Modern Java - A Guide to Java 8

wizardforcel

Published with GitBook

Table of Contents

Modern Java - A Guide to Java 8

Author: winterbe

From: java8-tutorial

Modern Java - A Guide to Java 8

> “Java is still not dead—and people are starting to figure that out.”

Welcome to my introduction to Java 8. This tutorial guides you step by step through all new language features. Backed by short and simple code samples you'll learn how to use default interface methods, lambda expressions, method references and repeatable annotations. At the end of the article you'll be familiar with the most recent API changes like streams, functional interfaces, map extensions and the new Date API. **No walls of text, just a bunch of commented code snippets. Enjoy!**

This article was originally posted on my blog. You should follow me on Twitter.

Table of Contents

Default Methods for Interfaces

Java 8 enables us to add non-abstract method implementations to interfaces by utilizing the `default` keyword. This feature is also known as virtual extension methods.

Here is our first example:

```
interface Formula {
    double calculate(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

Besides the abstract method `calculate` the interface `Formula` also defines the default method `sqrt`. Concrete classes only have to implement the abstract method `calculate`. The default method `sqrt` can be used out of the box.

```
Formula formula = new Formula() {
    @Override
    public double calculate(int a) {
        return sqrt(a * 100);
    }
};

formula.calculate(100);     // 100.0
formula.sqrt(16);           // 4.0
```

The formula is implemented as an anonymous object. The code is quite verbose: 6 lines of code for such a simple calculation of `sqrt(a * 100)`. As we'll see in the next section, there's a much nicer way of implementing single method objects in Java 8.

Lambda expressions

Let's start with a simple example of how to sort a list of strings in prior versions of Java:

```
List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");

Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
});
```

The static utility method `Collections.sort` accepts a list and a comparator in order to sort the elements of the given list. You often find yourself creating anonymous comparators and pass them to the sort method.

Instead of creating anonymous objects all day long, Java 8 comes with a much shorter syntax, **lambda expressions**:

```
Collections.sort(names, (String a, String b) -> {
    return b.compareTo(a);
});
```

As you can see the code is much shorter and easier to read. But it gets even shorter:

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

For one line method bodies you can skip both the braces `{}` and the `return` keyword. But it gets even shorter:

```
names.sort((a, b) -> b.compareTo(a));
```

List now has a `sort` method. Also the java compiler is aware of the parameter types so you can skip them as well. Let's dive deeper into how lambda expressions can be used in the wild.

Functional Interfaces

How does lambda expressions fit into Java's type system? Each lambda corresponds to a given type, specified by an interface. A so called *functional interface* must contain **exactly one abstract method** declaration. Each lambda expression of that type will be matched to this abstract method. Since default methods are not abstract you're free to add default methods to your functional interface.