# How to (Actually) become a Better Programmer

**This guide is for anyone who knows *something* about programming and wants to improve on their craft. If you’ve never programmed before, I suggest you first start somewhere else.**

Like most programmers these days, I am self-taught.

My programming journey started with reading online tutorials, trying silly side projects, and watching YouTube videos until my code finally started working. I never took a full class, never read a big book about best programming practices, nothing of the sort.

I wish I had.

Being able to learn by yourself, using these amazing online resources, is a great thing. It also means, however, that the quality of your education and thus the speed/efficiency of that learning process isn’t guaranteed.

One tutorial might teach you method A. The other might say that method A is absolutely horrible, and you should use method B!

Then you find several code snippets and just copy/paste them to your project, not even considering how they work, their performance cost, or if it’s good practice at all.

**For example:** many, *many* tutorials teach you to incrementally update timers in some sort of update function. If you have a countdown, they teach you to do “timer -= deltaTime” every frame. I did this for years in all my projects.

But you already guessed it: this is horrible coding. It consumes resources every frame, and if you have many timers, your update function is horrible to read and maintain. Better to use e.g. built-in timers (if your software/library supports it) or an instruction queue where you plan the “time has run out” event.

**Note:** this guide is mostly written from the perspective of a game and web developer. Although most of it applies to any sort of programming/coding, the examples will largely be from my own experience.

## Golden Rules

In this guide, I want to teach you many “**golden rules**” I’ve picked up over the years. Things I learnt after countless mistakes and tutorials pointing me in the wrong direction. Improvements you can make to your code across the board, no matter your goal, no matter your language.

I was tempted to call these “**tips and tricks**”, but that really doesn’t do them any justice. They are more of a … philosophy, a way of thinking about problems, a better way of communicating with computers (and your future code-reading self).

Read this guide, apply the principles to whatever your current project is, and grow substantially as a coder :)

*Remark:* many of my projects are online and freely available, so you can check the source code. And yes, you will see that in each of those projects, I make countless mistakes about which I’ll now say: “never do that!” And I will continue to make mistakes. I do not write this guide pretending to be a perfect programmer, I merely want to share the things I have learned.

*Remark:* yes, university courses or books can *also* steer you in the wrong direction. In fact, I went to university and had an obligatory Java course. Imagine my surprise when the teacher had trouble getting his computer to work and started teaching us very outdated methods for programming.

The message is: try to absorb as many information/philosophies/ideas about programming as you can! Don’t get stuck one way, just because that happens to be the first method you learned.

## Golden Rule: Code, code, code

Before we even begin the guide, we need to consider the golden rule of all programming: **you must code, a lot**.

The only way to truly improve as a programmer, to truly learn these principles by heart and create good, strong habits, is by programming *a lot*. Start side projects, try different languages and systems, experiment, tackle something you were too afraid to try two years ago.

When you read this guide, I hope you can immediately apply the information to improve your skill and your code. But you’ll probably forget most of it. Or you won’t see *why* it should be a good idea. Maybe you feel some things are just overly abstract or convoluted. Only repetition, experimentation, and experience can help you internalize these ideas and see when they are (or aren’t) applicable.

## Table of Contents

Thisguide will expand over time, as I add articles whenever I think something is worth sharing. I also try to divide them into logical sections, to make them easier to read and apply.

Nevertheless, it’s highly recommended to go through them in chronological order, because I start with the most fundamental principles first.

1. Defining “Good” Code
2. Acceptances
3. Modules & Data Transformations
4. Pure Functions
5. Conditionals
6. Readability (Clean Code)
7. Data Manipulation
8. Performance (big O and stuff)
9. Dependencies & Updates
10. General Principles
11. Testing
12. ??

# Defining “Good” Code

## What is “good” code?

In order to *check* if your code (or skill as a programmer) is actually improving, you need some metric for “good code”.

This is extremely dependent on context.

If I know my game will need quite some computer resources and do some heavy lifting, then *performance* has a high priority.

If I’m making a website, I might care more about making it *search engine friendly* and keeping the file size *small*. (Because my servers have a limited bandwidth per month. Yes, I am broke and can’t pay for more. I see it as a *challenge*.)

If I’m developing an application of significant size, with multiple people working on that same project, then the ability to *read* and *maintain* the code becomes vital.

## List of Metrics

However, as you already saw above, you *can* define a general list of metrics to keep in the back of your mind.

Obviously, the most important metric is **code should work**. It should do what it’s supposed to do, hopefully 100% of the time.

Here is the rest of my personal list (in no particular order):

* Readability
* Maintenance
* Flexibility
* Performance
* Size
* Robustness
* Practicality (quick hacks might be better than general solutions)

Here are some quick explanations for each metric. (I’ll elaborate on these metrics and what they mean during the guide.)

**Readability =** how easy it is to read your code and understand what it does.

**Maintenance** = how easy it is to keep the code updated, to fix bugs and test specific cases, to work on the codebase with a large team over large periods of time

**Flexibility** = how easy it is to extend the code, adapt it, customize, use different parts for different purposes, perhaps switch direction halfway through a project.

**Performance** = well, obviously, the faster code runs the better

**Size =** the actual size (on the hard drive) of your code files, the number of files/folders/dependencies in your project, the verbosity of the code (including comments).

**Robustness** = how well the code is guarded against exceptions, rare errors, bad input, updates or regressions. I gave this its own category (instead of putting it under maintenance/flexibility), because it’s just too important.

For example, many of my older programs/games are prone to weird crashes, simply because I never considered the full scope of things that can go wrong. Which makes them virtually unplayable, because people don’t think the game is good or don’t trust the game anymore.

**Practicality** = does the code work well *in practice*? It’s nice that something should theoretically work, or that you have some abstract piece of code that looks beautiful, but code is meant to be *used in practice*. Sometimes a quick hack is the best solution.

## My ideal workflow

Personally, my number 1 priority is always **working efficiency**. I want my code to allow me to work at peak efficiency, with no delays, no roadblocks. That’s why I am obsessed with keeping my projects small, with minimal dependencies, using software that boots within a second.

(I’ve noticed that people can be extremely productive this way. With this setup, I’m now able to accomplish much more in a day than before in a week. But then again, I also quit social media during that time and disabled internet on my smartphone.)

But it might be something different for you.

I know people who love splitting their projects into a solid structure with a huge number of files, because they like that structure (and the fact that individual files will have very few lines).

Find out what’s most important to you, what you like working on and working with, and hold you code to that standard.

## A real-life example

I’ve recently developed a new type of game: **Pizza Peers – Local Multiplayer using your Smartphone ((LINK HIER)).**

You start the game on your computer, but you *play* using your smartphone. The game was an experiment in making extremely accessible local multiplayer games: all players just need to visit the website on their phone, and they can play a game together!

I’m glad to see the experiment was a success and I’m developing more games of this “genre”, but the development was, as you can expect, a rocky road.

The game is just a website, which meant I had the following priorities:

* **Size**: the website and all the assets should get downloaded to phones/computers very quickly.
* **Performance**: a web game cannot do the same things as a desktop game. Each step of the way, I needed to think about performant ways to include something. And some game ideas were scrapped because they were too resource intensive.

It was also the first game of this type, which meant I had to completely figure out all parts: the networking (peer to peer connections), the latest version of the Phaser engine for web games.

For that reason, I focused heavily on *documenting* the code (aka **readability**). Almost every line has a comment attached. Sometimes I have a large comment block containing links, exceptions, things to look out for. I learnt these things *while* programming the game and had to write them down to remind myself.

I did not focus on the other aspects. I knew the game would be small, just a solo project that could be done within a month, and that’s how the code was written. Extending, customizing, adapting, updating the game will all be very hard to do (if they ever need to be done). Because of the tiny game size, and the fact it’s free, I made no additional effort for robustness.

You might think “well, that’s just bad practice”, to which I say “but it did allow me to invent a new type of game and create it from start to finish in under two weeks”

Hopefully that gets the point across.

(It gets even worse. The map is randomly generated, and maybe *one out of hundred generations* something goes wrong, and it gets stuck in an infinite loop. I spent a whole day rewriting that code and optimizing/fixing where possible, but could not find the source of the issue. So eventually, I settled on checking whether generation was successful, and if not … just keep trying again and again until we have something that works. Brute force solution, ugly, but it allowed me to finish the game and “ship” it.)

## What’s the purpose of code?

The second basic question to ask ourselves is: what should code do? What is code *supposed* to accomplish at its core?

Okay, we might already be entering controversial territory, but bear with me. I do not want to impose any methodology; I merely want to share knowledge and what I think is a useful way to code.

The purpose of code is **data transformation.**

IMAGE HERE (to somehow help make the point)

Think about it.

For example, say I’m developing a game. When the user clicks somewhere on the screen, that is *data* I receive. That click needs to do something. So I check what has been clicked by *transforming* the input coordinates to my world.

Oh, I see, the player has clicked on a tile! Now the hero must walk to that location. Let’s *transform* the (x,y) coordinates of the hero to the next tile, every frame, until we arrive at our location.

Once you separate code from the idea of data transformations, it actually becomes *harder* to reason about the code, because you create several layers of abstraction between the code and what it should actually do. Not to mention that performance and robustness are severely impacted.

There are many different “methodologies” which usually claim to be the best way to program. To my surprise, many of them do not even consider what I just told you.

These methods were usually developed to make teaching code easier (to students or beginners), or because of some abstract notion regarding the real world or what beautiful code should look like. Which is fine, I guess, but doesn’t make for good code once you become serious about it. Or, to give it a positive spin, I think code could be *better* by embracing different philosophies.

Of course, I will explain this with a few examples below.

## An example: OOP

One such method is “Object-Oriented Programming” (OOP). It teaches you to turn everything into a self-contained object (and *inherit* from other objects when possible).

Want to have several animals in your game? Create a general “Animal” class. Then create a “Dog” class that inherits from Animal.

It sounds very logical, intuitive, easy to wrap your head around. This is also what they taught in my Java course (at university) and all subsequent courses involving programming and algorithms.

However, just because something resembles the real world … does that mean it’s the best way to program it?

IMAGE HERE (Inheritance, Dogs/Whales, etc.)

### Example: Dogs & Whales

Indeed, a Dog is a type of Animal. So is a Whale. But when you are creating a game, I’m quite sure that the *actual functionality* of the Dogs and Whales in your game will be miles apart.

(Does sound like a cool game, though. “Dogs & Whales: An Unexpected Story”)

And when the functionality is far apart, you’ll be writing a lot of custom code for each class and repeating yourself.

On top of that, if all the complexity is in the Dogs and Whales classes … then what does the Animal class actually do? It just becomes a generic wrapper that slows down your code and creates unnecessary function calls.

What if we added a Bird later in the game? Finally, we made the dogs walk and the whales swim, do we need a completely different codebase for flying!

Looking at this, the OOP structure becomes more of an obstacle, something that slows you down and makes you rethink each decision.

Then there’s the problem of **multiple inheritance**. If you have a class Animal, and a class Collectible, what do you do with a *Collectible Animal* in your game?

IMAGE HERE (of multiple inheritance)

Not to speak about the fact that things **don’t exist on their own**. You might have hundreds of animals in your game. Usually, it’s cheaper, quicker and less error-prone to reason about them *simultaneously*, instead of calling functions on their objects one by one.

### Example 2: Doors & Nails

To make the point clearer, let’s consider another example common in games: **doors**.

Let’s say we have several types of doors:

* Regular: walk up to it, press a button, it opens.
* Decoration: cannot be interacted with, just for visuals
* Breakable: shoot/slam it to break it.
* Locked: find a key/solve a puzzle to unlock it

If you have any familiarity with OOP, you’ll immediately jump to this idea: “we’ll create a general Door class, and all the others inherit from that!”

But when you think about it … how much do these objects really have in common?

A locked door requires a completely different set of mechanisms, rules and visuals than a regular one. So does the breakable door. And the door purely for decoration does not need any of those things, so inheriting from the Door class would be a great waste of resources.

(These are quite succinct examples, I know. We’ll dig deeper into them during this guide.)

## Conclusion

Think about what defines “good code” and “good programming” to you and your team/company/product.

Whatever methodology you’ve learned, don’t get stuck with it. In some cases, a piece of software actually *does* mimic the real world very well, in which case OOP is useful. In many other cases, there’s no valid reason to model your whole codebase around the real world or some abstract notions of concepts.

I’ve singled out OOP here, simply because it’s very well known, but the same holds true for other philosophies.

Heck, I don’t even stick to the same philosophy each project!

However, trying out many different techniques over the years, I’ve always found one common denominator for when each technique worked optimally: they werefocused on **data transformations**.

During the guide, I’ll give many examples about what this means *in practice*.

# Acceptances

This is an unusual second chapter, but that’s exactly why I wrote it.

Many problems with programming (and code-based projects in general) come from people not *accepting* the real world. We are human: we make mistakes, we have limitations, we can’t perfectly communicate our desires to a stupid computer.

If you learn to accept what I’ll discuss in this chapter, you will be more productive, program more, and eventually create better and more polished programs. And as the saying goes: “real developers ship”.

I’m not saying it’s easy. I’m saying it’s necessary, both for your work and your own (mental) health. (Programming can be an extremely tough job, especially if the obstacles seem insurmountable, and you’re on your own. Use these acceptances to make life a little easier.)

## #1 Your code will break

I remember being *so* devastated when I had worked on something for a full day, or maybe several days, and it *still didn’t work*.

The guy in the tutorial made it sound so easy. He got it right the first time! And here I was, still banging my head against the wall.

Programming is hard. This means you’ll rarely get things right the first time. This means you might have a working piece of code … until a few weeks later, you add something new, and the whole thing breaks again.

It’s best to see this as a puzzle, a challenge, a learning process. Coding something is not the same as following the instructions and, voila, you’ve built yourself a nice IKEA closet. It’s creative, it’s a process of finding the issues and then finding ways to overcome them.

It's iterative and challenging by its very nature. So don’t be discouraged when your code breaks. Don’t be discouraged when it breaks the second time, the third time, and so on. Every time, simply find a way to fix it, which will (over time) teach you many ways to write better code.

Additionally, I see programmers who try to make their code work on *all possible platforms* from the *dawn of time until the heat death of the universe*. Okay, that’s obviously hyperbole, but you get my point: just make code work for your intended platforms in the foreseeable future.

At some point, that carefully crafted and future-proofed program of yours … it will break down. The fix might be easy, the fix might be nigh impossible, but that’s an issue for you (or your company) fifty years from now. Don’t think about it.

Accept that your code will break.

## #2 Your code will look ugly, for now

There are several programmers I’ve really come to admire. They write amazing games/products/libraries which do wonderful things. And, sometimes, these are open source and I can check out their code.

I was expecting some god-like level of programming. Beautiful code, easy to read, perfectly written.

Of course, that’s not what happened. Their code looked quite similar to my code, only slightly clearer and cleaner. At several points, they even included comments like: “this is an ugly fix, I need to find a better way to do this – coder’s initials 2020”

Sometimes the ugly hack was years old and still hadn’t been fixed. Sometimes there were several of these statements in a single file, each a more severe violation of good coding practice than the last.

The point is: focus on getting things working. The biggest thing holding you back, is being stuck and not doing anything.

Once you have *something*, you can improve and iterate. If you have *nothing*, you can’t.

So yes, write that ugly piece of code, use that weird hack to fix some issues, if it allows you to continue coding and to continue with other vital parts of the software.

Accept that your code will look ugly … for now.

## #3 Your code will be rewritten

For some part, this is an extension of what I’ve discussed above. You won’t get something right the first time, code will start ugly, and it will break, all of which requires you to rewrite stuff.

But rewriting is not always strictly “necessary”. It might be that you want to improve the structure of the project, so programming becomes easier in the future. It might be that you’ve changed a few ideas about how your game should work, requiring you to rewrite a piece of code that actually worked flawlessly.

Or it might be that someone else on your team takes over after you, and rewrites parts of your work. There are even whole games that spend months developing some amazing part of the game … only to throw it out or repurpose 95% of it later.

Never think of your code being final. It will be rewritten, either by you or someone else, and that’s only a good thing.

As I code these days, I set aside time regularly to “rewrite” the project. During development, I’ve marked areas that are ugly, or don’t perform well, or that are inconsistent with the rest of the codebase. On my “rewriting day”, I go over these areas and usually rewrite almost everything. It’s usually a huge performance boost, both for the game and my own working efficiency.

Accept that your code will be rewritten.

## #4 In hindsight, everything is stupid

You see, artists have this annoying inherent property: “as we work on a project, we get *better* at the skills required for this project”

This means that, halfway through the project, you’ll look back at your first lines of code and say: “What was I thinking?! This is horrible, I must improve this!”

Do not give in to this, because it’s a never-ending cycle. Once you’ve improved, all your older work looks stupid and dumb. You want to bring it to your new (hopefully higher) standards, but if you do that, then your recent work will look weak in comparison!

In hindsight, everything you did was stupid. With the skills and knowledge you’ve gained, previous projects will always look mediocre at best. Let them be that way.

(Heck, I’m only at chapter 2 of this guide, and I’m already itching to polish the first chapter.)

Accept that you’ve improved, but don’t need to improve your older work.

## Conclusion

If you internalize these acceptances, I’m sure you’ll find it freeing. You are more productive, quicker to iterate, and feel better about your work/progress in general.

The moment I learned this, was the exact moment I actually started *finishing* all my projects and *improving* every single day.

(Before that, I had tens of unfinished projects, all of which didn’t allow me to grow very much. I only did the part of the project I already understood, and when things got hard, I was like “I’m not smart enough for this!” or “This code is becoming a giant mess!”, and quit the project.)

With this out of the way, let’s look at some *golden rules* for managing the global structure of a project, in the next chapter.

# Modules

So far, I’ve told you that coding is about data transformations. (And I’ve given some examples about why other methodologies, such as Object Oriented Programming, might not be good ideas.)

In this chapter, I want to look at how you can implement these ideas at a high level. A way of structuring your whole codebase (and design philosophy) that makes it easy to read, to understand, to adapt and to maintain.

I’m talking about using a **module (or modular) system**.

This is certainly not the only way, but it’s the one that stands head and shoulders above anything else I tried. And I feel like the industry is slowly realizing this as well. (A common quote I encounter these days is: “prefer independent components that can be used in different places”)

## Modules

Any piece of software is supposed to *do* something. (Now everyone in unison: “it’s supposed to transform data!”)

It doesn’t make sense to create objects or classes based on what they *are*, or what they *resemble*, or to which *category* they belong. But it does make a whole lot of sense to create classes based on what they *do*.

For this, we need a module system.

* Each **module** has a single unique **functionality**.
* Each **“thing”** in the project is made up of any number of modules.

For example, let’s say we have a puzzle game. You win a level if you manage to grab all collectibles. Some objects are static (like a nice diamond to collect), some move around (let’s say a guard holding a key).

What do we do? We create two modules: **collectible** and **moving**.

The guards receive both modules, as they move around, but are also a collectible. But a regular diamond only gets the collectible module, as it doesn’t move.

## The power of Modules

Let’s say we introduce another mechanic: grabbing a key unlocks a specific door. A locked door, of course can’t be passed through.

What do we do? We create two modules: **locker** and **unlocker**.

The key (or whoever holds it) receives the unlocker module. The door it belongs to receives the locker module.

When you grab the key, it sees that it has an unlocker, sends a signal, the locked door knows to receive it and opens itself. It still resembles an object oriented approach, which makes it intuitive and understandable, but now much more flexible.

Now, allow me to remind you that these things can have *any other modules* attached to it as well. In some crazy mind-bending puzzle, a locked door might *also* be a collectible and moving!

That’s the power of modules. Create functionality once, use it to create things that do any combination of all possible functionalities.

## How would this work in code?

This really depends on your language and your tools (such as game engine or libraries).

Usually, you want to be able to *bind* modules to objects and certain events.

Something like this:

* When defining the object in code (or your editor), define the modules attached to it.
* When the object is loaded, it loops through all its modules and calls some “init()” function (within the module, if it exists).
* This function sets the right parameters (on itself or its parent object) and attaches itself to certain callbacks.
  + For example, objects might have a “click” event. A module that turns something red when clicked, will bind its own functionality to that event.
  + Or, objects might have an “update” call that’s called every frame. A module that needs to be checked every frame, will make sure it’s called whenever that happens.
* Now, during the game, when certain events happen, the right modules and functionality are invoked.

This doesn’t need to be constrained to individual entities, of course. I usually have “global modules” as well, which enable or disable certain systems on a global scale. For example, many games introduce mechanics only later in the game, which means they must not be enabled for the first few levels.

## An example (JavaScript)

Below is a very rudimentary example in JavaScript. It felt like the best language to use, because it’s simple and well-known.

Additionally, you can test the code by opening a console window in your browser, copy/pasting, then executing. To see what happens, call functions on the variable **dog**.

== moduleExample.js ==

Note how I could remove this whole barking functionality, just by removing the word “Bark” from the list of modules. Also note how I could add any other modules, simply by adding them to the list.

Of course, in the real world, this would need to be fleshed out a lot more! I think this will be a good **exercise** for you: try to figure out ways to improve the code above and implement a better module system.

Once you’re done, continue reading!

There will be many commonalities across objects (for reading, initializing and binding modules), which can best be put into a single master object. You want modules to be able to contact other modules. You want modules to get their own parameters, which you can also predefine.

These are the improvements I made in the code below

* I created a single “binds” dictionary, with keys being the sort of event (“click”, “collide”).
* I created a master object called “Entity”. You could inherit from this … but in practice, objects are already completely defined by their modules.
* I don’t predefine modules but pass them as arguments to the constructor. (Extra flexibility!)
* I’ve exposed the actual modules, so other objects can interact with them as well.

Suppose our dog now barks when you click it, and when two dogs collide, they go into a rage depending on the loudness of the barks! As dogs tend to do.

== moduleExampleImproved.js ==

## What should be a Module?

Now the question becomes: what should be a single module? Is “Collectible” too generic, or actually too narrow? Aren’t there many ways to collect something?

The answer is, again, that **each module should perform only one data transformation**. In other words: a module should do exactly one thing, all the time, no exceptions or different logic branches.

For example, let’s go back to that common problem of “multiple inheritance”. Suppose I have two types of collectibles in the game: one which you can grab by touching it, and another which you can only grab by sending a certain spell towards it.

They share one strong commonality: they are both collectibles. But they are very dissimilar otherwise: the way you interact with them is totally different.

One might be tempted to write one module with several branches. It might even be short and easy. A few if-statements, and your module is good to go for all collectibles!

But I’m here to tell you: don’t. A module should do exactly one thing. If you want to add other functionality, use a different module for that. *That’s where the whole power of this system comes from*.

So, give both objects a “collectible” module, which does nothing more than: if “collect” event is fired, remove itself and increment collectibles counter.

Now give the first object a “touch” module, which means it checks whether the player is touching it, and fires an event based on that.

Give the second object a “spell target” module, which checks if it’s been targeted by a spell, and fires an event based on that.

Now you know *exactly* what each module does. There is no branching, no set of possibilities, no hidden decisions being made. Additionally, you can probably reuse those “touch” and “spell target” modules on other objects in the game.

Hopefully, the idea about data transformations becomes clearer. A module transforms certain input data (“hey, we touched the player”) to certain output data (“fire this exact event with these parameters”).

Once you make the module too large, it can (and will) do several data transformations, obscuring the actual functionality.

Once the module says “hey, we touched the player” => “we either convert that input to a touch event, or we are supposed to explode and fly away”, it’s not about a single data transformation anymore.

## When not to use Modules

As with any other programming paradigm, modules create some overhead and layers of abstraction, in return for better code and an easier workflow.

I’ve had several programs that *really* needed to use any sliver of performance they could get. In that case, I did not use modules, but worked with the raw data instead.

“Objects” weren’t really a thing. I simply had several lists containing raw data (such as coordinates, or the color of something, or a Boolean for the state it was in), and worked with those instead. This is harder to visualize, and thus harder to code. But it’s faster than anything, because we don’t create whole structures or hierarchies around the data. This is as “pure” as a program gets.

I find the module system to be the most flexible, versatile, intuitive, easy and quick to write system I’ve ever seen. But you might feel differently or have projects with different needs, and that’s fine. Just give it a try.

## Conclusion

I’ve shown you the power of modules, the reasoning behind using them, and given you some ideas for implementing them. (Once you’ve created such a system once or twice, it should take no more of your time in future projects.)

I’ve also told you to use modules for single data transformations, and nothing more.

That is, however, easier said than done. In the next chapter I want to look at “Pure Functions” and how you can write them.

# Pure Functions

Modules can be seen as the “big building blocks” of your code. You have your world, you might have several subsystems (which can be modules themselves), and you might have entities (which are defined by their modules).

Now I want to go one level deeper. How do we create those modules to do only a *single* thing and do it well? How do we write the codebase to be about single and clear data transformations?

For that we use – what I call – *pure functions*.

Applying these principles will also help you structure your whole codebase to follow this data-oriented principle. It gives a good general philosophy, a way of thinking about your problems and possible solutions, without delving into any specifics.

## Golden Rule: Write as many functions as possible

Before I define pure functions, let me first tell you to **write as many functions as possible**. And I really mean that.

Well-named functions make your code *much* cleaner and easier to comprehend.

Additionally, they avoid repetition and the mistakes that come with it. (You might have implemented two things differently in different places, leading to confusion or errors down the line.)

Sometimes it will feel odd. You create a function for an operation that is only one (or a few) lines. Or you’re not sure whether you want a function for this operation *now*, because it might be used with some slight adaptations in the future.

I’m telling you to write those functions anyway. Code with more well-named and well-defined functions, is always better code.

And now I’m also telling you to **try to write as many *pure* functions as possible**. But what is that exactly?

## What is a Pure Function?

I define a **pure function** as something that does a **single data transformation**.

(Also known as “things should do *one* thing only, and they should do it well”.)

This means that functions will, on average, be very short and perhaps even “trivial” when you look at them. And that’s the whole point.

You’ll know exactly what the function will do, and what it can (or cannot) do. There are no hidden conditions, no different branches, no unintended side-effects.

I’m guilty of violating this principle this big time. My functions used to have names like “spawnEnemy()”, and then proceed to have 2000 lines of code that did many things only vaguely (or indirectly) related to actually spawning an enemy.

Or I’d have a function called “sumLives(a,b)” which didn’t just sum the lives of two players … it also overwrote the previous value for lives, and updated the User Interface to display the new value. A breeding ground for nasty bugs, that is. (I talk about this more under “avoid unintended consequences”.)

## How to know if a function is pure?

**Method 1:** check how many if-statements are in your function. You’d like there to be none, but a few of them are often necessary. If there are, however, many different branches within the same function, you’re in a bad shape.

**Method 2:** checking the size of the function (number of lines and number of arguments). Functions can be extremely short if you’re working with this system. There’s usually no need for functions with more than 3 arguments or 100 lines of code.

**Method 3:** checking data types. If a single function has many different variables, data types, and different return values, there’s probably a way to improve that.

**Method 4:** if you have a class (or some sort of object), check if *most of the methods* use *most of the properties in the class*. If half of the functions uses only half of the properties, then that object isn’t doing one thing and you can split it into two.

This also goes hand-in-hand with modules, as they will usually consist of one (or several) pure functions, making this process easier.

=== pureFunctions.js ===

*Remark*: in this case, I’m assuming the process for creating each monster is quite different. If they are (nearly) identical, of course, it’s not necessary to create unique functions for each of them. Additionally, I might create an “enemy factory” object instead that does exactly what it says: given some enemy stats, convert that data to the actual monster.

There are many ways to do this, depending on the context.

The main takeaway: you can split everything into functions (or objects/modules), such that each thing does one thing and one thing only. This also increases readability, avoids repeating yourself, and gives performance boosts.

## Reducing Arguments

A common expression is “the fewer arguments a function has, the better”

While this is true, it can be misunderstood. You can usually retrieve values another way, instead of passing them into the function as arguments.

That’s *not* what we mean here.

We mean that you give a function *as few arguments as possible* and that the function only uses the *arguments that it’s been given*.

(If you start poking around outside of the function scope, you’re usually not doing a single thing, and you’re making the code very vulnerable.)

A soft guideline is “no more than 3 arguments”, although you can usually apply the “one object trick” to use only a single argument.

What’s that trick? All programming languages should allow you to pass a certain object/list/structure into a function. Give that object all the properties/values you need. Now pass that object into the function as the only argument.

==== reducingArguments.js ====

(Of course, it’s preferable to work with that same structure in many different places. You don’t want to just create an object for the sake of reducing the number of arguments, that type of object should actually be a useful data structure within the code, used in many different places.)

## Avoid Unintended Consequences

Remember my example using “sumLives(a,b)”? Why was that code so bad? Can we pinpoint the exact reason?

I think we can: it’s all in the function name.

When I read a function “sumLives”, I assume it’s going to sum the two numbers I give it, and return that. It does not (heavily) imply the other two operations: overwriting the current values and updating the user interface.

Thus, calling this function will have unintended (and usually undesired) consequences, and it’s not a pure function.

See the code below for more examples.

==== avoidUnintendedConsequences.js ====

## Naming Functions

As you can see, writing pure functions is as much as about the *content* of the function, as it is about the *name* and the *arguments*.

You want the name of a function to state *exactly* what it does, no more, and no less.

Do not fall into the trap of naming a function “sumValues” when it actually does more than that.

But also don’t go overboard (somebody once called this “LEGO Naming”), writing “sumValuesOfTwoIntegersIfBothPositive”. Try to find the shortest, most concise way of communicating the intent of the function.

Most of all, however, you want a function to be **semantically valuable**. In other words, the name and arguments should have concrete **meaning** in actual (English) language.

Try to write names that give you the *context* and *meaning* of what you’re doing, instead of the abstract names that the programming language gives you (list, object, value, integer, etc.)

For example, say you have a piece of software that’s all about viewing videos. Then there are two ways to write the code for opening a video, if the user has permission, and I think you can guess which one is more useful.

==== namingFunctionsSemantically.js ====

(This is also sometimes called **underabstraction** or “Code in the Language of the Domain”. Search for it, it can make a huge difference on your code quality.)

## Conclusion

Hopefully, you understand why I advocate for the *module + pure function* system, and you have clear tools/concepts to implement them yourself.

This was all a bit technical, but don’t worry, now we can get into some more specific details. Once you can more easily apply to pieces of code and see their result.

The next chapter will be even more complaining about if-statements, so read on :)

# Conditionals

Conditionals (or “if-statements”) are the first things you learn when you start programming. And that’s good, because they are a vital tool, especially when you try to wrap your head around the logic of computer programs.

However, this also means that many people are stuck with them. For every problem, they immediately think “well if we use this statement, and check for that condition, and blabla, then it could work!”

And that’s a shame, because there are usually smarter, more powerful ways to accomplish the same tasks.

Because those conditionals are such a large part of programming, especially for beginners, I wanted to write a chapter early on about how to use … well, fewer of them.

## Golden Rule: Errors First

Memorize this mantra (play it in your head using Donald Trumps voice if you like): **“Errors First!”**

Instead of thinking “if conditions X and Y are met, then do A”, reverse that line of thought. It becomes:

“If condition X is NOT met, abort mission. If condition Y is NOT met, abort mission. Now do A.”

This allows us to remove nested if-statements and to split conditionals. Because usually (not always, but very often), conditionals are independent.

For example, let’s say a user wants to log in to your website. What could go wrong? Which errors could occur?

* The username might be invalid
* The password might be invalid
* The user might not exist
* The connection might fail to be established

These are independent of each other. If the password is invalid, you abort mission, regardless of the rest of the program state.

So, you can write each of them as a separate if-statement. And if such an error occurs, you can simply **return** out of the function.

(This chapter was almost called “Return Statement: The Unsung Hero”. It really is more powerful than most students/beginners are taught.)

Once you’ve checked all the errors … there’s nothing more to check! You are *sure* you can execute the thing you want to execute. So you simply do it: you log in the user.

==== errorsFirst.js ====

Another way to memorize this is: **do not nest if-statements** and **always check errors first**.

Hopefully you agree with me that the code is much cleaner, more readable, and less error-prone.

## Removing conditionals

The golden rule above (which is more like a *platinum rule*) leads to cleaner and more readable code. But it doesn’t actually remove if-statements, it just separates them. Can we do better?

Yes, we can!

If you’re smart about it, and perhaps a bit creative, there are usually ways to rewrite if-statements into other statements.

I will show you the most common cases below.

### Technique 1: Become Mathematical

Mathematics is all about finding general formulas and algorithms for problems, because mathematicians are lazy and don’t want to do any extra work, especially not twice. (I can say that, because I am a mathematician.)

This means there is *usually* a mathematical relation that is identical to your conditional.

This is best demonstrated with a few examples:

==== mathematicalConditionals.js ====

I highly recommend looking up Math functions/operators in your programming language. And whenever you have a problem, search online if somebody else already discovered a smarter way to do it.

### Technique 2: Use Arrays/Dictionaries

The very idea of conditionals is to call different blocks of code depending on the situation.

However, as we saw in the previous chapter, you don’t really want that. You want *pure functions* which do a single thing. Which also means you want to call *different functions* depending on the situation, not different blocks of code within the same function.

So, if possible, create an array holding all the functions you want to call, and then simply access the function you want directly.

==== arrayConditionals.js ====

Of course, in this case I’d instead add *modules* to the tiles and call those directly. But hey, I need to keep examples simple and self-contained.

And if you have many effects, you might want to put all those functions inside a single dictionary instead, calling the right key when needed.

But hopefully, you get the idea: you can usually remove conditionals and replace them with direct access to some sort of list or object.

### Technique 3: Boolean Return

If your function is supposed to return a Boolean (true/false) value based on some conditions, you can usually simplify it as follows.

==== booleanReturnConditional.js ====

This might seem a bit trivial, but if you do this consistently throughout a codebase, it makes a huge difference.

### Technique 4: Reuse Known Data

Another technique that is harder to spot, but very useful when you see it, is **reusing known data**.

For example, if you’ve calculated (or retrieved) the value of something early in a function, there’s no reason to grab it *again* later on. It’s easy to forget this, especially if you’ve been away from the code a while (and some time has passed between writing several functions).

==== reuseDataConditionals.js ====

(These examples might not be the best. It was hard coming up with a good set of simple examples for this point, because the use cases for these improvements are already quite advanced. So please contact me if you know a better example.)

### Technique 5: Miscellaneous

Many other methods exist, if you put some creative force behind it. Google for yourself: “programming without if-statements”

These methods usually work by transforming your data structure. By saving your data in a different way, such as an array, you can access and manipulate it without using conditionals.

## Know your Data

In this section I can (finally) give you a specific, concrete example why you should think in terms of *data transformations*.

Many conditionals can be removed *and* performance improved, if you know your data. Or, more precisely, the *domain* of your data.

If you know that 95% of the time the input will be 0, and only 5% of the time it’s a 1, you can optimize that.

How? By using a **default value** and making some assumptions about the data.

==== conditionalsKnowData.js ====

Similarly, if you *know* that some data transformation needs to be executed 99% of the time … why don’t we just abandon the check and execute it always?

For example, suppose we have a button that opens a window in our interface. We could check if the window is *already open*, and if so, not do anything. But when the user presses that button, there’s a 99% chance the window isn’t already open, right? So get rid of the conditional and just make the window visible in any case.

## Why does it matter? Performance?

You might think: “yeah, but replacing if-statements with solutions that are slightly harder to understand doesn’t seem that smart!”

And I’d agree. There are certainly cases where you can and should just use a bunch of if-statements.

However, there are some things to be said about the performance.

If-statements are fast. They should be – they are the building block of all programming. But that doesn’t mean they can’t be a bottleneck. It doesn’t mean your code can’t be even *faster*.

If you have code littered with thousands of conditionals, that will surely impact the results. If those conditionals were (almost completely) replaced with simpler equations and lookups, and smarter data structures, you will surely see performance gains.

Additionally, if you’re working with a *compiled* language in some way, getting rid of conditionals is even better. The computer has a hard time “predicting” the flow of a program, which means a program with many different branches will compile far worse. It might be obvious to *you* how your software works and what the if-statements will accomplish, but that won’t be the case for the “*dumb” computer*.

I admit that these are low-level optimizations, but I think it’s wise to make good coding practices a habit.

On top of that, with our increased attention to conditionals, we can do even more smart things!

## Traveling Conditionals

The *position* of a conditional matters a great deal.

You want to avoid something called **last-minute decisions**. These are hard for compiled languages, unpredictable for human coders, and a waste of performance in general.

What do I mean with that? I refer to a conditional that is used *near the end of a chain of instructions/commands*. In other words, the program is happily doing its thing, until suddenly it must make a very important decision near the end.

==== travelingConditionals.js ====

These can be avoided with a single technique: **move the conditional up the chain**. This can mean one of three things:

* Move it outside of a code block (usually a loop)
* Pre-calculate the value at the top of a function
* Or move it out of the function entirely, allowing the parent to decide.

Especially the last one is very powerful. The parent, the one *calling the function*, usually already knows which function it wants to call. Instead of making a last-minute decision, simply use the data available to the parent.

*Move the conditional up the chain.*

## Conclusion

Always remember to check **errors first**.

Never nest if-statements or make them too long. You can usually split them, move them up the chain, or remove them altogether (with math tricks, array tricks, or default values).

And most of all: avoid last-minute decisions. The earlier a statement is used, the more pure your functions, the easier it is to parse for the computer, and the easier it is to find bugs.

In the case of conditionals: less is more.

**TO DO:** The examples in the **Conditionals** chapter aren’t very good. They might even be bad practice, as they make code more convoluted.

# Readability (Clean Code)

You might disagree with me on a few things up until now. You might say that some techniques make code slightly less readable, or slightly harder to comprehend, or less “generally applicable”.

And you might be right. There’s always something to improve, there’s always a bit of personal preference or dependency on context.

I intentionally suppressed my urge to make previous examples cleaner and more general (or “complete”). To keep the examples concise and to-the-point, but most of all … to allow this chapter to stand out.

(Besides, there seems to be a never-ending debate about “clean code”. People can’t even agree on where curly braces should be placed.)

## Why is this so important?

Because the biggest part of writing code is not the actual *writing*. It’s the *reading*. Reading your own code from months ago, someone else’s code, the inner workings of some library you’re using.

You’re spending way more time trying to **understand** code than actually **writing** it.

On top of that, humans are visual creatures. Much more visual than many people realize.

If we’re able to format code in a visually clear way, we’re making the job a LOT easier on ourselves and others. We can skim over the code *without* missing important details. We can read some complex piece of code like it was just a regular English paragraph.

Think about it this way: if you open a website, and it’s just lines of text stretching the whole screen, without white space, or paragraphs … will you even try to read it? I think not. The information there might be extremely valuable, crucial even, but you’re not even giving it a shot because it’s not visually appealing.

What I’m going to tell you in this chapter, is nothing more than **applying principles from visual design** to your code.

Stuff that belongs together, should visually be together. Stuff that does not have a relation, should be far apart.

Things of great importance should take up great space, whilst minor operations should be one-liners.

This chapter will look specifically at code. If you want actually gain in-depth knowledge about graphic design, I highly recommend my own free tutorials: Design Tutorials (on Pandaqi) == https://pandaqitutorials.com/Design ==

I wrote them for this exact purpose: to introduce people to the world of graphic design in a fun and free way, because visual design is important in *any field*.

## Noisy Code

Lala

(Don’t put stuff on one line or make it as short as possible … just because you can.)

The biggest obstacle in reading/understanding code, is the NOISE. Avoid noisy code. Noise ratio = ratio of useful information against false or irrelevant info.

(Noisy Logging => don’t log everything, have clear guidelines what requires logging and in what format.)

(Poor code is not created arbitrarily or with malice, or by lack of skill. It’s created by system/convention, so change the convention.)

(If you have a comment for a block of code … make it a function. A function is defined as a *block of code* with one purpose.)

(Declare variables close to their usage. Usually means: only declare something when you can immediately initialize.)

(Name functions based on *intent*, on what you *want*, not just abstractly what the function does.)

(The purpose of a *constructor* is not just to initialize an object. It’s to create that object in a *well-defined* state. So you know what data you’re working with and that everything is in the correct configuration. => If there is a problem, it should be found *earlier* => moving up the chain.)

## Columns

Lala

## Curly Braces

Place them so that the *visual design* of your code is helped.

But, even better, most programming languages allow you to omit them if there’s only a single statement after the conditional.

So, try to write your code *in such a way that* there are only single statements!

## Comments

Lala

## The Undrawn Lines

Lala

(Both vertical and horizontal)

(Argument Lists here? Indentation)

## Descriptive variable names

Lala

(Not too short, certainly not too long. Remember: LEGO-naming!)

(Make your names as **fluid** as possible. It takes time for something to settle, to realize what it’s actually supposed to do and what the best structure is. So be prepared to rename variables, functions, classes etcetera as needed.)

## Encapsulate State

Lala

## “Safe” Coding

Too many lines, workarounds, conditions necessary. Use more direct code, with direct names and methods. Use a smarter way/hack if possible. The code only needs to work in practice, not in some magical realm of “maybe the code will need this someday”

**TO DO:** Update my comments on OOP to be about “class and inheritance based OOP” (the way it’s usually implement in modern times)

“OOP to me means only messaging, local retention/protection/hiding of state-process, and extreme late-binding of all things” – Alan Key (inventor of OOP)

**Interesting article** (especially the end “objects are experts; you tell them to do something and they do it for you”): <https://ovid.github.io/articles/alan-kay-and-oo-programming.html>

“We can build a complete programming model out of two separate pieces: the **computation model** and the **coordination model**.”=> You have nice little pieces that do their nice little (computation) jobs. Then you have different pieces that *coordinate* between all these modules.

TIP: Don’t go after trends, or switch languages/libraries/frameworks every few months, or listen to people saying they’ve “discovered” this great new thing.

Programming has been around for over fifty years. During that time, especially at the start, most of the good ideas and philosophies have already been figured out. Go back to the founders of programming, look at the wealth of knowledge produced over time.

(You’ll soon notice that those people saying “look at this great new thing!” are actually pointing at something that was already discovered 40 years ago. And that those early insights are still true today and will probably forever contain at least a bit of truth.)

General stuff:

* Good and meaningful variable names. Don’t make it ultra-short and explain it somewhere else (or use acronyms or something). Only do this if you have some piece of code you’re never going to look again and you don’t want it to take up too much space.
* Write code that expresses intent. Use function names that clearly say what (data transformation) they intent to do.
* “Leave your Code better than you Found it” => whenever you read something, or add/remove something, take a look at the code around and see if you can improve something just slightly.
* Tests are very important. Write quick, clear, easy to access tests for all parts of the system. HOWEVER, don’t do this too early (when working on a game or something), as it just takes a lot of time … and the code might be thrown away anyway.

When explaining an idea, add an **overdose** section: “this is a good idea/philosophy … but what happens when we apply it to EVERYTHING!?”

Good Software = Easy to Understand and Easy to Change

* DRY = Don’t Repeat Yourself
* POLA = Principle of Least Astonishment
* KISS = Keep It Simple Stupid
* Don’t guess whether something improves performance or stability or whatever. Measure it. Measure it again. If it’s not an improvement, don’t use it – keep it simple.
* Work on big picture skeleton, then fill in the details later. (Interface first, implementation later.)

Things to watch out for:

* Don’t overload names or concepts. The same exact word might mean *different* things in different places. (Or a system might be implemented one way in function A, but in a different way at function B.) Bad offenders are “general” words such as *layers* and *controllers*.
* Be careful with dependencies. If possible, you should have “singular directory dependencies” => no circular dependencies (one-directional). Also keep number of dependencies small. Make things as *independent* as possible.
* Organization is Forever. When things aren’t organized well, this chaos will exist forever, and only get worse. But when things are organized, you’ll have a strong core to build upon (and things will stay better organized).

From another video

* Avoid abbreviating variables
* Declare variables close to their usage
* Stop writing zombie code => commented code (because it was old, deprecated, a backup, etc.) that could come alive at any moment!

Things to cover:

* Do not modify stuff in place. Learn the different between copying by VALUE and by REFERENCE.
* Use vanilla code. Syntactic sugar and high-level functions are only good for *hiding* problems. (Unless you are *really* familiar with what something does and exactly how it works.)
* Commenting is extremely useful. However, it also has a nasty issue: comments can be wrong when the code changes (and you forget to update the comments). Ideally, code should be so clear and simple that it can be understood at a glance.
  + In other words, when you’re about to write a comment, take a step back and ask yourself: “can I express this code better so that I don’t *have* to write a comment?”
* Performance (and many other things) are a tradeoff between initialization speed, game speed, memory, and code quality.
* A bit about data structures. (How a data structure informs the code. Choosing a smart structure, will get rid of many problems, conditionals, obstacles, etc.)
* Whatever you want to do, chances are someone has done it before. Check out the different algorithms/ideas. Don’t reinvent the wheel.
* Do not think “maybe in the future the game will need X”, think about the now and the practical.
* **Check out that Godot page about contributing:** some really good stuff there.
* **For Games (mostly)**: split the logic and the visuals. (Where visuals can also be split between menus, GUI and the actual 2D/3D world.) The point is: they are two different datasets with two different purposes, so do not make them overlap/interweave AT ALL.
* Parents usually know which version of a function they are calling.
* Difference between an array of Vector3s, and a Vector3 containing arrays. Know when to use which.
  + More generally: Do not re-generate values you (should) already know. => In fact, you usually know more than you think. Group operations together based on similarity, so you can share results.
* Go analog. By writing/drawing stuff on actual paper, you must think about it more, activate different processes in your brain, and spot errors/solutions more easily. (For example, many writers report printing their books when proofreading, as they catch errors much easily on printed pages than their computer screen.)

“We propose […] that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to *hide* such a decision from the others.”

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." - Martin Fowler

You should not use comments to explain what the code is doing. You SHOULD use comments to explain WHY the code is doing it.

**GREAT RESOURCES:**

* **Game Programming Patterns:** <https://gameprogrammingpatterns.com/>
  + Free web version! Read it!
* 97 Things every Programmer Should Know: <https://github.com/97-things/97-things-every-programmer-should-know/blob/master/en/SUMMARY.md>
* Clean up your visual mess: <http://www.visualmess.com/>

More talks from the guy (Kevlin Henney):

* <https://www.youtube.com/watch?v=FyCYva9DhsI>
* <https://www.youtube.com/watch?v=AbgsfeGvg3E>

Good talk: <https://www.youtube.com/watch?v=ZsHMHukIlJY>

**1) noise code =** too many comments (especially obvious ones), too little activity per line, or too “safe” code (where a smarter hack would work)

**2) unsustainable spacing =** people read in narrow columns; don’t make lines too long. (Also, use spaces instead of tabs, wastes less space. Use visual vertical whitespace to SEPARATE ideas.)

What is clean design? One that supports visual thinking so people can meet their informational needs with a minimum conscious effort

Keep ideas that belong together visually together!

Line up things on the LEFT. (We start reading from the left. Don’t bunch up things/gather them on the MIDDLE or RIGHT.)

For example, parameter list:

**NO:** public int howNotToLayoutAMethodHeader(int firstArgument,  
 String secondArgument)

**YES:** public in ensureArgumentsAreAlignedLikeThis(  
 int firstArgument,  
 String secondArgument);

**3) lego naming** = do not just concatenate many different words. Makes for long terms, and meaning just disappears. We have names for things, usually small ones.

**BAD:** ConditionCheckers class with method checkCondition()

**BETTER:** Condition class with method isTrue()

By context, or definition, you usually already know what something is or should be. Don’t append it to the end. (Things like “Exception”, “Object”, “Component”, “Part”, …)

**4) underabstraction** = Code in the Language of the Domain

Use terms, terminology, ideas, concepts, etc. that match the domain/theme/concept of what you’re working on. Don’t keep it abstract and dumbed down, it only makes the code less meaningful.

It should read like an English sentence. For example:

**BAD:** if (portfolioIdsByTraderId.get(trader.getID()).containsKey(portfolio.getID()))

**BETTER**: if(trader.canView(portfolio))

**5) unencapsulated state** = don’t dig into the path of an object.

**BAD:** list.Items.Add.blabla.dede.hihi(“Iets”)

**BETTER:** list.Add(“Iets”)

Properly encapsulate/protect stuff that should be private, inside a bubble. But expose what needs to be exposed (like a “user interface” of an object).

Keep the abstraction properly private.

**6) getters and setters**  = avoid use?

Get and Set both have enough definitions/meaning to fill 30+ pages. When you code, you want to use something with a *precise* and *specific* meaning.

Don’t put “set” and “get” in front of EVERYTHING. Especially don’t automate getters and setters.

“Get something” is an imperative with an expected side effect. You get something … but you lose that at the other end.

**7) uncohesive tests** = “Test Driven Development” means that **tests** must **drive** the development. Tests must be clear specifications of the required functionality. Tests that are not written with their role as specifications in mind are confusing to read.

Organize tests as “Scenarios” you could encounter in real life.