Writing clean, understandable, and maintainable code is a skill that is crucial for every developer to master.

In this post, we will look at the most important principles to improve code quality and I will give you code examples for each of them.

Most examples are taken from Robert J. Martin's Clean Code. It is a programming classic and I suggest you read the whole text when you have time.

## How to Name Variables (and other things)

"There are only two hard things in Computer Science: cache invalidation and naming things."                                                                                                                – Phil Karlton

There is a reason why we do not use memory addresses and have names instead: names are much easier to recall. And, more importantly, they can give you more information about the variable, so someone else can understand its significance.

It can take some time to find a good name but it will save you and your team even more time in the future. And I am sure most readers have faced the situation where you visit your code only a few months later and have a hard time understanding what you did before.

### How to Create Meaningful Names

Do not use comments to explain why a variable is used. If a name requires a comment, then you should take your time to rename that variable instead of writing a comment.

"A name should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent."                 – Clean Code

**Bad:**

var d; // elapsed time in days

I have seen this type of code so many times. It is a common misconception that you should hide your mess with comments. Do not use letters like x, y, a, or b as variable names unless there is a good reason (loop variables are an exception to this).

**Good:**

var elapsedTimeInDays;

var daysSinceCreation;

var daysSinceModification;

These names are so much better. They tell you what is being measured and the unit of that measurement.

### Avoid Disinformation

Be careful about words that mean something specific. Do not refer to a grouping of accounts as accountList unless its type is actually a List. The word has a specific meaning and it may lead to false conclusions.

Even if the type is a list, accounts is a simpler and better name.

**Bad:**

var accountList = [];

**Good:**

var accounts = []

### Avoid Noise Words

Noise words are the words that do not offer any additional information about the variable. They are redundant and should be removed.

Some popular noise words are:

* The (prefix)
* Info
* Data
* Variable
* Object
* Manager

If your class is named UserInfo, you can just remove the Info and make it User. Using BookData instead of Book as class name is just a no-brainer, as a class stores Data anyways.

You can also read Jeff Atwood's blog post about SomethingManager naming [here](https://blog.codinghorror.com/i-shall-call-it-somethingmanager/).

### Use Pronounceable Names

If you can't pronounce a name, you can't discuss it without sounding silly.

**Bad:**

const yyyymmdstr = moment().format("YYYY/MM/DD");

**Good:**

const currentDate = moment().format("YYYY/MM/DD");

### Use Searchable Names

Avoid using magic numbers in your code. Opt for searchable, named constants. Do not use single-letter names for constants since they can appear in many places and therefore are not easily searchable.

### Bad:

if (student.classes.length < 7) {

// Do something

}

**Good:**

if (student.classes.length < MAX\_CLASSES\_PER\_STUDENT) {

// Do something

}

This is much better because **MAX\_CLASSES\_PER\_STUDENT**can be used in many places in code. If we need to change it to 6 in the future, we can just change the constant.

The bad example creates question marks in the reader's mind, like what is the importance of 7?

You should also make use of your language's constant naming and declaration conventions such as **private static final** in Java or **const**in JavaScript.

### Be Consistent

Follow the **one word for each concept** rule. Do not use fetch, retrieve, and get for the same operation in different classes. Choose one of them and use it all over the project so people who maintain the codebase or the clients of your API can easily find the methods they are looking for.

## How to Write Functions

### Keep them Small

Functions should be small, really small. They should rarely be 20 lines long. The longer a function gets, it is more likely it is to do multiple things and have side effects.

### Make Sure They Just Do One Thing

Functions should do one thing. They should do it well. They should do it only. – Clean Code

Your functions should do only one thing. If you follow this rule, it is guaranteed that they will be small. The only thing that function does should be stated in its name.

Sometimes it is hard to look at the function and see if it is doing multiple things or not. One good way to check is to try to extract another function with a different name. If you can find it, that means it should be a different function.

This is probably the most important concept in this article, and it will take some time to get used to. But once you get the hang of it, your code will look much more mature, and it will be more easily refactorable, understandable, and testable for sure.

### Encapsulate Conditionals in Functions

Refactoring the condition and putting it into a named function is a good way to make your conditionals more readable.

Here is a piece of code from a school project of mine. This code is responsible for inserting a chip on the board of the Connect4 game.

The isValidInsertion method takes care of checking the validity of the column number and allows us the focus on the logic for inserting the chip instead.

public void insertChipAt(int column) throws Exception {

if (isValidInsertion(column)) {

insertChip(column);

boardConfiguration += column;

currentPlayer = currentPlayer == Chip.RED ? Chip.YELLOW : Chip.RED;

} else {

if (!columnExistsAt(column))

throw new IllegalArgumentException();

else if (isColumnFull(column - 1) || getWinner() != Chip.NONE)

throw new RuntimeException();

}

}

Here is the code for isValidInsertion, if you are interested.

private boolean isValidInsertion(int column) {

boolean columnIsAvailable = column <= NUM\_COLUMNS && column >= 1 && numberOfItemsInColumn[column - 1] < NUM\_ROWS;

boolean gameIsOver = getWinner() != Chip.NONE;

return columnIsAvailable && !gameIsOver;

}

Without the method, if condition would look like this:

if (column <= NUM\_COLUMNS

&& column >= 1

&& numberOfItemsInColumn[column - 1] < NUM\_ROWS

&& getWinner() != Chip.NONE)

Gross, right? I agree.

### Fewer Arguments

Functions should have two or fewer arguments, the fewer the better. Avoid three or more arguments where possible.

Arguments make it harder to read and understand the function. They are even harder from a testing point of view, since they create the need to write test cases for every combination of arguments.

### Do not use Flag Arguments

A flag argument is a boolean argument that is passed to a function. Two different actions are taken depending on the value of this argument.

For example, say there is a function that is responsible for booking tickets to a concert and there are 2 types of users: Premium and Regular. You can have code like this:

public Booking book (Customer aCustomer, boolean isPremium) {

if(isPremium)

// logic for premium book

else

// logic for regular booking

}

Flag arguments naturally contradict the principle of single responsibility. When you see them, you should consider dividing the function into two.

### Do Not Have Side Effects

Side effects are unintended consequences of your code. They may be changing the passed parameters, in case of passing by reference, or maybe changing a global variable.

The key point is, they promised to do another thing and you need to read the code carefully to notice the side-effect. They can result in some nasty bugs.

Here is an example from the book:

public class UserValidator {

private Cryptographer cryptographer;

public boolean checkPassword(String userName, String password) {

User user = UserGateway.findByName(userName);

if (user != User.NULL) {

String codedPhrase = user.getPhraseEncodedByPassword();

String phrase = cryptographer.decrypt(codedPhrase, password);

if ("Valid Password".equals(phrase)) {

Session.initialize();

return true;

}

}

return false;

}

}

Can you see the side-effect of this function?

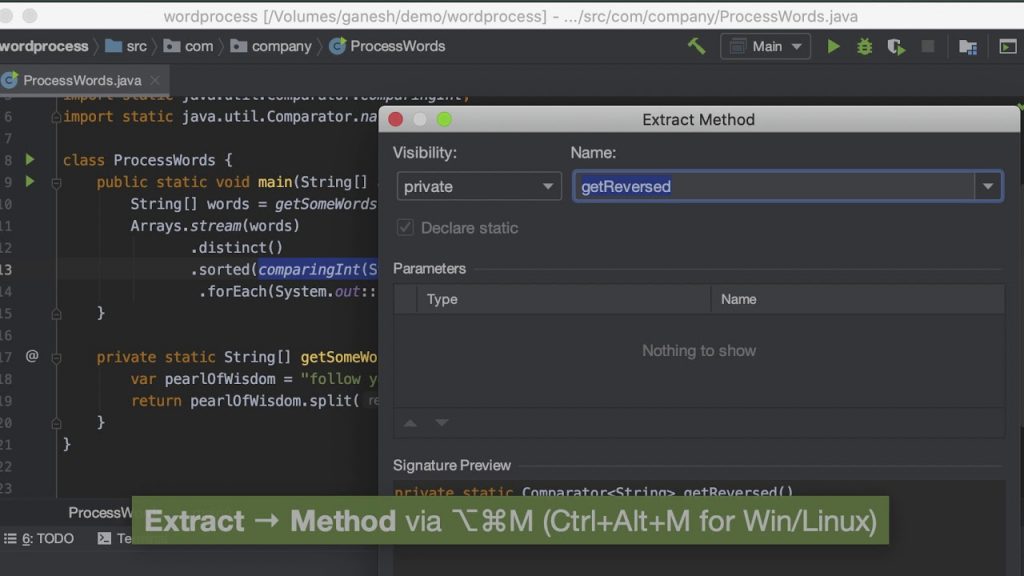
It is checking the password, but when the password is valid, it is also initializing the session which is a side-effect.

You can change the name of the function to something like checkPasswordAndInitializeSession to make this effect explicit. But when you do that, you should notice that your function is actually doing two things and you should not initialize the session here.

### Don't Repeat Yourself

Code repetition may be the root of all evil in software. Duplicate code means you need to change things in multiple places when there is a change in logic and it is very error prone.

Use your IDE's refactoring features and extract a method whenever you come across a repeated code segment.



## Bonus

### Do not leave code in comments

Please, do not. This one is serious because others who see the code will be afraid to delete it because they do not know if it is there for a reason. That commented out code will stay there for a long time. Then when variable names or method names change, it gets irrelevant but still nobody deletes it.

Just delete it. Even if it was important, there is version control for that. You can always find it.

### Know your language's conventions

You should know your language's conventions in terms of spacing, comments, and naming things. There are style guides available for many languages.

For example, you should use camelCase in Java but snake\_case in Python. You put opening braces on a new line in C# but you put them on the same line in Java and JavaScript.

These things change from language to language and there is no universal standard.

Here are some useful links for you:

* [Python Style Guide](https://www.python.org/dev/peps/pep-0008/)
* [Google's Javascript Style Guide](https://google.github.io/styleguide/jsguide.html)
* [Google Java Style Guide](https://google.github.io/styleguide/javaguide.html)

# Conclusion

Clean coding is not a skill that can be acquired overnight. It is a habit that needs to be developed by keeping these principles in mind and applying them whenever you write code.

Thank you for taking your time to read and I hope it was helpful.

If you are interested in reading more articles like this, you can subscribe to my [blog](http://erinc.io/).