Theory: Recursion

51 minutes 0 / 5 problems solved

Skip this topic Start practicing

1941 users solved this topic. Latest completion was about 1 hour ago.

§1. Recursion basics

As you know, a method can call another method. What is even more interesting, a method can call itself. This possibility is known as **recursion** and the method calling itself is named **recursive method**.

As a regular method, any **recursive method** can contain parameters and return something as well as it can take or return nothing.

But how many times should a method call itself? It should be limited. The method must have a special condition to stop the recursion, otherwise, the call stack will overflow and the execution will stop with an error.

To write **recursive methods** you should consider the solution of a problem as a smaller version of the same problem.

§2. The factorial example

The classic example of the recursion is a math function calculating **the factorial**.

> If you have forgotten or did not know, the **factorial** of a non-negative integer **n** is the product of all positive integers from **1** to **n** inclusively. E.g., the factorial of 4 is 1 * 2 * 3 * 4 = 24. The factorial of 0 equals 1.

Here is a recursive method which does the same using **the recursive call**:

```
public static long factorial(long n) {
    if (n == 0 || n == 1) {
        return 1; // the trivial case
    } else {
        return n * factorial(n - 1); // the recursive call
    }
}
```

This method has one long parameter and returns a long result. The implementation includes:

- the trivial case that returns the value 1 without any recursive calls;
- the reduction step with the recursive call to simplify the problem.

We suppose, the **passed argument >= 0**. If the passed value is 0 or 1, the result is 1, otherwise, we invoke the same method decreasing the argument by one.

Let's invoke the method passing different arguments:

```
long fact0 = factorial(0); // 1 (by definition)
long fact1 = factorial(1); // 1
long fact2 = factorial(2); // 2 (1 * 2)
long fact3 = factorial(3); // 6 (1 * 2 * 3)
long fact4 = factorial(4); // 24 (1 * 2 * 3 * 4)
```

As you can see, it returns the expected results.

But what happens if a recursive method never reaches a base case? The stack will never stop growing. If a program's stack exceeds the limit size, the `StackOverflowError` occurs. It will crash the execution.

§3. Replacing recursion by a loop

Every recursive method can be written iteratively using a loop.

Let's rewrite the factorial method in this way:

Current topic:

Recursion

Topic depends on:

- Call stack
- Recursion basics

Table of contents:

```
public static long factorial(long n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

You can be sure that the result will be the same.

§4. Types of recursions

There are several types of recursions.

1) **Direct recursion.** A method invokes itself like the considered factorial method.

2) **Indirect recursion.** A method invokes another method that invokes the original method.

3) **Tail-recursion.** A call is tail-recursive if nothing has to be done after the call returns. I.e. when the call returns, the result is immediately returned from the calling method.

In other words, **tail recursion** is when the recursive call is the last statement in the method.

The considered recursive method for calculating factorial is not tail-recursion because after the recursive call it multiplies the result by a value. But it can be written as a tail recursive function. The general idea is to use an additional argument to accumulate the factorial value. When **n** reaches **0**, the method should return the accumulated value.

```
public static long factorialTailRecursive(long n, long accum) {
    if (n == 0) {
        return accum;
    }
    return factorialTailRecursive(n - 1, n * accum);
}
```

And write a special wrapper to invoke it more convenient:

```
public static long factorial(long n) {
    return factorialTailRecursive(n, 1);
}
```

4) **Multiple recursion.** A method invokes itself recursively multiple times. The well-known example is calculating the **N-th** Fibonacci number using the recursion.

The recurrent formula:

```
Fib(n) = Fib(n - 1) + Fib(n - 2); Fib(0) = 0, Fib(1) = 1.
```

The Fibonacci sequence starts with: `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...`

```
public static int fib(int n) {
    if (n <= 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

This solution is very inefficient, it's just an example of **multiple recursion**. Try to start the method passing **45** as the argument. It takes too much time. If you replace the recursion with a loop it will work much faster. Another possible optimization is the technique named memoization.

Start practicing

Comments (2) Hints (0) Useful links (2) Show discussion