Combining Lambda Expressions

You may have noticed the presence of default methods in the functional interfaces of the `java.util.function` package. These methods have been added to allow the combination and chaining of lambda expressions.

Why would you do such a thing? Simply to help you write simpler, more readable code.

Chaining Predicates with Default Methods

Suppose you need to process a list of strings, to keep only the strings that are non-null, non-empty, and shorter than 5 characters. The way this problem is stated is the following. You have three tests on a given string:

- non-null;
- non-empty;
- shorter than 5 characters.

Each of these tests can easily be written with a very simple, one line predicate. It is also possible to combine those three tests into one single predicate. It will look like this code:

```
1 Predicate<String> p = s -> (s != null) && !s.isEmpty() && s.length() < 5;
```

But the JDK allows you to write this piece of code in that way:

```
1 Predicate<String> nonNull = s -> s != null;
2 Predicate<String> nonEmpty = s -> !s.isEmpty();
3
```

```
Predicate<String> shorterThan5 = s -> s.length() < 5;

Predicate<String> p = nonNull.and(nonEmpty).and(shorterThan5);
```

Hiding the technical complexity and exposing the intent of the code is what combining lambda expressions is about.

How is this code implemented at the API level? Without diving too deep in the details, what you can see is the following:

- `and()` is a method
- It is called on an instance of `Predicate<T>`: it is thus an instance method
- It takes another `Predicate<T>` as an argument
- It returns a `Predicate<T>`

Since only one abstract method is allowed on a functional interface, this `and()` method has to be a default method. So from the API design point of view, you have all the elements you need to create this method. The good news is: the `Predicate<T>` interface has a `and()` default method, so you do not have to do it yourself.

By the way, there is also a `or()` method that takes another predicate as an argument, and also a `negate()` method that does not take anything.

Using these, you can write the previous example in this way:

```
Predicate<String> isNull = Objects::isNull;
Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNullOrEmpty = isNull.or(isEmpty);
Predicate<String> isNotNullNorEmpty = isNullOrEmpty.negate();
Predicate<String> shorterThan5 = s -> s.length() < 5;

Predicate<String> p = isNotNullNorEmpty.and(shorterThan5);
```

Even if this example may be pushing the limits a little too far, you can dramatically improve the expressiveness of your code by leveraging method references and default methods.

Creating Predicates with Factory Methods

Expressiveness can be pushed one step further by using factory methods defined in functional interfaces. There are two of them on the `Predicate<T>` interface.

In the following example, the predicate `isEqualToDuke` tests a string of characters. The test is true when the tested string is equal to "Duke". This factory method can create predicates for any type of objects.

```
Predicate<String> isEqualToDuke = Predicate.isEqual("Duke");
```

The second factory method negates the predicate given as an argument.

```
Predicate<Collection<String>> isEmpty = Collection::isEmpty;
Predicate<Collection<String>> isNotEmpty = Predicate.not(isEmpty);
```

Chaining Consumers with Default Methods

The `Consumer<T>` interface also has a method to chain consumers. You can chain consumers with the following pattern:

```
Logger logger = Logger.getLogger("MyApplicationLogger");
Consumer<String> log = message -> logger.info(message);
Consumer<String> print = message -> System.out.println(message);

Consumer<String> longAndPrint = log.andThen(print);
```

In this example, `printAndLog` is a consumer that will first pass the message to the `log` consumer and then pass it to the `print` consumer.

Chaining and Composing Functions with Default Methods

The difference between chaining and composing is a little subtle. The result of both operations is in fact the same. What is different is the way you write it.

Suppose you have two functions `f1` and `f2`. You can chain them by calling [`f1.andThen(f2)`. Applying the resulting function to an object will first pass this object to `f1` and the result to `f2`.

The `Function` interface has a second default method: `f2.compose(f1)`. Written in this way, the resulting function will first process an object by passing it to the `f1` function and then the result is passed to `f2`.

What you need to realize is that to get the same resulting function, you need to call `andThen()` on `f1` or `compose()` on `f2`.

You can chain or compose functions of different types. There are obvious restrictions though: the type of the result produced by `f1` should be compatible with the type consumed by `f2`.

Creating an Identity Function

The `Function<T, R>` interface also has a factory method to create the identity function, called `identity()`.

Creating the identity function can thus be done using the following simple pattern:

```
1 | Function<String, String> id = Function.identity();
```

This pattern is applicable for any valid type.

Last update: *October 26, 2021*

Writing Lambda Expressions as Method References ← **Combining Lambda Expressions** → Writing and Combining Comparators

Home > Tutorials > Lambda Expressions > Combining Lambda Expressions