yes
Fields	Any type (with modifiers)	Only `public static final` constants
Access Modifiers	public, protected, private allowed	Only public (for methods)
Use Case	Shared code + contract	Pure contract

🧠 Best Practices

- Use **abstract classes** when:
  - You want to provide **base functionality**.
  - You expect future **code reuse** in child classes.
  - You need constructors or maintain state.

- Use **interfaces** when:
  - You want to define a **contract** with no implementation.
  - You need **multiple inheritance**.

✅ Summary Table

Feature	Abstract Class
Can be instantiated?	❌ No
Can have constructors?	✅ Yes
Can have concrete methods?	✅ Yes
Can have abstract methods?	✅ Yes
Can be extended?	✅ Yes
Can implement interfaces?	✅ Yes
Access Modifiers allowed?	✅ All (except private abstract)
Can have static/final?	✅ Yes

Document by ***Suyash***😇