

Theory: Function composition

⌚ 36 minutes 0 / 0 problems solved

Skip this topic

Start practicing

In this topic, you will learn a new technique for working with functions called **function composition**. It is a mechanism for combining functions to obtain more complicated functions that originally comes from math. In a sense, it can be considered as a **design pattern** in functional programming. You can use this pattern to compose standard functions, operators, predicates and consumers (but not suppliers). Let's take a look at examples.

§1. Composing functions

The functional interface `Function<T, R>` has two default methods `compose` and `andThen` for composing new functions. The main difference between these methods lies in the execution order.

Generally, `f.compose(g).apply(x)` is the same as `f(g(x))`, and `f.andThen(g).apply(x)` is the same as `g(f(x))`.

Here is an example with two functions: `adder` and `multiplier`.

```
1  Function<Integer, Integer> adder = x -> x + 10;
2  Function<Integer, Integer> multiplier = x -> x * 5;
3
4  // compose: adder(multiplier(5))
5
System.out.println("result: " + adder.compose(multiplier).apply(5));
6
7  // andThen: multiplier(adder(5))
8
System.out.println("result: " + adder.andThen(multiplier).apply(5));
```

In this case, the `compose` method returns a composed function that first applies `multiplier` to its input, and then applies `adder` to the result. The `andThen` method returns a composed function that first applies `adder` to its input, and then applies `multiplier` to the result.

Here is the output:

```
1  result: 35
2  result: 75
```

Operators can be used in the same way as functions.

The methods `compose` and `andThen` do not modify the functions that are combined. Instead, they return new functions. This is true for all the next examples.

§2. Composing predicates

All functional interfaces representing predicates (`Predicate<T>`, `IntPredicate` and others) have three methods for composing new predicates: `and`, `or` and `negate`.

There are two predicates in the example below: `isOdd` and `lessThan11`.

2 required topics

✗ [The concept of patterns](#) ▾

✗ [Standard functional interfaces](#) ▾

1 dependent topic

[Stream pipelines](#) ▾

Table of contents:

[↑ Function composition](#)

[§1. Composing functions](#)

[§2. Composing predicates](#)

[§3. Composing consumers](#)

[§4. Conclusion](#)

[Discussion](#)

```

1      IntPredicate isOdd = n -
> n % 2 != 0; // it's true for 1, 3, 5, 7, 9, 11 and so on
2
3      System.out.println(isOdd.test(10)); // prints "false"
4      System.out.println(isOdd.test(11)); // prints "true"
5
6      IntPredicate lessThan11 = n -
> n < 11; // it's true for all numbers < 11
7
8      System.out.println(lessThan11.test(10)); // prints "true"
9      System.out.println(lessThan11.test(11)); // prints "false"

```

Let's negate the first predicate:

```

1      IntPredicate isEven = isOdd.negate(); // it's true for 0, 2, 4, 6, 8,
10 and so on
2      System.out.println(isEven.test(10)); // prints "true"
3      System.out.println(isEven.test(11)); // prints "false"

```

Here we have a new predicate that tests whether the value is even rather than odd.

Now let's combine both `isOdd` and `lessThan11` predicates together by using `or` and `and` methods:

```

1      IntPredicate isOddOrLessThan11 = isOdd.or(lessThan11);
2
3      System.out.println(isOddOrLessThan11.test(10)); // prints "true"
4      System.out.println(isOddOrLessThan11.test(11)); // prints "true"
5      System.out.println(isOddOrLessThan11.test(12)); // prints "false"
6      System.out.println(isOddOrLessThan11.test(13)); // prints "true"
7
8      IntPredicate isOddAndLessThan11 = isOdd.and(lessThan11);
9
1     System.out.println(isOddAndLessThan11.test(8)); // prints "false"
1     System.out.println(isOddAndLessThan11.test(9)); // prints "true"
1     System.out.println(isOddAndLessThan11.test(10)); // prints "false"
1     System.out.println(isOddAndLessThan11.test(11)); // prints "false"
3

```

As you can see, these methods are equivalent to logical operators `&&` and `||`, but they work with functions rather than their values.

§3. Composing consumers

It can be a little surprising, but it is also possible to combine consumers by using the `andThen` method. It just returns a new consumer that consumes the given value several times in a chain.

In the following example, we use `andThen` to print a value two times, but it is possible to do it more times.

```

1      Consumer<String> consumer = System.out::println;
2
Consumer<String> doubleConsumer = consumer.andThen(System.out::println);
3      doubleConsumer.accept("Hi!");

```

Here is the output:

1	Hi!
2	Hi!

In a real situation, you could use it to output a value to different destinations, like a database or a logger.

Although it is possible to combine **consumers**, it is not possible to combine **suppliers**.

§4. Conclusion

Function composition allows developers to build new functions from existing ones and use them whenever you want. All kinds of predicates, functions, operators, and consumers (except for suppliers) have methods for that purpose.

 Report a typo

131 users liked this piece of theory. 2 didn't like it. What about you?




Start practicing

[Comments \(2\).](#) [Hints \(0\).](#) [Useful links \(0\).](#) [Hide discussion](#)

G

Share something, GulbalaDeveloper

Post ☐ Place a bounty 

Sort by:

Most popular ▼

 **SimonA** [12 months ago](#)

This is really cool.

 5  [Reply](#) [Report](#)

 **just-for-fun** [10 months ago](#)

``consumer1.andThen(consumer2)`` works like T pipe!

 [Reply](#) [Report](#)