Theory: Interface

35 minutes 0 / 5 problems solved

§1. Why interface

The general idea of OOP and one of its principles is an abstraction. It means that real-world objects can be represented by their abstract models. Designing models is about focusing on the essential features of the objects and discarding the others. To understand what it means, let's take a look at a pencil. A pencil is an object that we can use to draw. Other properties such as material or length may be important to us sometimes but do not define the idea of a pencil.

Imagine we need to create a graphical editor program. One of the basic functions of the program is drawing. Before drawing the program asks a user to select a drawing tool. It can be a pen, pencil, brush, highlighter, marker, spray, and others. Each tool from a set has its own specific features: a pencil and a spray leave different marks and that matters. But there is also an essential feature that unites them: the ability to draw.

Now let's consider `Pencil` class, which is an abstraction of a pencil. As we already discussed the class at least should have `draw` method, that accepts a model of a curve. This is a crucial function of a pencil for our program. Suppose `Curve` is a class that represents some curve:

```
class Pencil {
    ...
    public void draw(Curve curve) {...}
}
```

Let's define classes for other tools, for example, a brush:

```
class Brush {
    ...
    public void draw(Curve curve) {...}
}
```

Each of them has method `draw`, although does it in its own fashion. The ability to draw is a common feature for all of them. Let's call this feature `DrawingTool`. Then we can say that if a class has `DrawingTool` feature, then it should be able to draw, that means the class should have `void draw(Curve curve) {...}` method.

Java allows declaring this feature by introducing interfaces. This is how our interface looks like:

```
interface DrawingTool {
    void draw(Curve curve);
}
```

It declares `draw` method without implementation.

Now let's mark classes that are able to draw by adding `implements DrawingTool` to the class declaration. If a class implements an interface, it has to implement all declared methods:

```
class Pencil implements DrawingTool {
    ...
    public void draw(Curve curve) {...}
}

class Brush implements DrawingTool {
    ...
    public void draw(Curve curve) {...}
}
```

Current topic:

Interface

Topic depends on:

✕ Polymorphism

Topic is required for:

Abstract class vs interface

Default methods

✓ Anonymous classes

Strategy

Command

Builder

Decorator

Observer

Abstract factory

The Collections Framework overview

Standard logger

Custom threads

Now just a quick look at the class declaration is enough to understand that class is able to draw. In other words, the main idea of an interface is *declaring functionality*.

Another important profit of introducing interfaces is that you can use them as a type:

```
DrawingTool pencil = new Pencil();
DrawingTool brush = new Brush();
```

Now both a pencil and a brush objects have the same type. It means that both classes can be treated in a similar way as a `DrawingTool`. This is another way of supporting **polymorphism**, which helps to design reusable drawing function of the graphical editor program.

```
void drawCurve(DrawingTool tool, Curve curve) {
    System.out.println("Drawing a curve " + curve + " using a " + tool);
    tool.draw(curve);
}
```

In many cases, it is more important to know what an object can do, instead of how it does what it does. This is a reason why interfaces are commonly used for declaring a type of variable.

§2. Declaring interfaces

An interface can be considered as a special kind of a class that can't be instantiated. To declare an interface you should write the keyword `interface` instead of `class` before the name of the interface:

```
interface Interface { }
```

An interface can contain:

- public constants;
- abstract methods without an implementation (the keyword `abstract` is not required here);
- default methods with implementation (the keyword `default` is required);
- static methods with implementation (the keyword `static` is required).
- private methods with implementation

> If the modifiers are not specified once the method is declared, its parameters will be **public abstract** by default.

Keyword `abstract` before a method means that the method cannot have a body, just declare a signature. `default` methods will be discussed further in details.

An interface can't contain fields (only **constants**), constructors, and non-public abstract methods. Let's declare an interface containing all possible members:

Table of contents:

↑ Interface

§1. Why interface

§2. Declaring interfaces

§3. Implementing interfaces

§4. Implementing and extending multiple interfaces

§5. Marker interfaces

§6. Static methods

§7. Conclusion

Feedback & Comments

```
interface Interface {


 int INT_CONSTANT = 0; // it's a constant, the same as public static final int
INT_FIELD = 0

    void instanceMethod1();

    void instanceMethod2();

    static void staticMethod() {

        System.out.println("Interface: static method");

    }



    default void defaultMethod() {

    System.out.println("Interface: default method. It can be overridden");

    }



    private void privateMethod() {

    System.out.println("Interface: private methods in interfaces are acceptabl
e but should have a body");

    }

}
```

Static, default, and private methods should have an implementation in the interface!

Let's take a closer look at this interface. The variable `INT_CONSTANT` is not a field here, it's a static final constant. Two methods `instanceMethod1()` and `instanceMethod2()` are abstract methods. The `staticMethod()` is just a regular static method. The default method `defaultMethod()` has an implementation but it can be overridden in subclasses. The `privateMethod` has an implementation as well and can be used to decompose `default` methods.

§3. Implementing interfaces

A class can implement an interface using the keyword `implements`. We should provide implementations for all abstract methods of the interface.

Let's implement the interface we've considered earlier:

```
class Class implements Interface {

    @Override
    public void instanceMethod1() {
        System.out.println("Class: instance method1");
    }

    @Override
    public void instanceMethod2() {

        System.out.println("Class: instance method2");

    }

}
```

Now we can create an instance of the class and call its methods:

```
Interface instance = new Class();

instance.instanceMethod1(); // it prints "Class: instance method1"
instance.instanceMethod2(); // it prints "Class: instance method2"

instance.defaultMethod();   // it prints "Interface: default method. It can be overridden"
```

Note that `instance` variable has `Interface` type, although it is ok to use `Class` for denoting type.

```
Class instance = new Class();
```

§4. Implementing and extending multiple interfaces

One of the important interface features is **multiple inheritance**.

A class can implement multiple interfaces:

```
interface A { }
interface B { }
interface C { }

class D implements A, B, C { }
```

An interface can extend one or more other interfaces using the keyword `extends`:

```
interface A { }
interface B { }
interface C { }

interface E extends A, B, C { }
```

A class can also extend another class and implement multiple interfaces:

```
class A { }

interface B { }
interface C { }

class D extends A implements B, C { }
```

All the examples above do not pose any problems.

Multiple inheritance of interfaces is often used in the Java standard class library. The class String, for example, implements three interfaces at once:

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
// ...
}
```

§5. Marker interfaces

In some situations, an interface can have no members at all. Such interfaces are called **marker** or **tagged interfaces**. For example, a well-known interface `Serializable` is a marker interface:

```
public interface Serializable{
}
```

Other examples of marker interfaces are `Cloneable`, `Remote`, etc. They are used to provide essential information to the JVM.

§6. Static methods

You can declare and implement a static method in an interface

```
interface Car {
    static float convertToMilesPerHour(float kmh) {
        return 0.62 * kmh;
    }
}
```

To use a static method you just need to invoke it directly from an interface

```
Car.convertToMilesPerHour(4.5);
```

The main purpose of interface static methods is defining utility functionality that is common for all classes implementing the interface. They help to avoid code duplication and creating additional utility classes.

§7. Conclusion

An interface is a special kind of class that cannot be instantiated. The main idea of an interface is declaring functionality. Interfaces help to abstract from specific classes and emphasize the functionality. It makes software design more reusable and clean. This is a good practice to design classes by using interfaces instead of classes. To implement an interface keyword `implements` is used. Opposite to a class, an interface can extend several interfaces. A class can implement multiple interfaces.

Report a typo

453 users liked this theory. **13** didn't like it. **What about you?**

Start practicing

Comments (23) Hints (0) Useful links (2) Show discussion