## Programming Principles

Part of *Programming*. Alternative title: “How to Actually become a Better Programmer”

Use my earlier work (see folder) when needed.

1. Introduction
2. What is “good” code?
3. Some truths to accept
4. Comments & Intent
5. Readability (clean code)
6. Data transformations / manipulation
7. Modularity
8. Testing
9. Conditionals
10. Performance (big O and stuff, also pitfalls or premature, *measure it*)
11. All writing I rewriting => refactoring and doing things 2 or 3 times is basically *required*.
12. Dependencies & Updates
13. General Principles

Why chapter 4 and 5? Most of the time, you will be *reading* code, *checking* your own code (or that of others). As such, readability and code that shows what it *wants* to do is crucial. Comments will be outdated and useless, visual noise. Ugly formatting is as bad as code that doesn’t work. => These can be quite short though, as the real examples and tips come *in the rest of the course*.

Why “testing” so early? Because you have no clue how to value your programming if you’re not sure if it *works at all (or exactly as intended)*. Additionally, these force you to be modular and use data transformations, otherwise testing is a chore. Finally, they force you to consider edge cases.

**REMEMBER:** Make this accessible to new programmers as well. So start with the basics. With some real *real* simple examples of optimization.

* Plan events in the future, instead of checking each frame
* Work with events in general, instead of checking each frame
* Save information you need often, instead of getting/calculating it again each frame

In the introduction =>

*Good programming is not about computers, or syntax, or the specific language you use. In fact, if I were to teach coding, I wouldn’t involve a computer or specific syntax until the final stages. Programming is problem solving and logical thinking—the words you end up typing are an afterthought.*

*To show this, I’ve created my own very simple scripting language. During this course, I’ll often add an interactive window with an example. You can type anything you want in there and check what happens! Play around!*

*This language is so simple that it can barely even be called a language. And yet … it is able to explain all the major concepts pretty well.*

## Specific Tips

* Often, the solution isn’t *smarter code* or *better/faster code* or *new features*. The solution is to pick better **data structures** and a completely different general **architecture.**
  + Use my word dictionary as example! How switching to an “index” made all of it possible. (While the conversion is quite simple.)
  + Similarly, most of it comes down to how you use memory. That’s the biggest bottleneck, and the biggest potential for accidental waste.
  + Keep things close in memory. Stop allocating lots of memory, the immediately giving it back. (Or using many redundancies.)
  + I remember this story by Jonathan Blow about how he found out the developers of some popular game just use one big, stupid array for a certain algorithm. He was mad and told them how stupid it was, and the many ways to optimize it. But now, years later, he understood. It solved the problem, it was fast to code, it was easy to understand/read. The performance hit, even if it were big (which it wasn’t), would not be more important.
* Use the Rust idea: only one owner, you pass in what you need specifically, and get something new out of it
* If possible, make the parent know about the child, but not the other way around. (Parent calls direct functions on child. Has reference to it. Child *has no idea*. If needed, it can only have a “loose coupling”: a signal that might or might not be accepted by a parent.)
  + Call down, signal up.
  + (Even so, if possible, using signals for *all* things is nice.)
* When naming variables, go from *biggest category* to *smallest category*. (“context” and “offlineContext” are hard to understand immediately, and link in our brain. Instead, “contextOnline” and “contextOffline” are clearly different versions of the same general thing. => This is from my Web Audio API experience)
* Sometimes you don’t *know* the problem. If needed, you can still *solve* it (by rewriting the code in a different way, or going around it). Often, though, it’s better to research the cause diligently. Often this reveals a *major* issue lying in wait, which would’ve completely destroyed everything at some later date.
* The idea of **non pessimization** (Casey Muratori). There’s no need for extremely detailed or intricate programming hacks. Just don’t do absolute stupid, gratuitous, wasteful things (on accident or not). If you never pick the absolute worst options, you will not need those optimization hacks.
* **Don’t use null / nil.** You can pick a language that doesn’t support it (like Rust). You can also write your code in a smarter way.
  + **Example:** instead of returning null when trying to access a Cell that’s out of bounds (of a grid/map), return a Cell that is marked as *invalid* or *out of bounds*.
  + **Example:** instead of returning null when some array indexing function doesn’t work, provide checks upstream to prevent the function ever being called with invalid arguments.
* **About setters/getters.** If at all possible, avoid them.
  + Setters make your object mutable—a bad idea. (The object should be fixed or only the object *itself* should change *itself*.)
  + Getters extract information from your object to be used … however. (It violates the principle “Tell an object what to do, instead of asking it for information and then manipulating it yourself”
  + In summary: “Strive for immutability where possible. When mutation is really needed, ask it to mutate for you, instead of setting/extracting values from the outside.”

**Good video:** <https://www.youtube.com/watch?v=tKbV6BpH-C8>

Great comment: “One lesson that always stuck with me since college was that software engineering doesn’t just have one time-cost. It has three: runtime cost, development time cost, and repair/maintenance cost. Which happen to be a perfect allegory to the performance, velocity, adaptability triangle.”

## Research on creating your own Programming Language

General process:

* Write grammar
* Parse text into *tokens*
* Collect tokens into *abstract syntax tree*
* Execute AST

A **terminal** is a part of the grammar that is done and needs no further evaluation (like the final *int* literal). A **non-terminal** is anything else; something that needs to be resolved using some other grammar rule.

The **leaves** of an abstract syntax tree are the actual values. (So the literals put in, the specific kind of operator or expression it’s part of, any *literal finished data*.) The **nodes** are about combining those in the right way.

To execute this code, you simply ask the root to “execute”, which calls execute on all its children, and so forth. Nodes simply pass on the “execute” call (and return what they get). Only the **leaves** actually have a value, and only the **operators/keywords** actually perform a calculation or operation.

Best in-detail resources:

* <https://medium.com/@gvanrossum_83706/peg-parsing-series-de5d41b2ed60>
* <https://ruslanspivak.com/lsbasi-part9/>

Check the lexical grammar of JavaScript here: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar>

* Line-terminating characters
* Comments
* Identifiers (link a *value* with a *name*)
* Keywords (identifiers with a special meaning, not used to link value->name)
* Literals (literal pieces of information)
  + Object and Array literals are *expressions* that consist of a *series* of tokens

**Step 1:** parse the text. Divide it into “Token” objects which could have a …

* Literal (Number, Boolean, String)
* Variable (knows its name / where to look it up or change it)
* Operators (which does something to the one or two values around it, like math)
* Structure (a reference to a list/bag)
* Expression (a set of tokens which should *resolve* to a single value?)
* Statement (resolves that expression to a value, then does something with it => the basic structure of all code => like print, assign, input, condition)

**Step 2:** step through the tokens to assign meaning. This is probably recursive.

Functional Parsing video (20 min) by Numberphile: <https://www.youtube.com/watch?v=dDtZLm7HIJs> => **pretty good introduction!**

Another great article: <https://tomassetti.me/parsing-in-javascript/> => (Goes a bit further in depth, about left and right recursive, the two steps (LEXER and PARSER), an image of an Abstract Syntax Tree.)

After that, this is a *great* resource on building a simple parser. Just three short articles, all the steps, explained well and minimal implementation: <https://lisperator.net/pltut/parser/>

How to write down grammars (**Backus Naur Form**) + a good, short, readable example of it

* <https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form>
* <https://stlab.cc/legacy/how-to-write-a-simple-lexical-analyzer-or-parser.html>

Then check this article: <https://www.geeksforgeeks.org/recursive-descent-parser/>

It explains the *recursive descent parser* (which the others also explain, if I’m not mistaken). And it explains how to solve the “left recursion” (x = x + 5) problem with smart grammar => more on *that* problem here:

* <https://www.tutorialspoint.com/what-is-left-recursion-and-how-it-is-eliminated>
* <https://www.youtube.com/watch?v=H7iGUr2W5N8>

A **great** YouTube playlist with lectures about parsing and abstract syntax trees: <https://www.youtube.com/watch?v=4m7ubrdbWQU>

Related “parsing algorithms” (more theoretical) course by the same guy: <https://www.youtube.com/watch?v=VZ5DJopq5JA>

Sources:

* <https://hackernoon.com/building-your-own-programming-language-from-scratch>
* <https://www.guru99.com/compiler-design-lexical-analysis.html>
* An **amazing** tool that parses and creates an **Abstract Syntax Tree** for any JavaScript code! => <https://www.101computing.net/abstract-syntax-tree-generator/>

## Pirate Lang

### Variables

**Keyword:** “be”

* A be 5
* someVar be “blap”

**Keyword**: “steals from”

* A steals from (someVar + 5)

### Conditional

**Keyword:** “when … be”

* When A be 10

### Functions

**Keyword:** “sail to <name> with booty <arguments>”

Other ideas:

* Bottle / message in a bottle
* Hook
* Seven seas
* Raise the sail
* Plunder
* Treasure
* Treasure map
* Lock/key
* Gold
* …