(http://baeldung.com)

Static and Default Methods in Interfaces in Java

Last modified: February 20, 2018

by baeldung (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/) +

1. Overview

Java 8 brought to the table a few brand new features, including lambda expressions (https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html), functional interfaces (http://www.baeldung.com/java-8-functional-interfaces), method references (http://www.baeldung.com/java-8-double-colon-operator), streams (http://www.baeldung.com/java-8-streams), Optional (http://www.baeldung.com/java-optional), and *static* and *default* methods in interfaces.

Some of them have been already covered in this article (http://www.baeldung.com/java-8-new-features). Nonetheless, *static* and *default* methods in interfaces deserve a deeper look on their own.

In this article, we'll discuss in depth **how to use *static* and *default* methods in interfaces** and go through some use cases where they can be useful.

2. Why Default Methods in Interfaces Are Needed

Like regular interface methods, **default methods are implicitly public** — there's no need to specify the *public* modifier.

Unlike regular interface methods, they are **declared with the *default* keyword at the beginning of the method signature**, and they **provide an implementation**.

Let's see a simple example:

```java
public interface MyInterface {

    // regular interface methods

    default void defaultMethod() {
        // default method implementation
    }
}
```

The reason why *default* methods were included in the Java 8 release is pretty obvious.

In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface, all the implementations will be forced to implement them too. Otherwise, the design will just break down.

Default interface methods are an efficient way to deal with this issue. They **allow us to add new methods to an interface that are automatically available in the implementations**. Thus, there's no need to modify the implementing classes.

In this way, **backward compatibility is neatly preserved** without having to refactor the implementers.

3. Default Interface Methods in Action

To better understand the functionality of *default* interface methods, let's create a simple example.

Say that we have a naive *Vehicle* interface and just one implementation. There could be more, but let's keep it that simple:

```
public interface Vehicle {

    String getBrand();

    String speedUp();

    String slowDown();

    default String turnAlarmOn() {
        return "Turning the vehicle alarm on.";
    }

    default String turnAlarmOff() {
        return "Turning the vehicle alarm off.";
    }
}
```

And let's write the implementing class:

```java
public class Car implements Vehicle {

    private String brand;

    // constructors/getters

    @Override
    public String getBrand() {
        return brand;
    }

    @Override
    public String speedUp() {
        return "The car is speeding up.";
    }

    @Override
    public String slowDown() {
        return "The car is slowing down.";
    }
}
```

Lastly, let's define a typical *main* class, which creates an instance of *Car* and calls its methods:

```java
public static void main(String[] args) {
    Vehicle car = new Car("BMW");
    System.out.println(car.getBrand());
    System.out.println(car.speedUp());
    System.out.println(car.slowDown());
    System.out.println(car.turnAlarmOn());
    System.out.println(car.turnAlarmOff());
}
```

Please notice how the *default* methods *turnAlarmOn()* and *turnAlarmOff()* from our *Vehicle* interface are **automatically available in the *Car* class**.

Furthermore, if at some point we decide to add more *default* methods to the *Vehicle* interface, the application will still continue working, and we won't have to force the class to provide implementations for the new methods.

The most typical use of default methods in interfaces is **to incrementally provide additional functionality to a given type without breaking down the implementing classes.**

In addition, they can be used to **provide additional functionality around an existing abstract method**:

```
public interface Vehicle {

    // additional interface methods

    double getSpeed();

    default double getSpeedInKMH(double speed) {
       // conversion
    }
}
```

4. Multiple Interface Inheritance Rules

Default interface methods are a pretty nice feature indeed, but with some caveats worth mentioning. Since Java allows classes to implement multiple interfaces, it's important to know **what happens when a class implements several interfaces that define the same *default* methods**.

To better understand this scenario, let's define a new *Alarm* interface and refactor the *Car* class:

```
public interface Alarm {

    default String turnAlarmOn() {
        return "Turning the alarm on.";
    }

    default String turnAlarmOff() {
        return "Turning the alarm off.";
    }
}
```

With this new interface defining its own set of *default* methods, the *Car* class would implement both *Vehicle* and *Alarm*:

```
public class Car implements Vehicle, Alarm {
    // ...
}
```

In this case, **the code simply won't compile, as there's a conflict caused by multiple interface inheritance** (a.k.a the Diamond Problem (https://en.wikipedia.org/wiki/Multiple_inheritance)). The *Car* class would inherit both sets of *default* methods. Which ones should be called then?

To solve this ambiguity, we must explicitly provide an implementation for the methods:

```java
@Override
public String turnAlarmOn() {
    // custom implementation
}

@Override
public String turnAlarmOff() {
    // custom implementation
}
```

We can also **have our class use the *default* methods of one of the interfaces**.

Let's see an example that uses the *default* methods from the *Vehicle* interface:

```java
@Override
public String turnAlarmOn() {
    return Vehicle.super.turnAlarmOn();
}

@Override
public String turnAlarmOff() {
    return Vehicle.super.turnAlarmOff();
}
```

Similarly, we can have the class use the *default* methods defined within the *Alarm* interface:

```java
@Override
public String turnAlarmOn() {
    return Alarm.super.turnAlarmOn();
}

@Override
public String turnAlarmOff() {
    return Alarm.super.turnAlarmOff();
}
```

Furthermore, it's even **possible to make the *Car* class use both sets of default methods**:

```
@Override
public String turnAlarmOn() {
    return Vehicle.super.turnAlarmOn() + " " + Alarm.super.turnAlarmOn()
}

@Override
public String turnAlarmOff() {
    return Vehicle.super.turnAlarmOff() + " " + Alarm.super.turnAlarmOff
}
```

5. Static Interface Methods

Aside from being able to declare *default* methods in interfaces, **Java 8 allows us to define and implement *static* methods in interfaces**.

Since *static* methods don't belong to a particular object, they are not part of the API of the classes implementing the interface, and they have to be **called by using the interface name preceding the method name**.

To understand how *static* methods work in interfaces, let's refactor the *Vehicle* interface and add to it a *static* utility method:

```
public interface Vehicle {

    // regular / default interface methods

    static int getHorsePower(int rpm, int torque) {
        return (rpm * torque) / 5252;
    }
}
```

Defining a *static* method within an interface is identical to defining one in a class. Moreover, a *static* method can be invoked within other *static* and *default* methods.

Now, say that we want to calculate the horsepower (https://en.wikipedia.org/wiki/Horsepower) of a given vehicle's engine. We just call the *getHorsePower()* method:

```
Vehicle.getHorsePower(2500, 480));
```

The idea behind *static* interface methods is to provide a simple mechanism that allows us to **increase the degree of cohesion (https://en.wikipedia.org/wiki/Cohesion_(computer_science))** of a design by putting together related methods in one single place without having to create an object.

Pretty much **the same can be done with abstract classes.** The main difference lies in the fact that **abstract classes can have constructors, state, and behavior**.

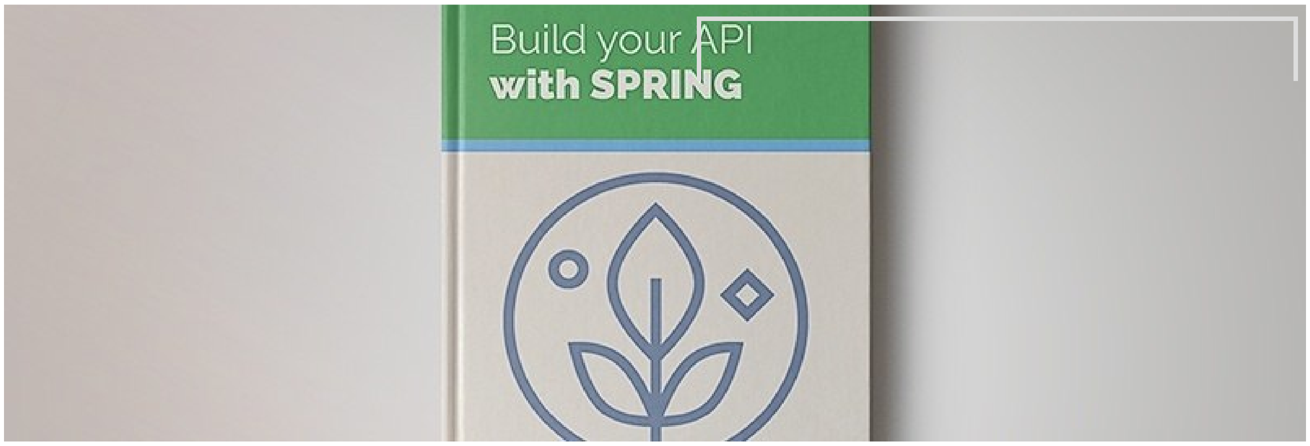
Furthermore, static methods in interfaces make possible to group related utility methods, without having to create artificial utility classes that are simply placeholders for static methods.

6. Conclusion

In this article, we explored in depth the use of *static* and *default* interface methods in Java 8. At first glance, this feature may look a little bit sloppy, particularly from an object-oriented purist perspective. Ideally, interfaces shouldn't encapsulate behavior and should be used only for defining the public API of a certain type.

When it comes to maintaining backward compatibility with existing code, however, *static* and *default* methods are a good trade-off.

And, as usual, all the code samples shown in this article are available over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-8).



(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-main-1.2.0.jpg)

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-icn-1.0.0.png)

Learning to "Build your API **with Spring**"?

Enter your Email Address

>> Get the eBook

CATEGORIES

SPRING (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
HTTPCLIENT 4 TUTORIAL (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
SPRING PERSISTENCE TUTORIAL (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SECURITY WITH SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
THE COURSES (HTTP://COURSES.BAELDUNG.COM)
CONSULTING WORK (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
WRITE FOR BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACT (HTTP://WWW.BAELDUNG.COM/CONTACT)
COMPANY INFO (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
TERMS OF SERVICE (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
PRIVACY POLICY (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)
EDITORS (HTTP://WWW.BAELDUNG.COM/EDITORS)

MEDIA KIT (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)