Theory: Default methods

13 minutes 0 / 5 problems solved Skip this topic Start practicing

219 users solved this topic. Latest completion was about 2 hours ago.

As you probably remember, interface methods are abstract by default. It means that they can't have a body, instead, they just declare a signature. Some kind of methods can have a body nevertheless. Such methods are called `default` and are available since Java 8.

Current topic:

Default methods

Topic depends on:

× Interface

Topic is required for:

Functional interfaces

Table of contents:

§1. Methods with a body

Default methods are opposite to abstract ones. They have an implementation:

```java
interface Feature {
    default void action() {
        System.out.println("Default action");
    }
}
```

To denote that method is a `default`, the keyword `default` is reserved. Remember, that an interface method is treated as `abstract` by default. So you need to indicate this explicitly by putting `default` keyword before methods with a body, otherwise, a compilation error happens.

Although default methods are implemented, you cannot invoke them directly from an interface like `Feature.action()`. You still need to have an object of a class that implements the interface:

```java
class FeatureImpl implements Feature {
}
...

Feature feature = new FeatureImpl();
feature.action(); // Default action
```

If you want to customize a default method in a class, just override it like a regular method:

```java
class FeatureImpl implements Feature {
    public void action() {
        System.out.println("FeatureImpl specific action");
    }
}
...

Feature feature = new FeatureImpl();
feature.action(); // FeatureImpl-specific action
```

Sometimes default methods are huge. To make it possible to decompose such methods, Java allows declaring private methods inside an interface:

```java
interface Feature {
    default void action() {
        String answer = subAction();
        System.out.println(answer);
    }

    private String subAction() {
        return "Default action";
    }
}
```

§2. Why are they needed

As it was mentioned in the Interface topic, the main idea of an interface is declaring functionality. Default methods extend that idea. They don't just declare functionality but also implement it. The main reason is supporting backward compatibility. Let's consider an example.

Suppose you program a game that has several types of characters. These characters are able to move within a map. That is represented by `Movable` interface:

```
interface Movable {
    void stepAhead();
    void turnLeft();
    void turnRight();
}
```

So we have the interface and many classes that implement it. For example, a `Batman` character:

```
class Batman implements Movable {
    public void stepAhead() {...}
    public void turnLeft() {...}
    public void turnRight() {...}
}
```

Once you decide that characters should be able to turn around. It means you need to add `turnAround` method to the `Movable`. You may implement the method for all classes implementing the interface. Another way is declaring a `default` method in the interface. Then you don't have to implement it in all classes.

Another example when the situation is getting even worse is when we are talking about interfaces that are part of the Java standard library. Suppose Java maintainers decided to extend a commonly used interface with a new method in the next release. It means if you are going to upgrade the Java version and there are classes implementing the interface in your code, you have to implement the new method. Otherwise, your code won't compile.

Sometimes default methods help to avoid code duplication. Indeed in our case `turnAround` methods may look the same for all classes.

```
interface Movable {
    void stepAhead();
    void turnLeft();
    void turnRight();

    default void turnAround() {
        turnLeft();
        turnLeft();
    }

}
```

If you want to customize a default implementation for `Batman`, just override it:

```
class Batman implements Movable {
    public void stepAhead() {...}
    public void turnLeft() {...}
    public void turnRight() {...}
    public void turnAround() {
        turnRight();
        turnRight();
    }
}
```

§3. The diamond problem

Suppose we have another interface `Jumpable` that represents the ability to jump. The interface contains abstract methods for jumping in place, jumping with turning left and right. It also has a `default` method for a turnaround jumping with the same signature as `Movable`.

```
interface Jumpable {
    void jump();
    void turnLeftJump();
    void turnRightJump();
    default void turnAround() {
        turnLeftJump();
        turnLeftJump();
    }
}
```

`Spiderman` has both abilities of `Movable` and `Jumpable`, so its class implements both interfaces. Note, interfaces have `default` method `turnAround` with the same signature, but different implementations. Which one should be chosen for the class? To avoid ambiguity, the compiler force a programmer to provide the implementation explicitly, otherwise it raises a compilation exception.

```
class Spiderman implements Movable, Jumpable {
    // define an implementation for abstract methods
    public void stepAhead() {...}
    public void turnLeft() {...}
    public void turnRight() {...}
    public void jump() {…}
    public void turnLeftJump() {...}
    public void turnRightJump() {...}


    // define an implementation for conflicted default method

    public void turnAround() {

        // define turnaround for the spiderman

    }

}
```

You can also choose one of the default implementations instead of writing your own.

```
class Spiderman implements Movable, Jumpable {
    ...
    public void turnAround() {
        Movable.super.turnAround();
    }
}
```

The problem is known as **the diamond problem**.

§4. Conclusion

Some interface methods have a body. Such methods are called default and have `default` keyword before a signature. The main idea of supporting default methods is providing backward compatibility. That allows you to add new methods to existed interface without changing all classes that implements the interface. Remember, that if a class implements several interfaces and some of them have a default method with the same signature, you have to define implementation for the method in the class. This happens because the compiler can not decide what implementation should be used.

Report a typo

29 users liked this theory. 1 didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

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