(/)

New Features in Java 8

Last updated: January 16, 2024

Written by: baeldung (https://www.baeldung.com/author/baeldung)

Baeldung (/)

Reviewed by: Slaviša Avramović (https://www.baeldung.com/editor/slavisa-author)

Core Java (https://www.baeldung.com/category/java/core-java)

>= Java 8 (https://www.baeldung.com/tag/jdk8-and-later)

This article is part of a series:

1. Overview

(/)

In this tutorial, we'll have a quick look at some of the most interesting new features in Java 8.

We'll talk about interface default and static methods, method reference and Optional.

We have already covered some the features of the Java 8 release — stream API (/java-8-streams-introduction), lambda expressions and functional interfaces (/java-8-lambda-expressions-tips) — as they're comprehensive topics that deserve a separate look.

2. Interface Default and Static Methods

Before Java 8, interfaces could have only public abstract methods. It was not possible to add new functionality to the existing interface without forcing all implementing classes to create an implementation of the new methods, nor was it possible to create interface methods with an implementation.

Starting with Java 8, interfaces can have ***static*** and ***default*** methods that, despite being declared in an interface, have a defined behavior.

2.1. Static Method

Consider this method of the interface (let's call this interface *Vehicle*):

```
static String producer() {
    return "N&F Vehicles";
}
```

The static *producer()* method is available only through and inside of an interface. It can't be overridden by an implementing class.

To call it outside the interface, the standard approach for static method call should be used:

```
String producer = Vehicle.producer();
```

2.2. Default Method

Default methods are declared using the new ***default* keyword.** These are accessible through the instance of the implementing class and can be overridden.

Let's add a *default* method to our *Vehicle* interface, which will also make a call to the *static* method of this interface:

```
default String getOverview() {
    return "ATV made by " + producer();
}
```

Assume that this interface is implemented by the class *VehicleImpl*.

For executing the *default* method, an instance of this class should be created:

```
Vehicle vehicle = new VehicleImpl();
String overview = vehicle.getOverview();
```

3. Method References

Method reference can be used as a shorter and more readable alternative for a lambda expression that only calls an existing method. There are four variants of method references.

3.1. Reference to a Static Method

The reference to a static method holds the syntax ***ContainingClass::methodName*.**

We'll try to count all empty strings in the *List<String>* with the help of Stream API:

```
boolean isReal = list.stream().anyMatch(u -> User.isRealUser(u));
```

Let's take a closer look at lambda expression in the *anyMatch()* method. It just makes a call to a static method *isRealUser(User user)* of the *User* class.

So, it can be substituted with a reference to a static method:

```
boolean isReal = list.stream().anyMatch(User::isRealUser);
```

This type of code looks much more informative.

3.2. Reference to an Instance Method

The reference to an instance method holds the syntax ***containingInstance::methodName***.

The following code calls method *isLegalName(String string)* of type *User*, which validates an input parameter:

```
User user = new User();
boolean isLegalName = list.stream().anyMatch(user::isLegalName);
```

3.3. Reference to an Instance Method of an Object of a Particular Type

This reference method takes the syntax ***ContainingType::methodName***.

Let's look at an example:

```
long count = list.stream().filter(String::isEmpty).count();
```

3.4. Reference to a Constructor

A reference to a constructor takes the syntax ***ClassName::new***.

As constructor in Java is a special method, method reference could be applied to it too, with the help of ***new*** as a method name:

```
Stream<User> stream = list.stream().map(User::new);
```

4. *Optional<T>*

Before Java 8, developers had to carefully validate values they referred to because of the possibility of throwing the *NullPointerException (NPE)*. All these checks demanded a pretty annoying and error-prone boilerplate code.

Java 8 *Optional<T>* class can help to handle situations where there is a possibility of getting the *NPE*. It works as a container for the object of type *T*. It can return a value of this object if this value is not a *null*. When the value inside this container is *null*, it allows doing some predefined actions instead of throwing *NPE*.

4.1. Creation of the *Optional<T>*

An instance of the *Optional* class can be created with the help of its static methods.

Let's look at how to return an empty *Optional*:

```
Optional<String> optional = Optional.empty();
```

Next, we return an *Optional* that contains a non-null value:

```
String str = "value";
Optional<String> optional = Optional.of(str);
```

Finally, here's how to return an *Optional* with a specific value or an empty *Optional* if the parameter is *null*:

```
Optional<String> optional = Optional.ofNullable(getString());
```

4.2. Optional<T> Usage (/)

Let's say we expect to get a *List<String>*, and in the case of *null*, we want to substitute it with a new instance of an *ArrayList<String>*.

With pre-Java 8's code, we need to do something like this:

```
List<String> list = getList();
List<String> listOpt = list != null ? list : new ArrayList<>();
```

With Java 8, the same functionality can be achieved with a much shorter code:

```
List<String> listOpt = getList().orElseGet(() -> new ArrayList<>());
```

There is even more boilerplate code when we need to reach some object's field in the old way.

Assume we have an object of type *User* that has a field of type *Address* with a field s*treet* of type *String*, and we need to return a value of the *street* field if some exist or a default value if *street* is *null*:

```
User user = getUser();
if (user != null) {
    Address address = user.getAddress();
    if (address != null) {
        String street = address.getStreet();
        if (street != null) {
            return street;
        }
    }
}
return "not specified";
```

This can be simplified with *Optional*:

```
Optional<User> user = Optional.ofNullable(getUser());
String result = user
  .map(User::getAddress)
  .map(Address::getStreet)
  .orElse("not specified");
```

In this example, we used the *map()* method to convert results of calling the *getAdress()* to the *Optional<Address>* and *getStreet()* to *Optional<String>*. If any of these methods returned *null*, the *map()* method would return an empty *Optional*.

Now imagine that our getters return *Optional<T>*.

In this case, we should use the *flatMap()* method instead of the *map()*:

```
Optional<OptionalUser> optionalUser = Optional.ofNullable(getOptionalUser());
String result = optionalUser
  .flatMap(OptionalUser::getAddress)
  .flatMap(OptionalAddress::getStreet)
  .orElse("not specified");
```

Another use case of *Optional* is changing *NPE* with another exception.

So, as we did previously, let's try to do this in pre-Java 8's style:

```
String value = null;
String result = "";
try {
    result = value.toUpperCase();
} catch (NullPointerException exception) {
    throw new CustomException();
}
```

And the answer is more readable and simpler if we use *Optional<String>*:

```
String value = null;
Optional<String> valueOpt = Optional.ofNullable(value);
String result = valueOpt.orElseThrow(CustomException::new).toUpperCase();
```

Notice that how to use *Optional* in our app and for what purpose is a serious and controversial design decision, and explanation of all its pros and cons is out of the scope of this article. But there are plenty of interesting articles devoted to this problem. This one (http://blog.joda.org/2014/11/optional-in-java-se-8.html) and this one (http://blog.joda.org/2015/08/java-se-8-optional-pragmatic-approach.html) could be very helpful to dig deeper.

5. Conclusion

(/)

In this article, we briefly discussed some interesting new features in Java 8.

There are of course many other additions and improvements spread across many Java 8 JDK packages and classes.

But the information illustrated in this article is a good starting point for exploring and learning about some of these new features.

Finally, all the source code for the article is available over on GitHub. (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-8)

Next »
New Features in Java 9 (/new-java-9)

SECURITY WITH SPRING (/SECURITY-SPRING)

(/)

SPRING REACTIVE SERIES (/SPRING-REACTIVE-SERIES)

ABOUT

ABOUT BAELDUNG (/ABOUT)

THE FULL ARCHIVE (/FULL_ARCHIVE)

EDITORS (/EDITORS)

OUR PARTNERS (/PARTNERS/)

PARTNER WITH BAELDUNG (/PARTNERS/WORK-WITH-US)

EBOOKS (/LIBRARY/)

FAQ (HTTPS://WWW.BAELDUNG.COM/LIBRARY/FAQ)

BAELDUNG PRO (/MEMBERS/)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)