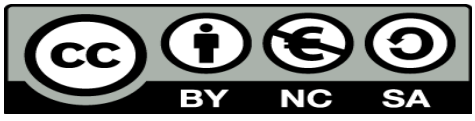
Programming challenges in Java

1. Introduction to Java. Basic sentences and flow control

Nacho Iborra

IES San Vicente



This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/4.0/

Table of Contents

Programming challenges in Java

1. Introduction to Java

Java is an object-oriented programming language created by Sun Microsystems in the middle of the 90's. Its main purpose was to implement programs that were able to run in multiple platforms, such as different operating systems, and even some electrical appliances (refrigeratos, washing machines...).

This document is a quick and short overview of the Java programming language. It is not suitable if you don't know anything about programming, but it may be useful for those of you who have a knowledge of any other programming language (specially those very similar to Java, such as C, C++ or C#), and want to learn how to "switch" to this language.

1.1. Required software

If you want to develop Java applications, you need to install Java JDK (*Java Development Kit*). You can get it from the Oracle Java web site. Besides, you will need an IDE to create your Java projects. Some of the most popular IDEs are Eclipse and NetBeans. You can get them from their respective official web sites (Eclipse and NetBeans)

You can also download and install a bundle with both NetBeans IDE and Java JDK, from this web site.

2. First steps with Java

2.1. Our first Java program

A basic Java program structure to say "Hello world" would be like this (in a file called *MyClass.java*):

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

We always need to define a class, even if we only need a *main* function or method. Every public class must have the same name that the source file that contains it.

2.2. Basic data types

To deal with basic data types (integer or floating point numbers, characters, boolean values and strings) we have the following data types in Java:

- `byte`, `short`, `int`, `long` for integer numbers
- `float` and `double` for real numbers
- `char` for characters
- `boolean` for boolean types, whose values may be *true* or *false*
- `String` for strings

There are no modifiers such as *signed/unsigned*, as in other languages such as C++. The ranges of these data types are more or less the same than in other languages.

2.2.1. Some useful conversions

There are some type conversions that are very usual in Java application. For instance, if we want to convert from String to integer, we can use `Integer.parseInt` method. In the same way, if we want to convert from String to float or double, we can use `Float.parseFloat` o r `Double.parseDouble` methods, respectively. Here's an example of how to use them:

```
int number = Integer.parseInt("42");
```

2.3. Basic operations

The most common basic operations that can be done in Java are:

- Arithmetic operations: +, -, *, /, %(module), ++(increment), --(decrement)
- Assignments: =, +=, -=, *=, /=, %=
- Comparisons: >, >=, <, <=, ==, !=

- NOTE: the comparison == DOES NOT WORK with strings. You must use the *equals* method (`string1.equals(string2)`)

- Logical: && (AND), || (OR), ! (NOT)

2.4. Basic input / output

2.4.1. Basic input: Scanner

In order to get the user input, the easiest way may be through the *Scanner* object. We need to import *java.util.Scanner* class first, and then we create it in the function/method where we want to ask the user to enter data, and then we can call some of its methods. Some of them are *nextLine* and *nextInt*:

```
import java.util.Scanner;
...
public class ClassName
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int number = sc.nextInt();
        String text = sc.nextLine();
        sc.close();
    }
```

There are some other methods, such as *nextFloat, nextBoolean*... but they are very similar to *nextInt*, and they help us read specific data types from the input, instead of reading texts and then converting them into the corresponding type (as *Console.ReadLine* does in C#). You can introduce this data separated by whitespaces or new lines (Intro).

2.4.2. Be careful when combining data types

Let's suppose that you have to read this information from the input:

```
23 43
Hello world
```

You may think that you need to use *nextInt* method twice, and then *nextLine* method to read the last string, but this approach is NOT correct: when you use *nextInt* to read the integer values, you don't read the end of line that exists beyond the number 43, so, when you use *nextLine* method once, you just read this new line, but not the second line. The correct sequence would be this one:

```
int number1 = sc.nextInt();
int number2 = sc.nextInt();
sc.nextLine();                  // Read and ignore end of first line
String text = sc.nextLine();
```

2.4.3. Basic output: System.out.print(ln)

On the other side, you can use the *System.out.print* o r *System.out.println* instruction (depending on whether you want a new line at the end or not). You can join multiple values by using the link operator (+)

```
int result = 12;
System.out.println("The result is " + result);
```

2.5. Constants declaration

We declare constants in Java by declaring the data as *final* (and *static*, if we want to share the data among all the elements). Typically these constants are placed at the beginning of the class. We can set them public, protected or private, depending on which classes are going to access these values.

```
class MyClass
{
    public static final int MAX_USERS = 10;
    ...
}
```

3. Conditions and loops

In this section we are going to see the different selective and iterative structures that you can use in Java. In all of them you can/must use the braces { ... } to group everything that must be contained in the corresponding clause.

3.1. Selective structures

3.1.1. if .. else if .. else

You can use the *if* and *if .. else if .. else if .. else* structures as in many other languages.

```
if (number > 0)
{
    System.out.println("It is positive");
}
else if (number < -10)
{
    System.out.println("It is under -10");
}
else
{
    System.out.println("It is between -10 and 0");
}
```

3.1.2. switch

Besides, there's a *switch* clause, with their corresponding *cases* and a *default* case. The data managed in the *switch* clause must be a primitive type; strings are NOT allowed in early versions of Java (Java 6 and earlier). The *break* at the end of each *case* is not compulsory, but if you don't put it, you go to next *case*.

```
switch(number)
{
    case 0: System.out.println("It is 0"); break;
    case 1: System.out.println("It is 1");
    case 2: System.out.println("It is 2"); break;
    default: System.out.println("Unknown number");
}
```

In previous example, if number is 1, it would output the messages "It is 1" and "It is 2", since there is no *break* clause at *case 1*.

3.2. Loops

Java has the most common iterative structures that (almost) every programming language has: *while, do..while* and *for*. The way you use them is similar to other languages such as C or C#.

3.2.1. while

We will use this loop when we don't know how many iterations are expected, and we don't even know if there will be one iteration. This example counts from 1 to 10:

```
int n = 1;
while (n <= 10)
{
```

```
    System.out.println(n);
    n++;
}
```

3.2.2. do..while

We will use this loop when we don't know how many iterations are expected, but we know that there will be (at least) one iteration. It is very usual when we ask the user to type something and then check the input and ask the user again. If we do the same loop than in previous example with a do..while structure, it would look like this.

```
int n = 1;
do
{
    System.out.println(n);
    n++;
} while (n <= 10);
```

3.2.3. for

We will use this loop when we know how many iterations are expected. The counter from 1 to 10 should be done with this structure preferably, and it would look like this:

```
for (int n = 1; n <= 10; n++)
{
    System.out.println(n);
}
```

Note that we can declare variables in *for* loops (and in the middle of other code, as in other languages such as C#).

3.2.4. Another "for"

There is another way of using the *for* clause, applied to collections or arrays. It consists in using a variable with the same type, this way:

```
for (int number: numbers)
    System.out.println("" + number);
```

where *numbers* is expected to be a collection or array of integers.

This structure is equivalent to the *foreach* structure of other languages such as C#, and is expected to be used in a read-only way (only to check the values, but not to modify them)

4. Challenges for this session

4.1. Sample challenge: Hello world

Let's have a look at this challenge from *Acepta el reto*. It asks you to read a number N and print N times the string "Hola mundo.". To solve this challenge, we could implement something like this in Java:

```
import java.util.Scanner;

public class Challenge116_Hola_mundo
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int times = sc.nextInt();

        for (int i = 0; i < times; i++)
            System.out.println("Hola mundo.");
    }
}
```

Try to upload this code to *Acepta el reto* and see how it works fine:

4.2. Try yourself: Boredom at table talk

Let's go on with another challenge. Read it here and try to solve it in both languages.

4.3. Try yourself: Last factorial digit

Here you can read this challenge. As usual, try to solve it in both languages.