(http://baeldung.com)

Functional Interfaces in Java 8

Last modified: May 5, 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/)

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

1. Introduction

This article is a guide to different functional interfaces present in Java 8, their general use cases and usage in the standard JDK library.

2. Lambdas in Java 8

Java 8 brought a powerful new syntactic improvement in the form of lambda expressions. A lambda is an anonymous function that can be handled as a first-class language citizen, for instance passed to or returned from a method.

Before Java 8, you would usually create a class for every case where you needed to encapsulate a single piece of functionality. This implied a lot of unnecessary boilerplate code to define something that served as a primitive function representation.

Lambdas, functional interfaces and best practices of working with them, in general, are described in the article "Lambda Expressions and Functional Interfaces: Tips and Best Practices" (/java-8-lambda-expressions-tips). This guide focuses on some particular functional interfaces that are present in the *java.util.function* package.

3. Functional Interfaces

All functional interfaces are recommended to have an informative *@FunctionalInterface* annotation. This not only clearly communicates the purpose of this interface, but also allows a compiler to generate an error if the annotated interface does not satisfy the conditions.

Any interface with a SAM(Single Abstract Method) is a functional interface, and its implementation may be treated as lambda expressions.

Note that Java 8's *default* methods are not *abstract* and do not count: a functional interface may still have multiple *default* methods. You can observe this by looking at the *Function's* documentation (https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html).

4. Functions

The most simple and general case of a lambda is a functional interface with a method that receives one value and returns another. This function of a single argument is represented by the *Function* interface which is parameterized by the types of its argument and a return value:

```
public interface Function<T, R> { … }
```

One of the usages of the *Function* type in the standard library is the *Map.computeIfAbsent* method that returns a value from a map by key but calculates a value if a key is not already present in a map. To calculate a value, it uses the passed Function implementation:

```
Map<String, Integer> nameMap = new HashMap<>();
Integer value = nameMap.computeIfAbsent("John", s -> s.length());
```

A value, in this case, will be calculated by applying a function to a key, put inside a map and also returned from a method call. By the way, **we may replace the lambda with a method reference that matches passed and returned value types**.

Remember that an object on which the method is invoked is, in fact, the implicit first argument of a method, which allows casting an instance method *length* reference to a *Function* interface:

```
Integer value = nameMap.computeIfAbsent("John", String::length);
```

The *Function* interface has also a default *compose* method that allows to combine several functions into one and execute them sequentially:

```
Function<Integer, String> intToString = Object::toString;
Function<String, String> quote = s -> "'" + s + "'";

Function<Integer, String> quoteIntToString = quote.compose(intToString);

assertEquals("'5'", quoteIntToString.apply(5));
```

The *quoteIntToString* function is a combination of the *quote* function applied to a result of the *intToString* function.

5. Primitive Function Specializations

Since a primitive type can't be a generic type argument, there are versions of the *Function* interface for most used primitive types *double*, *int*, *long*, and their combinations in argument and return types:

- *IntFunction*, *LongFunction*, *DoubleFunction:* arguments are of specified type, return type is parameterized
- *ToIntFunction*, *ToLongFunction*, *ToDoubleFunction:* return type is of specified type, arguments are parameterized
- *DoubleToIntFunction*, *DoubleToLongFunction*, *IntToDoubleFunction*, *IntToLongFunction*, *LongToIntFunction*, *LongToDoubleFunction* — having both argument and return type defined as primitive types, as specified by their names

There is no out-of-the-box functional interface for, say, a function that takes a *short* and returns a *byte*, but nothing stops you from writing your own:

```
@FunctionalInterface
public interface ShortToByteFunction {

    byte applyAsByte(short s);

}
```

Now we can write a method that transforms an array of *short* to an array of *byte* using a rule defined by a *ShortToByteFunction*:

```
public byte[] transformArray(short[] array, ShortToByteFunction function
    byte[] transformedArray = new byte[array.length];
    for (int i = 0; i < array.length; i++) {
        transformedArray[i] = function.applyAsByte(array[i]);
    }
    return transformedArray;
}
```

Here's how we could use it to transform an array of shorts to array of bytes multiplied by 2:

```
short[] array = {(short) 1, (short) 2, (short) 3};
byte[] transformedArray = transformArray(array, s -> (byte) (s * 2));

byte[] expectedArray = {(byte) 2, (byte) 4, (byte) 6};
assertArrayEquals(expectedArray, transformedArray);
```

6. Two-Arity Function Specializations

To define lambdas with two arguments, we have to use additional interfaces that contain "*Bi*" keyword in their names: *BiFunction*, *ToDoubleBiFunction*, *ToIntBiFunction*, and *ToLongBiFunction*.

BiFunction has both arguments and a return type generified, while *ToDoubleBiFunction* and others allow you to return a primitive value.

One of the typical examples of using this interface in the standard API is in the *Map.replaceAll* method, which allows replacing all values in a map with some computed value.

Let's use a *BiFunction* implementation that receives a key and an old value to calculate a new value for the salary and return it.

```
Map<String, Integer> salaries = new HashMap<>();
salaries.put("John", 40000);
salaries.put("Freddy", 30000);
salaries.put("Samuel", 50000);

salaries.replaceAll((name, oldValue) ->
  name.equals("Freddy") ? oldValue : oldValue + 10000);
```

7. Suppliers

The *Supplier* functional interface is yet another *Function* specialization that does not take any arguments. It is typically used for lazy generation of values. For instance, let's define a function that squares a *double* value. It will receive not a value itself, but a *Supplier* of this value:

```
public double squareLazy(Supplier<Double> lazyValue) {
    return Math.pow(lazyValue.get(), 2);
}
```

This allows us to lazily generate the argument for invocation of this function using a *Supplier* implementation. This can be useful if the generation of this argument takes a considerable amount of time. We'll simulate that using Guava's *sleepUninterruptibly* method:

```
Supplier<Double> lazyValue = () -> {
    Uninterruptibles.sleepUninterruptibly(1000, TimeUnit.MILLISECONDS);
    return 9d;
};

Double valueSquared = squareLazy(lazyValue);
```

Another use case for the Supplier is defining a logic for sequence generation. To demonstrate it, let's use a static *Stream.generate* method to create a *Stream* of Fibonacci numbers:

```
int[] fibs = {0, 1};
Stream<Integer> fibonacci = Stream.generate(() -> {
    int result = fibs[1];
    int fib3 = fibs[0] + fibs[1];
    fibs[0] = fibs[1];
    fibs[1] = fib3;
    return result;
});
```

The function that is passed to the *Stream.generate* method implements the *Supplier* functional interface. Notice that to be useful as a generator, the *Supplier* usually needs some sort of external state. In this case, its state is comprised of two last Fibonacci sequence numbers.

To implement this state, we use an array instead of a couple of variables, because **all external variables used inside the lambda have to be effectively final**.

Other specializations of *Supplier* functional interface include *BooleanSupplier*, *DoubleSupplier*, *LongSupplier* and *IntSupplier*, whose return types are corresponding primitives.

8. Consumers

As opposed to the *Supplier*, the *Consumer* accepts a generified argument and returns nothing. It is a function that is representing side effects.

For instance, let's greet everybody in a list of names by printing the greeting in the console. The lambda passed to the *List.forEach* method implements the *Consumer* functional interface:

```
List<String> names = Arrays.asList("John", "Freddy", "Samuel");
names.forEach(name -> System.out.println("Hello, " + name));
```

There are also specialized versions of the *Consumer* — *DoubleConsumer*, *IntConsumer* and *LongConsumer* — that receive primitive values as arguments. More interesting is the *BiConsumer* interface. One of its use cases is iterating through the entries of a map:

```
Map<String, Integer> ages = new HashMap<>();
ages.put("John", 25);
ages.put("Freddy", 24);
ages.put("Samuel", 30);

ages.forEach((name, age) -> System.out.println(name + " is " + age + " y
```

Another set of specialized *BiConsumer* versions is comprised of *ObjDoubleConsumer*, *ObjIntConsumer*, and *ObjLongConsumer* which receive two arguments one of which is generified, and another is a primitive type.

9. Predicates

In mathematical logic, a predicate is a function that receives a value and returns a boolean value.

The *Predicate* functional interface is a specialization of a *Function* that receives a generified value and returns a boolean. A typical use case of the *Predicate* lambda is to filter a collection of values:

```
List<String> names = Arrays.asList("Angela", "Aaron", "Bob", "Claire", "
List<String> namesWithA = names.stream()
  .filter(name -> name.startsWith("A"))
  .collect(Collectors.toList());
```

In the code above we filter a list using the *Stream* API and keep only names that start with the letter “A”. The filtering logic is encapsulated in the *Predicate* implementation.

As in all previous examples, there are *IntPredicate*, *DoublePredicate* and *LongPredicate* versions of this function that receive primitive values.

10. Operators

Operator interfaces are special cases of a function that receive and return the same value type. The *UnaryOperator* interface receives a single argument. One of its use cases in the Collections API is to replace all values in a list with some computed values of the same type:

```
List<String> names = Arrays.asList("bob", "josh", "megan");

names.replaceAll(name -> name.toUpperCase());
```

The *List.replaceAll* function returns *void*, as it replaces the values in place. To fit the purpose, the lambda used to transform the values of a list has to return the same result type as it receives. This is why the *UnaryOperator* is useful here.

Of course, instead of *name -> name.toUpperCase()*, you can simply use a method reference:

```
names.replaceAll(String::toUpperCase);
```

One of the most interesting use cases of a *BinaryOperator* is a reduction operation. Suppose we want to aggregate a collection of integers in a sum of all values. With *Stream* API, we could do this using a collector, but a more generic way to do it is, would be to use the *reduce* method:

```
List<Integer> values = Arrays.asList(3, 5, 8, 9, 12);

int sum = values.stream()
  .reduce(0, (i1, i2) -> i1 + i2);
```

The *reduce* method receives an initial accumulator value and a *BinaryOperator* function. The arguments of this function are a pair of values of the same type, and a function itself contains a logic for joining them in a single value of the same type. **Passed function must be associative**, which means that the order of value aggregation does not matter, i.e. the following condition should hold:

```
op.apply(a, op.apply(b, c)) == op.apply(op.apply(a, b), c)
```

The associative property of a *BinaryOperator* operator function allows to easily parallelize the reduction process.

Of course, there are also specializations of *UnaryOperator* and *BinaryOperator* that can be used with primitive values, namely *DoubleUnaryOperator*, *IntUnaryOperator*, *LongUnaryOperator*, *DoubleBinaryOperator*, *IntBinaryOperator*, and *LongBinaryOperator*.

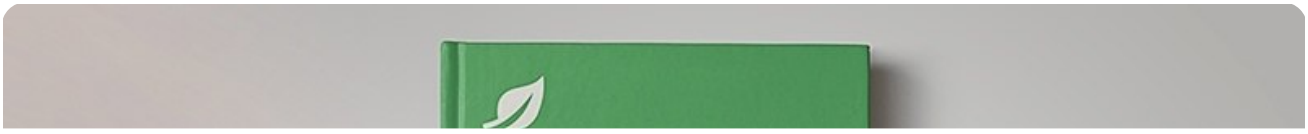
11. Legacy Functional Interfaces

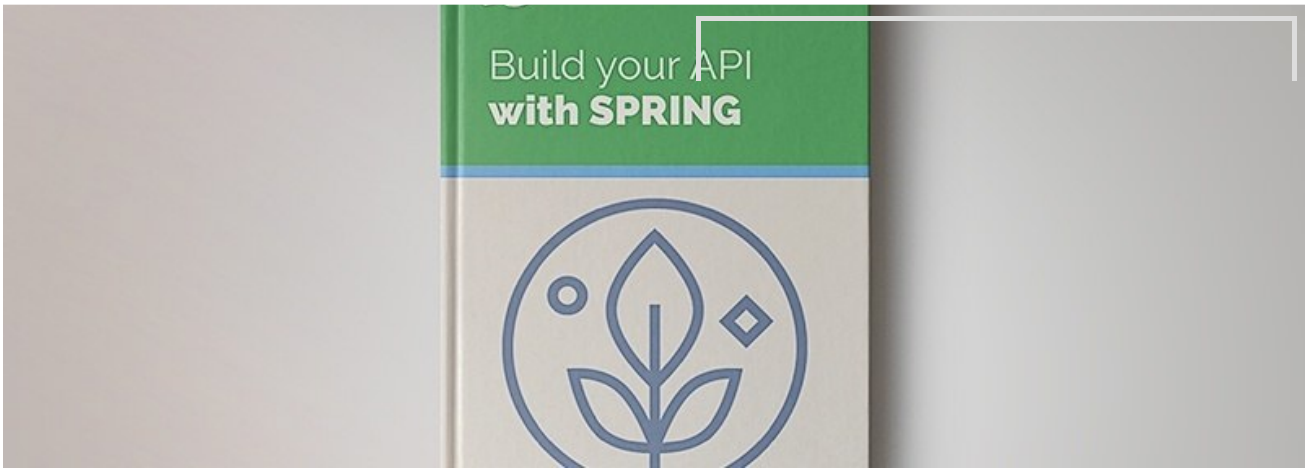
Not all functional interfaces appeared in Java 8. Many interfaces from previous versions of Java conform to the constraints of a *FunctionalInterface* and can be used as lambdas. A prominent example is the *Runnable* and *Callable* interfaces that are used in concurrency APIs. In Java 8 these interfaces are also marked with a *@FunctionalInterface* annotation. This allows us to greatly simplify concurrency code:

```
1  Thread thread = new Thread(() -> System.out.println("Hello From Another
2  thread.start();
```

12. Conclusion

In this article, we've described different functional interfaces present in the Java 8 API that can be used as lambda expressions. The source code for the article is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-8).





(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-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)

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/EAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)