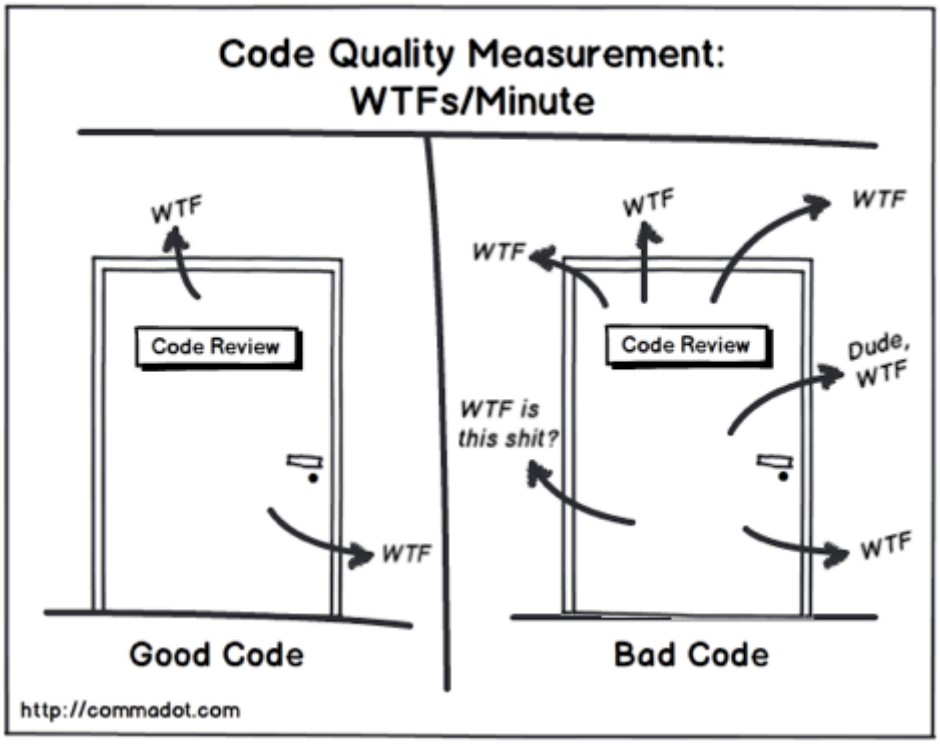
by Nacho Iborra

Development Environments
Block 1
Unit 9: Programming best practices

1.9.1. Introduction to clean code

As an introduction of what we want you to "master" as good programming guidelines, let's read some extracts from the book *Clean Code*, by Robert C. Martin. To begin with, here you can see a graphical representation of how to measure code quality:



1.9.1.1. The importance of practice

There are two parts to learning craftsmanship: knowledge and work. You must gain the knowledge of principles, patterns, practices, and heuristics that a craftsman knows, and you must also grind that knowledge into your fingers, eyes, and gut by working hard and practicing.

I can teach you the physics of riding a bicycle. Indeed, the classical mathematics is relatively straightforward. Gravity, friction, angular momentum, center of mass, and so forth, can be demonstrated with less than a page full of equations. Given those formulae I could prove to you that bicycle riding is practical and give you all the knowledge you needed to make it work. And you'd still fall down the first time you climbed on that bike.

Coding is no different. [...]

Learning to write clean code is hard work. It requires more than just the knowledge of principles and patterns. You must sweat over it. You must practice it yourself, and watch yourself fail. You must watch others practice it and fail. You must see them stumble and retrace their steps. You must see them agonize over decisions and see the price they pay for making those decisions the wrong way.

1.9.1.2. Effects of bad code

I know of one company that, in the late 80s, wrote a killer app. It was very popular, and lots of professionals bought and used it. But then the release cycles began to stretch. Bugs were not repaired from one release to the next. Load times grew and crashes increased. I remember the day I shut the product down in frustration and never used it again. The company went out of business a short time after that.

Two decades later I met one of the early employees of that company and asked him what had happened. The answer confirmed my fears. They had rushed the product to market and had made a huge mess in the code. As they added more and more features, the code got worse and worse until they simply could not manage it any longer. It was the bad code that brought the company down.

1.9.1.3. The impossibility of a grand redesign

[...] *Eventually the team rebels. They inform management that they cannot continue to develop in this odious code base. They demand a redesign. Management does not want to expend the resources on a whole new redesign of the project, but they cannot deny that productivity is terrible. Eventually they bend to the demands of the developers and authorize the grand redesign in the sky.*

A new tiger team is selected. Everyone wants to be on this team because it's a green-field project. They get to start over and create something truly beautiful. But only the best and brightest are chosen for the tiger team. Everyone else must continue to maintain the current system.

Now the two teams are in a race. The tiger team must build a new system that does everything that the old system does. Not only that, they have to keep up with the changes that are continuously being made to the old system. Management will not replace the old system until the new system can do everything that the old system does.

This race can go on for a very long time. I've seen it take 10 years. And by the time it's done, the original members of the tiger team are long gone, and the current members are demanding that the new system be redesigned because it's such a mess.

1.9.1.4. Some reasons why bad code exists

Have you ever waded through a mess so grave that it took weeks to do what should have taken hours? Have you seen what should have been a one-line change, made instead in hundreds of different modules? These symptoms are all too common.

Why does this happen to code? Why does good code rot so quickly into bad code? We have lots of explanations for it. We complain that the requirements changed in ways that thwart the original design. We bemoan the schedules that were too tight to do things right. We blather about stupid managers and intolerant customers and

useless marketing types and telephone sanitizers. But the fault, dear Dilbert, is not in our stars, but in ourselves. We are unprofessional.

This may be a bitter pill to swallow. How could this mess be our fault? What about the requirements? What about the schedule? What about the stupid managers and the useless marketing types? Don't they bear some of the blame?

No. The managers and marketers look to us for the information they need to make promises and commitments; and even when they don't look to us, we should not be shy about telling them what we think. The users look to us to validate the way the requirements will fit into the system. The project managers look to us to help work out the schedule. We are deeply complicit in the planning of the project and share a great deal of the responsibility for any failures; especially if those failures have to do with bad code! "But wait!" you say. "If I don't do what my manager says, I'll be fired." Probably not. Most managers want the truth, even when they don't act like it. Most managers want good code, even when they are obsessing about the schedule. They may defend the schedule and requirements with passion; but that's their job. It's your job to defend the code with equal passion.

To drive this point home, what if you were a doctor and had a patient who demanded that you stop all the silly hand-washing in preparation for surgery because it was taking too much time? Clearly the patient is the boss; and yet the doctor should absolutely refuse to comply. Why? Because the doctor knows more than the patient about the risks of disease and infection. It would be unprofessional (never mind criminal) for the doctor to comply with the patient.

So too it is unprofessional for programmers to bend to the will of managers who don't understand the risks of making messes.

> **Proposed exercises**:

> **1.9.1.1.** Do you agree with these reasons? Do you think there can be any other reasons why bad code exists?

1.9.1.5. Some definitions of "good code"

Bjarne Stroustrup, inventor of C++ and author of *The C++ Programming Language*

I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.

Grady Booch, author of *Object Oriented Analysis and Design with Applications*

Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.

Dave Thomas, founder of OTI, godfather of the *Eclipse* strategy

Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.

Michael Feathers, author of *Working Effectively with Legacy Code*

I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them. Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better. All of those things were thought about by the code's author, and if you try to imagine improvements, you're led back to where you are, sitting in appreciation of the code someone left for you—code left by someone who cares deeply about the craft.

Ward Cunningham, inventor of Wiki, inventor of Fit, coinventor of eXtreme Programming. Motive force behind Design Patterns. Smalltalk and OO thought leader. The godfather of all those who care about code

You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem.

And, to finish with this part, when talking about clean code, you must keep in mind the Boy Scout rule: *Leave the campground cleaner than you found it*

> **Proposed exercises**:

> **1.9.1.2.** Which definition do you like most, or think is the most suitable? Why?

> **1.9.1.3.** How can the Boy Scout rule be applied to code typing?

1.9.2. Dealing with variable names

Once you learn what a variable is and its main purpose (store values that can be modified along the program execution), you should 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 the first units we will assign names to variables, but later we will assign them to functions, parameters, classes, files, namespaces and so on.

1.9.2.1. 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;
```

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++)
{
    ...
```

1.9.2.2. Avoid misunderstandings, small variations and noisy words

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?
- Small variations in variable names. 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.
- Noisy words, i.e., 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`.

1.9.2.3. Add meaningful context

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`.

1.9.2.4. 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.

1.9.2.5. Other desirable features

Besides all the recommendations explained before, names should follow some other rules, such as:

- 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 `ddmmyyyy`, for instance, since it would be difficult for you to pronounce this name in a discussion.
- 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.
- 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, and lots of IDEs that help us find out the type of each variable easily, so don't use these prefixes.

1.9.2.6. 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:

```
const int MAXIMUM_SIZE = 100;
```

1.9.3. 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*).

1.9.3.1. 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.
- **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 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.

1.9.3.2. 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 registered
int total;
```

We can avoid 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. 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.

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

- **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.

```
// ================== 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. 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.

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

- **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. 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.

> **Proposed exercises**:

> **1.9.3.1.** 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);
    }
}
```

1.9.3.2. 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. Negative number");
        else
            Console.WriteLine("Finishing. 10 numbers");
    }
}
```

1.9.3.3. 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.

1.9.4. Code spacing and formatting

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.

1.9.4.1. 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 C# program, for instance, we would have something like this (pay attention to where blank lines are added):

```
using System;

public class Program
{
    public static void Main()
    {
        int personAge;
        string personName;

        Console.WriteLine("Tell me your name:");
        personName = Console.ReadLine();

        Console.WriteLine("Tell me your age:");
        personAge = Convert.ToInt32(Console.ReadLine());

        if (personAge > 18)
            Console.WriteLine("You are an adult, {0}",
                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

1.9.4.2. 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 == "John") || (personName == "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()
    {
        Console.WriteLine("Hello");
        if (…)
        {
            Console.WriteLine("Inside an if");
        }
    }
}
```

1.9.5. Functions

Functions are a way to organize the code in our programs. They help us split the code into different modules, and call/use each module whenever we need it. However, if they are incorrectly written, they can be part of the problem instead of being part of the solution. In this section we will learn how functions must be organized and some basic rules to write good code with them.

1.9.5.1. The size matters

The (maybe) most important rule that a function must accomplish is that it must be small. But the adjective *small* is quite fuzzy... how many lines are considered small? Well, some years ago the experts said that a function should not be longer than a page. But the font types then only allowed some lines per page, and

now we could write up to 100 lines in a single page. Does this mean that we could have functions of 100 lines? The answer is NO... a function should contain about 20 lines of code as much.

To do this, we should divide our code in functions, and make calls between them properly. This would cause that the indentation level of a function should not be greater than one or two (control structures such as `if`, `while` or similar, and a function call inside them)

1.9.5.2. Single tasks

Which is the best way to divide our code so that each function is no longer than 20 lines? We must divide the program in tasks, and this tasks in subtasks, if needed, and so on. Each function should implement one task of one level.

For instance, consider the following program: we validate a user log in, and then, we print his profile on the screen. The main program would be composed of two high level tasks:

1. Validate user
2. Print profile

So we should define a function for each one of these tasks. Then, we go on dividing each function in subtasks. For instance, for the first function:

1. Validate user
    1. Print "User login:" on the screen
    2. Get user login from keyboard
    3. Print "User password:" on the screen
    4. Get user password from keyboard
    5. Check user in the database
    6. If correct, finish this function
    7. If not correct, go on to step 1.1

Almost all of these subtasks are simpler enough to do them in just one or two lines of code, so we do not need more levels.. apart from subtask 1.5, which would need a new level:

5. Check user in the database
    1. Connect to database
    2. Look for the login and password introduced
    3. Set if the user exists or not

If we translate this schema into code, we would get something like this:

```
public static void Main()
{
    ValidateUser();
    PrintUserProfile();
}

public static void ValidateUser()
{
    bool result = false;
    do
    {
        Console.WriteLine("User login:");
        string login = Console.ReadLine();
        Console.WriteLine("User password:");
        string password = Console.ReadLine();
        result = CheckUserInDatabase(login, password);
    } while (!result);
}

public static void CheckUserInDatabase(string login, string password)
{
    ...
}

...
```

1.9.5.3. Function names

In the same way that variables must have appropriate names, functions must also have names that explain clearly what they do. Try to apply the same pattern to similar functions, and try to use a verb in the name, since functions DO something. For instance, if we have a bunch of functions to validate some data entered by the user, we could call these functions `validateNumber`, `validatePostalCode`, `validateTelephoneNumber`... In all these functions we use a verb (*validate*), and the same pattern (*validateAAA*, *validateBBB*...). It would be a mistake to call a function `validateNumber`, and use `checkPostalCode` for the second one, for instance.

Regarding the uppercase and lowercase use, functions follow the same rules than variables explained in previous sections. In Java, for instance, every function (apart from the constructors) start with lowercase, and every new word inside the name starts with an uppercase letter (*camel case*):

```
public void myFunction() { … }
```

In C#, this rule applies only to non-public functions. If a function is public, it starts with an uppercase letter (*pascal case*):

```
public void MyFunction() { … }
int anotherFunction() { … }
```

1.9.5.4. Functions and spacing

Regarding functions, we must pay attention to these rules explained before when talking about how to format and space our code. To begin with, there should be a blank line between each pair of functions:

```
public void Function1()
{
    ...
}

public int Function2()
{
    ...
}
```

Functions that are closely related should be placed closed to each other, so that we do not need to explore the whole source file to find one of them from the other. This affinity can be derived from either dependence between functions or similar functionality.

- Regarding dependence, if one function A calls another one B, then A should be placed before B.
- Regarding functionality, consider a bunch of functions with the same prefix or suffix (`addNames` , `addCustomers` ...). These functions should be close between them.

1.9.5.5. Other features

There are some other features that are desirable in functions.

- **Avoid side effects**: the code of a function should not modify anything outside this function directly, nor do anything not implicitly declared in its name. For instance, if we have a function called `validateUser` , it should not store anything in a file.

- **Follow the structured programming rule**: according to structured programming, every code block (for instance, a function) should have only one entry and one exit. This means that every function should have just one `return` statement, and every loop one finish point. So we should avoid `break` or `continue` instructions inside loops.

> **Proposed exercises**:

> **1.9.5.1.** The following example asks the user to tell how many names he is going to type. Then, it initializes an array of the specified size, and stores the names in it. Finally, it sorts the names

alphabetically and prints them. Take a look at the code and try to improve it in terms of code cleanliness.

```
using System;

public class SortNames
{
    static void PrintNames(string[] names)
    {
        Console.WriteLine("List in alphabetical order:");
        foreach(string name in names)
        {
            Console.WriteLine(name);
        }
    }
    public static void Main()
    {
        int totalNames;
        string[] names;
        string aux;
        Console.WriteLine("Enter the total number of names:");
        totalNames = Convert.ToInt32(Console.ReadLine());
        names = new string[totalNames];
        Console.WriteLine("Enter the {0} names:", totalNames);
        for(int i = 0; i < totalNames; i++)
        {
            names[i] = Console.ReadLine();
        }
        for (int i = 0; i < totalNames - 1; i++)
            for (int j = i+1; j < totalNames; j++)
                if (names[i].CompareTo(names[j]) > 0)
                {
                    aux = names[i];
                    names[i] = names[j];
                    names[j] = aux;
                }
        PrintNames(names);
    }
}
```

1.9.5.2. The following code shows some functions that return some values, and a *Main* program that uses them. Try to improve the code in terms of cleanliness.

```
using System;

// Returns if a number is even or odd
public static boolean isEven(int number)
{
    if (number % 2 == 0)
        return true;
    else
        return false;
}

// Looks for a word in a string array and returns the
// position in which it is found.
// If the word is not found, it returns -1
public static int search(string[] array, string word)
{
    for (int i = 0; i < array.Length; i++)
    {
        if (array[i] == word)
            return i;
    }
    return -1;
}

public static void Main()
{
    Console.WriteLine("Enter a number");
    int number = Convert.ToInt32(Console.ReadLine());
    bool even = isEven(number);
    if (even)
        Console.WriteLine("The number is even");
    else
        Console.WriteLine("The number is odd");
    string[] words = {"hello", "goodbye", "one", "two", "three", "four", "red", "yel
    Console.WriteLine("Enter a word");
    string word = Console.ReadLine();
    int result = search(words, word);
    if (result >= 0)
        Console.WriteLine("Word found at " + result);
    else
        Console.WriteLine("Word not found");
}
```

1.9.6. Other elements to consider

To finish with this unit, let's have a quick overview about some other elements that need to be carefully written, such as classes and error handling.

1.9.6.1. Using classes

Although you may not know what a class is by now, you may need to know some useful concepts in order to write them properly:

- **Standard structure of a class**: the standards say that a class must start with a list of instance variables or attributes, and then the constructors and methods. Regarding the variables list, we normally put the constants before, and then the variables. For instance, this could be the code of a class called *Person* that manages the name and age of people:

```
class Person
{
    string name;
    int age;

    public Person(string aName, int anAge)
    {
        name = aName;
        age = anAge;
    }

    public void SayHello()
    {
        Console.WriteLine("Hello {0}, you are {1} years old",
            name, age);
    }
}
```

- **Encapsulation**: it is also recommended to keep our attributes private (or protected, if we are going to inherit from that class). Classes and objects hide their data behind abstractions and expose functions that operate on that data. This is called *encapsulation*.
- **Cohesion and small classes**: the more instance variables a method manipulates, the more cohesive that method is to its class. A class where every variable or attribute is manipulated by every method is maximally cohesive. Our aim is to have a high cohesion, since that would mean that methods and variables are co-dependent and act as a logical whole. Besides, if we try to maintain cohesion (this is, having attributes used by (almost) every method), we will probably get small classes, since it is difficult to have a class with a large amount of variables or methods that are co-dependent.
- **Class naming**: a class is usually a noun in the requirements specification, so a class name should be typically a noun. To be more precise, it should be a singular noun, such as Person, Shape, Animal, StringBuffer... Besides, the class name should give enough information about the aim of this class, and what responsibilities it has. For instance, a class called String may make us think that it will handle all the

methods needed to deal with texts, such as taking substrings, looking for a partial text, replacing text and so on.

1.9.6.2. Error handling

Error handling is important in our programs because user may make mistakes when introducing data, or some devices may fail when, for instance, we try to read or write data from/to a file. However, if we do not use error handling properly, it may damage our code and hide its logical structure. In this section we will see some techniques that help us write code that is clean and robust.

1.9.6.2.1. Exceptions and error codes

In earlier programming languages exceptions did not exist, and the only ways to detect an error were either creating a boolean variable to store if some operation was correct, or returning an integer representing an error code (depending on this value, we could figure out the error itself).

However, whenever a programming language allows exception handling, it is better to use exceptions rather than error codes. Whenever a code block can cause an error, you should try to catch it (or throw an exception to another function). But there are some aspects that you should take into account when dealing with exceptions:

- **Provide context with exceptions**: when we get an error message from a program running, it is important that this message is understandable, and we can figure out what went wrong. So, whenever we throw an exception, we should provide information about the source of the error and the location. For instance, we can mention the operation that failed, and the type of failure: "Error reading from file. File 'data.txt' not found in /home/user".

- **Write *try..catch* blocks first**: it may be a good idea to write your `try..catch` block before starting with the lines of code that will be in it. The try block defines a scope, as `if` or `while` do, and when we add it, we must be aware that the execution can be aborted at any point, and we must leave the program in a consistent state in the catch block. Besides, starting a function that may cause an exception with a try..catch block tells the user of that code that an exception can be expected, no matter which or when.

- **Differentiate the exception types if required**: in many programs, it may be useful to show different error messages for different exceptions. For instance, an error message if a file could not be found, and another different message if we could not read a given element from it.

1.9.6.2.2. Returning null

There are a lot of methods in many official APIs that return `null` when something goes wrong. For instance, there are methods that, when we are reading data from a file, return `null` when there is no more information left to read. Returning `null` can be a source of new errors, since it forces the programmer to check if the return value of the function is null:

```
String value = myObject.methodThatCanReturnNull();

if (value != null)
{
    // Rest of the code that matters
}
```

Note that, if we do this, we are indenting the code one extra level to the right, telling that this is not the first level of code inside the function... but it is. Besides, there are some other lines that are not checked. What if myObject is null? Indeed, we should not check the `null` value in any part of our code.

So, what to do instead of returning null? We have some options:

- Throw an exception with an appropriate message
- Return a special value that represents an error. For instance, a negative index when a word is not found in an array of strings.

In the same way, it is not a good idea to pass a null value as an argument to any method or function, because the method could throw a `NullPointerException` unintentionally.

> **Proposed exercises**:

> **1.9.6.1.** The following exercise asks the user to introduce how many ages he is going to type, and then it calculates the average of these ages. Check the code in terms of cleanliness and robustness.

```
using System;

public class AgeAverage
{
    public static void Main()
    {
        int numberOfAges;
        int[] ages;

        Console.WriteLine("Enter the number of ages:");
        numberOfAges = Convert.ToInt32(Console.ReadLine());
        ages = new int[numberOfAges];

        for (int i = 0; i < numberOfAges; i++)
        {
            ages[i] = Convert.ToInt32(Console.ReadLine());
        }
        int total = 0;
        for (int i = 0; i < numberOfAges; i++)
            total += ages[i];

        int average = total / numberOfAges;
        Console.WriteLine("Average is " + average);
    }
}
```