Basic elements of a Java program

Writing clean code

1. Introduction to clean code

When we are writing a program, we should not only think about what the program needs to do. We should also ask some other questions to ourselves, such as:

- What if I have to take this project back in two years? Will I understand the code?
- What if anyone else has to take this project in the future? Will he/she understand the code?

After these questions, you should think of a way to write your code so that it will be easy to read and understand. This is where clean code rules come into play.

1.1. What is clean code?

You can find many examples and good explanations to this question in the book *Clean Code*, by Robert C. Martin. Here we just summarize some of these ideas:

- Code must be elegant and easy to read, simple and direct. *Clean code reads like well-written prose* (Grady Booch).
- Logic should be straightforward to make it hard for bugs to hide.
- Performance should be close to optimal so as not to tempt people to make further changes.
- Keep in mind the Boy Scout rule: *Leave the campground cleaner than you found it*

1.2. The importance of practice

Writing clean code not only consists in reading documents like this one to keep in mind some rules. It also consists in putting into practice these rules continuously. For instance, you can read how to ride a bike, but you won't learn how to do it until you practice.

Besides, if we don't start writing clean code from the beginning of a project, there may be some terrible consequences later: projects can grow too much, and then it may be hard to apply clean code rules to the whole code: the time we spend fixing the code in the future may affect deadlines, maintenance, future versions...

1.3. Why bad code exists?

Although everyone should apply clean code rules in their programs, and we can easily see the benefits of working this way, there are many reasons why bad code exists:

- Too tight schedules
- Unexperienced project managers
- Programmer's docility (he/she doesn't want to get fired)
- Boredom (always doing the same kind of projects)
- ...

1.4. What's coming next?

In this document we are going to focus on some basic aspects of clean code rules, such as how to assign variable names and how to place comments in our code.

1.5. Code conventions

Different programming languages use different coding conventions. We should follow the conventions of the language we are using in order to produce consistent code.

Related to java code conventions, the following are recommended:

- Java Code conventions: https://www.oracle.com/technetwork/java/codeconventions-150003.pdf
- Google Java Code conventions https://google.github.io/styleguide/javaguide.html

2. Dealing with variable names

Names are essential in programming, since we will assign a name to (almost) everything we include in our program. At this point you should already know what a variable is and its main purpose (store values that can be modified along the program execution). But you should not assign a variable name carelessly. You should use **meaningful names** for your variables.

When reading the name of a variable (or any other element in the code), it must answer some basic questions, such as why it exists, what it does and how it is used. If a name requires a comment to explain its meaning, then it is not a suitable name. For instance, if we want to store in a variable the age average of a list of people, we should NOT do this:

```
int a;          // Age average
```

We could do this instead:

```
int ageAverage;
```

Some other aspects that we should take into account when dealing with variable names:

- Try not to use too similar names. Variables like `totalRegisteredUsers` and `totalUnregisteredUsers` only differ in two letters, and you could use the wrong one in a given

piece of code. It's better to call them `registered` and `anonymous`.

- Add **meaningful context** when it's necessary. For instance, if a variable is named `account`, what does it mean? A user account? a bank account? It's better to be more specific, and call it `bankAccount`, for instance
- Choose **one word per concept**: if you declare many variables in many parts of your code to refer to a user login, you should always call them in the same way: `user`, or `login`, for instance, but don't change the name in each situation.
- Don't use short names, such as `n`, or `e`, because it will be difficult to find your variable among other similar words in the text.
- Try to use readable names. It's better to use a name like `birthDate` than `ddmmyyyy`, because you can pronounce this name in a conversation.

2.1. Uppercase or lowercase?

The use of uppercase and lowercase letters in names depend on the programming language itself. There are mainly four naming standards:

- **Camel Case**: it is used in languages such as Java or Javascript. Every word in the variable name starts with upper case, apart from the first word. For instance:

```
String personName;
```

- There is a subset of camel case standard, called **Pascal Case** in which the first word of the name also starts with uppercase. This subset is employed by C# to define public elements (private elements are named using camel case). For instance:

```
string personName;
public int PersonAge;
```

- **Snake Case**: it is used in languages such as PHP. Variable words are separated by underscores:

```
$person_name = "Nacho";
```

- **Kebab Case**: variable words are separated by hyphens. It is not very popular among programming languages, since many of them don't allow the hyphen as part of the variable name (so as not to mix it up with the substraction operator). There are some few examples, such as Lisp or Clojure.

```
(def person-name "Nacho")
```

- **Upper case**: it is used in many languages to define constants. The words of the name are usually separated by underscores, as in snake case standard:

```
static final int MAXIMUM_SIZE = 100;
```

3. Placing comments

Well-placed comments help us understand the code around them, whereas misplaced comments can damage the understanding of the code. Some programmers think that comments are failures, and should be avoided as much as possible. One of the reasons argued is that they are hard to maintain. If we change the code after writing a comment, we may forget to update the comment, and thus it would talk about something that is no longer present in the code.

Another reason to avoid comments is that they are tightly linked to bad code. When we write bad code, we often think that we can write some comments to make it understandable, instead of cleaning the code itself.

In this section we will learn where to put comments. Firstly, we will see what type of comments are necessary (what we call *good comments*), and then we will see what comments are avoidable (*bad comments*).

3.1. Good comments

The following comments are considered necessary:

- **Legal comments**, such as copyright or authorship, according to the company standards. This type of comments are normally placed at the beginning of each source file that belongs to the author or company.
- **Introduction comments**, a short comment at the beginning of each source file (typically classes) that explains the main purpose of this source file or class. This comment is usually placed along with a legal comment at the beginning of a source file:

```
/*
 This class stores information about a user account

 Created by Nacho Iborra
 */

public class User
{
    ...
}
```

- **Explanation of intent**. These comments are used when:

    - We tried to get a better solution to the problem but we could not, and then we explain that a part of the code could be improvable.
    - There is a part of the code that does not follow the same pattern than the code around it (for instance, an integer variable among a bunch of floats), and we want to explain why we have used this instruction or data type.

- **TODO comments**, which are placed in incomplete parts. They help us remember all the pending tasks. This type of comments have become so popular that a lot of IDEs automatically detect and highlight them.
- **API documentation**. Some programming languages, such as Java or C#, let us add some comments in some parts of the code so that these comments are exported to HTML or XML format, and become part of the documentation.

3.2. Bad comments

The following are examples of bad comments that we can avoid...

Some type of **information comments** can be avoided by changing the name of the element that they are explaining. For instance, if we have this comment with this variable:

```
// Total number of customers
int total;
```

We can avoid the comment by renaming the variable this way:

```
int totalCustomers;
```

Redundant comments, i.e. comments that are longer to read than the code they are trying to explain, or they are just unnecessary, because the code is self-explanatory. For instance, the following comment is redundant, since the code it is explaining is quite understandable:

```
/* We ask the user two numbers and add them */
Scanner sc = new Scanner(System.in);
System.out.println("Enter two numbers");
int number1 = sc.nextInt();
int number2 = sc.nextInt();
System.out.println(number1 + number2);
```

Comments without context, i.e. comments that are not followed by the corresponding code. For instance, the following comment is not completed with appropriate code. We say we are writing data into a file, but nothing is executed after that. Maybe there was some piece of code, but it was removed.

```
/* We ask the user two numbers and add them */
Scanner sc = new Scanner(System.in);
System.out.println("Enter two numbers");
int number1 = sc.nextInt();
int number2 = sc.nextInt();
System.out.println(number1 + number2);
// We print the result in a text file
```

There should be no **mandated comments**. Some people think that every variable, for instance, must have a comment explaining its purpose. But that is not a good decision, since we can avoid most of these comments by using appropriate variable names.

Also, there should be no **journal comments**: sometimes an edit registry is placed at the beginning of a source file. It contains all the changes made to the code, including the date and the reason of the change. But nowadays, we can use version control applications, such as GitHub, to keep this registry out of the code itself.

Some time ago, some programmers used to place some **position markers and/or code dividers**, to quickly find a place in the code, or to separate some code blocks that are quite long. Both types of comments are not recommended if code is properly formatted.

```
// ================== VARIABLES ====================
int age;
String name;
...
// ================== MAIN =========================
public static void main(String[] args)
{
    ...
    ////// FINAL RESULT
}
```

Closing brace comments are also not recommended. They are placed at every closing brace to explain which element is this brace closing. These comments can be avoided, since most of current IDEs highlight each pair of braces when we click on them, so that we can match each pair automatically.

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

Warnings are used when we have some code that may cause problems in certain situations, because it needs to be reviewed. It is very usual to find some code blocks completely commented, and a warning message explaining the problem with it. These comments should be turned into "TODO" comments, in order to warn the programmer that this code needs to be reviewed in the future, instead of just removing the comments.

> **Exercise 1:**
>
> This program asks the user to introduce three numbers and gets the average of them. Discuss in class which parts of the code are not clean or could be improved, regarding variable names and comments.

```
import java.util.Scanner;

public class AverageNumbers
{
    public static void main(String[] args)
    {
        // Variables to store the three numbers and the average
        int n1, n2, n3;
        int Result;
        Scanner sc = new Scanner(System.in);

        // We ask the user to enter three numbers
        System.out.println("Introduce three numbers:");
        n1 = sc.nextInt();
        n2 = sc.nextInt();
        n3 = sc.nextInt();
        // The result is the average of these numbers
        /* We could have used a float number instead,
            but we decided to keep this program as
            simple as we could */
        Result = (n1+n2+n3)/3;
        System.out.println("The average is " + Result);
    }
}
```