EARLY DRAFT!
Rev 86 (22 June 2023)

with code samples in TypeScript

THE CRAFT OF SOFTWARE DEVELOPMENT – A PRACTICAL INTRODUCTION

SEBASTIAN FELLING

The Craft of Software Development
— A Practical Introduction

Critical and honest comments on your reading experience as well as suggestions for future improvements are highly appreciated. Please send your comments via e-mail or check the book's website to get in contact.

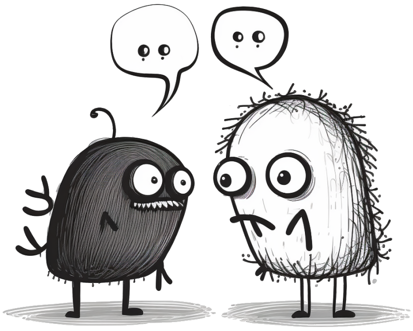
www.craftofsoftware.dev
seb@craftofsoftware.dev

Book Version: Rev 86 (22 June 2023)

Generated on Thu, 22 Jun 2023 07:38:50 GMT

Commit #n.a. / Renderer v1.7.1.0

Profile: Debug-Digital



To Alicia.
Wherever life may take you,
your loving heart
and your sharp mind
will always guide you.

To all the teachers and scientists
teaching us critical thinking,
independence, humbleness,
and, above all, humanness.

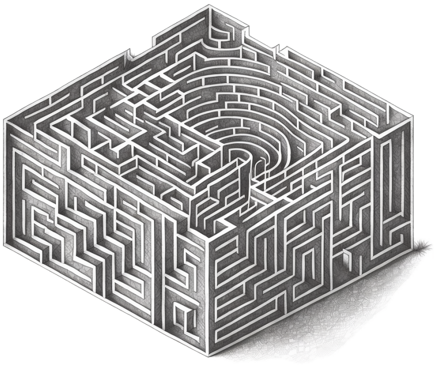


TABLE OF CONTENTS

CHAPTERS

INTRODUCTION

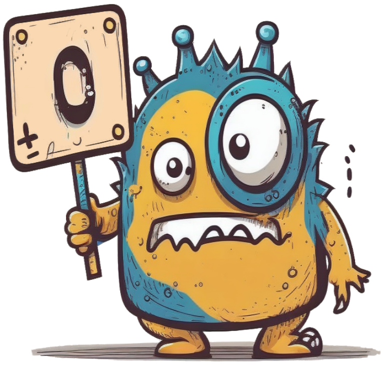
HANDS-ON EXERCISES

APPENDICES

KEYWORDS

ILLUSTRATIONS

LISTINGS



WHAT THIS BOOK IS ABOUT AND WHO MIGHT WANT TO READ IT

The introductory chapter discusses the motivation behind this book and its target audience. It also lays out the contents of its parts, introduces some programming lingo, and answers questions the readers might have. [A1]

WHAT IS THIS BOOK ABOUT?

This book teaches the craft of software development through practical exercises. It exemplifies and explains the principles and best practices of the field as well as the reasoning behind them. The key learning objective is writing maintainable software in the real world through an informed application of such principles and practices.

WHAT IS THE TARGET AUDIENCE?

The following chapters are written for readers who are already familiar with the very basics of programming but seek practical guidance on learning the techniques necessary to apply their knowledge to solve real-world problems.

I'm sure your next question is: Am I ready for this book? Before we discuss some of the defails of that question, let's test your general programming language. If the code in ➔ Listing 0.1 makes sense to you (independent of whether you already know TypeScript or not), you are most likely ready for this book:

sample-code.ts

```
class Mail
{
    constructor(
        private _recipient: string
    )
    {}

    public get recipient()
    {
        return this._recipient;
    }
    public textBody: string;
    public subject: string;
}

const mail = new Mail("seb@craftofsoftware.dev");
mail.subject = "Greetings!";
mail.textBody = "Hello, Seb!";

mail.send();
```

Listing 0.1: Some sample code to self-assess your programming knowledge.

IS THIS THE RIGHT BOOK FOR ME?

The target audience of this book is relatively broad. Thus, a few questions might help you decide if this is the right book for you:

✗ You have never written a single line of code but would like to know how to do so? **This book is too steep of a learning curve for you.** But don't worry! Find an introduction for beginners ➔ FURTHERREADING , learn the basics of programming, and then come back here. Once you have learned to write simple applications (like the famous Tic Tac Toe game, or a To-Do List), you are ready to dive into the chapters of this book.

✓ You already know how to write a simple FizzBuzz or Tic Tac Toe app (a text-based console or JavaScript app is enough) but fail at writing more sophisticated solutions because you don't know where to start, or because complexity amounts and you end up with ➔ Spaghetti Code? **This is your book!** It will show you how to plan and execute real-world projects writing clean and maintainable code that you (or your colleagues) will find easy to read and extend.

DEVELOPER SPEAK

SPAGHETTI CODE is a derogatory word for code that is hard to reason about because it lacks structure. It tends to be needlessly convoluted, filled with unwieldy control flow logic, and generally mixes concerns that had better be separated. A lack of abstraction or overuse of the wrong kinds of abstraction usually come with it.

✗ You are pretty good at programming and want to learn all there is about TypeScript? **This is not your book, either.** While it does use TypeScript in its source code listings and projects, it only uses it as a vehicle to get practical aspects across. This is not a book teaching the peculiarities of TypeScript!

✓ You have learned the basics of JavaScript or TypeScript and would like to create your first very own applications but are stuck in ➜Tutorial Hell? You know how to swim but lose your confidence in deep waters because of a lack of guidance? **This is your book, too!** It will show you how to start with a given problem and work your way towards a fully functioning solution in code.

DEVELOPER SPEAK

TUTORIAL HELL is what novice developers frequently find themselves in. After having learned the basics of programming through web tutorials and videos online, they venture into their first solo project (without any guidance from a tutor) and get stuck because there is no clear way forward. Instead of pushing through, they resort to yet another tutorial, thus ending up in a vicious circle that is hard to escape.

If you have done some basic programming before but in a language other than TypeScript (the language we will use here), you are good to go as long as you are willing to learn it. Don't worry, we won't go deep into the paculiarities of types. In fact, we will only use a small subset of all of TypeScript's features (classes, properties, inheritance, and the like). To help you get up to speed, there is a short introduction to the language in the appendix: ➜Appendix C: Short Introduction to TypeScript (p. 73).

Should you still feel unsure about whether this is the right book for you, check out the ➜Table of Contents (p. 4) as well as the keyword index there. Both should give you a good idea of what to expect.

WHAT KIND OF CONTENT CAN I EXPECT?

This is what we will be covering:

- The introductory part (Part I: Introduction) lays the groundwork for the coming chapters and discusses general notions of the field of software development. It also debunks some myths about what development or developers are.
- This is a teaching book, so expect lengthy discussions with lots of exercises and examples. These will make up most of the entire contents (Part II: Hands-On Exercises). I try to keep discussions on point, so I will occasionally push lengthier comments and food for thought into the ➜Appendix A: Endnotes (p. 61) when these are deemed important but might distract from the current discourse. These notes are referenced in superscript using an **Ax** notation, where **A** stands for *Appendix* and **x** stands for the number of the note, e.g. *A1*, *A2*, etc.
- No teaching book would live up to its name without exercises. You can expect a lot of exercises in this book. You will stumble upon these throughout the text in the form of ✎ prompts marked with a pen. They prompt you to stop reading immediately (really, stop immediately!) and start acting, e.g. writing code, taking notes, do some brainstorming or thinking. When you think you

are done (or stuck), you can resume reading and compare your results to the results offered in the discussion thereafter. I strongly encourage you to do these exercises, as they are the most important part of this book. You will learn a lot from them!

- Terminology is key, so expect a good deal of "developer speak", which is explained in colored boxes the first time a term is used. The concepts and ideas contained therein are usually accompanied by examples, though sometimes these can also be found in the context around the colored boxes. You can find an alphabetical listing of all these keywords in the ➜**Table of Contents** (p. 4), which also lets you jump to these directly.
- There's only so much you can discuss in a single book. For those that seek further material worth consuming, I list additional resources (e.g. books and links to web sites) in "further reading" boxes that are referenced using **prompts marked with a mortar board** ➜**FurtherReading** . These boxes can generally be found at the end of every chapter. Most of these come with a link where you can read free content or at least download sample chapters of commercial books.
- Finally, key principles of software development are referenced using **visual marks with a compass** . This is done to emphasize their importance as they guide us through the often tough course of decision making.

Programming, Software Development, Computer Science: What's The Difference?

This book takes a pragmatic approach to software development. We are not going to engange in lengthy theoretical discussions (although theory is touched upon for explanation and to fuel engagement). Accordingly, we won't make a difference between programming and software development. For all practical reasons, they are the same thing.

One point deserves special mention, though: We are not dealing with computer science here, or at least not with the academic subject thereof. Instead, we will apply its practical implications in the context of software development, which is understood here as a practical craft, not an academic field. While computer scientists are concerned with academic, largely theoretical questions to further the science itself, software developers apply such knowledge to create solutions outside of academia.

Having said that, software development and computer science are not at odds with one another. Rather, they complement each other and share great overlap. Computer science is an ideal base to start a career as a software developer. However, it is not the only way to do so, nor is it better than any of its alternative ways of getting into the field. [1]

---

1. Gatekeeping is a vanity that holds no value. Instead of discouraging people, we will focus on enbling them. Computer science courses don't teach magic. Anything that is taught in curricula at colleges and universities can just a well be learned outside of them.

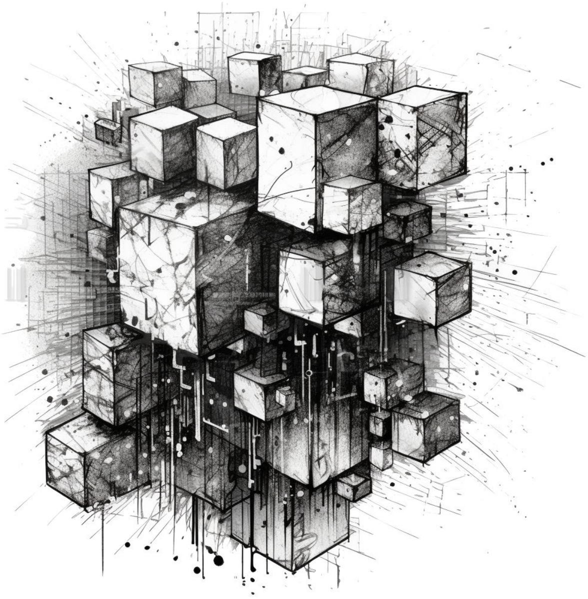
IS SOFTWARE DEVELOPMENT REALLY A CRAFT?

There are many reasons why software development can be considered a craft, and I find them all convincing.

If we interpret craft as something that is produced skillfully by hand, then software development fits the bill perfectly. Our hands type code and draw diagrams, they guide our mouse or pointer devices to create docs and illustrations, all of which play an important part in our everyday job.

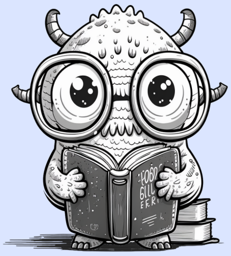
A craft is a practical job that cannot be learned merely from theory. While theoretical knowledge (such as computer science) is an ideal basis for learning software development, the craft itself can only be learned through hands-on experience, which takes years to gather and does not stop even if we feel comfortable and confident in what we are doing.

One last point worth mentioning: Crafts are passed on from experienced craftspeople, who teach those that are willing to learn them. An ideal way to learn a craft is through a mentorship. Software development is a great match as its practice is heavily based on experience, which can be passed on from a mentor to a mentee. Viable alternatives to mentorships exist in the form of group learning on any of the 🎓 plenty coding platforms ➔ FURTHERREADING , most of which offer free contents and exercises.



Further Reading

If you are completely new to programming, the following sources introduce you to the very basics:



How Computer Programming Works. Daniel Appleman. Apress. 2000.
https://link.springer.com/book/9781893115231

How Computers Work. Roger Young. Apress. 2. 2009.

Once you have covered the basics, it's time to dive into your first programming language and the essentials of data structures and algorithms. These are great picks that are easily approachable:

C# Programming for Absolute Beginners: Learn to Think Like a Programmer and Start Writing Code. Radek Vystavěl. Apress. 2021.
https://doi.org/10.1007/978-1-4842-7147-6

Automate the Boring Stuff with Python. Practical Programming for Total Beginners. Al Sweigart. No Starch Press. 2nd Edition. 2015.
https://automatetheboringstuff.com

Codeless Data Structures and Algorithms: Learn DSA Without Writing a Single Line of Code. Armstrong Subero. Apress. 2020.
https://doi.org/10.1007/978-1-4842-5725-8

Mentorships are a great way to learn the craft of software development. If you do not have access to a mentor, group learning sites can be a viable alternative:

- ➦https://www.codecademy.com/
- ➦https://www.freecodecamp.org/
- ➦https://www.w3schools.com/
- ➦https://www.codewars.com/

TRUTHS AND MYTHS ABOUT SOFTWARE DEVELOPMENT

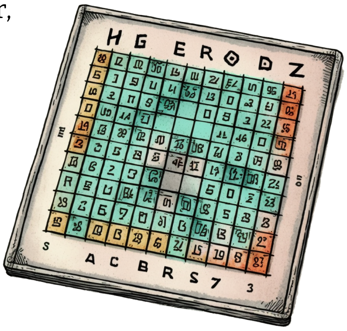
This chapter discusses some false assumptions people frequently make about software developers and their job. We will also debunk myths and peek into the everyday job of a common developer.

Software development as both the subject of intellectual exploration and professional means of income is a vast and incredibly interesting area. Thus, it is pursued by an increasing number of people through academic and vocational training, or simply as a hobby. Novice developers take to coding schools, bootcamps, and the social networks searching for answers to their many questions. What is so striking about these is the fact that many of them seem to be based on frequently recurring distorted views of the field, and sometimes plain wrong assumptions.

PROGRAMMING IS FOR THE TECHNICALLY INCLINED AND MATHEMATICAL GENIUSES ONLY!?

No, it is not. Plain and simple. At least generally it is not. Of course, there are branches that require a fair amount of solid math skills and technical aptitude. If you want to work in fields like cryptography (the field where developers deal with encoding and decoding secrets), machine learning, or computational finance (where developers help solve practical problems in finance), solid math skills is a requirement. In many other branches, though, math can and will definitely help you in your everyday work but a solid high school-level understanding of it will get you a long way. And the rest you can pick up and learn through daily practice → FURTHERREADING .

In any case, you need not be a math genius to become a good developer, so don't be discouraged by people saying so. And, more importantly, don't be afraid of math! If anything, math is your best friend when it comes to learning the formal aspects of programming and will help you acquire a whole new language for talking about real-world problems.



Developers Are Loners That Don't Like To Socialize And Are Bad At Communication!?

Nothing could be further from the truth! Believing that developers sit in front of computer screens all day and avoid social contact is a huge misconception. Indeed, software development is an act that involves *a lot* of social contact. That's because hardly any software is written by a single person, most projects are a team effort.

Those teams can range from anything of two persons doing ➦ Pair Programming in the same room to hundreds of developers working across the globe. Just imagine the kind of communication effort that it takes to get a project through the finish line, and you begin to understand why a good command of one's mother tongue (and the English language, too, in case they are not the same) is an absolute must.

A key aspect of communication is language. As we will see shortly, language is a frequently underappreciated skill that needs to be practiced and honed (a fact we tend to lose sight of because, after all, language comes for free, right?). Developers need to be rock-solid **communicators**, which means that they need to be rock-solid masters of their language. Think about it: Code is largely text, documentation is largely text, even project management is primarily based on written language. Whatever products we create, they most likely involve text as a medium of relaying information.

Developer Speak

Pair Programming is the act of programming as a shared, joint experience done by two developers at the same time on the same codebase. Usually the two individuals take turns so that one of them writes code while the other one watches. Discussing and reasoning about the whys and hows of said code is an important aspect of it. The benefits of pair programming are a deeper understanding of the codebase, higher quality code results, and learning new programming techniques.

Developers Write Code All Day!?

Among all the stereotypes discussed here, this is probably the one where expectation and reality clash the most. Of course, a developer's responsibility is to write code. But that doesn't mean they write code *all day*. In fact, the average developer probably spends a maxium of half their workday writing code, if not much less (the actual number depends on organizational complexity, e.g. roles and team size, of course). The rest of the day is spent on equally important jobs such as discussing requirements and implementation, planning releases, reviewing code, writing documentation, to name just a few.

And then there is also the act of *reading code*, which is an art in and of itself. It can be argued that reading code is more important for a developer than writing it. As some estimates have it, every line of code is written only once but read at least ten times. From a quantitative point of view, developers spend about ten minutes reading for every line of code that takes them one minute to write.

Consequently, we better focus our efforts on making code easily readable and understandable instead of opting for an easy and quick writing experience. Sadly, the reality out there is sometimes the exact opposite and developers (both novice and experienced) optimize code for their writing experience while neglecting the effort needed to read it.

DEVELOPER SPEAK

LINES OF CODE (LOCs) refers to a quantitative measurement of source code as the total amount of lines of text over all code files of a project. It is a notoriously unreliable indicator of a developer's productivity, which unfortunately doesn't keep some management folks from using it as such anyway.

Developers write code for machines!?

One might be inclined to assume this, but it's generally not true. Yes, the ultimate need to write code comes from the fact that machines don't understand natural language (a kind of code itself, if you come to think of it). What kind of language do machines understand instead? As it turns out, they are pretty good with digits and computations. But this is where human beings usually fall short, we're much better with natural words. [A2]

If you write low-level code, that is code in a machine language or very close to it (e.g. Assembly), then you are writing code for the machine. This kind of code tends to read rather difficult (for us human beings) due to its low-level details and minute style. This, of course, affects the way we maintain such code. Low-level code works well for hardware-related operations but is rather cumbersome to use for anything closer to the more abstract application-level.

Higher-level languages (such as JavaScript, TypeScript, Java, C, C++), on the other hand, use the power of abstraction and work well for application-level development. These languages overcome the shortcomings of machine language and were specifically created to be easily readable by humans, not machines. This is also why they do not run on the 'bare metal' directly but are translated to machine language either statically (compiler) or on-the-fly (e.g. through a Virtual Machine).

Developer Speak

Abstraction is a key concept in software development. It is on one side of a continuum that has implementation ("concretion") as the opposing extreme. Abstraction generally refers to a view of things from a higher-up perspective. One that is more general and leaves out the lower-level details.

For example, if we look at a school setting and see the students Alice and Bob, who are taking German und French classes with Frau Müller on Thursdays, we are dealing with a rather concrete view of 'school'. The opposing view, the abstraction, would be to look at it from a functional perspective: An environment where students learn languages. The abstract view gets rid of irrelevant details in favor of a more holistic picture.

The problem with high-level languages is that writing readable code does not come for free and is certainly not a given. It is akin to the craft of authoring books and literature: Writing alone is a necessary requirement to become a good writer, but it is not a sufficient one. Like any craft, mastering it takes practice. Thus, writing code that is formally correct to be run by a machine, functionally correct to fulfill its requirements, and still readable and maintainable by humans, is what the job of software development is all about. To say it with the words of an experienced developer:

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
>
> — **Martin Fowler (Refactoring: Improving the Design of Existing Code)**

Throughout this book, we will explore many practices and principles that help us write readable and maintainable code.

There is only one "Right" way of doing sofware development!?

We live in a complex world and the field of software development is no different. There's an old adage among developers stating that "there are no silver bullets" [2] , and that little gem holds up rather nicely.

In the same mood it can safely be stated that there is no single "right" way of doing things in our field. We always have a choice among different, competing techniques and technologies, so there are always nuances to be considered. [A3] This is both a blessing and a burden at the same time: because it forces us to think less in terms of right and wrong, and more in terms of gradual and highly nuanced ➚ trade-offs.

---

[2] The WikiWikiWeb attributes the adage to the famous Frederick P. Brooks, Jr., see ➚ https://wiki.c2.com/?NoSilverBullet

Developer Speak

A **TRADE-OFF** is simply a compromise, a deal between two (usually opposing) goods: you get one thing in exchange for another. Or, more realistically speaking, you get more of one thing in exchange for less of another. To give a simple example, every purchase we make is such a trade-off: We give money and get a certain commodity in return.

Software development is all about trade-offs because software systems touch on so many qualities that compete with one another. An advantage we gain at one end is likely to be paid for with a disadvantage at another. The key is finding the right balance between these two that allows us to achieve our goals while not putting too many obstacles in our way. A4

Certainly, broad generalizations in tech have a strong appeal — especially among novice developers, who lack the experience and deep understanding of their choices and thus seek simple answers to their questions. [3] But that doesn't make them right. Sadly, many developer veterans resort to them, too, partly for a lack of willingness to explain and discuss things, partly because they really believe in them.

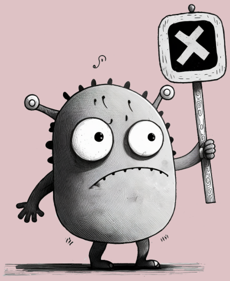
And then, there are enough developers out there opinionated to a point where they refuse to accept views opposing their own experience. While that might be human and understandable, there is no point in defending a single truth like a religious zealot. Instead, we need to accept that every developer speaks from a single point of experience. No two developers are alike and experiences vary greatly, thus we ave to come to terms with the fact that **there are multiple truths out there**. Some go well with one another, others are downright mutually exclusive. That's okay, let's embrace differing (even opposing) views and pick the tools and practices that work for us. For if we do opt for simplistic views and simple answers, instead, we might just end up with another case of Maslow's hammer. [4]

---

[3] Think of the common questions on social media along the lines of "What is the best programming language/backend tech/IDE?"

[4] Also known as *The Law of the Instrument*: When all you have is a hammer, everything becomes a nail. See: ➜ https://www.wikiwand.com/en/Law_of_the_instrument

WORD OF CAUTION



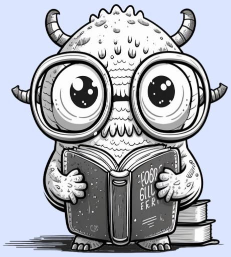
The following chapters are the result of over twenty years of developing software in various contexts and teaching the craft for over five years. The lessons I share here should be considered advice that has proven worthwhile and useful on *my* journey. I am not proposing anything as the ultimate truth or as a solution without alternatives.

Rather than following my advice blindly, I'd like you to consider it critically and see if it works for you. I am optimistic you can find great value in much of the content presented here but might occasionally opt for alternative solutions. That's absolutely okay, because only you know best what works in your situation.

Should you come across parts where your experience differs greatly from mine, please do not hesitate to contact me and let me know. I'm grateful for honest feedback and I trust we can have lots of wonderful discussions based on mutual respect and a passion for coding.

FURTHER READING

In case you want to brush up your high school math, here are a few sources that look at math from a developer's perspective:



- **Good Math. A Geek's Guide to the Beauty of Numbers, Logic, and Computation.** Mark Chu-Carroll. The Pragmatic Programmers, LLC. 2013.
  https://pragprog.com/titles/mcmath/good-math/
- **Math for Programmers. 3D graphics, machine learning, and simulations with Python.** Paul Orland. Manning Publications Co. 2020.
  https://www.manning.com/books/math-for-programmers
- **Doing Math with Python.** Amit Saha. No Starch Press. 2015.
  https://nostarch.com/doingmathwithpython

PROJECT: A SIMPLE MARKDOWN PROCESSOR

It's time to dive into some code: Our first hands-on project involves a simple Markdown processor. It will give us ample opportunity to explore practical approaches to solving real-world problems. We will cover some best practices and introduce basic terminology of modern software development.

DEVELOPER SPEAK

Markdown is a lightweight markup language, i.e. a way to enrich plain text with structural information and visual formatting. It's so simple in form that it can be considered an anti-markup (hence the name Mark*down*).

Markdown was invented by John Gruber and Aaron Swartz in 2004 as a way to allow users "to write [formatted text] using an easy-to-read, easy-to-write plain text format, then convert it to structurally valid [HTML]" [5] .

Since Gruber's and Swartz's first proposal, Markdown has evolved into various flavors, and multiple specifications exist. *CommonMark* has become the de-facto standard in tech. [6]

WHY REINVENT THE WHEEL?

Under real-world conditions we would most likely *not* implement our own Markdown processor. The common adage is: **Don't reinvent the wheel**. And indeed, there are plenty of Markdown processors out there. When faced with a technological problem, it's good practice to look for existing solutions first before embarking on writing our own. If we can use what is already available, why waste time and effort on re-implementing it?

Well, it turns out there are a few good reasons that justify reinventing the wheel. ✎ Can you think of some?

---

[5] See https://daringfireball.net/projects/markdown/

[6] See https://spec.commonmark.org/

A very good reason is: *for learning purposes*. Practice makes perfect, and that's where re-implementing solutions to common problems holds a lot of value. If a plethora of solutions already exist (preferebly in the language of your choice), then chances are you can go and look for help, or find inspiration, or learn from other people's mistakes, and so on. That makes for a pretty good learning experience where you can get your hands dirty, and still have a safety net. And that's exactly what we are doing here, too. So, let's get our hands dirty!

Picking A Development Environment

We won't spend any time setting up our work environment here. I trust that you are familiar with your needs and have already picked your favorite code editor or IDE. If you are unsure what environment to pick, please have a look at **Appendix B: Setting Up A Development Environment** (p. 72).

Formatting Text With Markdown

Markdown is a popular markup language used in technical writing and on blogs. Even though it is concise, simple, and easy on the eye, it still allows for a broad range of text formats. Unlike markup languages (e.g. HTML, XML), which can easily look convoluted, its intention is "to be as easy-to-read and easy-to-write as is feasible" [7] .

Have a look at the Markdown file in **Listing 2.1**. It shows plain text that is interspersed with special characters. These characters control the visual formatting of the text.

`markdown-example.md`

```
# This is an example of Markdown

Here comes a simple paragraph.

We can also create lists:

* This is a list item.
* This is another list item.

Ordered lists are possible, too:

1. This is the first item.
2. This is the second item.

And here is another paragraph containing *italic text* and a [link](https://www.craftofsoftware.dev/).

## Here is a level-2 heading

We can also add **bold** text.
```

---

[7] See **https://daringfireball.net/projects/markdown/syntax**

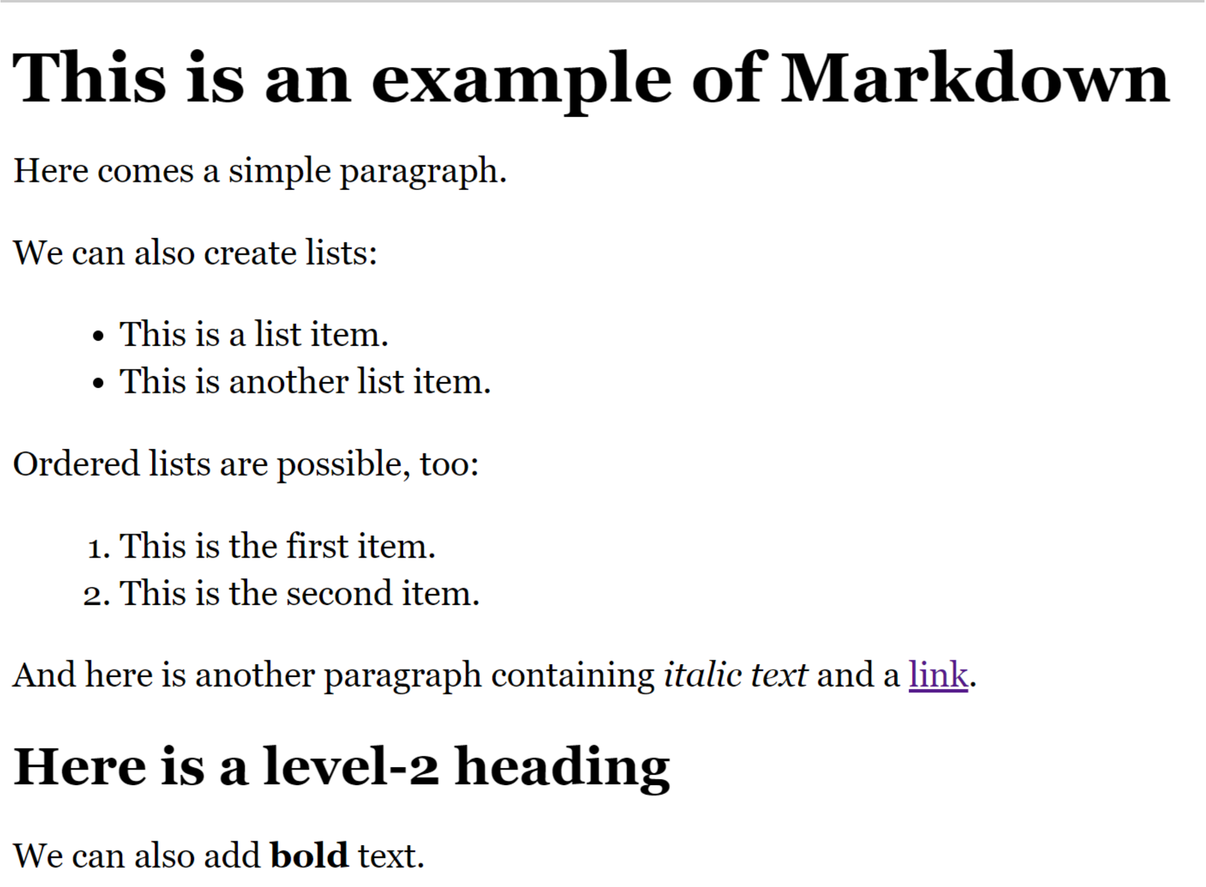
Listing 2.1: A sample Markdown file

Running the plain text through a Markdown processor [8] presents us with the outcome we see in Image 2.1

Comparing the Markdown source with the visual results, the semantics of the special characters should become apparent. **Can you infer those semantics?**

This is how Markdown works: Lines beginning with `#` and `##` are rendered as primary and secondary headings, respectively. Lines starting with `*` are turned into list items, whereas those starting with numbers like `1.`, `2.`, etc. create numbered list items. Notice that the trailing single space in all those cases is important as without them the lines would be just recognizes as regular text.

So far, we have listed block-level elements (based on lines). There are inline elements, too: Enclosing text within `*` renders it italic, `**` makes it bold. Even links can be created using the `[Text](https://url/)` pattern.



This is an example of Markdown

Here comes a simple paragraph.

We can also create lists:

- This is a list item.
- This is another list item.

Ordered lists are possible, too:

1. This is the first item.
2. This is the second item.

And here is another paragraph containing *italic text* and a link.

Here is a level-2 heading

We can also add **bold** text.

Image 2.1: The rendered Markdown sample This is how our Markdown file is rendered after running it through a processor.

---

[8] That's a program that translates Markdown text into visuals, e.g. HTML. Dingus is a good example, see https://daringfireball.net/projects/markdown/dingus.

The translation of Markdown into formatted text looks straightforward. Let's see if we can come up with a solution in code that takes Markdown text as input and spits out the corresponding text formatting. For the purpose of this exercise, we will translate Markdown into simple HTML.

Note that we won't cover the entire CommonMark specification since such an endeavor would be beyond the scope of this book. We will limit our implementation to the features shown in Listing 2.1, that is headings (primary and secondary), paragraphs, lists (unordered and ordered), links, and italic/bold text.

A Core Principle of Software Development

Before we set out to write our first lines of code, let us quickly discuss a core principle of software development that shall serve as an invaluable guideline for our work in general, and the coming chapters in particular:

> Complex, working systems evolve from less complex, working systems.
>
> — **Gall's Law (serving as a principle of software development)**

Gall's Law has, in fact, not originated in software development but in the field of systems design, a school of thought that has great overlap with software engineering. It serves as such an important guideline that it is not too far-fetched calling it a "principle of software development". Let's take it apart and see why it is so important.

The word "system" refers to a non-trivial software solution. Any codebase we write can be described as such a system. These systems should ideally be "complex" *and* "working". The combination of these two adjectives is key here. If we were just talking about "complex systems" alone (without the "working" part), we would include *failing* systems. Indeed, creating just complex systems is an easy thing to achieve as complexity is naturally given in non-trivial codebases and (without proper safeguards in place) failure is an all-too-common consequence. But such failing systems are not what Gall's Law is about.

"[C]omplex, working systems" are also not merely *working* systems, which (again) could be pretty easy to achieve if such systems were just trivial in nature. Obviously, the challenge here is the combination of a system that "works" *despite* all of its inherent "complexity".

There is a second aspect to the principle that deserves discussing: "Complex, working systems" do not appear out of the blue. They are also not mere coincidences or whims of nature. Instead, those systems evolve in a piecemeal fashion: iteratively und incrementally.[A5] They are not born out of overly complicated specifications but fall into place through disciplined trial-and-error, through informed decision-making, designing, and testing under a tight feedback loop.[A6]

Now, what makes Gall's Law so valuable to inexperienced developers? Its logic implies that every complex, working solution ultimately started out as the simplest, working version of itself.

Unsurprisingly, this is exactly how we build software in real life: starting with the simplest solution, then adding value incrementally and iteratively.

Unlike common belief sometimes has it, the process of writing software is not linear. Developers don't write solutions from cover to cover, if you will. Far from! Rather, the entire process works iteratively, where each iteration looks at the existing codebase, rearranges some parts, rewrites other parts, deletes again other parts entirely, and adds completely new parts. Yes, you heard that right: Sometimes, in order to go a step ahead, we first have to go two steps back! If we realize that something that was written previously doesn't work out, we are well advised to consider deleting or replacing it. There is nothing wrong with that. Indeed, being afraid of deleting code is like writing an essay without being allowed to use the backspace key. It's a recipe for disaster.

The idea of writing a solution linearly from start to finish is an idealized notion that does not hold for non-trivial software systems. So, the key takeaway here for new developers is: Always start with the simplest solution imaginable, and improve it one step at a time. [9]

STARTING WITH THE SIMPLEST SOLUTION IMAGINABLE...

We will follow **Gall's Law** rigidly, so we start with the simplest solution imaginable and let it grow gradually. To start off, we want to write some logic that takes a single line of Markdown (a simple paragraph) as an input and generates the corresponding HTML.

The simplest meaningful logical unit in a high-level programming language such as TypeScript is a function, so this is what we will use for now. We need a simple function that receives as input some Markdown like `Here comes a simple paragraph.` and returns the corresponding HTML `<p>Here comes a simple paragraph.</p>`. **Write such a function** . [10]

When done, compare your solution to the function `toHtml(...)` in **Listing 2.2**:

markdown-processor/processor-v1.ts

```
1  function toHtml(markdownLine: string): string
2  {
3      return `<p>${markdownLine}</p>`;
4  }
5
6  const html = toHtml("Here comes a simple paragraph.");
7  console.log(html);
```

Listing 2.2: Parsing a simple markdown paragraph

There's not much going on code-wise. All we do is wrap the input string in paragraph tags (line 3). It is worth mentioning that we check the output (the HTML code returned from our function) manually by logging it to the console (line 7). This kind of poor man's **Unit Testing** will suffice for the time being, though we will find a more convenient method of testing soon.

---

[9] This very idea is also the premise of a practice called 'Test-Driven-Development', which we will read about later.

[10] For the remainder of this chapter, we will assume that the Markdown we feed into our functions does not contain any special characters like `<` or `>` that we would otherwise have to escape in our HTML.

Developer Speak

Unit Testing is the automated (code-based) testing of source code usually done from within the development environment.

Unit tests are pieces of code that quickly execute small, meaningful parts ("units") of a solution to assert that their behavior is as expected.

For example, when writing a tic tac toe game, a unit test would arrange a new round in a state where player X has already crossed two consecutive cells and is about to start a new turn. The test would have the player act so that they cross the third consecutive X. The test would then assert that the game logic is handling this state correctly by declaring X the winner of the match.

… And Extending It One Feature at a Time

Let's add a new feature: Processing headings. Whenever there is a line that starts with a single `#` followed directly by a space character, we want our processor to return a primary heading (an `h1` in HTML). **Modify your solution so that it handles both simple paragraphs and primary headings.**

Given our previous solution, we just rewrite the function body as in Listing 2.3:

markdown-processor/processor-v2.ts

```
function toHtml(markdownLine: string): string
{
    if(markdownLine.startsWith("# "))
    {
        return `<h1>${markdownLine.substring(2)}</h1>`;
    }

    return `<p>${markdownLine}</p>`;
}
```

Listing 2.3: Extending our code in order to support headings

The code in Listing 2.3 receives as input `# This is an example of Markdown` and returns `<h1>This is an example of Markdown</h1>`. We make sure the code works as expected by adding another test, see Listing 2.4.

markdown-processor/processor-v2.ts

```
const htmlForHeading = toHtml("# This is an example of Markdown");
console.log(htmlForHeading);
```

Listing 2.4: Adding a new test that checks if our headings support works

Our new feature is working. And more importantly, our previous feature (paragraph input) still works as expected, too. Remember that we keep *adding* tests, so all previous tests will be run together with all the new ones we add.

This is good progress but the use case is still very limited. Both solutions only ever handle a single line of Markdown. Let's see if we can come up with a solution that can handle multiple lines. From now on, we will add a test first and then add code to make the test pass. This gives us a better starting point for writing code: Figuring out a good use case and example of what we need *before* coding out the solution. So, let's add the test we see in ➔**Listing 2.5**:

`markdown-processor/processor-v3.ts`

```
30 const html = toHtml(`# This is an example of Markdown
31 Here comes a simple paragraph.`);
32 console.log(html);
```

Listing 2.5: Adding a new test that checks if multiple lines are supported

If we ran the test now, it would produce something along the lines of `<h1>This is an example of Markdown\nHere comes a simple paragraph.</h1>`, which is not what we want. This happens, of course, because we still have to extend our code and implement the new feature. ✎ **Modifiy your solution now to make the test pass!**

We could do something like this:

`markdown-processor/processor-v3.ts`

```
function toHtml(markdown: string): string
{
    let html = "";
    const markdownLines = markdown.split("\n");

    for(const markdownLine of markdownLines)
    {
        if(markdownLine.startsWith("# "))
        {
            html += `<h1>${markdownLine.substring(2)}</h1>`;
        }
        else
        {
            html += `<p>${markdownLine}</p>`;
        }
    }

    return html;
}
```

Listing 2.6: Extending our code to support multiple lines

The code in ➔**Listing 2.6** handles multiple lines of text that each include either a paragraph or a primary heading. If we feed it the test Markdown given on lines 30 to 31 of our test code (➔**Listing 2.5**), it returns the expected HTML `<h1>This is an example of Markdown</h1><p>Here comes a simple paragraph.</p>`. If we run the test now, it passes. Yay!

Separating the Testing Logic

Our code in ➜Listing 2.6 works as required but has some issues that need addressing. ✎ **Check the code carefully: What bothers you?**

Here's what bothers me: For starters, checking the generated output manually through the console gets tiresome. Every time we add a new feature, we have to log (and check) the generated output to make sure our solution works as expected.

Additionally, the testing code gets unwieldy due to verbosity and repetition. The problem is that with each addition we not only have to test the new feature but all previous features, too. After all, when extending our code, we don't want to break existing functionality. This kind of testing is called ➜Regression Testing and is good practice in software development.

Developer Speak

Regression Testing is a methodology in software testing that makes sure that code functions as expected even after making changes (like bug fixes or feature additions).

Regression testing can be done with all types of tests and lends itself perfectly to unit testing. The key idea is to create a suite of tests that check the outward behavior(s) of a unit. When extending or otherwise modifying the code of that unit, the test suite gets executed and (if the tests pass) guarantees that the existing behavior of the unit has not been broken.

To make life easier with regression testing, we have to come up with a plan. ✎ **Can you think of a way to get rid of the repetition and tedious checks?**

Clearly, we could write a little helper function that gets rid of the repetition and the manual checks. Basically, we would have the computer do the checking for us.[11] The `check(...)` function in ➜Listing 2.7 takes two parameters: the `markdown` that gets passed to our `toHtml(...)` function, and the `expectedHtml` code that is expected to be generated from it.

The code here is fairly simple: We generate HTML from the given Markdown code, which is then compared to the expected HTML (that is given, too). If they are identical, the test passes and we simply see a short log in the console output. Should the test fail, details are error-logged to the console to give us a clue as to where the problem might be.

---

[11] Keep in mind: Computers are very good at repetition. They never grow tired and keep working at a fast pace. Whenever you find yourself dealing with repetetive work in code or elsewhere, consider offloading that work to your computer!

markdown-processor/processor-v4.ts

```
function check(markdown: string, expectedHtml: string)
{
    const generatedHtml = toHtml(markdown);

    if(generatedHtml == expectedHtml)
    {
        console.info(`Test passed: ${markdown}`);
    }
    else
    {
        const errorMsg = [
            `Test failed: ${markdown}`,
            `   Expected: ${expectedHtml}`,
            `  Generated: ${generatedHtml}`
        ].join("\n");

        console.error(errorMsg);
    }
}
```

Listing 2.7: The new check(...) helper function facilitates testing

With our test helper function in place, we can now simplify our tests by calling the function for every feature and passing in the parameter pairs needed, ➜Listing 2.8.

markdown-processor/processor-v4.ts

```
check("Here comes a simple paragraph.",
      "<p>Here comes a simple paragraph.</p>");

check("# This is an example of Markdown",
      "<h1>This is an example of Markdown</h1>");

check("# This is an example of Markdown\nHere comes a simple paragraph.",
      "<h1>This is an example of Markdown</h1><p>Here comes a simple paragraph.</p>");
```

Listing 2.8: Testing code passed to the test helper function

Basically, what we have done here is hide the testing logic inside a function. This technique is called ➜**encapsulation** and is one of the basic techniques used in software development.

DEVELOPER SPEAK

ENCAPSULATION is a technique in software development by which logic is separated inside a self-contained unit. The idea is to hide (*encapsulate*) the complexity of the logic from the user (➚client) of such a unit and let them access it through a well-defined ➚interface. Encapsulation promotes the reusability of (frequently used) code, loose coupling, as well as ➚information hiding, all three central concepts to keep software maintainable.

An **INTERFACE** is a well-defined way of communication between two independent systems. When two systems interact with one another, there is usually a **CLIENT** that requests the services of a **SERVER**. The interaction is thus done by the client calling functions on the server, and the server responding to these. The logic thus executed behind the scenes in either system is called **IMPLEMENTATION**, which is hidden from the prying eyes of the other side. Positively speaking, the client does not need to know how the server works (**INFORMATION HIDING**), and the server does not need to know what clients are using its services.

THE HIERARCHICAL STRUCTURE OF MARKDOWN

Another issue with our current code will become obvious when we add yet another feature. So far, our use cases are pretty simple: Headings and paragraphs. They are easy to figure out. All we have to do is check the beginning of every line of text and we know what we are dealing with. That's because all types of formats we support work on entire lines.

Even though Markdown is simple, it does have *some degree* of complexity. If you have a look at how bold and italic text is marked up (see lines 15 and 19 in ➚Listing 2.1), you realize that we can nest Markdown syntax within a single line. We could, for example, have italicized text inside a header, as in `# This header has *italicized* text`.

Nesting makes the syntax a lot more complex and requires us to rethink our current implementation. So far, we check whether we're dealing with a heading or a regular paragraph and spit out the relevant HTML tags with the text given. But now we will have to do additional checks for bold and italic text. And not just that, the syntax is also no longer confined to the beginning of a new line.

The major change in processing here is caused by the fact that we are no longer dealing with a flat, linear structure. Instead, Markdown text is hierarchical, as seen in ➚Image 2.2.

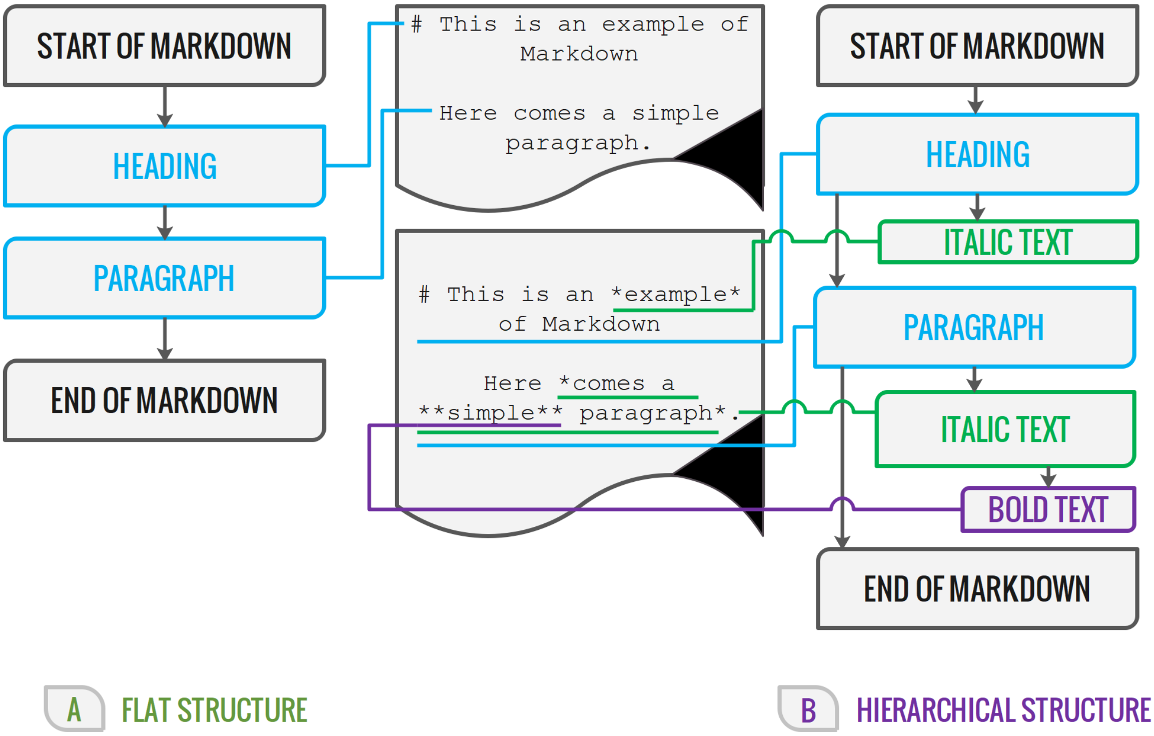


Image 2.2: The hierarchical structure of markdown text The examples show that Markdown text isn't simply linear and flat (A) but hierarchical (B).

Let's see how we can modify our existing solution to handle markup like the following: `This is a paragraph with *italic* text.`. First, **add a new test**. It could look like that one in **Listing 2.9**.

`markdown-processor/processor-v5.ts`

```
check("This is a paragraph with *italic* text.",
      "<p>This is a paragraph with <em>italic</em> text.</p>");
```

Listing 2.9: A new test to see if italic formatting works

Of course, running the code now would give us a failing test as we haven't implemented the new feature yet. **Modify your code to accomodate the new test.** If you don't get it right the first time, don't worry. Have a look at **Listing 2.10**, which contains the modified version of our `toHtml(...)` function.

`markdown-processor/processor-v5.ts`

```
function toHtml(markdown: string): string
{
    let html = "";
    const markdownLines = markdown.split("\n");

    for(const markdownLine of markdownLines)
    {
        if(markdownLine.startsWith("# "))
        {
            html += `<h1>${markdownLine.substring(2)}</h1>`;
        }
        else
        {
            // here begins the paragraph
            html += '<p>';

            let emTagOpened = false;

            // iterate through the characters
            for(let char of markdownLine.split(""))
            {
                // a * triggers the beginning or end of italicized text
                if(char == "*")
                {
                    if(!emTagOpened)
                    {
                        html += "<em>";
                        emTagOpened = true;
                    }
                    else
                    {
                        html += "</em>";
                        emTagOpened = false;
                    }
                }
                else
                {
                    html += char;
                }
            }

            // here ends the paragraph
            html += '</p>';
        }
    }

    return html;
}
```

Listing 2.10: Our modified function now supports italics inside paragaphs

Alright, let's run our tests and see. All tests pass, which means that the new feature works as expected. But, let's be honest: that code is... ouch! It works but it's getting convoluted and really difficult to read.

Can you spot the problematic parts?

Look, we have just doubled the **levels of indentation (LOIs)**, going from three (cp. **Listing 2.6**) to six. That's tough. Levels of indentation? Why would we care about those? Well, turns out there is a thing called **complexity**. And we have to talk about it.

Code Complexity

The total number of levels of indentation in a function is a good measure of its complexity: the more indentations it has, the higher its complexity usually is. Every level of indentation commonly works as a separate logical level of reasoning. With control flow statements such as `if(...) else(...)`, `for(...)`, and `while(...)` we execute (or skip over) lines of code based on certain conditions. These conditions are difficult for us to keep track of: the more conditions we control inside a function, the more complex the function becomes.

Developer Speak

Levels of Indentation (LOIs) are a visual aid that helps us reason about source code. Code is not arranged in a strictly serial, flat order but nested into different levels of abstractions that correspond to different levels of horizontal indentation where each level usually works as a logical unit. Such identation is achieved by branching off the code flow via control structures such as `if` and `for`, where the code inside the control block is indented. The indentation is removed once the control block is exited.

Let's see how complexity increases with each level of indentation. In Listing 2.10, what is our context on line 4? So far, we know that we are given a `string` of Markdown called `Markdown` and are to return its HTML representation. The contextual knowledge at this point is still quite manageable.

But it gets more complex: On line 9 we have already split the Markdown string into separate lines and are now iterating over those. The `markdownLine` variable contains the line of the current iteration. On line 11 we know that the current Markdown line starts with a `#` sign and is likely a primary heading. Contrarily, the alternative branch of the `if` control structure (lines 15+) assumes that the current `markdownLine` does not start with a `#` sign and hance we are likely dealing with a paragraph.

Let's fast forward here and find ourselves in considerably deeper complexity right away: On line 29 we know that the current `markdownLine` is likely a paragraph, we have separated the single characters of the line and are now iterating over them. We have come across a `*` char and the state of the `emTagOpened` variable tells us that no `<em>` tag (an HTML tag used to emphasize text, which browsers usually print in italics) had yet been opened, so we just open one.

By now it should become clear how indendations make code complex. The more contextual information we add to a function, the more difficult it will be to make sense of. In general, complexity is difficult to handle because it requires us to remember subtle nuances of our logic. Nuances that are easy to forget or simply overlooked and thus demand all of our attention. Psychologically speaking, complexity increases the *mental load* our brain has to deal with when reasoning about code.[A7]

Increasing the mental load takes a toll on us in terms of slower thinking, higher error rates, and faster cognitive exhaustion. Generally, more complex code makes our codebase more difficult to maintain, which leads to more bugs, a slower turnaround of features, and increased costs. Complexity is a *constant* battle in software development. Every single line of code we add to a codebase (even if it is a very simple one) potentially increases its complexity. In this regard, complexity cannot be avoided unless we stop coding altogether.

Ultimately, complexity can bring any project to a halt because it's only a matter of time until the marginal costs of having to maintain yet another line of code turn prohibitive. This is why we have to go to such great lengths to keep unnecessary complexity as low as possible. In fact, we can state this very effort as our prime responsibility:

> As software developers, our prime responsibility is managing complexity.
>
> — **Every Developer's Prime Responsibility**

What does it mean to manage complexity? Given that code complexity naturally creeps into any codebase, we are fighting on two fronts:

1. We have to keep the total number of LOCs to a minimum,
2. while making the code we *do* add to the codebase as maintainable as possible.

Item (1) is a quantitative measure that tells us that every line of code we do *not* write works in our favor. This is underlined by the widely known adage: *Junior developers try to figure out what lines of code to write, whereas senior developers try to figure out what lines of code not to write.* A good developer knows that solving a real-world problem through code is always a last resort. If possible, we try to solve problems before having to write any (additional) code.

In contrast, item (2) is a qualitative measure. It is necessary because we cannot avoid writing code altogether. If we do resort to writing code, we should at least make sure it is maintainable. Maintainability concerns many different properties: readability, adaptability, and clear communication of intent (to name just a few). We will learn more about these qualities later.

With this new knowledge about code complexity and why it's good to keep it to a minimum, let's return to our Markdown processor and see how we can address the complexity issues we have bumped into.

The logic we have implemented so far is required because there's no way we can meet the given requirements without it. So, if Removing code is not an option, we have to tackle complexity from a qualitative angle. Our focus should be on reducing the number of LOIs. One way of achieving this is splitting our `toHtml(...)` function up into smaller chunks. This technique is called **➔ Functional Decomposition** and is a very powerful way of managing complexity.

DEVELOPER SPEAK

FUNCTIONAL DECOMPOSITION is a technique in software design that breaks complex problems down into multiple simpler, more manageable pieces. Each piece is an independent unit that can be implemented and tested separately. The overall problem is then solved by combining the results of the individual pieces. It is important to note that the word "functional" here does not refer to functions in the sense of programming but the more general concept of functionality. Functional decomposition is thus a technique that is applied beyond the realm of programming.

FUNCTIONALLY DECOMPOSING OUR CODE BASE

Let's see which parts of our current logic can be extracted into their own functions. We first need to identify the independent pieces it consists of and then *extract* these (i.e. move them into their own functions). We can then call those subfunctions and thus restore the overall functionality.

The first thing we notice in **Listing 2.10** is that our `toHtml(...)` function is currently doing two very different things: On the one hand, it is separating the Markdown input into its individual lines and then processing each line separately. On the other hand, it is generating the resulting HTML output. Handling Markdown input and HTML output are clearly two separate concerns that should be separated functionally. [A8]

✎ **Let's extract the HTML generation logic and move it into its own class**. A `StringBuilder` class handles everything that deals with building the bits that make up our final HTML, see **Listing 2.11**.

markdown-processor/processor-v6.ts

```
class StringBuilder
{
    private _text: string = "";

    public append(text: string)
    {
        this._text += text;
    }
    public toString(): string
    {
        return this._text;
    }
}
```

Listing 2.11: We extracted the HTML generation logic into its own class.

You may be wondering: Why call the new class `StringBuilder` instead of `HtmlBuilder`? After all, we are using it to build HTML. That's a fair point. However, look closely at what that class *actually does*. **Can you think of a reason now?** You will see that it doesn't do anything HTML-specific. It simply builds up strings, hence the name `StringBuilder`. In fact, the class isn't even aware of *what types of strings* (semantics!) it builds. It could just as well be used to build, for example, XML or JSON strings instead of HTML. Calling the class `HtmlBuilder` would thus be overly specific and might prevent you from using it for any other string concatenations later on, or - even worse - reimplement that very logic in a similar class. [A9]

Next, we will extract the logic of our `toHtml(...)` function that handles each Markdown line by extracting it into its own `processMarkdownLine(...)` function. It takes two arguments: the `markdownLine: string` to handle, and the `stringBuilder: StringBuilder` we just introduced. Our modified `toHtml(...)` function now looks like in **Listing 2.12**.

markdown-processor/processor-v6.ts

```
function toHtml(markdown: string): string
{
    const stringBuilder = new StringBuilder();
    const markdownLines = markdown.split("\n");

    for(const markdownLine of markdownLines)
    {
        processMarkdownLine(markdownLine, stringBuilder);
    }

    return stringBuilder.toString();
}
```

Listing 2.12: The modified toHtml(...) function.

The new `processMarkdownLine(...)` function is quite straight forward. All it does is delegate the concatenation [12] of HTML code to the new `StringBuilder` instead of doing this itself. See **Listing 2.13**.

markdown-processor/processor-v6.ts

```
function processMarkdownLine(markdownLine: string, stringBuilder: StringBuilder)
{
    if(markdownLine.startsWith("# "))
    {
        stringBuilder.append(`<h1>${markdownLine.substring(2)}</h1>`);
    }
    else
    {
        // here begins the paragraph
        stringBuilder.append("<p>");

        let emTagOpened = false;

```

---

[12] A fancy word for joining two or more strings.

```
46         // iterate through the characters
47         for(let char of markdownLine.split(""))
48         {
49             // a * triggers the beginning or end of italicized text
50             if(char == "*")
51             {
52                 if(!emTagOpened)
53                 {
54                     stringBuilder.append("<em>");
55                     emTagOpened = true;
56                 }
57                 else
58                 {
59                     stringBuilder.append("</em>");
60                     emTagOpened = false;
61                 }
62             }
63             else
64             {
65                 stringBuilder.append(char);
66             }
67         }
68 
69         // here ends the paragraph
70         stringBuilder.append("</p>");
71     }
72 }
```

Listing 2.13: The new processMarkdownLine(...) function.

So far, we have managed to decrease the overall complexity inside `toHtml(...)`: What used to be 48 LOCs and 6 LOIs (cp. **Listing 2.10**) are now 12 LOCs and 2 LOIs (cp. **Listing 2.12**). That's a pretty decent start, but there is more to do. We haven't solved our complexity issue entirely yet as a large part of complexity was just pushed further down the line and now resides inside `processMarkdownLine(...)` (41 LOCs and 5 LOIs). This is where we will continue looking for more independent pieces that we can extract.

Analyzing `processMarkdownLine(...)` in **Listing 2.13**, we identify yet again two logical parts that warrant further decomposition. Thinking of **Separation of Concerns**, **what is the problem here?**

The principle of SOC is clearly violated here: While lines 35 to 38 deal with headings, lines 41 to 70 deal with paragraphs. **Let's separate those two concerns by extracting these into separate functions**.

Listing 2.14 shows what the code could look like after the extraction:

markdown-processor/processor-v7.ts

```
function processMarkdownLine(markdownLine: string, stringBuilder: StringBuilder)
{
    if(markdownLine.startsWith("# "))
    {
        headingToHtml(markdownLine, stringBuilder);
    }
    else
    {
        paragraphToHtml(markdownLine, stringBuilder);
    }
}
```

Listing 2.14: The modified processMarkdownLine(...) function.

The newly created functions are aptly named `headingToHtml(...)` and `paragraphToHtml(...)` and shown in ➚**Listing 2.15**.

markdown-processor/processor-v7.ts

```
function headingToHtml(markdownLine: string, stringBuilder: StringBuilder)
{
    stringBuilder.append(`<h1>${markdownLine.substring(2)}</h1>`);
}
function paragraphToHtml(markdownLine: string, stringBuilder: StringBuilder)
{
    // here begins the paragraph
    stringBuilder.append("<p>");

    let emTagOpened = false;

    // iterate through the characters
    for(let char of markdownLine.split(""))
    {
        // a * triggers the beginning or end of italicized text
        if(char == "*")
        {
            if(!emTagOpened)
            {
                stringBuilder.append("<em>");
                emTagOpened = true;
            }
            else
            {
                stringBuilder.append("</em>");
                emTagOpened = false;
            }
        }
        else
        {
            stringBuilder.append(char);
        }
    }

    // here ends the paragraph
    stringBuilder.append("</p>");
79 }
```

Listing 2.15: The extracted functions.

When modifying our code so heavily, it is of utmost importance that we keep testing its behavior regularly with the unit tests we have in place. As long as these tests pass, we are good to keep going. Should any test fail, we know that we have broken something and need to fix it before we can continue. If we can't find a fix, we can always revert the last changes and continue from where the code still worked.

So far, we are good: All tests still pass despite all the considerable modifications we have made to our code base. ✎ **Can you figure out how much we have reduced the complexity here?** We have reduced the LOCs of `processMarkdownLine(...)` from 41 to 11 and the LOIs are down from 5 to 2. Admittedly, we have pushed some complexity further down into `paragraphToHtml(...)` and will deal with this later. Right now, it's time to implement a new feature.

New feature: italics inside paragraphs and headings

We have already implemented the logic to display italicized text inside paragraphs. But we also need to allow italicized text to be displayed inside headings. We don't want to duplicate any code, so we will need to find a way to reuse the existing logic that handles Markdown like `This is a paragraph with *italic* text.` (see ➦Listing 2.9).

Again, we will start with a test that checks for the expected behavior we want. Here's the code:

markdown-processor/processor-v8.ts

```
check("# This is *italic* text",
      "<h1>This is <em>italic</em> text</h1>");
```

Listing 2.16: The new unit test.

The existing logic that handles `*` for italic text in Markdown can be found in lines 52 to 75 in ➦Listing 2.15. We should extract that logic into a separate function `inlineMarkdownToHtml(...)`, as shown in ➦Listing 2.17.

markdown-processor/processor-v8.ts

```
function inlineMarkdownToHtml(markdownText: string, stringBuilder: StringBuilder)
{
    let emTagOpened = false;

    // iterate through the characters
    for(let char of markdownText.split(""))
    {
        // a * triggers the beginning or end of italicized text
        if(char == "*")
        {
            if(!emTagOpened)
            {
                stringBuilder.append("<em>");
                emTagOpened = true;
            }
            else
            {
                stringBuilder.append("</em>");
                emTagOpened = false;
            }
        }
        else
        {
            stringBuilder.append(char);
        }
    }
}
```

Listing 2.17: The extracted logic that handles inline Markdown.

Now, all that is left to do is modify our existing functions `headingToHtml(...)` and `paragraphToHtml(...)` to delegate the handling of inline markdown to the new `inlineMarkdownToHtml(...)` function. See Listing 2.18.

`markdown-processor/processor-v8.ts`

```
function headingToHtml(markdownLine: string, stringBuilder: StringBuilder)
{
    stringBuilder.append("<h1>");

    inlineMarkdownToHtml(markdownLine.substring(2), stringBuilder);

    stringBuilder.append("</h1>");
}
function paragraphToHtml(markdownLine: string, stringBuilder: StringBuilder)
{
    stringBuilder.append("<p>");

    inlineMarkdownToHtml(markdownLine, stringBuilder);

    stringBuilder.append("</p>");
}
```

Listing 2.18: The modified functions delegate the handling of inline Markdown.

And done! Our test in Listing 2.16 passes now, as do all the previous tests. We have successfully extended the feature set of our solution.

AND SOME MORE: BOLD TEXT

Let's have a quick glance at our initial Markdown sample (see ➦ Listing 2.1). We still have to implement some missing features. We'll handle bold text next (in Markdown `**bold**`).

Basically, handling bold text is not much different from handling italics. Instead of looking for `*` we just need to look for `**` as delimiters. Let's define this behavior in a new unit test:

markdown-processor/processor-v9.ts

```
check("This is a paragraph with **bold** text.",
      "<p>This is a paragraph with <strong>bold</strong> text.</p>");

check("# This is a heading with **bold** text.",
      "<h1>This is a heading with <strong>bold</strong> text.</h1>");
```

Listing 2.19: The new unit tests check for bold text.

If we run the test now, it should fail and return the following output:

```
1 Test failed: This is a paragraph with **bold** text.
2    Expected: <p>This is a paragraph with <strong>bold</strong> text.</p>
3   Generated: <p>This is a paragraph with <em></em>bold<em></em> text.</p>
4
5 Test failed: # This is a heading with **bold** text.
6    Expected: <h1>This is a heading with <strong>bold</strong> text.</h1>
7   Generated: <h1>This is a heading with <em></em>bold<em></em> text.</h1>
```

What's the matter with those empty `<em></em>` tags (lines 3 and 7)? ✎ **Go through the code and figure out what's happening!** As it turns out, the `**` are not yet interpreted as delimiters for bold text (we haven't implemented that feature yet). Instead, the first `*` is interpreted as the start of italic text, the second `*` as the end of italic text. The text in between is empty, hence the empty `<em></em>` tags.

Handling `**` delimiters for bold text isn't much different from handling `*` delimiters for italic text. To implement this feature, we just have to modify the `inlineMarkdownToHtml(...) function`, as shown in ➦ Listing 2.20.

markdown-processor/processor-v9.ts

```
function inlineMarkdownToHtml(markdownText: string, stringBuilder:
↪ StringBuilder)
{
    let lastUnconsumedChar: string | null = null;
    let emTagOpened = false;
    let strongTagOpened = false;

    // iterate through the characters
    for(let char of markdownText.split(""))
    {
        // a * triggers the beginning or end of italicized/bold text
        if(char == "*")
        {
            if(lastUnconsumedChar == "*")
            {
                // **, so dealing with bold text
                if(!strongTagOpened)
                {
                    stringBuilder.append("<strong>");
                    strongTagOpened = true;
                }
                else
                {
                    stringBuilder.append("</strong>");
                    strongTagOpened = false;
                }

                lastUnconsumedChar = null;
            }
            else
            {
                lastUnconsumedChar = char;
            }
        }
        else
        {
            // *, so dealing with italic text
            if(lastUnconsumedChar == "*")
            {
                if(!emTagOpened)
                {
                    stringBuilder.append(`<em>${char}`);
                    emTagOpened = true;
                }
                else
                {
                    stringBuilder.append(`</em>${char}`);
                    emTagOpened = false;
                }
            }
            else
            {
                stringBuilder.append(char);
            }

            lastUnconsumedChar = null;
        }
    }
}
```

Listing 2.20: The modified inlineMarkdownToHtml(...) function now supports bold text.

The key change here lies in the fact that the interpretation of a single `*` character is no longer unambiguous. Previously, it marked the beginning or end of italic text (depending on whether we had already opened the `<em>` tag or not). Now, its interpretation depends on whether there is another `*` following it or not. If there is, it is a delimiter for bold text. Otherwise, it's a delimiter for italic text.

To distinguish between `*` and `**` characters, we save the very first occurrence of a `*` character in the variable `lastUnconsumedChar` (l. 60) and do not act on it (cp. l. 88) (hence the notion of *unconsumed*). Only when processing the subsequent character, we know whether we are dealing with a `**` and hence bold text (cp. ll. 72 - 84) or a `*` and hence italic text (cp. ll. 93 - 112).

Once more, let's run our tests to make sure we have implemented the feature as expected. Indeed, the tests pass.

Having implemented one feature after another, we have been focussing on the implementation level for quite some time now. Implementing feature after feature may give you a sense of productivity, but it can easily lead to losing sight of the bigger picture. It's good practice, then, to stop coding at given intervals and assess the entirety of the code we have written. Let's do that now. Here's the core code (excluding the testing and string building logic):

markdown-processor/processor-v10.ts

```
15 function toHtml(markdown: string): string
16 {
17     let stringBuilder = new StringBuilder();
18     const markdownLines = markdown.split("\n");
19 
20     for(const markdownLine of markdownLines)
21     {
22         processMarkdownLine(markdownLine, stringBuilder);
23     }
24 
25     return stringBuilder.toString();
26 }
27 
28 function processMarkdownLine(markdownLine: string, stringBuilder: StringBuilder)
29 {
30     if(markdownLine.startsWith("# "))
31     {
32         headingToHtml(markdownLine, stringBuilder);
33     }
34     else
35     {
36         paragraphToHtml(markdownLine, stringBuilder);
37     }
38 }
39 
40 function headingToHtml(markdownLine: string, stringBuilder: StringBuilder)
41 {
42     stringBuilder.append("<h1>");
43 
44     inlineMarkdownToHtml(markdownLine.substring(2), stringBuilder);
```

```

    stringBuilder.append("</h1>");
}
function paragraphToHtml(markdownLine: string, stringBuilder: StringBuilder)
{
    stringBuilder.append("<p>");

    inlineMarkdownToHtml(markdownLine, stringBuilder);

    stringBuilder.append("</p>");
}

function inlineMarkdownToHtml(markdownText: string, stringBuilder: StringBuilder)
{
    let lastUnconsumedChar: string | null = null;
    let emTagOpened = false;
    let strongTagOpened = false;

    // iterate over the characters
    for(let char of markdownText.split(""))
    {
        // a * triggers the beginning or end of italicized/bold text
        if(char == "*")
        {
            if(lastUnconsumedChar == "*")
            {
                // **, so dealing with bold text
                if(!strongTagOpened)
                {
                    stringBuilder.append("<strong>");
                    strongTagOpened = true;
                }
                else
                {
                    stringBuilder.append("</strong>");
                    strongTagOpened = false;
                }

                lastUnconsumedChar = null;
            }
            else
            {
                lastUnconsumedChar = char;
            }
        }
        else
        {
            // *, so dealing with italic text
            if(lastUnconsumedChar == "*")
```

```
                {
                    if(!emTagOpened)
                    {
                        stringBuilder.append(`<em>${char}`);
                        emTagOpened = true;
                    }
                    else
                    {
                        stringBuilder.append(`</em>${char}`);
                        emTagOpened = false;
                    }
                }
                else
                {
                    stringBuilder.append(char);
                }

                lastUnconsumedChar = null;
            }
        }
}
```

Listing 2.21: Our entire codebase after having implemented seven features.

Assessing our code critically, we have to acknowledge that complexity has crept in once again. ✎ **Can you spot it?** The `inlineMarkdownToHtml(...)` function is a sight for sore eyes. It's rather long and convoluted at 58 LOCs and 5 LOIs. It's time to improve this. [A10]

COMPLEXITY HITS YET AGAIN. REFACTORING TO THE RESCUE!

To be clear, there's nothing wrong with hitting complexity. Indeed, it is often better to hit complexity instead of desperately trying to avoid it by resorting to "overly smart" and unnecessarily abstract solutions (so-called ➜ **premature abstractions**). [A11]

Developer Speak

Premature Abstractions are early generalizations in code based not on actual needs but on speculation and a misguided notion of keeping the source "clean".

Abstractions are premature if they add indirection without a clear benefit. Frequently, such a "need" is driven by ideals of a clean codebase or speculation about how parts of a codebase will be used in the future.

Even developers cannot predict the future, and hence it is better to wait with generalizations until the need for them becomes obvious. This idea is the essence of the **You Ain't Gonna Need It (YAGNI)** guideline.

Keeping things simple and allowing some complexity to build up is necessary in order to figure out what kind of abstractions are actually needed. As long as we keep looking out for improper abstractions and act on such *unnecessary* complexity by ➦refactoring our code, we can effectively overcome it. Problems arise the moment that we postpone that refactoring step and allow complexity to build up to a point where it brings development to a halt.

Developer Speak

Refactoring is the process of improving the (internal) structure of code without changing its (external) behavior. It is a key technique for keeping code clean and maintainable. Refactoring is an iterative process that is best done in small steps and with tests in place. That way you can be sure that your code still works as expected afterwards.

Think of refactoring as a way to sweep through your code, very much like you would sweep your home: You remove a few things inside or move them around, or even add a new piece of furniture. But from outside, your home still looks the same, still behaves the same. It's still the same home (you wouldn't notice any difference from the outside), just a bit more organized and comfortable to live in.

Let's analyze the `inlineMarkdownToHtml(...)` function to see what's so complex about it and how we can decompose it. There are a few things that stand out:

1. The notion of unconsumed characters (cp. the variable `lastUnconsumedChar` in lines 59, 69, 93, etc.) is getting out of hand quickly. It's a rather convoluted way of keeping track of the current context. We need to find an easier way of handling this.
2. Since `*` and `**` denote both the start *and* end of text that shall be displayed as italic and bold, respectively, we have to keep state of this accordingly (cp. lines 72, 77, 95, 100). The way we have

implemented the tracking of such contextual state is too unwieldy. Here, too, we have to find an easier way of handling this so we can simply our code.

Having pinpointed the problems in our code, we can set out to look for potential solutions. Let's start with the first problem (unconsumed characters). Skipping over a `*` character to see how it makes sense in hindsight (`*` vs. `**`) adds some indirection that complicates things needlessly. Let's get rid of it. **Can you think of a way to reverse this flow, i.e. to look ahead a character?**

Let's step back and look at the code from a distance: What are we really trying to achieve through such **lookaheads**? We aren't interested in characters, really, but in how these characters make up larger structures, so-called **tokens**. What's the difference between characters and tokens? While `*` itself is just a character with an arbitrary meaning, it becomes a (more meaningful) token when interpreted in terms of its wider context and what it stands for, e.g. the beginning or end of emphasized text (which we will turn into italics in HTML). In the same vein, `**` is *two characters* that become *one token* when interpreted as the beginning or end of doubly emphasized text (which we will turn into bold text in HTML).

Developer Speak

A **LOOKAHEAD** is an operation on a stream of symbols (e.g. a collection of characters) whereby the current position in that stream is preserved even though a symbol ahead of the current position is already checked ("peeked at").

Lookaheads are common operatioms found in **PARSERS**, programs that analyze formal languages by breaking them down into their structural parts called **TOKENS**. Such formal languages follow syntactic rules (a grammar) that define how well-defined sentences look like.

When analyzing markup such as Markdown and disassembling it into its structural parts, we are getting our feet wet in a technique called *parsing*. A *parser* makes sense of structural markup through a *tokenizer*, which is the logic that takes as input a stream of characters (markup) and returns a stream of tokens (meaningful units in terms of the target language, in our case: Markdown). The process is illustrated in **Image 2.3**.

Notice that there is not necessarily a one-to-one relationship between characters and tokens. For example, the `**` two-character sequence is translated into a single token, namely `Double Emphasis`. Conversely, there can be tokens that do not find any correspondence to characters, e.g. the `Beginning of Stream`, `End of Stream`, and `End of Line` tokens, which are all just implied.

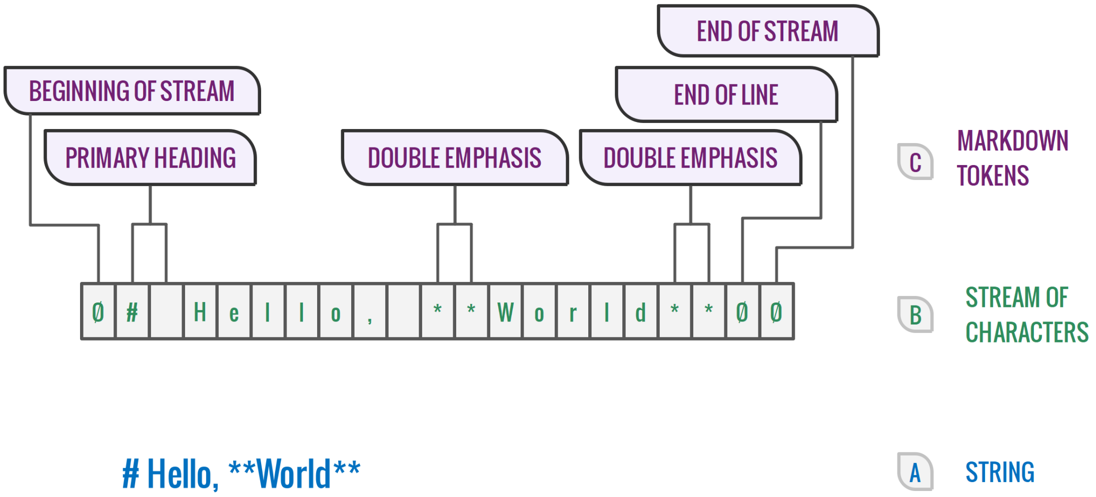


Image 2.3: Markdown tokens vs. a stream of characters A simple string [A] interpreted as a stream of characters [B] and a stream of Markdown tokens [C].

No worries, we are not going to write a full-blown parser here. But we *will* write a simple tokenizer that will help us simplify our code. Such a tokenizer will analyze the Markdown input and translate its characters into tokens, for example:

- `*` and `**` characters will be translated into `Single Emphasis` and `Double Emphasis` tokens, respectively.
- `#` and `##` characters will be translated into `PrimaryHeading` and `SecondaryHeading` tokens, respectively.

Given the above tokens, your task now is to **ponder the question of how you would write such a tokenizer**. You don't have to write any code (though, please feel free to give it a try!), just thinking about the intricacies of a potential solution will yield worthwhile insights.

REFACTORING THE TEXT DOMAIN (CHARACTERS)

Keep in mind that a tokenizer that translates characters into Markdown tokens works on two different levels of abstraction: a stream of characters (input) vs. a stream of tokens (output), which I will henceforth call *Text Domain* and *Markdown Domain*, respectively.

Following **Separation of Concerns**, it is a good idea to keep those two domains separate so that we don't increase complexity needlessly by mixing two independent concerns. Starting with the *Text Domain*, the logic to handle a stream of characters could look like this:

markdown-processor/processor-v11.ts

```
class Char
```

```
{
    constructor(
        public readonly value: string)
    {}

    public is(char: Char): boolean
    {
        return this.value == char.value;
    }
}
class Chars
{
    public static readonly NewLine = new Char("\n");
}
class CharStream
{
    private _normalizedText: string;
    private _position = -1;

    constructor(text: string)
    {
        this._normalizedText = CharStream.normalize(text);
    }

    public get length(): number
    {
        return this._normalizedText.length;
    }
    public get isEmpty(): boolean
    {
        return this.length == 0;
    }
    public get isAtStart(): boolean
    {
        return this.isEmpty
            || this._position == 0;
    }
    public get hasReachedEndOfStream(): boolean
    {
        return this.isEmpty
            || this._position >= this.length;
    }
    public get hasReachedEndOfLine(): boolean
    {
        return this.currentChar.is(Chars.NewLine)
            || this.isLastCharOfStream;
    }
    public get isLastCharOfStream(): boolean
    {
        return !this.isEmpty
```

```
            && this.hasReachedEndOfStream;
    }
    public get isFirstCharOfStream(): boolean
    {
        return !this.isEmpty
            && this.isAtStart;
    }
    public get isFirstCharOfLine(): boolean
    {
        return this.isFirstCharOfStream
            || this.peek(-1).is(Chars.NewLine);
    }
    public get currentChar(): Char
    {
        return this.getCharAtPosition(this._position);
    }

    public tryConsume(value: string): boolean
    {
        if(!(value?.length > 0))
        {
            throw new Error("Value must not be empty or null.");
        }

        for(var i = 0; i < value.length; i++)
        {
            const expectedChar = value.substring(i, i + 1);
            const givenChar = this.peek(i).value;

            if(givenChar != expectedChar)
            {
                return false;
            }
        }

        this.advance(value.length - 1);
        return true;
    }
    public advance(range?: number): boolean
    {
        range ??= 1;

        if(this.hasReachedEndOfStream)
        {
            return false;
        }

        this._position += range;
        return true;
    }

    private peek(offset: number): Char
    {
        return this.getCharAtPosition(this._position + offset);
    }
    private getCharAtPosition(position: number): Char
    {
        if(   position < 0
           || position >= this.length)
        {
            throw new Error("End of stream reached.");
        }

        var value = this._normalizedText.substring(position, position + 1);
        return new Char(value);
    }
    private static normalize(text: string): string
    {
        // normalize new lines across platforms
        return (text ?? "").replace("\r", "");
    }
}
```

Listing 2.22: The Text Domain handles a stream of characters

All classes listed in **➚Listing 2.22** deal with the lowest level of abstraction, namely characters. Notice how this is a self-contained domain that does *not* reference tokens or even Markdown, which are concepts that belong to an entirely different domain (more on this later). There are three classes in total that ware worth discussing.

The `Char` class represents a single character identified by its `value`. It allows comparing a character instance to other characters via `is(...)` so we can establish equality based on character value.

A convenience class `Chars` lists the most common characters we are dealing with. These serve as constants that help us reference frequently used characters without having to hard-code them everywhere.

Finally, the `CharStream` class helps us navigate through a stream of characters. It keeps a pointer to the current `_position` in the stream and allows looking at its current character (`currentChar`), advancing to the next character (`advance(...)`), and peeking ahead a custom amount of characters (`peek(...)`). It also features some positional properties to allow checking whether the end of a line (`hasReachEndOfLine`) or the stream has been reached (`hasReachedEndOfStream`), etc.

Two functions inside the `CharStream` class deserve special attention:

The `normalize(...)` function is used to normalize the input across different platforms (Windows, Mac, Linux) by removing all carriage return characters (`\r`). Line breaks are thus always represented by single new line characters (`\n`). [13]

---

[13] Historically, Windows uses the carriage return, line feed sequence `\r\n` to denote line breaks whereas Unix just uses `\n`.

The `tryConsume(...)` function advances the position of the stream if (and only if) the current characters match the a given `value` string. This is an all-or-nothing operation, meaning that either the entire string given is matched (and the position inside the stream advanced) or nothing happens at all. (This is useful for matching multi-character tokens such as `**` or `##` later on.)

That's all there is to characters and the *Text Domain*. Next, let's see how we can build upon this code to parse Markdown text into Markdown tokens.

REFACTORING THE MARKDOWN DOMAIN (TOKENS)

Climbing up one level of abstraction, the *Markdown Domain* could look like this:

markdown-processor/processor-v11.ts

```
class Token
{
    constructor(
        public readonly value: string)
    {}

    public is(token: Token): boolean
    {
        return this.value == token.value;
    }
}
class Tokens
{
    public static readonly SingleEmphasis = new Token("*");
    public static readonly DoubleEmphasis = new Token("**");
    public static readonly PrimaryHeading = new Token("# ");
    public static readonly SecondaryHeading = new Token("## ");
    public static readonly EndOfLine = new Token("\endofline");
    public static readonly BeginningOfStream = new Token("\beginningofstream");
    public static readonly EndOfStream = new Token("\endofstream");
}
class Tokenizer
{
    private _currentToken: Token = Tokens.BeginningOfStream;

    constructor(
        private readonly _charStream: CharStream)
    {}

    public get currentToken(): Token
    {
        return this._currentToken;
    }
    public get hasReachedEndOfStream(): boolean
    {
        return this._currentToken == Tokens.EndOfStream;
    }

    public advanceToNextToken(): boolean
    {
        if(this.hasReachedEndOfStream)
        {
            return false;
        }

        this._charStream.advance();
        this._currentToken = this.getToken();
        return true;
    }

    private getToken(): Token
    {
        if(this._charStream.hasReachedEndOfStream)
        {
            return Tokens.EndOfStream;
        }
        else if(this._charStream.hasReachedEndOfLine)
        {
            return Tokens.EndOfLine;
        }
        else if(   this._charStream.isFirstCharOfLine
                && this._charStream.tryConsume(Tokens.PrimaryHeading.value))
        {
            return Tokens.PrimaryHeading;
        }
        else if(   this._charStream.isFirstCharOfLine
                && this._charStream.tryConsume(Tokens.SecondaryHeading.value))
        {
            return Tokens.SecondaryHeading;
        }
        else if(this._charStream.tryConsume(Tokens.DoubleEmphasis.value))
        {
            return Tokens.DoubleEmphasis;
        }
        else if(this._charStream.tryConsume(Tokens.SingleEmphasis.value))
        {
            return Tokens.SingleEmphasis;
        }

        // when other options are exhausted, this must be a literal token
        return new Token(this._charStream.currentChar.value);
    }
}
```

Listing 2.23: The Markdown Domain handles Markdown tokens

The *Markdown Domain* consists of these three classes:

The `Token` class represents a Markdown token that consists of a string `value` and can be compared via `is(...)` to other tokens to check for equality based on that value.

The `Tokens` class is another convenience class that lists the tokens recognized by the `tokenizer`. It contains all the tokens already identified in ➔ **Image 2.3**, plus the `SingleEmphasis` token.

The core logic of this domain resides in the `tokenizer` class. It takes a `CharStream` instance as a dependency, from which it consumes characters and parses these into tokens – one after another. This is done by calling `advanceToNextToken()` in an indefinite loop until the end of the stream is reached (you may check this throught the `hasReachedEndOfStream` property). Inside the loop, we can access the `currentToken`.

The actual parsing of characters into tokens is done in `getToken()`. This is where the current position inside the character stream is checked to infer the correct token from this context. Please note that the order of checks can be important here: For example, if we checked for the `SingleEmphasis` token before the `DoubleEmphasis` token, we might end up with a false positive as `*` alone always matches `**`.

The checks themselves are pretty self-explanatory, except maybe for the last one (l. 304): This is the default "fallback" case that is used if none of the other checks return a match. It basically represents any literal character that is not part of a more specialized token, which is useful for representing plain text characters.

Refactoring the HTML Domain

Text and *Markdown* are not the only (sub-) domains our solution consists of. After all, we are translating Markdown into HTML, so we need to represent HTML as well. This is done in the *HTML Domain*, which could look like this:

markdown-processor/processor-v11.ts

```
class HtmlBuilder
{
    private _html = "";
    private _stack: string[] = [];

    constructor() {}

    private get isStackEmpty(): boolean
    {
        return this._stack.length == 0;
    }

    public peek(): string | null
    {
        if(this.isStackEmpty)
        {
            return null;
        }

        return this._stack[this._stack.length - 1];
    }
    public push(tag: string)
    {
        this.makeSureTagIsValidOrThrow(tag);

        const normalizedTag = this.normalize(tag);
        this._stack.push(normalizedTag);
        this._html += `<${normalizedTag}>`;
    }
    public pop(tag: string)
    {
        if(this.isStackEmpty)
        {
            throw new Error("Stack empty, nothing to pop.");
        }

        this.makeSureTagIsValidOrThrow(tag);
        const normalizedTag = this.normalize(tag);

        if(!this.canPop(normalizedTag))
        {
            throw new Error(`Must close tag <${this.peek()}> first.`);
        }

        this._stack.pop();
        this._html += `</${normalizedTag}>`;
    }
    public popRemaining()
    {
        while(!this.isStackEmpty)
        {
            const peekedTag = this.peek()!;
            this.pop(peekedTag);
        }
    }
    public write(text: string)
    {
        // TODO: we should validate the input here and escape special characters
        this.makeSureTextIsValidOrThrow(text);
        this._html += text;
    }
    public toHtml(): string
    {
        return this._html;
    }

    private canPop(normalizedTag: string)
    {
        return this.peek() == normalizedTag;
    }
    private normalize(tag: string): string
    {
        return tag.trim().toLowerCase();
    }
    private makeSureTagIsValidOrThrow(tag: string)
    {
        const validTagRegex = /^[a-z][a-z0-9]*$/im;

        if(!validTagRegex.test(tag))
        {
            throw new Error(`Invalid tag: ${tag}`);
        }
    }
    private makeSureTextIsValidOrThrow(text: string)
    {
        const validTextRegex = /[^<>]*/im;

        if(!validTextRegex.test(text))
        {
            throw new Error(`Invalid text: ${text}`);
        }
    }
}
```

Listing 2.24: The HTML Domain handles building up the HTML output

The domain consists of a single file `HtmlBuilder`, which is responsible for building up the HTML source code that we return as output. Notice how the class does not depend on any other classes from our solution, and thus is highly independent. This is a good thing, as it makes the class easier to test and reuse.

Before venturing into the logic of the `HtmlBuilder`, let's quickly discuss the syntactic peculiarities of HTML itself. HTML is a ➔**stack**-based language, which means that it is built up from nested elements in a last-in-first-out fashion.

DEVELOPER SPEAK

A **STACK** is an abstract data type that represents a collection of elements with two basic operations: The *push* operation adds an element to the collection, while the *pop* operation removes the most recently added element that has not yet been removed.

You can visualize the collection as a stack of plates: Pushing adds a plate on top of an existing pile of plates, while popping removes the topmost plate. This is why stacks are also called *LIFO* (last in, first out) data structures.

What does it mean for HTML to be stack-based? It means that its structural markup is represented by a stack of nested elements. We can push and pop as many elements onto and from the stack as long as we do this in a *LIFO* fashion, as illustrated in ➔Image 2.4.

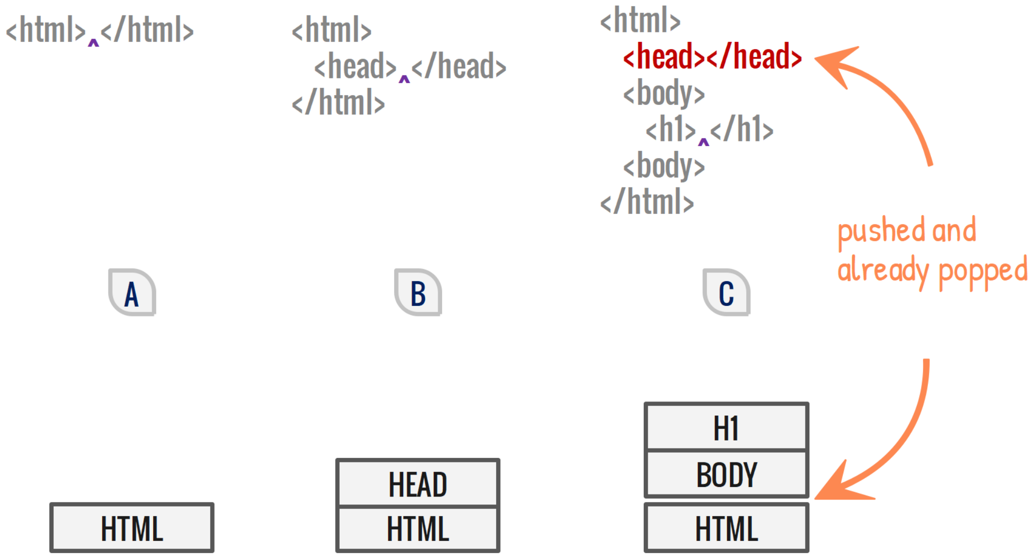


Image 2.4: HTML as a stack-based language HTML elements are stacked on top of each other in a last-in-first-out fashion. The stacks on the right represent the state of the HTML elements given at the position of the caret sign (‸).

The stack-based character of HTML is directly reflected in the `HtmlBuilder`: It starts out with an empty `_stack` and offers operations to `push(...)` and `pop()` tag elements onto and from the stack. A `peek()` function allows us to look at the topmost element on the stack without removing it. A `popRemaining` function is used to pop all remaining elements from the stack.

The `HtmlBuilder` keeps both track of the stacked elements and the HTML source code that is built up from these elements: Whenever an element is pushed or popped, the corresponding HTML tag is appended to the `_html` string. A `write(...)` function allows us to add arbitrary literal characters between the tags. Note that any such input should be properly validated and escaped to prevent security vulnerabilities (omitted here for brevity, `makeSureTextIsValidOrThrow()` only checks for angle brackets, which are reserved for tags).

Finally, the `toHtml()` function returns the final HTML source code. Note that any tags pushed and popped to and from the stack are validated inside the `makeSureTagIsValidOrThrow()` function and also normalized inside `normalize`. The latter is done because HTML is not case-sensitive, so all tags are handled as lower-case strings.

BRINGING IT ALL TOGETHER: TRANSLATING MARKDOWN TO HTML

Having refactored our solution into clearly separated domains of *HTML*, *Markdown*, and *Text*, we can now set out to tie lose ends and bring the entire logic together. This is done in the `MarkdownProcessor` class, which is responsible for translating Markdown into HTML. It looks like this:

markdown-processor/processor-v11.ts

```
class MarkdownProcessor
{
    public static toHtml(markdown: string): string
    {
        const charStream = new CharStream(markdown);
        const tokenizer = new Tokenizer(charStream);
        const htmlBuilder = new HtmlBuilder();

        while(tokenizer.advanceToNextToken())
        {
            this.translateTokenToHtml(tokenizer.currentToken, htmlBuilder);
        }

        htmlBuilder.popRemaining();

        return htmlBuilder.toHtml();
    }

    private static translateTokenToHtml(token: Token, htmlBuilder: HtmlBuilder)
    {
        if(token.is(Tokens.PrimaryHeading))
        {
            htmlBuilder.push("h1");
        }
        else if(token.is(Tokens.SecondaryHeading))
        {
            htmlBuilder.push("h2");
        }
        else if(token.is(Tokens.EndOfLine))
        {
            // headings and paragraphs are terminated at the end of a line
            const topmostItem = htmlBuilder.peek();

            if(topmostItem &&
               ["h1", "h2", "p"].includes(topmostItem))
            {
                htmlBuilder.pop(topmostItem);
            }
        }
        else if(token.is(Tokens.SingleEmphasis))
        {
            // if we already have an <em> tag on the stack,
            // this * will close it, otherwise we open it
            if(htmlBuilder.peek() == "em")
            {
                htmlBuilder.pop("em");
            }
            else
            {
                htmlBuilder.push("em");
            }
        }
        else if(token.is(Tokens.DoubleEmphasis))
        {
            // if we already have a <strong> tag on the stack,
            // this ** will close it, otherwise we open it
            if(htmlBuilder.peek() == "strong")
            {
                htmlBuilder.pop("strong");
            }
            else
            {
                htmlBuilder.push("strong");
            }
        }
        else if(!token.is(Tokens.EndOfStream))
        {
            if(htmlBuilder.peek() == null)
            {
                htmlBuilder.push("p");
            }

            htmlBuilder.write(token.value!);
        }
    }
}
```

Listing 2.25: The Markdown Processor brings all subdomains together

The `MarkdownProcessor` class reminds us of our initial `toHtml()` function. It still takes as input a `markdown: string` and returns HTML as a `string`. Remember that refactoring means keeping the outward behavior (and interface) of a unit intact, while improving its inner structure. This is exactly what we did: The `toHtml()` function still behaves as expected (it still translates Markdown into HTML), but its inner structure has changed. Instead of doing all the work itself, it now delegates to the various classes of our solution.

If we run our initial tests, we should see that all tests pass. There are also two new tests in place to cover *Secondary Headings*, which are now implemented as well. Checking the complexity of our solution, we see that no function holds more than 3 LOIs. The longest function is `translateTokenToHtml(...)` inside our `MarkdownProcessor`, which is still very much manageable with only slightly over 50 LOCs.

EXTENDING OUR SOLUTION: UNORDERED LIST ITEMS

We have refactored our initial solution nicely and removed a lot of complexity, thus making the codebase easier to read and maintain. We will now see how maintainable code helps scale our solution when new requirements come up (a thing that happens all-too frequently in the real world). Going back to the Markdown features in ➔ Image 2.1, we see that we have yet to implement support for unordered list items. Let's do this now.

We start, as always, by adding a new test case:

markdown-processor/processor-v12.ts

```
check('* This is a list item\n* This is another list item',
      '<ul><li>This is a list item</li><li>This is another list item</li>
 ↪ </ul>');
```

Listing 2.26: Adding a new test case for unordered list items

Next, we need to add a new token type `UnorderedListItem` to our `Tokens` class:

markdown-processor/processor-v12.ts

```
public static readonly UnorderedListItem = new Token("* ");
```

Listing 2.27: Adding a new token type: UnorderedListItem

Now we need to extend our `Tokenizer` so that it recognizes unordered list items:

markdown-processor/processor-v12.ts

```
else if(   this._charStream.isFirstCharOfLine
        && this._charStream.tryConsume(Tokens.UnorderedListItem.value))
{
    return Tokens.UnorderedListItem;
}
```

Listing 2.28: Extending the Tokenizer class to recongize UnorderedListItems

Finally, our `MarkdownProcessor` has to translate `UnorderedListItem` tokens into HTML:

markdown-processor/processor-v12.ts

```
else if(token.is(Tokens.UnorderedListItem))
{
    // if we have already opened a <ul> tag, we can add the item
    // otherwise, we need to open the list first
    if(htmlBuilder.peek() == "li")
    {
        htmlBuilder.pop("li")
    }
    else
    {
        htmlBuilder.push("ul");
    }

    htmlBuilder.push("li");
}
```

Listing 2.29: Extending the MarkdownProcessor class to translate UnorderedListItem tokens

That's it already! Our newly added test should pass now. That's all it took to add a completely new feature: 22 LOCs and an added test case. This is the power of maintainability: It allows us to scale our solution with relative ease provided that we keep a look out for complexity.

CONCLUSION, RECAP, EXERCISES

Our simple Markdown processor is done. We have implemented some of the most important features. Of course, our solution is far from finished or perfect, and yet it has taught us many valuable lessons. We would still have to check for and implement more advanced use cases as well as test edge cases. We will discuss these topics in the context of other projects in later chapters, but for now the topic of Markdown shall be closed.

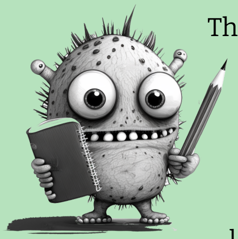
In this chapter, we set out to gain some hands-on experience under real-world conditions. We started with a clear goal of developing a simple Markdown processor. Our solution was guided by Gall's Law and the idea of starting with the simplest idea imaginable. This approach served us well in keeping our codebase simple while implementing some basic features.

The feature implementations were driven by unit tests, which we wrote after the implementation code. Understanding that testing is also a good way to understand what our codebase is required to do from a behavioral viewpoint, we set out to write tests first, that is: before implementing the features. We also realized how an ever growing body of test cases helps with spotting regressions.

At a certain point complexity crept in and we had to start rearranging our code. We used the technique of **Functional Decomposition** to break down a large function into smaller, more cohesive functions. This worked well and allowed us to implement some more features until (you name it!) complexity struck yet again.

At that point, we had to resort to some more heavy refactoring, during which we broke down our codebase into several clearly separated subdomains – lead by the principle of Separation of Concerns. The final solution kept complexity to a minimum and allowed us implement new features with relative ease.

exercises



The following exercises are meant to sharpen your skills and apply the knowledge you have gained in this chapter. They are not required to continue with the book, but they will help you internalize the concepts and techniques we have discussed so far. If you do not want to do all the exercises, feel free to skip some and pick those that you deem most useful.

- **Exercise 2.1**: Have a look at Listing 2.1 and Image 2.1. There are two features we haven't implemented in our solution: ordered list items and links. Extend our codebase by implementing these.
- **Exercise 2.2**: Check our implementation of the *Text Domain* in Listing 2.22. Why didn't we name the `Chars.Asterisk` variable simply `Chars.EmphasisCharacter` or similar?
- **Exercise 2.3**: Revisit the various sub-domains our codebase is divided into. What is the purpose of each sub-domain? Why do we need sub-domains at all?
- **Exercise 2.4**: Explain the following development terms and dichotomies using examples from the code we implemented: **implementation** vs. **interface**, **abstraction**, **information hiding**, **trade-off**, **technique** vs. **technology**, **functional decomposition**, **refactoring**.
- **Exercise 2.5**: Explain Gall's Law and how we followed it in our solution.
- **Exercise 2.6**: Find code in our solution that exemplifies the concept of **encapsulation**. Why do we need it? What are the benefits?
- **Exercise 2.7**: Find code in our solution that exemplifies the principle of Separation of Concerns. What is it? How does it help us?
- **Exercise 2.8**: What is a software developer's prime responsibility? Can you exemplify this using code from our solution?

APPENDIX A: ENDNOTES

This Appendix contains notes that expand on ideas in the chapters. These notes are not essential to the discourse of the book yet may provide additional explanations as well as food for thought to some readers.

A1 ON COUNTING THE DEVELOPER'S WAY (CH. 0)

You might be wondering why the chapter numbering starts at zero. Unlike in everyday life, where we count with natural numbers (and hence start counting at one), developers start counting at zero. This may seem unnatural at first and it certainly takes some time getting used to.

The reason behind this odd way of counting is that developers frequently work with data structures that keep multiple elements of the same type closely together (e.g. arrays) and thus require a contiguous block of memory (i.e. a block with no breaks in between its elements). The advantage here is that the location of each element inside the block can easily be calculated through its numerical index.

Instead of saving the memory location for each element separately (which would waste a lot of resources), we only need to save the memory location (called *base address*) of the block itself (which is the location of the first element, hence index zero) and the total size of the block (total number of elements). Using a simple calculation like $ma(i) = b + i * s$, where $ma(i)$ denotes the memory address of element i, b is the base address (location of the memory block), and s is the size of each element. This is a very efficient way of storing and retrieving data and is used in many programming languages. An example of such a calculation is shown in ➜ Image A1.1.

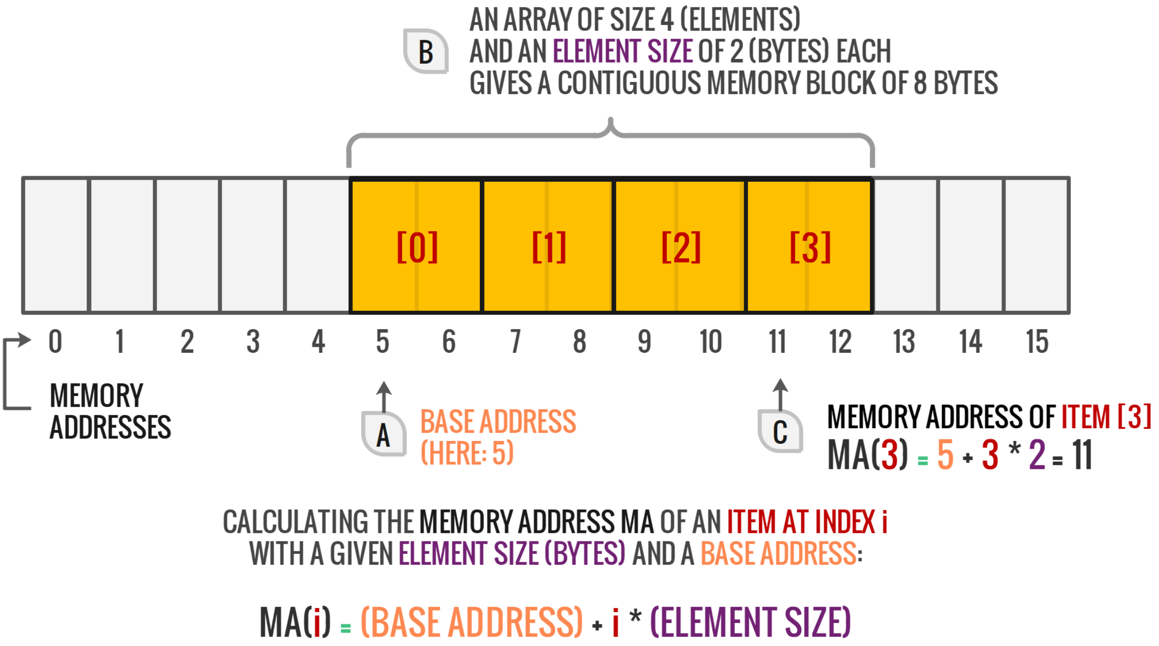

B
AN ARRAY OF SIZE 4 (ELEMENTS)
AND AN ELEMENT SIZE OF 2 (BYTES) EACH
GIVES A CONTIGUOUS MEMORY BLOCK OF 8 BYTES
[0]
[1]
[2]
[3]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
MEMORY ADDRESSES
A
BASE ADDRESS (HERE: 5)
C
MEMORY ADDRESS OF ITEM [3]
MA(3) = 5 + 3 * 2 = 11
CALCULATING THE MEMORY ADDRESS MA OF AN ITEM AT INDEX i
WITH A GIVEN ELEMENT SIZE (BYTES) AND A BASE ADDRESS:
MA(i) = (BASE ADDRESS) + i * (ELEMENT SIZE)

Image A1.1: Memory Allocation of an Array A simplified illustration of how an array is stored in memory. The array consists of four elements, each of the same size of 2 bytes [B]. The base address is 5 [A]. The element with index 0 is located at address 5, index 1 at address 7, index 2 at 9, index 3 at 11 [C].

The downside to this kind of counting is that when iterating over all items of an array with n elements in total, we need to keep in mind that the last element's index is *n-1* and not *n* (e.g. given ten items in an array, their indices are 0 to 9, not 1 to 10). This can be overlooked easily and lead to bugs that are so frequent that they even have a name: ➔ Off-By-One Error (OBE).

DEVELOPER SPEAK

OFF-BY-ONE ERRORS (OBEs) are common errors that occur when developers access elements of an array or other contiguous data structures counting from one (the natural way of counting) instead of zero, which is how computers store such data. For example: An array with ten elements has the indices 0 to 9, not 1 to 10. The last element is at index 9, not 10.

The following code listing exemplifies a typical Off-By-One Error when iterating over elements of an array:

```
const persons = ["Alice", "Bob", "Caesar", "Dora"];

for(let i = 0; i <= persons.length; i++)
```

```
4 {
5     // logs: Alice, Bob, Caesar, Dora, undefined
6     console.log(persons[i]);
7 }
```

Listing A1.1: A typical Off-By-One Error when iterating over the elements of an array.

The problem with Listing A1.1 is that it iterates five times despite the array only having four elements. The last iteration (when *i* is `4`) tries to access the fifth element of the array, which does not exist. Instead of throwing an error, JavaScript returns *undefined* in this case.

The culprit here is the condition `i <= persons.length` on line 3. It should actually be `i < persons.length` because the index that equals the length of the array is one too high (we start counting at zero, remember?).

A simple way of avoiding such OBEs is by ditching the "manual" `for` loop and use `for each` or a similar, more declarative [14] method that avoids manual counting. The following code listing shows how the OBE could have been avoided:

```
const persons = ["Alice", "Bob", "Caesar", "Dora"];

for(let person of persons)
{
    // logs: Alice, Bob, Caesar, Dora
    console.log(person);
}
```

Listing A1.2: A declarative programming style can avoid OBEs in for loops.

A2 ON GRAMMAR, WORDS, AND NATURAL VS. ARTIFICAL LANGUAGES (CH. 1)

Interestingly, we also call the basic unit of data that computers work with 'words'. In the same vein, computer languages have grammar (syntax) and vocabulary. The resemblance to natural languages doesn't stop there.

From a linguistic point of view, natural languages and programming languages can be placed on two very different complexity levels of language in general. If you have ever dealt with foreign languages at school (remember your French classes?), you certainly know that learning a natural language is so much more difficult than learning the syntax of a programming language.

---

[14] We will soon learn more about the meaning of *declarative* and what it entails. For the time being, just understand it as something synonymous with *expressive, direct, elucidating, communicative*.

A3 ON TECHNIQUE VS. TECHNOLOGY (CH. 1)

There is an important distinction to be made between two seemingly similar (yet fundamentally different) concepts: technique vs. technology.

Developer Speak

Technique (from the Greek word 'τέχνη': art) refers to an abstract way of doing things. It can be translated with 'craft' and 'knowledge', which are both involved in the process. Technique is meant to solve abstract, general problems independent of the concrete tools (see ➜Technology) used.

Technology (from the Greek word 'τεχνολογία': systematic treatment, related to but different from 'τέχνη'), on the other hand, refers to a concrete tool, usually a piece of software or hardware (or combination thereof) meant to solve specific problems.

An example will shed light on the differences between the two terms. Many applications allow multiple users to edit data concurrently (= "in parallel"). This can cause conflicts when two or more users edit the same data at the same time (concurrency issues). To mitigate the risks involved therein, you need to handle such conflicts gracefully, e.g. by allowing users to lock data before editing it (so that no other user can edit it at the same time). Knowing that these concurrency issues exist and how to address them in general is a technique ("conflict resolution"), i.e. abstract knowledge gained over years of experience. However, knowing what concrete options exist on a given hardware and software stack (e.g. MySQL vs. MongoDb) is technology-related knowledge.

While technique is usually cumulated, inferred wisdom that shows a certain resistance to change, evolves comparatively slowly, and adapts to new situations, technology is fast-paced and becomes obsolete rather quickly (see ➜Image A3.1).

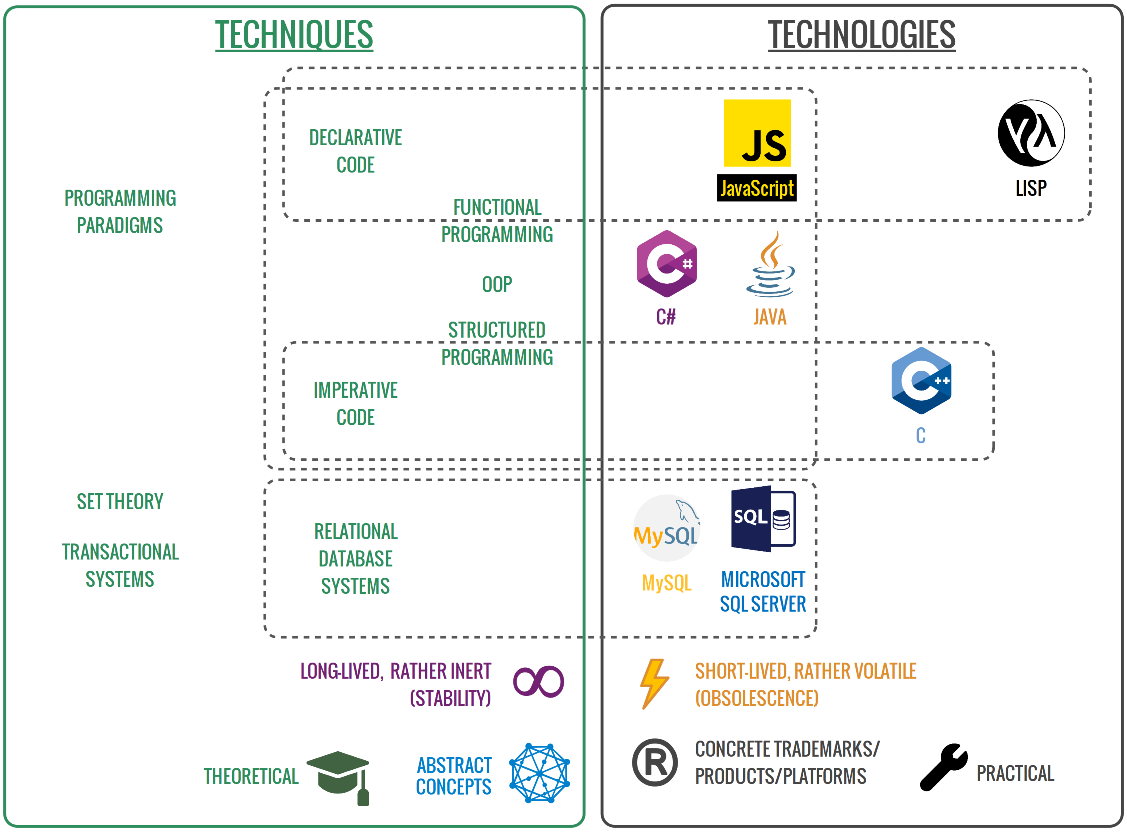


Image A3.1: Technique vs. Technology The two domains of technique vs. technology compared, with examples of what they entail (not exhaustive).

The key takeaway here is that learning technologies is time well invested if they are taken as means to an end. The end should always be practicing and furthering our understanding of the techniques behind them.

Obviously, technological knowledge is essential because as developers we craft tangible solutions, not theories. But the risk of acquairing soon-to-be obsolete knowledge is hanging over us like a sword of Damocles, which is why the meta knowledge of techniques is so important: it transfers over to new technologies easily and allows us to deal with the incessant influx of new, often short-lived technological products and changes by pivoting into new roles and perspectives without ever having to start from scratch again.

A4 ON TRADE-OFFS AND TRADE-OFF CONTINUA (CH. 1)

To exemplify what a typical trade-off could look like, imagine we're developing a note-taking app that allows users to create small pieces of texts. Those texts need to be persisted [15] between sessions so that when a user logs off the system and logs on again at a later point, all their texts are still there.

Now, we can either develop our own persistence storage (say, a relational database) or hook our app up to a third-party API that does the persistence for us (say, a cloud-based service). The trade-off to consider here is *ease of maintenance vs. vendor independence*: Creating our own database (Point **A** in ➔Image A4.1) gives us maximum independence (all software is under our control) but has significantly higher maintenance costs attached to it (because we have to develop and maintain the database system ourselves). Using a cloud-based, third-party storage service [Point **B**], on the other hand, gives us a maximum ease of maintenance (the third party will maintain the software for us) but also makes our product entirely dependent on it. Should the third-party decide to cease its service (a worst-case scenario), we are left with a non-functional app.

We can see clearly that the two opposing extremes (points **A** and **B**) are mutually exclusive and as such associated with significant disadvantages each (hence the *red zones*). Luckily, these two goods aren't simply binary but gradual in nature, so there is a continuum (varying degrees) between them. We are likely to find a "sweet spot" (point **C**) somwhere in the middle where we can get acceptable degrees of both goods (hence the *green zone*).

---

[15] A fancy word for "to save": to persist = to continue to exist, to save to permanent memory, e.g. hard disk (as opposed to volatile memory, e.g. RAM).

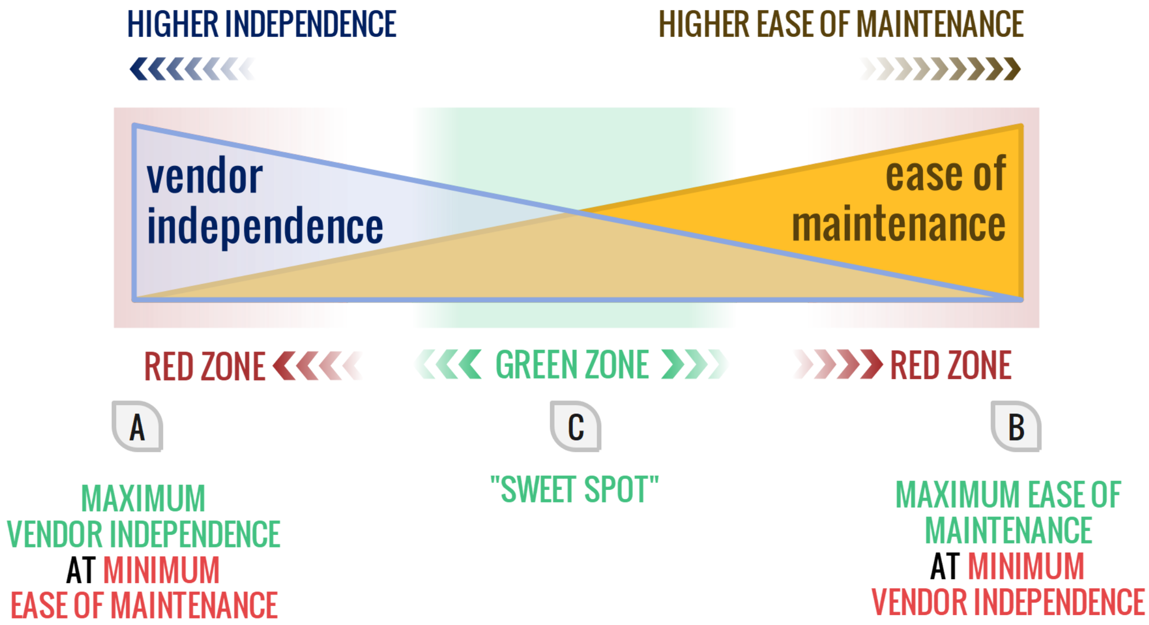

HIGHER INDEPENDENCE
HIGHER EASE OF MAINTENANCE
vendor independence
ease of maintenance
RED ZONE
GREEN ZONE
RED ZONE
A
C
B
MAXIMUM VENDOR INDEPENDENCE AT MINIMUM EASE OF MAINTENANCE
"SWEET SPOT"
MAXIMUM EASE OF MAINTENANCE AT MINIMUM VENDOR INDEPENDENCE

Image A4.1: A Typical Trade-Off Continuum The (somewhat idealized) trade-off depicted here spans a continuum between the two qualities "vendor independence" and "ease of maintenance". The red zones show areas that are generally unfavorable because one of the qualities tends towards zero. The green zone usually holds the "sweet spot" where a reasonable compromise between the two can be found.

In our example, we could opt for a self-hosted, off-the-shelf (third-party) database system. That way, we wouldn't have to develop the system ourselves but we would still host it in-house (*on premise*) so as not to depend entirely on a third-party service. This trade-off gives us a reasonable amount of vendor independence while keeping the ease of maintenance at a manageable level.

To clarify, decisions like this one can usually be analyzed under various opposing pairs of qualities, hence different types of continua and trade-offs co-exist at the same time. At the core of every decision we make lies our ability to reason about the bigger picture in terms of a multitude of co-existing trade-offs. Battles in tech are lost and won based on how me make sense of those.

A5 ON ITERATIVE VS. INCREMENTAL PROCESSES AND SOFTWARE DEVELOPMENT (CH. 2)

The two concepts *iterative* and *incremental* are sometimes used synonymously, even though they denote two different processes.

An *iterative* process involves repetition ("iteration"). Looping over a series of numbers (say from zero to 100) is an iterative process. So is riding a bicycle (our legs do the iterative work).

An *incremental* process, on the other hand, involves growth, an addition of something to what already exists. Incrementing an integer is (logically) an incremental process as it adds or subtracts a given amount to an already existing quantity, usually one. Climbing a flight of stairs is another example of an incremental process (each step adding to the sum of steps already climbed).

Iterative processes need not necessarily be incremental ones, and vice versa. Incrementing an integer variable in a loop is both incremental and iterative. However, one could also loop without incrementing, or increment without looping.

As far as software development is concerned, the process is usually both iterative (developers analyze, rearrange, and refine a codebase repetitively) and incremental (developers add small changes and features to the codebase each time they work on it).

A6 ON TIGHT FEEDBACK LOOPS (CH. 2)

Feedback is the response or reaction to an action. It's called that way because it "feeds back" into your decision-making by helping you evaluate your previous action. Such feedback is *tight* if the response follows quickly and allows you to adapt your actions constantly.

Let's consider a simple example (see ➚**Image A6.1**): Assuming we are dealing with JavaScript and want to know if zeroes are among the many expressions interpreted as `false` (type coercion). Instead of looking up the relevant bits in the docs, we can just have the runtime answer this question for us. We start our browser and enter `!0` into the console. The `!` is the logical NOT operator (negation).[16] It negates a boolean expression (`true` becomes `false`, and vice versa). The console spits out `true`, so it follows that `0`, while not a boolean itself, was indeed interpreted as `false`.

A tight feedback loop is essential in software development because it gives you immediate knowledge that you can act on. If such loops follow in rapid succession, they help you accumulate insights quickly and thus serve as a most valuable corrective in your decision-making.

---

[16] See ➚**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_NOT**

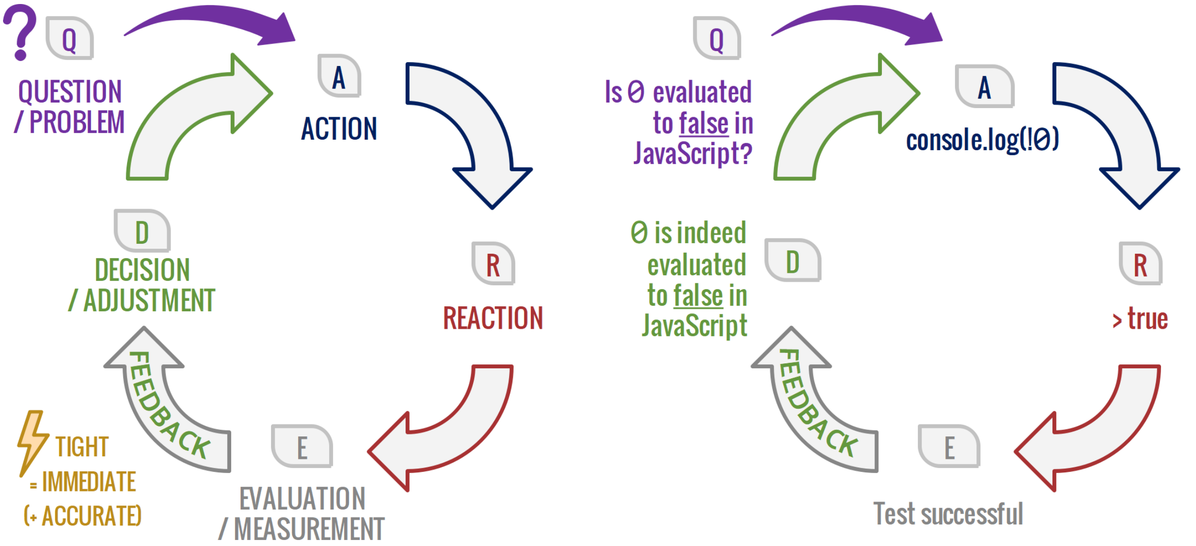


Image A6.1: Tight Feedback Loop A feedback loop starts with an **A**ction, awaits a **R**eaction, which it then **E**valuates in order to make a **D**ecision that guides the next action. Frequently, **Q**uestions or problems instigate the action that sets off the loop. A feedback loop is tight when it is accurate and all its parts execute instantly. Tight feedback looping tends to happen repetitively.

A7 ON BRAINS VS. COMPUTERS (CH. 2)

Our brain is an evolutionary mircale and quite adept at handling a plethora of information. Clearly, it needs to because we are constantly surrounded by it. It's not surprising then that our brain has often been used as a metaphor for computers (and vice versa). But the two are not the same. The brain is far from being as perfect a processor or storage device as the computer.

Unlike computers, which are very good at keeping track of details and remembering things (but plainly error out as soon as they run out of memory!), our brains are evolutionarily programmed to keep working even under intense pressure (mental load!). To do so, they resort to a little trick: When complexity rises above certain levels (for example, a large number of items to remember in our short-term memory), our brains tend to start rearranging information through generalization. This process involves dropping details in favor of broader subsumptions (trade-offs!). That way, we can still make sense of our complex world, but only on a higher abstractional level.

Now, generalization under a high mental load may work splendidly in real life. When reasoning about code, though, it can easily run into problems! The processing of information done by computers and brains are fundamentally different because both opt for different coping mechanisms.

A8 ON SEPARATION OF CONCERNS (CH. 2)

The practice of separating things that deal with separate concerns is a guiding principle aptly called **SEPARATION OF CONCERNS (SOC)**.

It is one of the most important principles in software development and should always warn you to not mix things together that don't belong together. As such, the principle is based on the universal idea of **cohesion** and works on every level: from the concrete implementation level to the abstract architectural level.

For example, when implementing a class that deals with sending e-mails, we are well advised to keep the e-mail details (recipient, subject, body) separate from the details of how the e-mail is sent (SMTP, IMAP, etc.). The former is a concern of the e-mail itself, whereas the latter is a concern of the e-mail *transport*. Mixing the two concerns together would make our code unnecessarily complex and less maintainable.

To use a more mundane example: Have you ever wondered why you keep your stuff at home in rooms, drawers, cupboards, and the like? Why not just throw everything into one big pile? The answer is that it would be impossible to find anything. The same principle applies to software: If you mix things together that don't belong together, you will end up with a big pile of code that is impossible to maintain.

A9 ON NAMING THINGS (CH. 2)

Surprisingly, naming stuff (variables, classes, modules, whatever) is one of the toughest things to get right in programming, and that `StringBuilder` in ↗ Listing 2.11 is a case in point.

Why is naming important? As already discussed in ↗ Truths and Myths About Software Development (p. 13) that we write code for our fellow developers, not for machines. We need to communicate effectively with our colleagues so they know how to use, fix, and extend our code. Communicating intent works best when we use ↗ identifiers that are precise and expressive.

DEVELOPER SPEAK

IDENTIFIERS are names given to variables, functions, types, classes, and labels in our code. Identifiers allow us to refer to parts of our code or state kept in memory. Unlike keywords, identifiers are not part of the programming language itself but are defined by the programmer. That's why they usually follow a strict naming convention and have to be unique in the context they are used. This also means that identifiers cannot overlap with existing keywords. For example: `for` is a keyword for a control structure in JavaScript and TypeScript and hence you cannot use the name as an identifier for, say, a variable.

The key to finding good names is constant, critical reflection of what we have already written. Nothing we write is set in stone. Everything can (and likely will be) changed at a later time. When needing to find a good name for a new variable, say, we can always go with what comes to mind first. There's nothing wrong with that. But then, we should come back to that name later and ask ourselves: Is this name still a good fit? Does it still express the intent of the code? If not, we should change it. The idea here is to approach something like the perfect name in small increments.

A10 On Assessing Complexity (Ch. 2)

You might be wondering at this point: How do I know if a function is too long or too convoluted? Rather than giving you any arbitrary hard limits, consider what the function does: If it implements some rather simple logic, thirty LOCs might already be too much, while fifty might still be okay for more elaborate logic. As for LOIs, I personally try to keep them to a maximum of three. If I have more than three, I try to extract the logic into separate functions.

The key idea, though, is to keep local contexts as small as possible: By keepig code coherent (similar things go together, different things are better kept separate) and not mixing different abstractional levels into the same units (functions, classes, modules, etc.). With time you will gain enough experience to be able to listen to your gut feeling regarding complexity. Until then, you can always ask a colleague for a second opinion.

A11 On Premature Abstractions (Ch. 2)

To give a real-world example of a premature abstraction: If you send a few letters by snail mail a year, you can probably get away with just buying stamps at the post office and stamping them manually. Should you decide to buy a postage meter, instead, and have the stamping done automatically, you are probably falling victim to a premature abstraction.

The additional time and expenses needed to buy that machine will most probably never be amortized over its lifetime. Chances are you ain't gonna need that postage meter (➜YAGNI) and you would have been better off just doing the stamping manually. If at some point you find yourself sending a few hundred letters a year, you can still decide to automate the process.

APPENDIX B: SETTING UP A DEVELOPMENT ENVIRONMENT

To be written.

APPENDIX C: SHORT INTRODUCTION TO TYPESCRIPT

To be written.