Day 5: basic recursion

1 Recursion

Recursion is the process of repeating items in a self-similar way. It is a concept that appears often in computer science and other branches of mathematics (e.g. number theory, fractals), but it has also attracted some interest from fields like the arts in recent times (look up "recursive images" on your favourite search engine!).

In programming, the most basic form of recursion happens when a method calls itself. This is something that can be done in any modern programming language, including Java, Java Decaf, and Groovy.

```
01    int countForeverUp(int previousCount) {
02        int newCount = previousCount + 1;
03        printLine(newCount);
04        countForeverUp(newCount); // the method calls itself
05    }
```

We are going to learn now the basic of recursive programming. This will help us understand a bit deeper how the stack works when a method is called, and it will also be useful when we start learning about dynamic data structures. We will go into the more advanced aspects of recursion during the second half of the course.

1.1 Classical examples: factorials and Fibonacci numbers

1.1.1 Factorials

The factorial of a natural number is represented with an exclamation mark (!) and defined[1] as:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

This definition makes it very easy to write a function that calculates the factorial of a number recursively:

```
01    int factorial(int n) {
02        if (n == 1) {
03          return 1;
04        } else {
```

[1] If you maths-inclined and would like to see how the factorial of a *real* (non-natural) number is defined, search for the Γ (gamma) function.

```
05            int result = n * factorial(n-1);
06            return result;
07        }
08    }
```

If the argument is 1, the function returns 1 (because 1! = 1). Otherwise, the function returns the argument multiplied by the factorial of the argument minus one; that function call will in turn return its argument (n-1) multiplied by the factorial of its argument minus one (n-2); and so on. If we want to calculate the factorial of 4, the resulting calculations look like this:

$$\begin{aligned}
&\text{factorial}(4) && \\
&4 \times \text{factorial}(3) && \text{(line 05)} \\
&4 \times (3 \times \text{factorial}(2)) && \text{(line 05)} \\
&4 \times (3 \times (2 \times \text{factorial}(1))) && \text{(line 05)} \\
&4 \times (3 \times (2 \times 1)) && \text{(line 03)} \\
&4 \times (3 \times 2) && \text{(line 06)} \\
&4 \times 6 && \text{(line 06)} \\
&24 && \text{(line 06)}
\end{aligned}$$

1.1.2 Fibonacci numbers

Another classical example of recursivity is the sequence of Fibonacci numbers[2], where every number is the addition of the preceding two. In other words, Fibonacci numbers are defined recursively as:

$$F(n) = \begin{cases} 1, & \text{if } n < 3 \\ F(n-1) + F(n-2), & \text{if } n \geq 3 \end{cases}$$

They have many applications in computer science and biology, among other fields. As they are defined recursively, a recursive method to calculate the n^{th} Fibonacci number is quite straightforward to write:

```
int fib(int n) {
    if ((n == 1) || (n == 2)) {
      return 1;
    } else {
        int result = fib(n-1) + fib(n-2); // method calls itself
        return result;
    }
}
```

[2]Fibonacci numbers were discovered by Italian mathematician Leonardo di Pisa (known as Fibonacci) in the early 13^{th} century as an ideal solution to the growth of the population of rabbits. The sequence of Fibonacci numbers starts as 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89... Fibonacci introduced arabic numerals to the West, so thanks to him we can write 2989 instead of MMCMLXXXVIII.

1.2 How does recursion work?

Recursion works as any other method call. There is no magic. When a statement in Java calls a method, the same process is always followed:

1. A new level is added to the stack.
2. The values given by the caller code are copied into the parameters in the new level.
3. The *hidden parameter* "this" is also copied into the stack, pointing to the object where the method is called. It is accessed with the reserved keyword `this` (e.g. `this.name = name`).
4. The code of the method is executed as usual. Any new local variable is created on the new level in the stack.
5. When a `return` statement is reached, the corresponding return value (if any) is stored.
6. The level in the stack is wiped out, and its local variables (and parameter values) forgotten. The return value (if any) is returned to the original caller code (and usually assigned to a variable).

This does not change in the case of a recursive call. If we take the calculation of 4! (factorial of 4) that we used before as an example, the computer stack changes are represented if Figure 1. First, a level is added to the stack and the parameters are copied (only `n` in this case[3], with value 4). The code of the method is executed line by line: as `n` is not equal to 1 (line 2), a new local variable `result` is created (line 5); its value is 1 plus the result of calling a method (line 5). A new level is added to the stack, and the parameter values are copied (only `n` again, but this time with value 3). A local variable `result` is created and then the method is called again (line 5) so we need to add a new level in the stack. The process is repeated once more, and we call the method with a value of 1 for `n`. At this point, the method returns 1 (line 3). Then the calling method can finally calculate the value of `result`, which is $2 \times 1 = 2$, and returns it (line 6). Then the calling method can calculate `result` itself, which in this case is $3 \times 2 = 6$, and return it (line 6). The calling method can then find the value of `result` (24) and return it (line 6).

As you can see, a method recursively calling itself is no different from a method calling another method. It is a bit convoluted to write it in English, but it is easy to follow the code if we are careful and remember that every method call *is independent* from the previous one even if they execute the *same* lines of code. For every method call, recursive or not, the calling code remains *on pause* while the new level in the stack is created and the code of the called method is executed (in the example, the value of `result` is not defined until a return value is received from the called method). When a result is returned, the calling code can continue its execution. When a method calls a method that calls a method, etc, several levels are added to the stack, and several methods are waiting for a return value. Recursive calls are exactly the same, the only difference being that the same code is executed over and over again *but on different levels of the stack*, which means that local variables are new and independent.

[3]The *hidden parameter "this" is not shown for clarity*

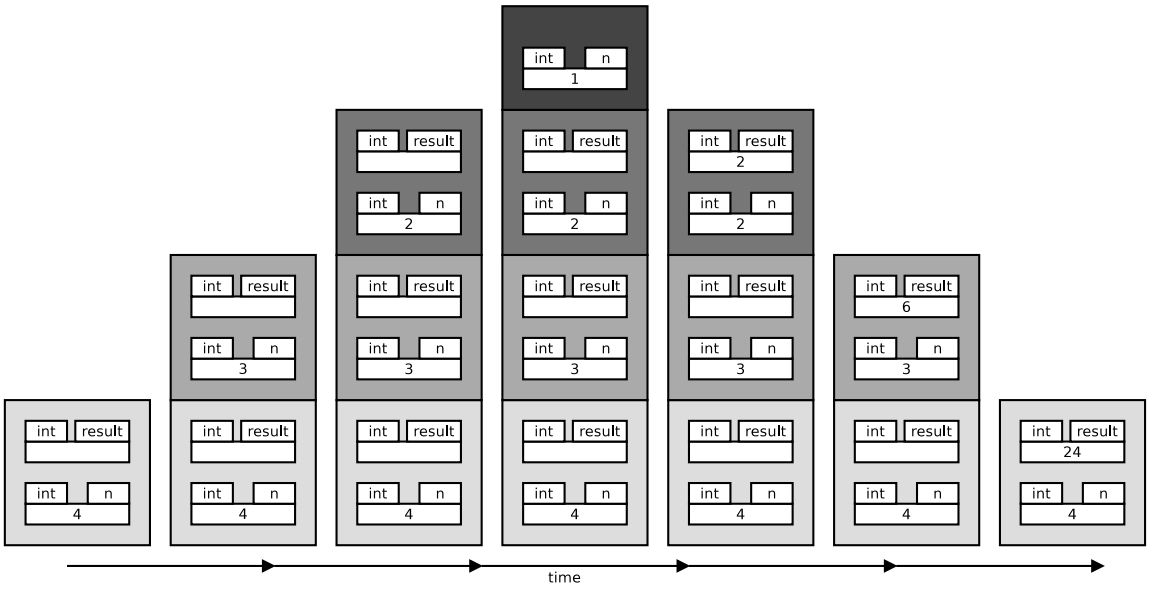

Figure 1: Evolution of the computer stack when factorial(4) is called. Every subsequent call of the method adds a new level to the stack, and the variable where the return value will be stored is left with an undecided value until the method call returns.

1.3 Pitfalls

At this point you are familiar with loops (`while`, `for`...) and you know that you must be careful with the way you set the condition of a loop to prevent infinite loops that block your programs. In a similar way, you must keep in mind a couple of things when you write recursive code.

Base case Every recursive program or method must have a base case for which the answer is known. In the case of the factorial, we know that 1! is 1; in the case of the Fibonacci numbers, we know that the answer for F(1) and F(2) is 1.

If a recursive method does not have a base case, the method will call itself continuously until the stack is overflown, i.e. until the computer tries to add a new level to the stack (to execute the next method) and finds there is no more memory to do it. At that point, the Java Virtual Machine will throw a `StackOverflowError`.

Look at the code on page 1. It does not have a base case, so it will run forever until the computer runs out of memory space for the stack.

No convergence Apart from a base case, it is important that each iteration of a recursive method tries to answer a simpler or more limited version of the same question. In other words, every time a method is called again its parameter must be a smaller number, or a shorter list, or —in general— must be nearer to the base case.

In simple cases like the factorial or the Fibonacci numbers, it is easy to see that the method will converge. This can be more difficult when the recursion involves more than one method, e.g. method A calls method B, and method B recursively calls A (note than A and B may be methods in different

classes[4]). You must be especially careful when you have a loop of several methods calling each other recursively because it may be quite complex to see whether they converge in all cases or not.

If your recursive method does not converge towards a base case, you will overflow the stack and the Java Virtual Machine will throw a `StackOverflowError`.

1.4 When to use recursion?

Recursive methods are a way of performing a computation more than once. In that respect, they are similar to loops. When a problem is solved by using loops, it is said to be solved *iteratively*; when recursive methods are used, it is solved *recursively*.

Every problem that can be solved iteratively can be solved recursively, and viceversa[5]. Some programming languages do not have loops, and some others do not have recursion. Most modern languages, including Java, Java Decaf, and Groovy have both.

There is no clear set of rules about when to use one or the other. In general, recursive code tends to be shorter, simpler, and clearer (but not always). On the other hand, recursive calls make heavy use of the stack, which can be an issue in small devices with limited memory.

As a general rule, you should use whatever you feel is more natural for the problem at hand. Sometimes the solution to a problem can be naturally thought in terms of "if only I had... it would easy to..."; for example, if you want to build a wall of 100 bricks, you can think "if only I had a wall of 99 bricks, it would be easy to add the 100th brick"; and if you wanted a wall of 99 bricks, it would be easy as long as you had a wall made of 98 bricks, and so on. Then you just need a trivial base case, like "building a wall of 0 bricks" and you have a recursive solution to your problem.

You should keep in mind that recursive solutions are sometimes a bit *harder to think*, especially when you are not familiar with the idea, but they are generally *easier to read*. As code is written once but read many times (for maintenance, extension, etc), it is usually a good idea to be a bit biased towards recursive approaches unless the amount of memory is really limited (and even in that case, remember that memory and computing power grow every year).

[4] A real story: There was a project in which we had a window, and inside of it a canvas on which different objects were painted. Different elements of the application needed to know where the origin was; some of them asked the window and some of them asked the canvas. If the window did not know where the origin was (e.g. at initialisation), it asked the canvas. The canvas set the origin at initialisation, and informed the window. If the user started a new canvas (e.g. by opening a new file), the new canvas did not know the origin but it could ask the window (which was already there). All was working well until the first day that a user, instead of starting the application and then a file, opened a file directly. The file opened in a canvas, that did not know where the origin was, so it asked the window; the window had not been told where the origin was, so it asked the canvas; and the recursive loop continued until the application crashed in front of the eyes of the (quite upset) user.

[5] This is one of the results of the Turing–Church thesis.