(http://baeldung.com)

The Double Colon Operator in Java 8

Last modified: August 31, 2017

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

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

Java 8 (http://www.baeldung.com/tag/java-8/)

1. Overview

In this quick article, we'll discuss the **double colon operator** (*::*) in Java 8 and go over the scenarios where the operator can be used.

2. From Lambdas to Double Colon Operator

With Lambdas expressions, we've seen that code can become very concise.

For example, to **create a comparator**, the following syntax is enough:

```
Comparator c = (Computer c1, Computer c2) -> c1.getAge().compareTo(c2.ge
```

Then, with type inference:

```
Comparator c = (c1, c2) -> c1.getAge().compareTo(c2.getAge());
```

But can we make the code above even more expressive and readable? Let's have a look:

```
Comparator c = Comparator.comparing(Computer::getAge);
```

We've used the *::* operator as shorthand for lambdas calling a specific method – by name. And the end, result is of course even more readable syntax.

3. How Does It Work?

Very simply put, when we are using a method reference – the target reference is placed before the delimiter *::* and the name of the method is provided after it.

For example:

```
Computer::getAge;
```

We're looking at a method reference to the method *getAge* defined in the *Computer* class.

We can then operate with that function:

```
Function<Computer, Integer> getAge = Computer::getAge;
Integer computerAge = getAge.apply(c1);
```

Notice that we're referencing the function – and then applying it to the right kind of argument.

4. Method References

We can make good use of this operator in quite some scenarios.

4.1. A Static Method

First, we're going to make use of **a static utility method**:

```
List inventory = Arrays.asList(
  new Computer( 2015, "white", 35), new Computer(2009, "black", 65));
inventory.forEach(ComputerUtils::repair);
```

4.2. An Instance Method of an Existing Object

Next, let's have a look at an interesting scenario – **referencing a method of an existing object instance**.

We're going to use the variable *System.out* – an object of type *PrintStream* which supports the *print* method:

```
Computer c1 = new Computer(2015, "white");
Computer c2 = new Computer(2009, "black");
Computer c3 = new Computer(2014, "black");
Arrays.asList(c1, c2, c3).forEach(System.out::print);
```

4.3. An Instance Method of an Arbitrary Object of a Particular Type

```
1  Computer c1 = new Computer(2015, "white", 100);
2  Computer c2 = new MacbookPro(2009, "black", 100);
3  List inventory = Arrays.asList(c1, c2);
4  inventory.forEach(Computer::turnOnPc);
```

As you can see, we're referencing the *turnOnPc* method not on a specific instance, but on the type itself.

At line 4 the instance method *turnOnPc* will be called for every object of *inventory*.

And this naturally means that – for *c1* the method *turnOnPc* will be called on the *Computer* instance and for *c2* on *MacbookPro* instance.

4.4. A Super Method of a Particular Object

Suppose you have the following method in the *Computer* superclass:

```
public Double calculateValue(Double initialValue) {
    return initialValue/1.50;
}
```

and this one in *MacbookPro* subclass:

```
@Override
public Double calculateValue(Double initialValue){
    Function<Double, Double> function = super::calculateValue;
    Double pcValue = function.apply(initialValue);
    return pcValue + (initialValue/10) ;
}
```

A call to *calculateValue* method on a *MacbookPro* instance:

```
macbookPro.calculateValue(999.99);
```

will also produce also a call to *calculateValue* on the *Computer* superclass.

5. Constructor References

5.1. Create a New Instance

Referencing a constructor to instantiate an object can be quite simple:

```
@FunctionalInterface
public interface InterfaceComputer {
    Computer create();
}

InterfaceComputer c = Computer::new;
Computer computer = c.create();
```

What if you have two parameters in a constructor?

```
BiFunction<Integer, String, Computer> c4Function = Computer::new;
Computer c4 = c4Function.apply(2013, "white");
```

If parameters are three or more you have to define a new Functional interface:

```
@FunctionalInterface
interface TriFunction<A, B, C, R> {
    R apply(A a, B b, C c);
    default <V> TriFunction<A, B, C, V> andThen( Function<? super R, ? e
        Objects.requireNonNull(after);
        return (A a, B b, C c) -> after.apply(apply(a, b, c));
    }
}
```

Then, initialize your object:

```
TriFunction <Integer, String, Integer, Computer> c6Function = Computer::
Computer c3 = c6Function.apply(2008, "black", 90);
```

5.2. Create an Array

Finally, let's see how to create an array of *Computer* objects with five elements:

```
Function <Integer, Computer[]> computerCreator = Computer[]::new;
Computer[] computerArray = computerCreator.apply(5);
```

6. Conclusion

As we're starting to see, the double colon operator – introduced in Java 8 – will be very useful in some scenarios, and especially in conjunction with Streams.

It's also quite important to have a look at functional interfaces for a better understanding of what happens behind the scenes.

The complete **source code** for the example is available in this GitHub project (https://github.com/eugenp/tutorials/tree/master/core-java-8) – this is a Maven and Eclipse project so that it can be imported and used as-is.

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

(http://www.baeldung.com/wp-

content/up
loads/2016
/05/baeld
ung-rest-
post-
footer-icn-
1.0.0.png)

Subscribe ▾ ▴ newest ▴ oldest ▴ most voted

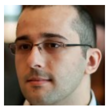
infoj (http://netjs.blogspot.com)

Guest

One point to note about method reference is that they provide a way to refer to a method, they don't execute the method.

Read more here – http://netjs.blogspot.com/2015/06/method-reference-in-java-8.html (http://netjs.blogspot.com/2015/06/method-reference-in-java-8.html)

+ 0 − 2 years ago



Eugen Paraschiv (http://www.baeldung.com/)

Guest

Definitely – the method reference syntax is not meant to trigger the execution of the method; however, in conjunction with lambdas – the execution is handled as well.

+ 0 − 2 years ago

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)