**1. Overview**

we'll take a closer look at functional interfaces and lambda expressions.

**2. Prefer Standard Functional Interfaces**

Functional interfaces, which are gathered in the **java.util.function** 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.

Let's consider an interface*Foo*:

@FunctionalInterface

**public** **interface** **Foo** {

String **method**(String string);

}

In addition, we have 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, we would write:

**Foo** foo = parameter -> parameter + " from lambda";

**String** result = useFoo.add("Message ", foo);

If we look closer, we'll 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>* from the java.util.function 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**

Now let's annotate our functional interfaces with *@FunctionalInterface.* At first, this annotation seems to be useless. Even without it, our interface will be treated as functional as long as it has just one abstract method.

However, let's 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 another abstract method/methods, rendering it unusable as a functional interface.

By 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 our application architecture easier to understand for other developers.

So we can 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**

We 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**(String string);

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

}

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

@FunctionalInterface

**public** **interface** **FooExtended** **extends** **Baz**, Bar {}

@FunctionalInterface

**public** **interface** **Baz** {

String **method**(String string);

**default** String **defaultBaz**() {}

}

@FunctionalInterface

**public** **interface** **Bar** {

String **method**(String string);

**default** String **defaultBar**() {}

}

Just as with regular interfaces, **extending different functional interfaces with the same default method can be problematic**.

For example, let's add the *defaultCommon()* method to the *Bar* and *Baz* interfaces:

@FunctionalInterface

**public** **interface** **Baz** {

String **method**(String string);

**default** String **defaultBaz**() {}

**default** String **defaultCommon**(){}

}

@FunctionalInterface

**public** **interface** **Bar** {

String **method**(String string);

**default** String **defaultBar**() {}

**default** String **defaultCommon**() {}

}

In this case, we'll get a compile-time error:

interface FooExtended inherits unrelated defaults **for** defaultCommon() from types Baz and Bar...

To fix this, the *defaultCommon()* method should be overridden in the *FooExtended* interface. We can provide a custom implementation of this method; however, **we can also reuse the implementation from the parent interface**:

@FunctionalInterface

**public** **interface** **FooExtended** **extends** **Baz**, Bar {

@Override

**default** String **defaultCommon**() {

**return** Bar.super.defaultCommon();

}

}

It's important to note that we have to be careful. **Adding too many default methods to the interface is not a very good architectural decision.** This should be considered 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 us to use an inner class to instantiate a functional interface; however, this can lead to very verbose code. We should prefer to use 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 we should review our whole older code base and change everything.**

**6. Avoid Overloading Methods With Functional Interfaces as Parameters**

We should use methods with different names to avoid collisions:

**public** **interface** **Processor** {

String **process**(Callable<String> c) **throws** Exception;

String **process**(Supplier<String> s);

}

**public** **class** **ProcessorImpl** **implements** **Processor** {

@Override

**public** String **process**(Callable<String> c) **throws** Exception {

// implementation details

}

@Override

**public** String **process**(Supplier<String> s) {

// implementation details

}

}

At first glance, this seems reasonable, but any attempt to execute either of the *ProcessorImpl*‘s methods:

**String** result = processor.process(() -> "abc");

Ends with an error with the following message:

reference to process is ambiguous

both method **process**(java.util.concurrent.Callable<java.lang.String>)

in com.baeldung.java8.lambda.tips.ProcessorImpl

and method **process**(java.util.function.Supplier<java.lang.String>)

in com.baeldung.java8.lambda.tips.ProcessorImpl match

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

String **processWithCallable**(Callable<String> c) **throws** Exception;

String **processWithSupplier**(Supplier<String> s);

**The second option is to perform casting manually,** which is not preferred:

**String** result = processor.process((Supplier<String>) () -> "abc");

**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 we use an inner class, it creates a new scope. We can hide local variables from the enclosing scope by instantiating new local variables with the same names. We can also use the keyword ***this*** inside our inner class as a reference to its instance.

Lambda expressions, however, work with enclosing scope. We can’t hide 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,*we 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 we execute the *scopeExperiment()* method, we'll get the following result: *Results: resultIC = Inner class value, resultLambda = Enclosing scope value*

As we can see, by calling *this.value* in IC, we can access a local variable from its instance. In the case of the lambda, *this.value* call gives us 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, we should 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 specifically 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 we 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;

};

**It is important to note, we shouldn't use this “one-line lambda” rule as dogma**. If we 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**. Consequently, adding a type to the parameters is optional and can be omitted.

We can 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 only requires parentheses around more than one parameter, or when there is no parameter at all. That's why it's safe to make our code a little bit shorter, and to exclude parentheses when there is only one parameter.

So we can 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.

We can 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**.

The lambda expression would be:

a -> a.toLowerCase();

We could substitute it with:

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 a compile-time error, **but that doesn’t mean that we should mark every target variable as *final.***

According to the “**effectively final**” concept, a compiler treats every variable as *final* as long as it is assigned only once.

It's 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 us 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 have the same state after execution of the lambda? No!

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