Clean code

What Is Clean Code?

- The code can be measured with either "good" or "bad" in the code review or by how many minutes it takes you to talk about it.
- A clean code should be elegant, efficient, readable, simple, without duplications, and well-written.
- You should add value to the business with your code.
- Clean code offers quality and understanding when we open a class.
- It is necessary that your code is clean and readable for anyone to find and easily understand. Avoid wasting others' time.

Clean Code Rules For Design

- Keep configurable data at highest levels
- Prefer polymorphism to selections (if/else, switch/case)
- Separate multi-threading code
- Prevent over-configuration
- Use dependency injection
- Understand inheritance (classes and direct dependencies)

Understand-Ability

- Try to code consistently. If you do something in a certain way, do it like that all the time.
- Give meaningful names to variables.
- Try to use objects instead of primitive types, where needed.
- Avoid negative and confusing conditionals.

Naming

- Go for a descriptive and unambiguous names
- Use pronounceable and searchable names
- Replace magic numbers with named constants
- Avoid encodings, such as appending prefixes and type information if not needed

Clean Code Rules For Functions

- The method should be easy to read and understand.
- The method should convey its intention.
- The methods should be small. Another rule for small methods is that they should be even lower.
- They must have up to 20 lines. (I think they should have up to 10 lines.)
- Methods should only do one thing: they should do it the right way and just do it.
- You should use names with words that say what it really does.
- The optimal number of parameters of a method is zero, after one and two.
- Three should be avoided, but if you think it should be used, have a good justification.
- Parameters of the Boolean type as a parameter already clearly states that it does more than one thing.
- Methods must do something and return something.
- Avoid duplication.

Pro Tip: Comments

- One of the most common reasons for the comments is because the code is bad.
- If you're thinking about writing a comment, then the code should be refactored.
- Comments do not save a bad code.

- Try to explain what the code causes to happen.
- Comments can be useful when placed in certain places.
- Create method names and informative variables instead of explaining the code with comments.
- Comments can be used to express the importance of certain points in the code.
- The best comment is one that needs to be written because your code already explained.
- Do not write comments with redundant, useless, or false information.
- They should not be used to indicate who changed or why, for that already exists in versioning.
- Don't comment code that will not be used, remove, it just pollutes the code and leaves no doubt in anyone reading.

Formatting

- Formatting should indicate things of importance since it is a developer of communication form.
- A messy code is hard to read.
- The readability of the code will take effect on all of the changes that will be made.
- Try to write a class with a maximum of 500 lines. Smaller classes are easier to understand.
- Set a limit of characters per line of code.
- A good character limit on a line is 120.
- Try to keep more next related concepts vertically to create a code stream.
- Use spaces between operators, parameters, and commas.

Objects and Data Structure

- Follow the Law of Demeter, which says that one *M* method of an object *O* can only consume services of the following types of objects:
    - The object itself, *O*.

    - The *M* parameters.
    - Any object created or instantiated by *M*.
    - Direct components of *O*.
- Make good abstraction and encapsulation.
- Do not make dumb objects.
- Objects hide the data abstraction and expose methods that operate the data.
- Data structures expose your data and do not have significant methods.

Error Handling

- Error handling should be planned carefully by all programmers.
- When wrong things occur, we have to get it to do the right things.
- We should give preference to launching an exception than treating it just to hide.
- Create messages with information about the error.
- Mention that it failed. Where was this failure? If possible, mention why it failed.
- Look at separate business rules for errors and error handling.
- Avoid returning a NULL in methods, preferably to return an empty object.
- Avoid passing NULL to the methods; this can generate NullPointerExceptions.

Classes

- By default, Java classes should start with the variables:
    - Static and constantly public.
    - Static and variable private.
    - Instances and variables privates.
    - Soon after comes the functions.
- The class name should represent your responsibility.
- The class must have only one responsibility.
- To know the size of the class is ideal or we should not measure her responsibility.
- You should try to make a brief description of the class.
- The methods should be:

- Small...
- ...and even lower.
- They must have only one responsibility.

Pro Tip: Source Code

- Separate concepts vertically
- Declare variables close to their usages (both horizontally and vertically)
- Keep dependent functions close.
- Keep similar functions close.
- Don’t break indention.
- Keep things simple and pretty.

Example :

Bad:

```
var d;
```

Good:

```
var elapsedTimeInDays;
```

Bad:

```
var accountList = [];
```

Good:

```
var accounts = [];
```

Bad:

```
if (student.classes.length < 7) {
   // Do something
}
```

Good:

```
if (student.classes.length < MAX_CLASSES_PER_STUDENT) {
    // Do something
}
```

Bad ✕

```
double limit = 13.89;        // unit not clear
from_to(int x,  int y,
        int x2, int y2);     // vague argument names

if (speed > limit &&
   t.h > 22 && t.h < 6){ ... }
```

Good:

```
MeterPerSecond limit = SPEED_LIMIT_NIGHT;
drive(Point origin, Point destination);

isNight    = (T_NIGHT_MIN < t.h && t.h < T_NIGHT_MAX);
isSpeeding = (limit < speed);
if (isSpeeding && isNight){ ... }
```

Bad ✕

```
class MyClass {
 public:
  int CountFooErrors(const std::vector<Foo>& foos) {
    int n = 0;  // Clear meaning given limited scope and context
    for (const auto& foo : foos) {
```

```
      ...
      ++n;
    }
    return n;
  }
  void DoSomethingImportant() {
    std::string fqdn = ...;  // Well-known abbreviation for Fully
Qualified Domain Name
  }
 private:
  const int kMaxAllowedConnections = ...;  // Clear meaning within
context
};
```

Good:

```
class MyClass {
 public:
  int CountFooErrors(const std::vector<Foo>& foos) {
    int total_number_of_foo_errors = 0;  // Overly verbose given
limited scope and context
    for (int foo_index = 0; foo_index < foos.size(); ++foo_index) {
// Use idiomatic `i`
      ...
      ++total_number_of_foo_errors;
    }
    return total_number_of_foo_errors;
  }
  void DoSomethingImportant() {
    int cstmr_id = ...;  // Deletes internal letters
  }
 private:
  const int kNum = ...;  // Unclear meaning within broad scope
};
```

Ref [Google C++ Style Guide]

[Clean Code Explained – A Practical Introduction to Clean Coding for Beginners (freecodecamp.org)]