OOPS IV - Abstract Classes & Interfaces

Interfaces

In Java, an interface is an abstract type that contains a collection of methods and constant variables.It can be thought of as a blueprint of behavior. It is one of the core concepts in Java and is used to achieve abstraction, polymorphism and multiple inheritances. An interface is a reference type in Java. It is similar to a class, but it cannot be instantiated. It can contain only constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

How to create an interface?

Let us create an interface for a Person

```
public interface Person {
    String getName();
    String getEmail();
}
```

Now let's create a class that implements the Person interface:

```
public class User implements Person {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public String getEmail() {
        return email;
    }
}
```

In an interface, we’re allowed to use:

- constants variables
- abstract methods
- static methods
- default methods

We also should remember that:

- we can’t instantiate interfaces directly
- an interface can be empty, with no methods or variables in it
- we can’t use the final word in the interface definition, as it will result in a compiler error
- all interface declarations should have the public or default access modifier; the abstract modifier will be added automatically by the compiler
- an interface method can’t be protected or final
- up until Java 9, interface methods could not be private; however, Java 9 introduced the possibility to define private methods in interfaces
- interface variables are public, static, and final by definition; we’re not allowed to change their visibility

Why use an interface?

1. Behavioral Functionality

We use interfaces to add certain behavioral functionality that can be used by unrelated classes. For instance, Comparable, Comparator, and Cloneable are Java interfaces that can be implemented by unrelated classes. Below is an example of the Comparator interface that is used to compare two instances of the Employee class:

```
public class Player {
    private int ranking;
    private String name;
    private int age;

    // constructor, getters, setters
}
```

Comparable is an interface defining a strategy of comparing an object with other objects of the same type. This is called the class’s “natural ordering.”

In order to be able to sort, we must define our Player object as comparable by implementing the Comparable interface.

```
public class Player implements Comparable<Player> {

    // same as before
    @Override
    public int compareTo(Player otherPlayer) {
        return Integer.compare(getRanking(), otherPlayer.getRanking());
    }
}
```

```
public static void main(String[] args) {
    List<Player> footballTeam = new ArrayList<>();
    Player player1 = new Player(59, "John", 20);
    Player player2 = new Player(67, "Roger", 22);
```

```
    Player player3 = new Player(45, "Steven", 24);
    footballTeam.add(player1);
    footballTeam.add(player2);
    footballTeam.add(player3);

    System.out.println("Before Sorting : " + footballTeam);
    Collections.sort(footballTeam);
    System.out.println("After Sorting : " + footballTeam);
}
```

2. Multiple Inheritances

Java classes support singular inheritance. However, by using interfaces, we're also able to implement multiple inheritances. For instance, in the example below, we notice that the Car class implements the Fly and Transform interfaces. By doing so, it inherits the methods fly and transform:

```
public interface Transform {
    void transform();
}

public interface Fly {
    void fly();
}

public class Car implements Fly, Transform {

    @Override
    public void fly() {
        System.out.println("I can Fly!!");
    }

    @Override
    public void transform() {
        System.out.println("I can Transform!!");
    }
}
```

3. Polymorphism

Polymorphism is the ability for an object to take different forms during runtime. To be more specific it's the execution of the override method that is related to a specific object type at runtime.

In Java, we can achieve polymorphism using interfaces. For example, the Shape interface can take different forms – it can be a Circle or a Square.

Let's start by defining the Shape interface:

```
public interface Shape {
    String name();
}
```

Now let's also create the Circle class:

```
public class Circle implements Shape {

    @Override
    public String name() {
        return "Circle";
    }
}
```

And also the Square class:

```
public class Square implements Shape {

    @Override
    public String name() {
        return "Square";
    }
}
```

Finally, it’s time to see polymorphism in action using our Shape interface and its implementations. Let’s instantiate some Shape objects, add them to a List, and, finally, print their names in a loop:

```
List<Shape> shapes = new ArrayList<>();
Shape circleShape = new Circle();
Shape squareShape = new Square();

shapes.add(circleShape);
shapes.add(squareShape);

for (Shape shape : shapes) {
    System.out.println(shape.name());
}
```

4. Default Methods in Interfaces

Traditional interfaces in Java 7 and below don’t offer backward compatibility. What this means is that if you have legacy code written in Java 7 or earlier, and you decide to add an abstract method to an existing interface, then all the classes that implement that interface must override the new abstract method. Otherwise, the code will break.

Java 8 solved this problem by introducing the default method that is optional and can be implemented at the interface level.

```
public interface Shape {
    default void draw() {
        System.out.println("Drawing a Shape");
    }
}
```

Summmary

Interfaces are used in the following scenarios:

- It is used to achieve abstraction.
- Due to multiple inheritance, it can achieve loose coupling.
- Define a common behavior for unrelated classes.

Abstract Classes

There are many cases when implementing a contract where we want to postpone some parts of the implementation to be completed later. We can easily accomplish this in Java through abstract classes. Before diving into when to use an abstract class, let’s look at their most relevant characteristics:

- We define an abstract class with the abstract modifier preceding the class keyword
- An abstract class can be subclassed, but it can’t be instantiated
- If a class defines one or more abstract methods, then the class itself must be declared abstract
- An abstract class can declare both abstract and concrete methods
- A subclass derived from an abstract class must either implement all the base class’s abstract methods or be abstract itself To better understand these concepts, we’ll create a simple example.

How to create an abstract class?

Let us create an abstract class for a Person You can create an abstract class by using the abstract keyword. Similarly, you can create an abstract method by using the abstract keyword.

```
public abstract Person {

    public abstract String getName();
    public abstract String getEmail();
}
```

Now let's create a class that extends the Person abstract class:

```
public class User extends Person {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public String getEmail() {
        return email;
    }
}
```

Why use an abstract class?

Now, let’s analyze a few typical scenarios where we should prefer abstract classes over interfaces and concrete classes:

- It is used to achieve abstraction.
- It can have abstract methods and non-abstract methods.
- When you don't want to provide the implementation of a method, you can make it abstract.
- When you don't want to allow the instantiation of a class, you can make it abstract.
- We want to encapsulate some common functionality in one place (code reuse) that multiple, related subclasses will share
- We need to partially define an API that our subclasses can easily extend and refine The subclasses need to inherit one or more common methods or fields with protected access modifiers
- Moreover, since the use of abstract classes implicitly deals with base types and subtypes, we’re also taking advantage of Polymorphism.

Sample Code Let’s have our base abstract class define the abstract API of a board game:

```
public abstract class BoardGame {

    //... field declarations, constructors

    public abstract void play();

    //... concrete methods
}
```

Then, we can create a subclass that implements the play method:

```
public class Checkers extends BoardGame {

    public void play() {
        //... implementation
    }
}
```

Reading List

- Comparators & Comparable Interface
- Interface Inheritance
- Functional Interfaces in Java 8
- Sample Java Collections Interface - Queue Interface
- Duck Typing
- OOP in Python