Writing Clean Code

Part I

Variables and comments

Nacho Iborra

IES San Vicente

Index of contents

1.Variable names....3
1.1.Names must be meaningful....3
1.2.Other desirable features....4
2.Comments....6
2.1.Good comments....6
2.2.Bad comments....6
3.Exercises....9
3.1.Exercise 1....9
3.2.Exercise 2....9
3.3.Exercise 3....10

1. Variable names

In the first units of this module you have learnt what a variable is and its main purpose (store values that can be modified along the program execution). In the main contents of the module you have been told to use meaningful names for your variables. In this section we will see this feature and some other rules that variable names should follow.

Names are essential in programming, since we will assign a name to (almost) everything we include in our program. In these first units we will assign names to variables, but later we will assign them to functions, parameters, classes, files, namespaces and so on.

1.1. Names must be meaningful

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, 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;
```

1.1.1. Exception to the rule: loops

If you are coding a loop, you will probably need an integer variable to store the value with the number of iterations performed. This variable can either have a meaningful name, if you can (or want to) use it later...

```
int count = 0;
while (count < 10)
{
   …
```

or just a short and typical name (for instance, i or n) to use it ONLY in the loop count:

```
for (int i = 0; i < 10; i++)
{
   …
```

1.1.2. Avoid misunderstandings

Besides, we must avoid names that might be "false friends", this is, they seem to mean something, whereas they are intended to mean something completely different. For instance, if we call a variable *account*, what is it used for? A bank account? A user account in a web site?

1.1.3. Avoid small variations

Two different variables must have two clearly different names. If we have a variable called *totalRegisteredCustomers*, it is not a good idea to have another one called *totalUnregisteredCustomers*, since we could mix them up when we read their names, and use the wrong one. Instead of this, we can use *registered* and *anonymous*, for instance.

1.1.4. Avoid noisy words

Noisy words are words that are part of the variable name but do not provide any additional information to this name. For instance, if a variable is called *nameString*, the word *String* is meaningless, since we should deduce that name is stored in a *string* variable. In the same way, a variable called *money* provides the same information than another called *moneyAmount*.

1.1.5. Add meaningful context

However, there are some words that are not noisy, but they help us set a variable name in an appropriate context. For instance, if we are talking about the data needed to store the address of someone (first name, last name, street, city, zipcode...), but we only see the variable *city* in the code, will we be able to deduce that this variable is storing part of the address? To be sure that we deduce this, we can add some information to the variable name, such as a preffix: *addressCity* is more likely to be associated with an address than just *city*.

1.1.6. Choose clarity over entertainment

Do not try to set variable names that are part of a joke that only some people would understand. Try to use names that everyone knows and understands.

1.1.7. Choose one word per concept

Try to use always the same name to express the same concept. For instance, if you implement several applications for several customers, and you use a variable to store the login of the people logging in, do not call this variable *user* in one application, *login* in another application, and so on. Use always the same word (*user* or *login*, in this case).

In the same way, do not use the same word to talk about different concepts. For instance do not use the word *sum* for all the final calculations, unless they are additions indeed. It is better to use *total*, or *result* in this case.

1.2. Other desirable features

Besides meaningfulness, names should follow some other rules, such as pronounceability or searchability.

1.2.1. Names must be pronounceable

You should name your variables in a way that allows you to pronounce this name to other people, so as to discuss about its value or code errors regarding this variable. If you are storing the birth date of someone in a variable, you could call it *birthDate*, but you should not call it *ddmmyyy*, for instance, since it would be difficult for you to pronounce this name in a discussion.

1.2.2. Names must be searchable

If you use short names in your variables (single letter names, for instance), it will be difficult for you to find each occurrence of this variable in your code, because it will be merged into other elements that will contain this short name as part of their names. For instance, if you call your variable *i,* then you fill find a lot of "ifs", "whiles" and other words while trying to find where you use this variable, unless your search utility allows you to search for the whole word.

1.2.3. Avoid prefixes or any additional encodings

Some years ago, it was very usual to find some kind of prefixes or suffixes in the variable names that revealed some information about the variable. For instance, we could call a variable *iAge* where prefix *i* showed that this variable was an integer. This habit was forced by some old programming languages where the data type was declared as a prefix. But nowadays there are lots of new programming languages that do not need this rule, so forget it.

1.2.4. Be careful with uppercase and lowercase

The use of uppercase and lowercase letters in names depend on the programming language itself. For instance, in Java every variable name starts with lowerCase, and every new word inside the name starts with an uppercase letter:

```
int personAge;
string personName;
```

However, in C# we use this rule only for variables that are not public (indeed, no variable should be public):

```
int personAge;
public string PersonName;     // This should be avoided
```

2. 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*).

2.1. 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.
- **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.
- **Warnings**, which 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.
- ***TODO* comments**, which are placed in uncompleted 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#, lets 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.

2.2. 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 registered
  int total;
  ```

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

  ```
  int totalRegisteredCustomers;
  ```

- **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 check if the age is greater than 18, and if so, we print a message
  // saying that "You are old enough"
  if (age > 18)
      Console.WriteLine("You are old enough");
  ```

- **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:

  ```
  if (age > 18)
  {
      Console.WriteLine("You are old enough");
  } else {
      // If user is not adult, he is logged out
  }
  ```

  There is some code missing. We say in the comment that, if user is not adult, it will be logged out of the application, but there is no code to log out the user below the comment. Maybe this log out is performed in other part of the code, but then this comment should be placed there.

- **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.

- **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.

- **Position markers and code dividers**. It is very usual to make comments to rapidly 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 (see next section).

  ```
  // =================== VARIABLES ====================
  int age;
  string name;
  …
  ```

```
// =================== MAIN =========================
public static void Main()
{
    …
    ////// FINAL RESULT
}
```

- **Closing brace comments**, which are placed at every closing brace to explain which element is this brace closing. For instance:

  ```
  while (n > 10)
  {
      if (n > 5)
      {
          …
      } // if
  } // while
  ```

  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.
- Avoid too much information in comments. Just explain the necessary to understand the code associated to the comment

3. Exercises

3.1. 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.

```
using System;

public class AverageNumbers
{
      public static void Main()
      {
            // Variables to store the three numbers and the average
            int n1, n2, n3;
            int Result;

            // We ask the user to enter three numbers
            Console.WriteLine("Introduce three numbers:");
            n1 = Convert.ToInt32(Console.ReadLine());
            n2 = Convert.ToInt32(Console.ReadLine());
            n3 = Convert.ToInt32(Console.ReadLine());
            // 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;
            Console.WriteLine("The average is {0}", Result);
      }
}
```

3.2. Exercise 2

This program asks the user to enter numbers until it enters a negative number, or a total of 10 numbers. Copy it into your IDE, and try to improve it with appropriate variable names and comments.

```
/*
 * (C) IES San Vicente 2016
 */
using System;

public class EnterNumbers
```

```
{
      public static void Main()
      {
            // Numbers entered by the user
            int number1;
            // Number count
            int number2 = 0;

            do
            {
                  // We ask the user to type a number
                  Console.WriteLine("Type a number: ");
                  number1 = Convert.ToInt32(Console.ReadLine());
                  number2++;
            } while (number2 < 10 && number1 >= 0); // do..while

            if (number1 < 0)
                  Console.WriteLine("Finishing. You entered a negative number");
            else
                  Console.WriteLine("Finishing. You entered 10 numbers");
      }
}
```

3.3. Exercise 3

Create a program called *RectangleDraw* that asks the user to introduce a rectangle base and height, and then prints a rectangle of the given dimensions. For instance, if the user sets a base of 5 and a height of 3, the program should print this rectangle (full of '*'):

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

Implement this program according to the rules explained in this document, regarding variable names and comments.