Control structures

Some additional concepts

In this document we are going to learn some additional concepts related with the control structures seen in this unit. We'll see how to work with more complex loop structures, and some additional elements that we can add to these loops. Also, we will learn some basic rules to write clean code regarding these control structures.

1. Nested loops

The iterative structures that we have learnt in previous documents can be combined, so that we can place one of them inside another. This complex structure is also called *nested loop*. It can be a *do..while* inside a *for*, or a *for* inside another *for*, or any other combination.

For instance, the following code prints a square filled with asterisks, whose size is determined by variable *size* (we assume that this variable has been previously declared and assigned):

```
for (int i = 1; i <= size; i++) {
    for (int j = 1; j <= size; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

> **Exercise 1**:
>
> Write a program called **Triangle** in which we ask a user to enter a height (integer) and then we write a reversed triangle like the following one (assuming a height of 5):

```
*****
 ****
  ***
   **
    *
```

> **Exercise 2**:

> Create a program called **Counter** that asks the user to write numbers between 1 and 100. The program must keep asking the number until it is a number in the valid range. For each valid number, it must count from this number to 1 in descending order.

2. Using "break" and "continue"

There are some special instructions that can be used inside loops to alter its natural behavior. These instructions are *break* and *continue*.

The **break** instruction exits the loop in which it is placed. For instance, this loop only counts from 1 to 5, because the *break* instruction exits before completing the whole count:

```
for (int i = 1; i <= 10; i++)
{
    System.out.println(i);
    if (i == 5)
        break;
}
```

The **continue** instruction forces a new iteration of the loop, without running the instructions below it. For instance, this loop counts from 1 to 10, but it does not print number 5, because *continue* instruction goes back to the beginning of the loop when *i* is 5, without printing this value at the end of the loop:

```
for (int i = 1; i <= 10; i++) {
    if (i == 5)
        continue;
    System.out.println(i);
}
```

These two instructions are **NOT recommended** in our programs, since they can be easily avoided in order to make our code more understandable. In the first case, if we just want to count from 1 to 5, we should just define a *for* loop from 1 to 5. In the second case, if we don't want to print number 5 in the screen, we can specify this condition in the *if* clause:

```
for (int i = 1; i <= 10; i++) {
    if (i != 5)
        System.out.println(i);
}
```

3. Writing clean code. Spacing.

In 01e previous documents we have seen some basic rules to write clean code, but now that we have learnt some additional concepts regarding control structures, our code can get a little bit more complex, and we need to apply some additional rules to keep it tidy.

Appropriate code formatting and spacing tells the reader that the programmer has paid attention to every single detail of the program. However, when we find a bunch of lines of code incorrectly indented and/or spaced, we may think that the same inattention may be present in other aspects of the code.

3.1. Vertical spacing

Let's see some simple rules to format and space your code vertically:

- As each group of lines represents a task, these groups should be separated from each other with a blank line. In a Java program, for instance, we would have something like this (pay attention to where blank lines are added):

```
import java.util.Scanner;

public class Program {
    public static void main(String[] args) {
        int personAge;
        String personName;
        Scanner sc = new Scanner(System.in);

        System.out.println("Tell me your name:");
        personName = sc.nextLine();

        System.out.println("Tell me your age:");
        personAge = sc.nextInt();

        if (personAge > 18)
            System.out.println("You are an adult, " + personName);
    }
}
```

- Concepts that are tightly related should be placed together vertically. For instance, if we declare two variables to store the name and age of a person, then we should place these declarations one after another, with no separations. This means that we should not add any comment that breaks the union:

```
String personName;
/*
 * This comment should not be written here!
 */
int personAge;
```

- Opening braces are put either at the end of the lines that need them (typical in programming languages such as Java or JavaScript) or at the beginning of the following line, with the same indentation than previous line (typical in programming languages such as C or C#). In this last case, they can act as blank lines of separation between blocks

```
// Java style (opening brace is NOT considered a blank line)
if (condition) {
    ...
}

// C# style (opening brace can be considered a blank line)
public static void Main()
{
    if (condition)
    {
        ...
    }
...
```

Regarding opening braces, you can decide which of these patterns you want to apply, but you must:

- Apply always the same pattern
- Use the same pattern than all the people in your team

3.2. Horizontal formatting

Regarding horizontal spacing or formatting, there are also some simple rules that we can follow.

- A line of code should be short (maybe 80 or 100 characters length, as much). Some IDEs show a vertical line (typically red) that sets the "ideal" limit for the length of each line. If it is going to be longer, we should cut it and divide the code in multiple lines. You can also apply other rules to determine the maximum line width: you should never have to scroll to the right to see your code, and it should be printable with the same appearance in a vertical page.

```
if ((personAge > 18 && personAge <= 65) ||
    (personName.equals("John")) || (personName.equals("Mary"))) {
    ...
}
```

- Horizontal spacing helps us associate things that are related, and disassociate things that are not. For instance, operators should be separated with a whitespace from the elements they are operating:

```
int average = (number1 + number2) / 2;
```

- Do not align the variable names vertically. It was very typical in old programming languages, such as assembly, but it makes no sense in modern programming languages, where there are lots of different data types. If you do this, you might tend to read the variable names without paying attention to their data types:

```
StringBuilder longText;
int           textSize;
String        textToFindAndReplace;
```

- The indentation is important, since it establishes a hierarchy. There are elements that belong to the whole source file, and others that are part of a concrete block. Indentation help us determine the scope of a group of instructions. In this way:
  - Class name is not indented
  - Functions or other elements inside a class are indented one level
  - Implementation of these functions are indented two levels
  - Block implementations inside function code (code of *if* or *while* clauses, for instance) are indented three levels
  - ... etc.

```
public class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello");
        if (...) {
            System.out.println("Inside an if");
        }
    }
}
```