Clean Code

Mohamed Emary

August 31, 2023

1 Clean Code From Odin Lesson Page

Learning to write clean code is a process of constant improvement. This lesson is meant to serve as a primer, a starting point for that journey.

Don't try to write perfectly clean code, this will only lead to frustration. Instead focus on gradual improvement, not perfection. Writing "spaghetti" is inevitable, everyone does it sometimes. Just keep these ideas in mind and with time and patience, your code will start to get cleaner.

Single characters can be used as variable names in the context of a loop or a callback function, but avoid them elsewhere.

What makes a good variable or function name.

- A good name is descriptive
- Use a consistent vocabulary. (example below)

```
// Good
function getPlayerScore();
function getPlayerName();
function getPlayerTag();
```

```
// Bad
function getUserScore();
function fetchPlayerName();
function retrievePlayer1Tag();
```

In the bad example, three different names are used to refer to the player and the actions taken. Additionally, three different verbs are used to describe these actions. The good example maintains **consistency in both variable naming and the verbs used**.

- **Variables** should always begin with a **noun** or an **adjective** (that is, a noun phrase) and ***functions*** with a ***verb***. (Two examples below)

Example 1:

```
// Good
const numberOfThings = 10;
const myName = "Thor";
const selected = true;
```

```
// Bad (these start with verbs, could be confused for functions)
const getCount = 10;
const isSelected = true;
```

Example 2:

```
// Good
function getCount() {
  return numberOfThings;
}
```

```
// Bad (it's a noun)
function myName() {
  return "Thor";
}
```

- Use searchable and immediately understandable names

Sometimes, it can be tempting to use an undeclared variable. Let's take another look at an example:

```
setTimeout(stopTimer, 3600000);
```

The problem is obvious. What does the undeclared variable 3600000 mean and how long is this timeout going to count down before executing stopTimer? Even if you know that **JavaScript understands time in milliseconds**, a calculation is needed.

Now, let's make this code more meaningful by introducing a descriptive variable:

```
const MILLISECONDS_PER_HOUR = 60 * 60 * 1000; // 3,600,000;

setTimeout(stopTimer, MILLISECONDS_PER_HOUR);
```

Letters in `MILLISECONDS_PER_HOUR` are all uppercase because this is a convention to be used when the programmer is absolutely sure that the variable is truly a constant that will never change.

- Choose a way to indent (Either tabs or spaces) and stick to it.
- Keep a line length that you generally don't exceed

Generally your code will be easier to read if you manually break lines that are longer than about 80 characters. Many code editors have a line in the display to show when you have crossed this threshold (Like that one vertical line I used to disable in jetbrains IDEs).

Code formatters like prettier can take care of the line length issue.

- Semicolons are mostly optional in JavaScript because the JS compiler **will automatically insert them** if they are omitted. This functionality CAN break in certain situations leading to bugs in your code so it is better to get used to adding semi-colons. *Again: consistency is the main thing.*
- Comments Part has some great advices that I think it's better to **READ FROM ODIN WEBSITE**.

2 Assignment Articles

2.1 Ten Principles for Keeping Your Programming Code Clean

1. Revise Your Logic Before Coding

Before blindly typing into the debugger of choice, some **flow diagrams or written pseudo-code** might come in handy to previously verify the logic behind those lines of code. Writing it down first can clarify many doubts or insecurities about complex functionality, and therefore save a lot of time. But most importantly, helping you get it right faster will also help you avoid all the messy code replacements and additions that tamper with the following principles.

2. Clearly Expose the Structure of the Page

This is done by using main containers with a representative ID

Example:

```
<div id="main-container">
  <div id="header">
    <div id="logo">...</div>
    <div id="main-menu">...</div>
  </div>
  <div id="content">
    <div id="left-column">...</div>

    <div id="center-column">...</div>
    <div id="right-column">...</div>
  </div>
  <div id="footer">
    <div id="footer-menu">...</div>
    <div id="disclaimer">...</div>
```

```
15    </div>
16  </div>
```

3. Use the Correct Indentation

indentation helps display the opening and closing points of each element used.

4. Write Explanatory Comments

A good comment should **tell why not how**.

5. Avoid Abusing Comments

What comments are NOT made for is:

- Writing explanatory notes to self (e.g. `/*Will finish this later...*/`).
- Blaming stuff on other people (e.g. `/*John coded this. Ask him.*/`).
- Writing vague statements (e.g. `/*This is another math function.*/`).

If the code will be documented via embedded comments, the team members need to make sure those **comments are there for a reason.**

Examples of good comment use are:

- Authoring specifications (e.g. `/*Coded by John, November 13th 2010*/`).
- Detailed statements on the functionality of a method or procedure (e.g. `/*This function validates the login form with the aid of the e-mail check function*/`).
- Quick notifications or labels that state where a recent change was made (e.g. `/*Added e-mail validation procedure*/`).

6. Avoid Extremely Large Functions

In the process of adding functionality to an application, its coded methods tend to grow accordingly. One can come across functions that consist of up to a hundred lines of code, and this tends to become confusing.

A better practice would be to **break up large functions into smaller ones**. This should have been avoided from the beginning if the first recommendation of **Revising Your Logic Before Coding** was carried out correctly.

7. Use Naming Standards for Functions and Variables

Whenever a variable or a function is created, its name should be **descriptive** enough as to give a general idea of what it does.

There are companies that have their own pre-established naming standards (e.g. The prefix `int_` for any numeric variables).

8. Treat Changes with Caution

This mainly involves:

- Keeping the correct indentations (e.g. when inserting an IF clause, its contents' indentations will be augmented).
- Commenting on the modification made or broadening the existing comments.
- Respecting standards in use.

9. Avoid Indiscriminate Mixing of Coding Languages

Like using In-line CSS styling and scattered JavaScript tags with short procedures within them. This will result in huge element tags with an embedded STYLE property, lots of interruptions in the flow of the structure because of embedded functions, and of course lots and lots of confusion.

Having the appropriate divisions between different coding languages will **give order to the logic applied**. This brings us, though, to the next consideration.

10. Summarize Your Imports

Even though it is much better to have additional coding languages imported from different files, this shouldn't be abused. If there are too many style sheets, they can probably be summarized into one or two.

This won't only save space and make things look cleaner, but it will also save loading time. Each imported file is an HTTP request that tampers with the performance of your application. So apart from being a consideration for tidiness, it is also a consideration for efficiency.

And, of course, this way one can avoid dealing with Internet Explorer's limit to the number of individual stylesheets.

To Sum Up

What's convenient isn't always what's best for the development process, since finding the convenient way to do something tends to drive us towards disregarding coding efficiency. Implied in this case for efficiency, there is a high need to keep up with standards in order to maintain a code that's readable in the future. Considering that it won't always be the same person who works on upgrading the same application, code should be sufficiently open and understandable for it to really support team work.

2.2 Coding Without Comments

While comments are neither inherently good or bad, they are frequently used as a crutch. You should **always write your code as if comments didn't exist**. This forces you to write your code in the simplest, plainest, most self-documenting way you can humanly come up with.

When you can't possibly imagine any conceivable way your code could be changed to become more straightforward and obvious – then, and only then, should you feel compelled to add a comment explaining what your code does.

> *Junior developers rely on comments to tell the story when they should be relying on the code to tell the story. Comments are narrative asides; important in their own way, but in no way meant to replace plot, characterization, and setting.*

The secret of code comments: **to write good comments you have to be a good writer.** Comments aren't code meant for the compiler, they're words meant to communicate ideas to other human beings. Since effective communication with other human beings is not exactly our strong suit. So it's better to stick to our strengths – that is, writing for the compiler, in as clear a way as we possibly can, and reaching for the comments only as a method of last resort.

Writing good, meaningful comments is hard. It's as much an art as writing the code itself.

As Sammy Larbi said in Common Excuses Used To Comment Code:

> *if you feel your code is too complex to understand without comments, your code is probably just bad. Rewrite it until it doesn't need comments any more. If, at the end of that effort, you still feel comments are necessary, then by all means, add comments . . . carefully.*

2.3 Code Tells You How, Comments Tell You Why

You should first strive to make your code as simple as possible to understand without relying on comments as a crutch. Only at the point where the code cannot be made easier to understand should you begin to add comments.

It helps to keep your audience in mind when you're writing code.

Knuth covers this idea in his classic 1984 essay on Literate Programming (pdf):

> *Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*
>
> *The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other.*

Note that more comments does not mean more readable code. Sometimes fewer comments makes for more readable code. Especially if it forces you to use meaningful symbol names instead.

No matter how simple, concise, and clear your code may end up being, it's impossible for code to be completely self-documenting. Comments can never be replaced by code alone. Just ask Jef Raskin:

> *Code can't explain why the program is being written, and the rationale for choosing this or that method. Code cannot discuss the reasons certain alternative approaches were taken. For example:*
>
> ```
> /* A binary search turned out to be slower than the Boyer-Moore algorithm
> for the data sets of interest, thus we have used the more complex, but faster
> method even though this problem does not at first seem amenable to a string
> search technique. */
> ```

What is perfectly, transparently obvious to one developer may be utterly opaque to another developer who has no context. Consider this bit of commenting advice:

> *You may very well know that* `$string = join('',reverse(split('',$string)));` *reverses your string, but how hard is it to insert* `# Reverse the string` *into your Perl file?*

Indeed. It's not hard at all. Code can only tell you how the program works; comments can tell you why it works. **Try not to shortchange your fellow developers in either area**.