**Overview**

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

We've already covered a few of these features in [another article](https://www.baeldung.com/java-8-new-features). Nonetheless, *static* and *default* methods in interfaces deserve a deeper look on their own.

In this tutorial, we'll learn **how to use *static* and *default* methods in interfaces,** and discuss some situations where they can be useful.

**2. Why Interfaces Need Default Methods**

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

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

Let's look at a simple example:

**public** **interface** **MyInterface** {

// regular interface methods

**default** **void** **defaultMethod**() {

// default method implementation

}

}Copy

The reason why the Java 8 release included *default* methods 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**. Therefore, we don't 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.

Suppose 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.";

}

}

Now let's write the implementing class:

**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.";

}

}

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

**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 common use of interface default methods is **to incrementally provide additional functionality to a given type without breaking down the implementing classes.**

In addition, we can use them 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, but there are 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. So which ones should we call?

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

@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:

@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:

@Override

**public** String **turnAlarmOn**() {

**return** Alarm.super.turnAlarmOn();

}

@Override

**public** String **turnAlarmOff**() {

**return** Alarm.super.turnAlarmOff();

}

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**

In addition to declaring *default* methods in interfaces, **Java 8 also allows us to define and implement *static* methods in interfaces**.

Since *static* methods don't belong to a particular object, they're not part of the API of the classes implementing the interface; therefore, 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 a *static* utility method to it:

**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.

Let's suppose 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));

Copy

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.

**The same can pretty much be done with abstract classes.** The main difference is that **abstract classes can have constructors, state, and behavior**.

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