Functions and error handling

Function definition

The programs that we have been typing so far are becoming more and more complex. It may be hard to maintain or keep clean a piece of code of some hundred lines and, besides, we may need to repeat the same piece of code in many parts of our program. In order to keep our code clean an reusable, we need to divide it into functions or modules. In this section we'll learn what a function is, how to define functions in Java and how to use them.

1. Basic function management

Functions let us arrange our code so that we can re-use a piece of code many times without having to duplicate the code. We just assign this piece of code a name (*function name*) and then we can *call* this code from other parts of our program. This paradigm is also called **modular programming**, since we define *modules* or functions to group our code in small subtasks, and call each subtask whenever we need it.

If we want to use functions within a class, we can declare them as `public static` if we want to call them from anywhere. We will learn in next units how to declare other types of functions and how to set their visibility, but for now we are going to deal with public, static functions.

When defining a function, we need to specify the return type (or `void` if the function does not return anything), the function name and a pair of parentheses. This function prints a welcome message in the screen:

```
public static void welcome(){
    System.out.println("Hello, welcome to this program!");
}
```

We can call this function from any other function of the same class (including `main` function) by using the function name and the parentheses:

```
public static void main(String[] args){
    welcome();
}
```

Note that function names always start in lower case in Java (in C#, they start in uppercase if they are public).

1.1. Returning values

Functions can return values. This values can be assigned to variables, or used in other expressions. In order to return a value from a function, its return type must be other than *void*, and we must use a `return` clause to specify the value returned. For instance, this function returns a welcome message:

```
public static string welcome(){
    return "Hello, welcome to this program!";
}
```

So, if we call this function from the *main* program, we will not see anything in the screen:

```
public static void main(String[] args){
    welcome();  // Nothing happens
}
```

We need to assign the return value to a variable, or use it in another expression:

```
public static void main(String[] args){
    // Option 1
    string result = welcome();
    System.out.println(result);

    // Option 2
    System.out.println(welcome());
}
```

2. Using parameters

Some functions need some additional data to do their job. These data can be passed to the function as **parameters**, some kind of variables that are specified within the parentheses, including the type of each parameter and its name. For instance:

```
public static void myOtherFunction (int a, String b){
    ...
}
```

Then, whenever we need to call this function, we need to specify the values of these parameters, in the same order:

```
public static void main (String[] args){
    ...
    myOtherFunction(3, "Hello");
}
```

2.1. Parameters by value or by reference

Passing a parameter **by value** means that we are passing a *copy* of the original value, so this original value can never be changed from within the function. This is the default situation for simple values, such as integers or floating point numbers:

```
public static void myFunction(int value){
    // This increment will have no effect after exiting the function
    value = value + 1;
    System.out.println("Inside function: " + value);
}

public static void main(String[] args)
{
    int number = 3;
    myFunction(number);         // Inside function: 4
    System.out.println(number); // 3
}
```

However, complex data such as arrays are always passed **by reference**. This means that we are passing a reference or pointer to the original values, and we can change these values from within the function *as long as* we don't change the whole reference, this is, as long as we don't reassign the whole variable. Let's have a look at this example:

```
public static void myFunctionWithArray(int[] data){
    data[0] = 10;           // OK
}

public static void myFunctionWithArray2(int[] data){
    data = {1, 2, 3, 4};    // No effect outside the function
}

public static void main(String[] args){
    int[] numbers = {1, 1, 1, 1};
    myFunctionWithArray(numbers);
    // Here numbers = {10, 1, 1, 1}
    myFunctionsWithArray2(numbers);
    // Here numbers = {10, 1, 1, 1}
}
```

The same happens with some other complex data, such as objects, as we will see in later units.

2.2. More on return types

Let's have a look at this function. It returns the maximum value of its two input parameters:

```
public static int maximum(int n1, int n2){
    if (n1 > n2){
        return n1;
    }
    else{
        return n2;
    }
}
```

From the point of view of *clean code*, a function should only have ONE return point, so that it's easy for us to find out where this function ends. In order to meet this requirement, we can rewrite the function this way:

```
public static int maximum(int n1, int n2){
    int result = n1;
    if (n2 > n1){
        result = n2;
    }

    return result;
}
```

2.3. Global and local variables

A **global variable** is a variable that has been defined outside of any function (and inside a class), so that it can be shared among all the functions of this class (and possibly other classes). In order for a variable to be global, we should declare it as *public* and *static* for now:

```
class MyClass{
    public static int myGlobalVariable;

    ...
}
```

A **local variable** is a variable that has been defined inside a function, and it does not exist outside this function:

```
class MyClass{
    public static void aFunction(){
        int number = 3;
        anotherFunction();
    }

    public static void anotherFunction(){
        System.out.println(number); // Error: number does not exist here
    }
}
```

Global variables are not a good choice, generally, since they can produce side effects. This is, we can accidentally change their values from any function of our program. The most recommended way of dealing with variables is to declare them as local, and pass them to other functions as parameters:

```
class MyClass
{
    public static void aFunction()
    {
        int number = 3;
        anotherFunction(number);
    }

    public static void anotherFunction(int n)
    {
        System.out.println(n); // OK
    }
}
```

> **Exercise 1**:
>
> Create a program called **Palindrome** with a function called `isPalindrome`. This function will take a string as a parameter and return a boolean indicating if this string is a palindrome (this is, a string that can be read the same backward as forward, ignoring upper or lower case, and whitespaces). Test this function from the *main* function with the texts *Hannah*, *Too hot to hoot* and *Java is the best language* (this last text is NOT a palindrome).

> **Exercise 2**:
>
> Create a program called **CountOcurrences** with a function called `countString`. This function will take two strings *a* and *b*, and an integer *n* as parameters, and it will return a boolean indicating if the string *b* is contained at least *n* times in the main string *a*. Try it from the main function with the main string a = `This string is just a sample string`, the substring b = `string` and the number n = 2 (it should return `true`).

3. Recursion

Recursion is the ability of a function to solve a task by calling itself multiple times with simpler versions of the problem to be solved.

3.1. Main components of a recursive function

In every recursive function we can find two components:

- **Base case**: the simplest problem that we can find, in which recursive calls finish. There can be more than one base case in a recursive function, but there must be at least one of them.
- **Recursive case**: every internal call to the same function with a simpler version of the problem.

So, whenever we try to solve a problem recursively, we need to think about the simplest value of this problem, and make the function tend to this simplest value through consecutive calls.

3.2. Some introductory examples

Let's get started with recursion by analyzing some simple examples...

3.2.1. The factorial

Factorial is a mathematical operation that consists in multiplying a number by all its descending sequence up to 1. For instance factorial of number 5 can be calculated as 5 * 4 * 3 * 2 * 1, and it's represented as *5!*.

We can see the factorial as a simple multiplication sequence, or as a recursive expression: the factorial of a given number can be decomposed in two parts: the number itself and the factorial of the previous number. So, factorial of number 5 can be seen as 5 multiplied by the factorial of number 4:

```
5 * 4 * 3 * 2 * 1 = 5 * 4!
```

Recursively, we can also calculate the factorials of all the subsequent numbers:

```
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1
```

As soon as we get to number 1, we've found our base case, the simplest number to calculate the factorial. After this point is reached, we can go back and calculate the rest of pending operations:

```
1! = 1
2! = 2 * 1 = 2
3! = 3 * 2 = 6
4! = 4 * 6 = 24
5! = 5 * 24 = 120
```

We can represent this as a recursive function in Java, this way:

```
public static int factorial(int number){
    // Base case
    if (number == 1){
        return 1;
    }
    // Recursive case
    else{
        return number * factorial(number - 1);
    }
}
```

Alternatively, we can also express this operation as an iterative algorithm as well:

```
public static inf factorialIterative(int number){
    int result = 1;
    for (int i = number; i > 1; i--){
        result = result * i;
    }
    return result;
}
```

> **NOTE**: recursive functions are an exception to the rule of a single *return* point. In recursive functions, we usually find a *return* point for the base case and another *return* point for the recursive case.

3.2.2. Fibonacci series

Let's see another example. Fibonacci series starts with numbers 0 and 1, and then next number is always generated as the sum of two previous numbers. So, we have this sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21...

How could we calculate the *nth* number of the Fibonacci series? We could define a function like this:

```
public static int fibonacci(int n){
    int previous, previous2, result;

    if (n == 0){
        result = 0;
    }
    else if (n == 1){
        result = 1;
    }
    else{
        previous2 = 0;
        previous = 1;
        for (int i = 2; i <= n; i++)
        {
            result = previous + previous2;
            previous2 = previous;
            previous = result;
        }
    }

    return result;
}
```

However, if we represent this function recursively, we can get this:

```
public static int fibonacci(int n)
{
    if (n == 0 || n == 1){
        return n;
    }
    else{
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

So, in some situations, recursion helps us write a shorter code to solve a problem.

> **Exercise 3**
>
> Create a program called **CountDigits** with a function called `countDigits` that receives a number and, recursively, counts the number of digit that this number has. Try it with number 1252 (4 digits), from the *main* function.

> **Exercise 4**
>
> Create a program called **PalindromeRecursive** with a function called `isPalindrome` that receives a string (with only alphabetical letters from a to z in lower case) and recursively determines if this string is palindrome or not, returning a boolean with the result. Try it with the same input values suggested for Exercise 1.

4. Using *main* arguments

As you may have noticed, *main* function has a String array as parameter:

```
public static void main(String[] args){
    ...
}
```

This means that we can pass as many arguments as we need to this *main* function from the command line. The first argument will be placed at index 0, the second argument at index 1 and so on.

```
public static void main(String[] args){
    if (args.length > 0){
        System.out.println("Received " + args.length + " args.");
    }
}
```

For instance, if we try to run a program called *Main* with this command line:

```
java Main Nacho 20
```

Then `args[0]` will be *Nacho* and `args[1]` will be 20.

> **Exercise 5**
>
> Repeat exercise 3 in another program called **CountDigitsMain** in which the number to be checked will be passed from the command line.