OOP in Java

Affan Rauf

September 20, 2024

What is an Abstract Class?

- An abstract class in Java is a class that cannot be instantiated directly.
- It is meant to be subclassed, and its abstract methods must be implemented in the subclasses.
- Abstract classes are used to represent generic concepts that can be shared among multiple subclasses.

Defining an Abstract Class

- Use the `abstract` keyword before the class keyword to define an abstract class.
- An abstract class can contain both abstract methods (without implementation) and concrete methods (with implementation).

```
abstract class Animal {
    // Abstract method (no implementation)
    abstract void makeSound();

    // Concrete method (implementation provided)
    void sleep() {
        System.out.println("This animal is sleeping.")
            ;
    }
}
```

Using an Abstract Class

- Subclasses of an abstract class must provide implementations for all abstract methods.
- The subclass can also override concrete methods if needed.
- An abstract class cannot be instantiated, but it can be used as a reference type.

```
class Dog extends Animal {
    // Implementing the abstract method
    @Override
    void makeSound() {
        System.out.println("The dog barks.");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.makeSound(); // "The dog barks."
        myDog.sleep();     // "This animal is sleeping
            ."
    }
}
```

When to Use Abstract Classes

- Use abstract classes when you have a clear base class that should define default behavior for subclasses.
- Use them when you want to enforce a certain structure across multiple related classes.
- Prefer abstract classes when your base class contains both some methods that are common to all subclasses and some that need to be implemented by each subclass.

What is an Interface in Java?

- An interface in Java is a reference type, similar to a class, that can contain only abstract methods (until Java 8) and static constants.
- Interfaces are used to specify a set of methods that a class must implement.
- In Java 8 and later, interfaces can also include default methods and static methods.

Defining an Interface

- Use the `interface` keyword to define an interface.
- Interfaces cannot have constructors and cannot hold instance fields (except static constants).
- All methods in an interface are implicitly abstract (until Java 8) and public.

```
interface Animal {
    void makeSound();  // Abstract method
    // Java 8 feature: default method
    default void sleep() {
        System.out.println("This animal is sleeping.")
            ;
    }
    // Java 8 feature: static method
    static void eat() {
        System.out.println("This animal is eating.");
    }
}
```

Implementing an Interface

- A class that implements an interface must provide implementations for all abstract methods of the interface.
- A class can implement multiple interfaces, allowing for multiple inheritance of type.

```
class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("The dog barks.");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound(); // "The dog barks."
        myDog.sleep();     // "This animal is sleeping
            ."
        // Calling a static method from the interface
        Animal.eat();      // "This animal is eating."
    }
}
```

When to Use Interfaces over Abstract Classes

- Ideal for defining a contract that multiple classes can implement in their own way.
- When you need to achieve multiple inheritance. A class can implement multiple interfaces but can only extend one abstract class.
- When different classes need to implement the same set of methods but aren't necessarily related by inheritance, interfaces are the way to go.
- When you need a class to adhere to multiple sets of behaviors or abilities that are unrelated, interfaces are preferable because they allow you to mix these behaviors.

Using Interfaces to Implement Polymorphism

```
interface Shape {
    void draw();
}
class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle.");
    }
}
class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle.");
    }
}
class Triangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Triangle.");
    }
}
```

Using Interfaces to Implement Polymorphism

```
public class ShapeDemo {
    public static void main(String[] args) {
        // Create instances of different shapes
        Shape circle = new Circle();
        Shape rectangle = new Rectangle();
        Shape triangle = new Triangle();

        // Array of Shape references
        Shape[] shapes = {circle, rectangle, triangle
            };

        // Polymorphic behavior
        for (Shape shape : shapes) {
            shape.draw(); // Calls the appropriate
                draw() method based on the actual
                object type
        }
    }
}
```

Interfaces vs. Abstract Classes

- Interfaces:
  - No constructors, can contain only abstract methods (until Java 8), default methods, static methods, and constants.
  - Can be implemented by any class, providing a way to share behavior across unrelated classes.
- Abstract Classes:
  - Can have constructors, instance fields, and both abstract and concrete methods.
  - Should be used when classes share a common base with shared functionality.

What is a Static Method?

- Belongs to the class rather than instances (objects) of the class
- Can be called without creating an instance of the class
- Can only access static variables and other static methods within the class

```
class MathUtils {
    public static int square(int number) {
        return number * number;
    }
    public static int add(int a, int b) {
        return a + b;
    }
    public static int factorial(int n) {
        if (n == 0) {
            return 1;
        }
        return n * factorial(n - 1);
    }
}
```

Using Static Methods

- You can call static methods directly using the class name without creating an instance of the class.

```
public class Main {
    public static void main(String[] args) {
        // Calling static methods of MathUtils class
        int squareOfFive = MathUtils.square(5);
        int sum = MathUtils.add(10, 20);
        int factorialOfFour = MathUtils.factorial(4);

        // Print the results
        System.out.println("Square of 5: " +
            squareOfFive);
        System.out.println("Sum of 10 and 20: " + sum)
            ;
        System.out.println("Factorial of 4: " +
            factorialOfFour);
    }
}
```

When to Use Static Methods

- **Utility or Helper Methods**: Methods that perform operations independent of the object state (instance variables) and are better suited as utility functions (e.g., ‘Math.sqrt()‘).
- **Shared Code Across Instances**: When you need a method that can be shared across all instances of a class (e.g., a method that calculates something based on parameters, not on instance variables).
- **Factory Methods:** When creating an instance of a class using a method that returns an instance of the class.