(http://baeldung.com)

Lambda Expressions and Functional Interfaces: Tips and Best Practices

Last modified: April 18, 2018

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

Now that Java 8 has reached wide usage, patterns, and best practices have begun to emerge for some of its headlining features. In this tutorial, we will take a closer look to functional interfaces and lambda expressions.

2. Prefer Standard Functional Interfaces

Functional interfaces, which are gathered in the **java.util.function (https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html)** package, satisfy most developers' needs in providing target types for lambda expressions and method references. Each of these interfaces is general and abstract, making them easy to adapt to almost any lambda expression. Developers should explore this package before creating new functional interfaces.

Consider an interface *Foo*:

```
@FunctionalInterface
public interface Foo {
    String method(String string);
}
```

and a method *add()* in some class *UseFoo*, which takes this interface as a parameter:

```
public String add(String string, Foo foo) {
    return foo.method(string);
}
```

To execute it, you would write:

```
Foo foo = parameter -> parameter + " from lambda";
String result = useFoo.add("Message ", foo);
```

Look closer and you will see that *Foo* is nothing more than a function that accepts one argument and produces a result. Java 8 already provides such an interface in *Function<T,R> (https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)* from the java.util.function (https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) package.

Now we can remove interface *Foo* completely and change our code to:

```
public String add(String string, Function<String, String> fn) {
    return fn.apply(string);
}
```

To execute this, we can write:

```
Function<String, String> fn =
  parameter -> parameter + " from lambda";
String result = useFoo.add("Message ", fn);
```

3. Use the *@FunctionalInterface* Annotation

Annotate your functional interfaces with *@FunctionalInterface (https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html).* At first, this annotation seems to be useless. Even without it, your interface will be treated as functional as long as it has just one abstract method.

But imagine a big project with several interfaces – it's hard to control everything manually. An interface, which was designed to be functional, could accidentally be changed by adding of other abstract method/methods, rendering it unusable as a functional interface.

But using the *@FunctionalInterface* annotation, the compiler will trigger an error in response to any attempt to break the predefined structure of a functional interface. It is also a very handy tool to make your application architecture easier to understand for other developers.

So, use this:

```
@FunctionalInterface
public interface Foo {
    String method();
}
```

instead of just:

```
public interface Foo {
    String method();
}
```

4. Don't Overuse Default Methods in Functional Interfaces

You can easily add default methods to the functional interface. This is acceptable to the functional interface contract as long as there is only one abstract method declaration:

```
@FunctionalInterface
public interface Foo {
    String method();
    default void defaultMethod() {}
}
```

Functional interfaces can be extended by other functional interfaces if their abstract methods have the same signature. For example:

```
@FunctionalInterface
public interface FooExtended extends Baz, Bar {}

@FunctionalInterface
public interface Baz {
    String method();
    default void defaultBaz() {}
}

@FunctionalInterface
public interface Bar {
    String method();
    default void defaultBar() {}
}
```

Just as with regular interfaces, extending different functional interfaces with the same default method can be problematic. For example, assume that interfaces *Bar* and *Baz* both have a default method *defaultCommon().* In this case, you will get a compile-time error:

```
interface Foo inherits unrelated defaults for defaultCommon() from types
```

To fix this, *defaultCommon()* method should be overridden in the *Foo* interface. You can, of course, provide a custom implementation of this method. But if you want to use one of the parent interfaces' implementations (for example, from the *Baz* interface), add following line of code to the *defaultCommon()* method's body:

```
Baz.super.defaultCommon();
```

But be careful. **Adding too many default methods to the interface is not a very good architectural decision.** It is should be viewed as a compromise, only to be used when required, for upgrading existing interfaces without breaking backward compatibility.

5. Instantiate Functional Interfaces with Lambda Expressions

The compiler will allow you to use an inner class to instantiate a functional interface. However, this can lead to very verbose code. You should prefer lambda expressions:

```
Foo foo = parameter -> parameter + " from Foo";
```

over an inner class:

```
Foo fooByIC = new Foo() {
    @Override
    public String method(String string) {
        return string + " from Foo";
    }
};
```

The lambda expression approach can be used for any suitable interface from old libraries. It is usable for interfaces like *Runnable*, *Comparator*, and so on. **However, this doesn't mean that you should review your whole older codebase and change everything.**

6. Avoid Overloading Methods with Functional Interfaces as Parameters

Use methods with different names to avoid collisions; let's look at an example:

```
public interface Adder {
    String add(Function<String, String> f);
    void add(Consumer<Integer> f);
}

public class AdderImpl implements Adder {

    @Override
    public  String add(Function<String, String> f) {
        return f.apply("Something ");
    }

    @Override
    public void add(Consumer<Integer> f) {}
}
```

At first glance, this seems reasonable. But any attempt to execute any of *AdderImpl's* methods:

```
String r = adderImpl.add(a -> a + " from lambda");
```

ends with an error with the following message:

```
reference to add is ambiguous both method
add(java.util.function.Function<java.lang.String,java.lang.String>)
in fiandlambdas.AdderImpl and method
add(java.util.function.Consumer<java.lang.Integer>)
in fiandlambdas.AdderImpl match
```

To solve this problem, you have two options. The **first** is to use methods with different names:

```
String addWithFunction(Function<String, String> f);

void addWithConsumer(Consumer<Integer> f);
```

The **second** is to perform casting manually. This is not preferred.

```
String r = Adder.add((Function) a -> a + " from lambda");
```

7. Don't Treat Lambda Expressions as Inner Classes

Despite our previous example, where we essentially substituted inner class by a lambda expression, the two concepts are different in an important way: scope.

When you use an inner class, it creates a new scope. You can overwrite local variables from the enclosing scope by instantiating new local variables with the same names. You can also use the keyword ***this*** inside your inner class as a reference to its instance.

However, lambda expressions work with enclosing scope. You can't overwrite variables from the enclosing scope inside the lambda's body. In this case, the keyword ***this*** is a reference to an enclosing instance.

For example, in the class *UseFoo* you have an instance variable *value:*

```
private String value = "Enclosing scope value";
```

Then in some method of this class place the following code and execute this method.

```
public String scopeExperiment() {
    Foo fooIC = new Foo() {
        String value = "Inner class value";

        @Override
        public String method(String string) {
            return this.value;
        }
    };
    String resultIC = fooIC.method("");

    Foo fooLambda = parameter -> {
        String value = "Lambda value";
        return this.value;
    };
    String resultLambda = fooLambda.method("");

    return "Results: resultIC = " + resultIC +
      ", resultLambda = " + resultLambda;
}
```

If you execute the *scopeExperiment()* method, you will get the following result: *Results: resultIC = Inner class value, resultLambda = Enclosing scope value*

As you can see, by calling *this.value* in IC, you can access a local variable from its instance. But in the case of the lambda, *this.value* call gives you access to the variable *value* which is defined in the *UseFoo* class, but not to the variable *value* defined inside the lambda's body.

8. Keep Lambda Expressions Short And Self-explanatory

If possible, use one line constructions instead of a large block of code. Remember **lambdas should be an expression, not a narrative.** Despite its concise syntax, **lambdas should precisely express the functionality they provide.**

This is mainly stylistic advice, as performance will not change drastically. In general, however, it is much easier to understand and to work with such code.

This can be achieved in many ways – let's have a closer look.

8.1. Avoid Blocks of Code in Lambda's Body

In an ideal situation, lambdas should be written in one line of code. With this approach, the lambda is a self-explanatory construction, which declares what action should be executed with what data (in the case of lambdas with parameters).

If you have a large block of code, the lambda's functionality is not immediately clear.

With this in mind, do the following:

```
Foo foo = parameter -> buildString(parameter);
```

```
private String buildString(String parameter) {
    String result = "Something " + parameter;
    //many lines of code
    return result;
}
```

instead of:

```
Foo foo = parameter -> { String result = "Something " + parameter;
    //many lines of code
    return result;
};
```

However, please don't use this "one-line lambda" rule as dogma. If you have two or three lines in lambda's definition, it may not be valuable to extract that code into another method.

8.2. Avoid Specifying Parameter Types

A compiler in most cases is able to resolve the type of lambda parameters with the help of **type inference (https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html)**. Therefore, adding a type to the parameters is optional and can be omitted.

Do this:

```
(a, b) -> a.toLowerCase() + b.toLowerCase();
```

instead of this:

```
(String a, String b) -> a.toLowerCase() + b.toLowerCase();
```

8.3. Avoid Parentheses Around a Single Parameter

Lambda syntax requires parentheses only around more than one parameter or when there is no parameter at all. That is why it is safe to make your code a little bit shorter and to exclude parentheses when there is only one parameter.

So, do this:

```
a -> a.toLowerCase();
```

instead of this:

```
(a) -> a.toLowerCase();
```

8.4. Avoid Return Statement and Braces

Braces and ***return*** statements are optional in one-line lambda bodies. This means, that they can be omitted for clarity and conciseness.

Do this:

```
a -> a.toLowerCase();
```

instead of this:

```
a -> {return a.toLowerCase()};
```

8.5. Use Method References

Very often, even in our previous examples, lambda expressions just call methods which are already implemented elsewhere. In this situation, it is very useful to use another Java 8 feature: **method references (https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html)**.

So, the lambda expression:

```
a -> a.toLowerCase();
```

could be substituted by:

```
String::toLowerCase;
```

This is not always shorter, but it makes the code more readable.

9. Use "Effectively Final" Variables

Accessing a non-final variable inside lambda expressions will cause the compile-time error. **But it doesn't mean that you should mark every target variable as *final.***

According to the "**effectively final (https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html)**" concept, a compiler treats every variable as *final,* as long as it is assigned only once.

It is safe to use such variables inside lambdas because the compiler will control their state and trigger a compile-time error immediately after any attempt to change them.

For example, the following code will not compile:

```
public void method() {
    String localVariable = "Local";
    Foo foo = parameter -> {
        String localVariable = parameter;
        return localVariable;
    };
}
```

The compiler will inform you that:

```
Variable 'localVariable' is already defined in the scope.
```

This approach should simplify the process of making lambda execution thread-safe.

10. Protect Object Variables from Mutation

One of the main purposes of lambdas is use in parallel computing – which means that they're really helpful when it comes to thread-safety.

The "effectively final" paradigm helps a lot here, but not in every case. Lambdas can't change a value of an object from enclosing scope. But in the case of mutable object variables, a state could be changed inside lambda expressions.

Consider the following code:

```
int[] total = new int[1];
Runnable r = () -> total[0]++;
r.run();
```

This code is legal, as *total* variable remains "effectively final". But will the object it references to have the same state after execution of the lambda? No!

Keep this example as a reminder to avoid code that can cause unexpected mutations.

11. Conclusion

In this tutorial, we saw some best practices and pitfalls in Java 8's lambda expressions and functional interfaces. Despite the utility and power of these new features, they are just tools. Every developer should pay attention while using them.

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 it can be imported and used as-is.

post-footer-main-1.2.0.jpg)

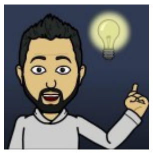
(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-icn-1.0.0.png)

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

Enter your Email Address

>> Get the eBook

Subscribe ▾ ▴ newest ▴ oldest ▴ most voted

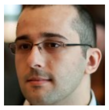


Guest

Nicola Malizia (https://unnikked.ga/)

Very helpful blog post! I read it with pleasure. I will definitively keep it in mind, thank you!

+ 2 − 2 years ago


Guest

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

Glad you enjoyed it Nicola. Cheers,
Eugen.

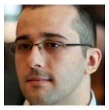
+ 1 − 2 years ago

Guest

Ola Kunysz

Good read, but it would have much more value if you use real life examples. If you replace Foo foo with some real code, it will be way easier to recall these practices in the right moment 🙂

+ 0 − 2 years ago


Guest

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

That's a good point, I'll add it to the content calendar of the site.
Cheers,
Eugen.

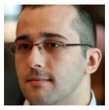
+ 0 − 2 years ago

Guest

Lukas Eder (http://www.jooq.org)

Regarding #10, this isn't true *in general*, but it is indeed a bad idea to implement lambdas with side-effects (stateful lambdas) in the context of some stream operations, e.g. `filter()`.

+ 0 − 2 years ago


Guest

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

Hey Lukas, I was actually thinking that I should run this article by you while I was reviewing it 🙂
Yeah, definitely agree – side effects are almost never a good thing as idempotence is such a usefull property. The more functional approach does help there, but since there's really no language level mechanism to ensure that, you need to be quite careful with that aspect.
Thanks for the feedback and keep in touch. Cheers,
Eugen.

+ 0 − 2 years ago

Alex Ve

Guest

In my opinion, #10 is more about showing a real denger of unexpected side-effects, than to offer expluatation of such side-effects. Maybe, it would be better to call #10 – "Avoid unexpected mutations of object variables while using lambdas" or something like that.

0

2 years ago

Brian Oxley

Guest

In your example:

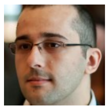
Foo foo = parameter -> buildString(parameter);

You may prefer an example where "buildString" takes an additional argument. Else I would prefer:

Foo foo = this::buildString;

0

2 years ago



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

Guest

The double colon operator is certainly a very useful addition to the language – I may explore it in a dedicated article. Cheers,
Eugen.

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)
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)