1 Methods

We have briefly introduced the concept of *methods*. In the last section we have seen that they have an *identifier* —like variables— and that they have (round) brackets. Sometimes we can put variables or values inside the brackets, as with methods `charAt()` or `substring()` of class String.

Let's look at methods in a bit more detail now, because they are very important.

1.1 Why are methods important?

Imagine that you are writing a program in which you need to make some checks on user input. For example, your program needs the user to introduce several logins/usernames and the program must make sure that they do not contain spaces and they are all lower case letters. You could have some code like:

```
String login = System.console().readLine();
boolean loginIsValid = true;
for (int i = 0; i < login.length(); i++) {
    char c = login.charAt(i);
    if (!Character.isLetter(c) || !Character.isLowerCase(c)) {
        loginIsValid = false;
    }
}
```

As you can see, the loop goes through the whole length of the string *login* checking that each and every character is a letter in lower case. If that is not the case (no pun intended), the boolean flag *loginIsValid* is set to `false` so the program knows that a new login must be asked from the user.

You can think of this code being necessary at different parts of your program. This can be useful when you are adding new users to the program, when you are changing the username of a user, and when you want to remove a user from the system, to name but a few. If you have to write the same code for every single place that you need it, you have two problems.

First of all, it is **boring**. You have to type it several times. Even if you copy–and–paste, you have to find the file where it is and this takes time and effort (normal programming projects have thousands of source code files, and sometimes they are quite long). Programmers like to make computers work for them and not the other way around.

Second, but most important, if you find an error (a so-called *bug*) in those lines of code, you **only need to fix it in one place**. If you have to fix it in several places, sooner or later you will forget to fix one of them because you are human. That means that your program will still be *buggy* even if you are sure you have fixed it, which is the worse thing than can happen to you. There is a very important rule in programming that is usually referred to as the DRY principle:

<div align="center">

DRY: Don't Repeat Yourself

</div>

Duplication of code will result in problems sooner or later, and we must always avoid it. This is what methods are for. They allow the programmer to put code in just **one place** that can be used from anywhere else in the program. This means that, for example, if you need to fix a bug,

you fix it in only **one place**; and if you need to improve the code to add a new feature, or to make it faster, or for any other reason, you only need to change it in **one place**. That way you are sure that you fix things once and forever.

Besides, separating code in methods also makes your code clearer, and that is good. Compare the easiness of reading the code above with the following statement:

```
String login = System.console().readLine();
boolean loginIsValid = containsOnlyLowerCaseLetters(login)
```

It is easy to read, isn't it?

1.2   Defining a method

A method is defined by its name, its return type, and the parameters inside the brackets. We will look at parameters in the next section.

A method's name must be an identifier like those used for variables: starting with a letter and consisting of letters, digits, and underscores ("_"). Actually, underscores are rarely used when you are programming in Java. Methods usually have names consisting of a single word (e.g. `length()`) or several words in so-called *camel case* starting with a verb (e.g. `isLetter()` or `containsOnlyLowerCaseLetters()`). Note that variables usually have identifiers that are a noun or an adjective, not methods' identifiers usually start with a verb (with the occasional 1-word noun, like `length()`).

The return type of a method is a data type, simple or complex, that is returned by the method when it finishes. When a method finishes, it must return a value of the appropriate data type by using the keyword `return`. For example, the method `containsOnlyLowerCaseLetters()` could look like the following code:

```
boolean containsOnlyLowerCaseLetters(String login) {
    boolean result = true;
    for (int i = 0; i < login.length(); i++) {
        char c = login.charAt(i);
        if (!Character.isLetter(c) || !Character.isLowerCase(c)) {
            result = false;
        }
    }
    return result;
}
```

As you can see, a method definition starts in Java with the return type, followed by the name of the method, followed by the parameters inside brackets. Note that the code is the same that we wrote before. The difference is we only need to write it once and can use it from anywhere else in the program just by calling it by its name. We are avoiding duplication of code. That is good.

This method has a return type "boolean", so we must create a boolean variable and return it at the end. The `return` statement is the last statement that is executed inside any method: once you return a value there is nothing else to do. This means that we can make our method a bit more efficient by returning early:

```
boolean containsOnlyLowerCaseLetters(String login) {
    for (int i = 0; i < login.length(); i++) {
        char c = login.charAt(i);
        if (!Character.isLetter(c) || !Character.isLowerCase(c)) {
            return false;
        }
    }
    return true;
}
```

Instead of traversing the whole string, we return `false` as soon as we find a character in the login that is not a letter or is not in lower case. If we arrive at the end of the string, that means that all characters are fine and we can return `true`. Note that the loop is automatically terminated when the method is finished, i.e. when the return value is given back. It does not matter if the loop has run to the end or not, or whether there is more code after the `return` statement.

This is important. It means that **you can only return from a method once** every time you call it. Even if you have several `return` statements, your program will only execute the first one it encounters. This does not mean that you can only return one piece of data from any method; remember that the return value can be any simple or complex data (like String or Person). By returning a complex type, you can return as much information as you want.

You have noticed that there is something inside the round brackets, a String called *login*, that is used inside the method. The variables inside the round brackets are called the *parameters* of the method.

1.3 Positional parameters

You can think of methods as small mini-programs: they get some input, they produce some output. The output is the return value, the input are the so-called *positional parameters*.

Positional parameters are declared in the same way as any other variable. The programmer must specify the type and give it a name, an identifier. Parameters can have any type, simple or complex, and are separated by commas. If a method does not return any value, the keyword `void` is used as a return data type (more on void methods below).

In the body of the method (what comes inside the curly brackets) parameters are used in the same way as other variables declared and initialised inside the method. Parameters are initialised when the method is called (see below).

Some methods do not have any parameters. If that is the case, the method is defined and called with an empty list of parameters, i.e. empty brackets. The method `length()` is an example of a method without parameters.

Positional parameters get their name because they come in some order and they must be called *in the same order*. Let's see how it is done.

1.4 Calling (i.e. using) methods

You have already seen how to use some methods, like `length()` or `substring(int,int)` of strings[1]. They are methods of a class, so you access them with the name of the variable and a dot:

---

[1]As you can see, now that you know that methods have parameters I will start referring to them in a proper way, specifying the type of their parameters.

```
int familyName = fullName.substring(6,12);
```

This is called *calling*, *executing*, or *running* the method (the first term is more common). You call a method by using its name and specifying the value of its parameters. It is important to call a method with the right parameters in the right order. For example, if you have a method like `repeat(String s, int times)` you cannot call it like `repeat(3, "Some text")` because you will get an error. Note that you do not need to say the type of the parameters: the computer already knows because they are specified in the method's definition.

1.5 Scope

In a way, you can see the execution of a method as the execution of a small program in which some of the variables (the parameters) are initialised in advance (with the values given by the caller).

A method has access to variables outside of it, but its own variables are hidden from the rest of the world. They cannot be read or modified from outside the method. In technical terms, the *scope* of variables inside the method is the method itself.

Actually, scope is not just a property of methods. In Java, scope is roughly defined by any pair of curly brackets, so loops and classes have scope too. That is why you must use the name of a class to access variables inside the class, because otherwise they are restricted to be used *in their scope*.

But what happens with the method's parameters? They are like variables for the method but they also come from outside the method, don't they? Good question.

When you call a method, the variables used to initialise the parameters are *copied*, and the copies are used instead of the original variables. This means that any change to the method's parameters is forgotten as soon as the method returns. You can check for yourself by running the following code:

```
void add1000(int number) {
    println "Starting method, parameter is " + number;
    number = number + 500;
    println "In the middle of method, parameter is " + number;
    number = number + 500;
    println "Ending method, parameter is " + number;
}
// program execution starts here
int myNumber = 0;
println "Starting program, my number is " + myNumber;
add1000(myNumber);  // method call
println "After the method is used, my number is " + myNumber;
```

The output is:

```
Starting program, my number is 0
Starting method, parameter is 0
In the middle of method, parameter is 500
Ending method, parameter is 1000
After the method is used, my number is 0
```

4

You can see that the value of `myNumber` never changes. The method made a copy of `myNumber`, used it —under the name `number`— inside the method, and then forgot it as soon as the `return` statement was reached.

Void methods

Note also that the method does not return anything (i.e. return type is `void`). When this is the case, i.e. there is nothing to return, the method ends after the final statement is reached or when the first `return` statement is found (with no value). There is no need to have a return statement, but it may be useful in some cases that we will see further down the line.

Methods that return void are sometimes called *procedures*. Methods that do return a value are sometimes called *functions* by analogy with mathematical functions.

1.5.1 Beware of parameters of complex types!

We have seen that methods make a copy of their parameters, and use the copy instead of using the actual variable. In other words, *what happens in the method remains in the method* with the only exception of the return value.

Actually, this is not completely true. Methods make copies of the "boxes" that are sent to them, but not of the objects they are pointing to if they are complex types. This means that changes to an object survive the method scope. Let's see how this works with a detailed example.

```
class Point {
    int x;
    int y;
}
// This method increments the int by 1 and
// moves the point to the right
void increment(Point point, int n) {
    n = n + 1;
    point.x = point.x + 1;
    point = null;
    println "  At the end of the method..."
    println "  The integer is " + n;
    println "  The point is " + point;
}
// Program execution starts here
Point myPoint = new Point();
point.x = 0;
point.y = 0;
int myInt = 0;
println "The integer is now " + myInt;
println "The point is now " + myPoint.x + "," + myPoint.y;
println "Calling method increment(Point, int)..."
increment(myPoint, myInt);
println "The integer is now " + myInt;
println "The point is now " + myPoint.x + "," + myPoint.y;
```

The output is:

```
The integer is now 0
The point is now 0,0
Calling method increment(Point, int)...
  At the end of the method...
  The integer is 1
  The point is null
The integer is now 0
The point is now 1,0
```

As you can see, the `int` (a simple type) is changed inside the method but this change is forgotten as soon as you leave the method. The `Point` is also modified inside the method, in two different ways: first, the value of one of its coordinates is changed; then, the `Point` itself is set to `null`. When we come out of the method, only the second change is forgotten. As you can see, the method copies the "boxes" of the parameters; but if the boxes are pointers and the objects that those boxes are pointing to are changed, those changes stay even after you leave the method.

Note. In some books you may find that passing simple types to methods as parameters is called *passing parameters by value*, while passing complex types is called *passing parameters by reference*. The bottom line is the same: changes to parameters passed by value do not survive the end of the method while changes to parameters passed by reference do. This is because the method only copies the "box" of a complex type, i.e. the pointer, not the object in memory that it points to.

1.5.2 Exercise

Draw a diagram of all the variables involved in the example above, and how they are copied and modified, including their pointers (when applicable); make sure that you understand what is the state of all variables (inside and outside the method) at every point.

1.6 Methods in classes

In the same way that classes can have member fields, they can also have member methods. You already know some examples like `length()`, which is a member method of String.

Member methods are written like any other method, only they are defined inside the class and can only be called on objects of the class using a dot (e.g. `str.length()` instead of just `length()`), in the same way that fields can only be called from the object (e.g. `point.x`). Let's see an example:

```
class UnidimensionalPoint {
    int x;
    int getX() {
        return x;
    }
    void setX(int x) {
        this.x = x;
    }
    UnidimensionalPoint clone() {
        UnidimensionalPoint copy;
```

```
        copy = new UnidimensionalPoint();
        copy.setX(x);
        return copy;
    }
}
```

The class `UnidimensionalPoint` has one coordinate and three methods: one for getting its only coordinate, one for changing it, and one for getting an exact copy of itself. This example introduces a new keyword, `this`, which means "this object we are now in". It is used to solve ambiguities like in the method `setX(int)`. Note that the field `x` of `UnidimensionalPoint` has the same name as the only parameter of method `setX(int)`. If we want to assign the value of one to the other we may write something like:

```
void setX(int x) {
    x = x;   // This does not work!
}
```

This does not really do anything. When a parameter has the same name as a field, the computer can only identify one of them. The rule that the computer uses is *local is always more important than global*, so the parameter `x` "hides" the field `x`. This is sometimes called *shadowing*.

In order to prevent shadowing, we can use different names, as in:

```
void setX(int newX) {
    x = newX;
}
```

Or we can use the `this` keyword as in the original example. The original statement `this.x = x;` can be read as "take the `x` field of *this* object (*point*) and assign the value of parameter `x` to it".

You can use the `this` keyword at any time to mean "this object". For example, we could have said `copy.setX(this.x)` in method `clone()`. Usually, `this` is only used when necessary, e.g. to prevent ambiguities and shadowing as in the example above.

Note. A method, like `getX()` that only returns the value of a field is sometimes called an *accessor method*, and very often a *getter*. A method, like `setX(int)` that only changes the value of a field is sometimes called a *mutator method*, and very often a *setter*.

Side effects

We have already seen that not all methods return a value, some of them return `void`, i.e. nothing; and they are can be called *procedures*. Even if they do not return anything, they can be useful because of their *side effects*. For example, we could have a Tamagotchi[2] class like this:

```
class Tamagotchi {
    int age = 0;
    void grow() {
        age++;
    }
}
```

---

[2]A Tamagotchi is a digital pet that was quite popular in the 1990s.

You can see that the method `grow()` does not take any parameter, and does not return any value, but it has an effect: it increases the age of the tamagotchi. This is called a side effect because it happens out of the method (but inside the class).

1.7 Flow of execution

We have already seen that the flow of execution of a program is not always linear, from the first line to the last line. Constructs like branches (`if...else`) or loops (`while`, `for`) can alter the flow and skip some lines and repeat some others. Methods also have a crucial effect on the flow of execution of a program.

First of all, when a method is defined nothing is executed. The program will only execute code inside a method is the method is actually called from somewhere else.

When a method is called, the next line being executed is the first line in the method, and then the flow continues in the method as in a mini-program. If a method calls another method, the execution flows to the first line of the other method. When the execution of a method finishes (either a return statement and/or the end of the method is reached), the flow continues from the point when the method was called.