Theory: Currying

36 minutes 0 / 5 problems solved

Since functions can be considered as objects, they can be returned as results from other functions. It allows us to use a special style of programming where we convey arguments to a function one by one and obtain functions as intermediate results. This is a fairly advanced technique, so you won't have to use it every time you can, but it is useful to be aware of such an option.

§1. Returning functions

When we consider a function as an object we can accept it as an argument and return it as a value.

Here's an example:

```
public static IntBinaryOperator sumF(IntUnaryOperator f) {
    return (a, b) -> f.applyAsInt(a) + f.applyAsInt(b);
}
```

The `sumF` method accepts an operator `f` with an integer argument and returns another operator with two integer arguments. In the method body, an anonymous function that takes two arguments is constructed and returned. This function applies `f` to each of its arguments and summarizes the results.

What can we do now? Let's just return the function from the method, save it to the `sumF` variable and apply some values to it:

```
// build a new sumOfSquares operator
IntBinaryOperator sumOfSquares = sumF(x -> x * x);

// the sum is equal to 125 (5 * 5 + 10 * 10)
long sum = sumOfSquares.applyAsInt(5, 10);
```

Here are some more examples:

```
// sum of two identities: 0 + 10 = 10
long sumOfIdentities = sumF(x -> x).applyAsInt(0, 10);

// sum with coefficients: 10 * 2 + 11 * 2 = 42
long sumWithCoefficient = sumF(x -> x * 2).applyAsInt(10, 11);

// sum of two cubes: 3 * 3 * 3 + 8 * 8 * 8 = 539
long sumOfCubes = sumF(x -> x * x * x).applyAsInt(3, 8);
```

As you can see, the possibility of returning functions provides an easy way to build complex and generalized functions.

§2. Currying functions

Currying is a technique for translating the evaluation of a function that takes multiple parameters into evaluating a sequence of functions, each with a single argument. The technique is called after mathematician Haskell Curry who originally developed it. Let's see how to use it in Java.

First, let's compare a regular function and a curried function:

```
IntBinaryOperator notCurriedFun = (x, y) -
> x + y; // not a curried function

IntFunction<IntUnaryOperator> curriedFun = x -> y -
> x + y; // a curried function
```

Current topic:

Currying

Topic depends on:

× Standard functional interfaces

Table of contents:

We can define a curried function with three arguments and then apply arguments one by one:

```
// curried function
IntFunction<IntFunction<IntFunction<Integer>>> fff = x -> y -> z -
> x * y + z;

// fff returns a curried function y -> z -> 2 * y + z
IntFunction<IntFunction<Integer>> ff = fff.apply(2);

// ff returns a curried function z -> 2 * 3 + z
IntFunction<Integer> f = ff.apply(3);


// f returns 7

int result = f.apply(1);
```

A shorter example:

```
// here the result is equal to 153
int anotherResult = fff.apply(10).apply(15).apply(3);
```

Let's rewrite the `sumF` method from the earlier example. Instead of returning a function from it, we can write a curried function and then use it in the same way:

```
Function<IntUnaryOperator, IntBinaryOperator> sumF =
        (f) -> (a, b) -> f.applyAsInt(a) + f.applyAsInt(b);

// build a new sumOfSquares operator in terms of sumF
IntBinaryOperator sumOfSquares = sumF.apply(x -> x * x);

// the sum is equal to 125 again
long sum = sumOfSquares.applyAsInt(5, 10);
```

As you see, **returning functions** and **currying** are very close concepts and both are based on **closures**.

§3. An example of currying

Suppose we would like to say `"Hi"` to our friends and `"Hello"` to our business partners. We can create a function that has two arguments: `what` and `who`. The function will apply `what` depending on the context.

```
Function<String, Consumer<String>> say = what -> who -
> System.out.println(what + ", " + who);
```

The friends' context:

```
List<String> friends = Arrays.asList("John", "Neal", "Natasha");
Consumer<String> sayHi = say.apply("Hi");

// many lines of code...

friends.forEach(sayHi);
```

The partner's context:

```

List<String> partners = Arrays.asList("Randolph Singleton", "Jessie James");
Consumer<String> sayHello = say.apply("Hello");

// many lines of code...

partners.forEach(sayHello);
```

The result:

```
Hi, John
Hi, Neal
Hi, Natasha
Hello, Randolph Singleton
Hello, Jessie James
```

In the real situation, we can get the list of persons from a database and pass the consumer as an argument from another part of our program.

§4. Conclusion

We learned how to return functions from other functions and what currying is. You can use this knowledge to create general functions and detail them depending on your needs.

Report a typo

49 users liked this theory. **2** didn't like it. **What about you?**

Start practicing

Comments (3) Hints (0) Useful links (0) Show discussion