Programming in Java (23/24) – Day 4: Methods

1 Introduction

We have briefly introduced the concept of *methods*. In the last day we have seen that they have a name, an *identifier* —like variables— and that they have (round) parentheses. Sometimes we can put variables or values inside the parentheses, 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. 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:

```java
        java.util.Scanner scan = new java.util.Scanner(System.in);
        String login = scan.nextLine();
        boolean loginIsValid = true;
        int i = 0;
        while (i < login.length()) {
            char c = login.charAt(i);
            if (!Character.isLetter(c) || !Character.isLowerCase(c)) {
                loginIsValid = false;
            }
            i++;
        }
```

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 worst thing that can happen to you—there is a hidden bug. There is a very important rule in programming that is usually referred to as the DRY principle:

DRY: Don't Repeat Yourself

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:

```
        java.util.Scanner scan = new java.util.Scanner(System.in);
        String login = scan.nextLine();
        boolean loginIsValid = containsOnlyLowerCaseLetters(login);
```

It is easy to read, isn't it?

The DRY principle is a key principle in programming, and many concepts in modern programming languages have been introduced to make it easier for programmers to follow the DRY principle.

2 Defining a method

A method is defined by its name, its return type, and the parameters inside the parentheses. 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*, while 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, which does not go *in* method `main`, but would be placed above or below it in the same class, or into a different class:

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

As you can see, a method definition starts in Java with the keyword **static**,[1] followed by the return type, the name of the method, followed by the parameters inside parentheses. The keyword **static** tells the Java compiler that the method can be used without an object of the class that contains the method, in contrast to, e.g., method `substring` in class `String`, which can be used only with a given instance of class `String`.[2]

Note that the code is the same that we wrote before. The difference is we only need to write it once and then we 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:

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

[1]We will later today also see methods that do not have the keyword **static**.

[2]But why did the designers of Java use a keyword called **static** for that rather than, say, "`standalone`"? An attempt at an explanation: the word **static** at the beginning of a method definition also indicates that when we compile code that uses that method, we (and the compiler) know for sure that it is the code *in that method* that will be executed. This may sound rather obvious at the moment: surely the code of the method that we are calling is also the code that will be executed? It will become clearer in later weeks that this is not always the case, when we talk about subclasses and overriding as a mechanism to redefine methods. There are also other limitations for **static** methods, which we will discuss later in the module.

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 parentheses, a String called *login*, that is used inside the method. The variables inside the round parentheses are called the *parameters* of the method.

3 Formal 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 inputs are called *formal parameters* (or *positional parameters*).

Formal parameters are declared in the same way as any other variable. To declare a formal parameter, 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 braces) 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 parentheses. The method `length()` in class `String` is an example of a method without parameters.

Formal parameters in Java are sometimes called *positional* parameters because they come in some order and they must get their values *in the same order*. Let's see how this is done.

4 Calling (i.e., using) methods

You have already seen how to use some methods, like `length()` or `substring(int,int)` of strings.[3] They are methods for the objects of a class, so you access them with the name of the variable and a dot:

```
        int length = name.length();
```

[3]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 and sometimes also their names.

This is called *calling*, *executing*, *invoking*, or *running* the method (the first term is the most 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 when you call a method: the computer already knows because they are specified in the method's definition.

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 braces, 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:

```java
public class D4Example1 {
    static void add1000(int number) {
        System.out.println("Starting method, parameter is " + number);
        number = number + 500;
        System.out.println("In the middle of method, parameter is " + number);
        number = number + 500;
        System.out.println("Ending method, parameter is " + number);
    }

    // program execution starts here
    public static void main(String[] args) {
        int myNumber = 0;
        System.out.println("Starting program, my number is " + myNumber);
        add1000(myNumber);  // method call
        System.out.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
```

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

Here myNumber is also called the *actual parameter* used for the method call. In the method call, the value of the actual parameter gets copied into the *formal parameter* of the method. This works analogously for methods that accept several parameters, such as substring(**int**,**int**).

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 return 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.

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 have complex types. This means that changes to an object (an instance of a class, a complex type) survive the method scope. Let's see how this works with a detailed example.

```java
class Point {
    int x;
    int y;
}
```

```java
public class D4Example2 {

    // This method increments the int by 1 and
    // moves the point to the right
    static void increment(Point point, int n) {
        n = n + 1;
        point.x = point.x + 1;
        point = null;
        System.out.println("  At the end of the method...");
        System.out.println("  The integer is " + n);
        System.out.println("  The point is " + point);
    }

    // Program execution starts here
    public static void main(String[] args) {
        Point myPoint = new Point();
        myPoint.x = 0;
        myPoint.y = 0;
        int myInt = 0;
        System.out.println("The integer is now " + myInt);
        System.out.println("The point is now " + myPoint.x + "," + myPoint.y);
        System.out.println("Calling method increment(Point, int)...");
        increment(myPoint, myInt);
        System.out.println("The integer is now " + myInt);
        System.out.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.

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.

6 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 if 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 where the method was called.

Let us discuss the following program as an example to see how the flow of execution works in Java (and also in other programming languages):

```java
1  public class D4Example3 {
2
3      static int add(int op1, int op2) {
4          int result = op1 + op2;
5          return result;
6      }
7
8      static void doSomething() {
9          int a = 5;
10         int sum = add(a, 10);
11         System.out.println(sum);
12     }
13
14     public static void main(String[] args) {
15         doSomething();
16     }
17 }
```

Running this program starts in line 14, with the `main` method. This adds a *frame* to the *method call stack* of the Java Virtual Machine (the `java` command that we saw on Day 1), which is running the program.

You can think of the method call stack as a stack of paper inside the computer's memory. Each paper sheet (i.e., stack frame) corresponds to a method call with information about which method was called, what contents the boxes defined in the method have (i.e., the current values of the variables in the method), and what the current statement of the method is. In our examples, we will refer to the statements by their line numbers.

Only the method call whose stack frame is currently on top of the stack can make progress in the program at a given time. In our analogy, we can write only on the paper sheet that is on top of the stack of paper (to change the information about the values of the variables and about the current statement).

The `main` method has one variable, its formal parameter variable `args`, which is added to the frame for the call to the `main` method on top of the method call stack. The first statement in the `main` method that is executed is the call to method `doSomething` in line 15. Whenever a method is called, a new frame is added on top of the call stack with the variables for the method. The frame that is now second-to-top keeps its information of the current statement set to line 15 until the call to `doSomething` completes its job, removing the stack entry for `doSomething`.

So now our call stack has two frames:

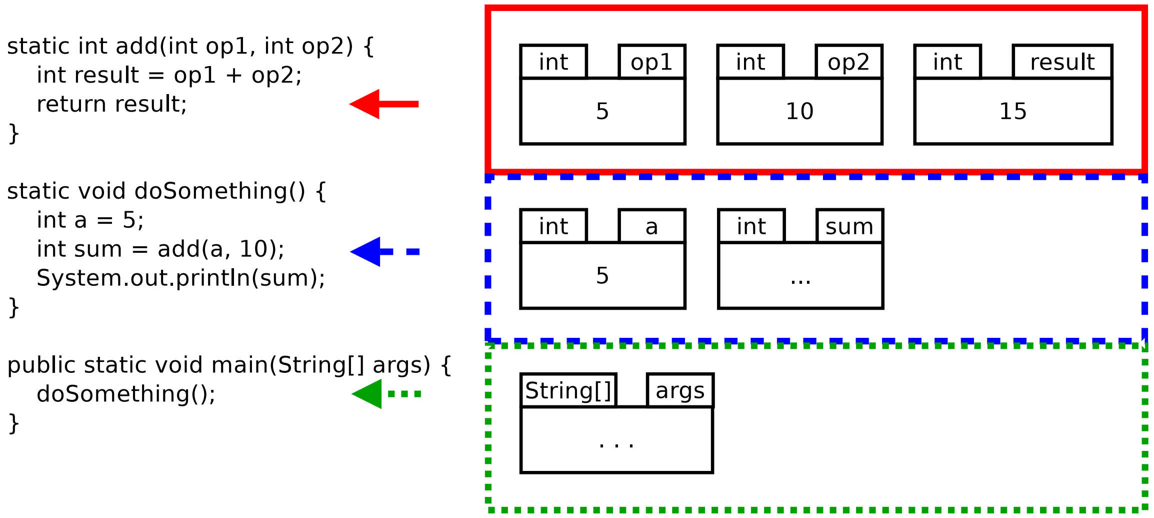
- At the bottom, there is the frame for method `main` with the box for its variable `args` and its current instruction set to line 15.
- On top, there is the frame for method `doSomething` with boxes for its variables `a` and `sum`, and its current instruction set to line 8.

Execution continues in the method doSomething whose frame is on top of the call stack. In line 9, variable a is initialised to the value 5, which is stored in the box in the top frame of the call stack. In line 10, method sum is called with the current value of a (i.e., 5) as the first *actual parameter* (or *argument*) and the value 10 as the second actual parameter.

This method call adds another frame to the stack, for method sum, starting in line 3. Method sum has three variables: op1, op2, and result. Here the variables op1 and op2 are *formal parameter variables*. The formal parameter variables get their initial values from the actual parameters that were given to the method call:

- The first formal parameter variable, op1, gets its value from the first actual parameter of the method call to sum – the value 5.
- The second formal parameter variable, op2, gets its value from the second actual parameter of the method call to sum – the value 10.

With these values of op1 and op2, in line 4 the variable result is initialised to 5 + 10 = 15. Then the current statement is updated to line 5. You can see the call stack at this moment in Figure 1. Here the stack frame for method main is dotted and green, the stack frame for the call to method doSomething is striped and blue, and the stack frame for method add on top is drawn with a solid line and red. The arrows indicate the current statements of each frame, with the statement for the top frame as the next one to run.



```
static int add(int op1, int op2) {
   int result = op1 + op2;
   return result;
}

static void doSomething() {
   int a = 5;
   int sum = add(a, 10);
   System.out.println(sum);
}

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

Figure 1: The call stack, just before the **return** statement of method add in line 4 of D4Example3 is executed. The variables of the frames *below* the top stack frame (a, sum, and args) *cannot* be accessed: only the frame on top is accessible. Note that if variables in a stack frame are of complex types, their "boxes" would be pointers pointing to objects in the (shared) heap. (For now, let us ignore the content of the box args, which is indeed of complex type.) The arrows mark the current statements of the respective stack frame and also their method. Only the top frame on the stack is currently running, all the frames on lower levels are waiting to become the frame on top level again so that they can continue.

When Java executes the **return** statement in line 5, the value 15 in variable `result` is returned back to the caller of method add, and the frame for the call to method add is removed from the top of the call stack and erased.

Now the flow of execution goes back to the current statement of the blue, striped stack frame for method `doSomething` that is now again on top of the stack. Just like removing the top sheet of paper from a paper stack reveals the content of the sheet right below it, now the content of the stack frame for `doSomething` becomes accessible again. The current statement is still the one in line 10, which initialises variable `sum` with the value that was just returned by the value from the method call to add, i.e., the value 15.

The next statement is then in line 16, the call to method `System.out.println` with the value 15 in variable `sum` as actual parameter. A new stack frame for the call to `System.out.println` is added on top of the call stack, and execution continues in that stack frame. When method `System.out.println` has completed its job, its stack frame is removed, and execution continues at the end of method `doSomething` in line 12. The method just ends (it has return type **void**, so this is okay), but we can treat this like a **return** statement that does not return any value.

Thus, the blue, striped stack frame for the call to `doSomething` from the green, dotted stack frame for `main` is removed from the top of the stack. Now finally the frame for our `main` method is on top again, and the method can continue with its next statement. But in line 16 there is nothing left to do, and the method returns. This removes the last frame from the call stack, and execution of our Java program terminates (i.e., it stops running).

7 Methods in classes (and the keyword `this`)

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`). Another difference is that we do not use the word **`static`** for member methods. Let's see an example:

```java
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 be tempted to write something like:

```java
    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:

```java
    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".

In a member method, 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()`. Some programmers use **this** only when necessary, e.g., to prevent ambiguities and shadowing as in the example above. Other programmers use **this** whenever this is possible, so one sees immediately that the variable belongs to the object. Either style is fine, but you should use a consistent style to make your code easy to read.

So, member methods can access the instance variables of the class in which they are defined via the keyword **this**, in the example by writing `this.x`, or without further qualification, by just writing the name of the variable `x` (unless we have shadowing). This is why member methods of a class are often also called *instance methods*.

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 sometimes 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[4] class like this:

```java
class Tamagotchi {
    int age = 0; // field of the class, but outside of method grow()
    void grow() {
        age++;
    }
}
```

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 outside of the method (but inside the class).

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

8 for loops

At the beginning of this day, we wrote a method containsOnlyLowerCaseLetters(String) that used a **while** loop to determine if a given String had only lower-case letters.

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

The loop visits every character of the String one by one. In this loop we can express in advance *how many times* the code in their body will run (here: as many times as the value of login.length()). Such loops are often written in a slightly more succinct way as **for** loops:

```
    static 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;
    }
```

Let us look at the first line of the loop in detail:

```
        for (int i = 0; i < login.length(); i++)
```

This first line is often called the *loop header*.

- The line begins with the keyword **for**.
- After the opening parenthesis, we see `int i = 0`. This statement is often called the *initialisation statement* of the loop. Here it declares and initialises a variable i that can be used in the loop header and in the loop body, but not after the loop. In other words, the *scope* of i is restricted to the loop header and the loop body. The initialisation statement is executed once, when program execution reaches the loop. A variable like i that is initialised here is sometimes called the "loop variable".
- In the next component of the loop header, we see `i < login.length()`. This is an example of a *loop condition*. Similar to a **while** loop, this condition is checked before each time the loop

body may be run. If the condition evaluates to `true`, the loop body is run another time. If the condition evaluates to `false`, execution of the program continues after the `for` loop.

- In the final component of the loop header before the closing parenthesis, we see `i++`. This is an example of a *loop increment* statement. It is executed right after the loop body has been executed and before the loop condition is checked. Even though this statement is usually called a loop *increment*, we are allowed to use an arbitrary statement here (e.g., `i--` or `i = i * 2`).
- The three components between the parentheses of the loop header of a `for` loop are separated from each other by semi-colons "`;`".

Have another look at the version with the `while` loop that we saw earlier. It has the same ingredients, but they are spread out over the program text in a different way.

Even though a `for` loop may look a bit complicated at first glance, it has the benefit of communicating to other programmers clearly what is happening: the initialisation of the loop variable, the loop condition, and the loop increment are all shown in one place.